Universidad Miguel Hernández

Departamento de Física y Arquitectura de Computadores

Diseño, estudio y evaluación de librerías numéricas en lenguajes de alto nivel para arquitecturas paralelas

Memoria presentada para optar
al grado de Doctor por:

Vicente Galiano Ibarra

dirigida por:

Violeta Migallón Gomis

José Penadés Martínez

UNIVERSITAS
Miguel
Hernández

Dña. Violeta Migallón Gomis y D. José Penadés Martínez, Catedráticos de Universidad de la Universidad de Alicante,

CERTIFICAN:

Que la presente memoria *Diseño, estudio y evaluación de librerías numéricas en lenguajes de alto nivel para arquitecturas paralelas*, ha sido realizada bajo su dirección, en el Departamento de Física y Arquitectura de Computadores de la Universidad Miguel Hernández, por el Ingeniero D. Vicente Galiano Ibarra, y constituye su tesis para optar al grado de Doctor.

Para que conste, en cumplimiento de la legislación vigente, autorizan la presentación de la referida tesis doctoral ante la Comisión de Doctorado de la Universidad Miguel Hernández, firmando el presente certificado.

Alicante, 1 de abril de 2007.

Fdo.: Violeta Migallón Gomis José Penadés Martínez

UNIVERSITAS
Miguel
Hernández

D. Gabriel Ruiz Ruiz, Profesor Titular de Escuela Universitaria y director del Departamento de Física y Arquitectura de Computadores de la Universidad Miguel Hernández,

CERTIFICA:

Que la presente memoria *Diseño, estudio y evaluación de librerías numéricas en lenguajes de alto nivel para arquitecturas paralelas*, realizada bajo la dirección de Dña. Violeta Migallón Gomis y D. José Penadés Martínez, en el Departamento de Física y Arquitectura de Computadores de la Universidad Miguel Hernández, por el Ingeniero D. Vicente Galiano Ibarra, constituye su tesis para optar al grado de Doctor.

Para que conste, en cumplimiento de la legislación vigente, autoriza la presentación de la referida tesis doctoral ante la Comisión de Doctorado de la Universidad Miguel Hernández, firmando el presente certificado.

Orihuela, 1 de abril de 2007.

Fdo.: Gabriel Ruiz Ruiz

UNIVERSITAS
Miguel
Hernández

A Mavi,

A mis padres, Jose y Susi.

UNIVERSITAS
Miguel
Hernández

Resulta curioso que los agradecimientos sean lo último que se escribe en una tesis y aparezcan al principio de la misma. Es un momento en el que uno busca en su interior y recapitula todo el esfuerzo, el trabajo y, sobre todo, la ayuda recibida por aquellos a los que ahora, quiero expresar mi más sincera gratitud. Especialmente, quiero agradecer a mis directores, José Penadés y Violeta Migallón, por la confianza depositada en mí, por tantos esfuerzos conjuntos y sobre todo por dirigirme con un saber hacer que lo ha convertido en un agradable viaje.

Asimismo, quiero agradecer a mis compañeros del Departamento de Física y Arquitectura de Computadores el apoyo que me han proporcionado, en especial, mi gratitud va dirigida a Héctor, Miguel, Otoniel, Pablo, Adrián y Manuel por esos buenos momentos al estilo *Camera Café*.

También quiero dar las gracias a Osni Marques y a Tony Drummond por invitarme a realizar una corta estancia en el Lawrence Berkeley National Laboratory y por facilitarme el acceso a los recursos informáticos del Centro de Computación Científica del Departamento de Energía de Estados Unidos. En especial, quiero agradecer a Tony Drummond su apoyo explícito para la realización y difusión de este trabajo.

En mi mundo personal y privado también ha estado presente la tesis, y han contribuido a la finalización de la misma mi familia y mis amigos. A mis hermanos Jose y Jasi, y a los que se han convertido en mi familia desde hace cuatro años, Aurelio y Pepita, agradecerles el apoyo constante y el ánimo recibido. A mis amigos Esteban, Boria, Pacheco, Ángel, David, Antonio, Juan, Viken y especialmente, a mi primo y amigo Vicente, gracias por estar cuando os he necesitado.

Mis mayores agradecimientos son para las personas a las que he dedicado este trabajo. A mis padres, Pepe y Susi, quiero agradecerles y corresponderles de algún modo por tanta dedicación con sus hijos, y animarles en su lucha personal, para que sigan estando ahí. También he dedicado este trabajo a mi esposa, Mavi, que ha sido una incansable compañera de viaje, alentándome y animándome continuamente para poder llevar esta tesis a buen puerto. Mavi, mi vida, ya hemos llegado al destino llamado tesis, pero nos quedan tantos otros caminos por hacer juntos...

UNIVERSITAS
Miguel
Hernández

Índice

Prólogo

El desarrollo de aplicaciones científicas y de ingeniería en un entorno de alto rendimiento suele ser un proceso complejo que requiere de la adecuada información sobre los recursos computacionales disponibles y sobre las herramientas software necesarias. En consecuencia, ante una de estas aplicaciones, los científicos e ingenieros deben utilizar una gran cantidad de tiempo en el desarrollo y ejecución de sus códigos sobre entornos computacionales de altas prestaciones. Esto exige de un conocimiento, en ocasiones complejo, sobre los recursos adecuados que se utilizan. En esta línea de trabajo, proponemos un conjunto de herramientas útiles en aplicaciones de la computación científica que requieran elevadas exigencias computacionales, y que faciliten la labor de desarrollo.

Nuestro trabajo se centra en el desarrollo de un conjunto de interfaces de alto nivel a la colección de librerías ACTS [16, 81]. Estas librerías consisten en un conjunto de rutinas útiles en el desarrollo de aplicaciones científicas de altas prestaciones utilizando código robusto y escalable. La utilización de estas librerías incluidas en ACTS puede resultar compleja para un programador cuya área de investigación no tenga relación con la computación paralela. De este modo, tratamos de acercar este conjunto de librerías a un lenguaje de alto nivel sin necesidad de perder eficiencia en el rendimiento de dichas librerías. De los diferentes lenguajes de alto nivel disponibles actualmente, se optó por el lenguaje Python [96] por sus características particulares. Estas características se pueden resumir en que Python es un lenguaje de código abierto, intuitivo, con una corta curva de aprendizaje, interpretado, orientado a objetos, con un tipo de datos muy dinámico, y cuyo punto fuerte es la capacidad de fusionar de una manera sencilla librerías escritas en otros lenguajes de programación.

En esta memoria presentamos las librerías PyACTS y PyPNetCDF que tratan de cubrir las necesidades descritas anteriormente. En el capítulo 1, se verá el estado actual de las arquitecturas uniprocesador o de procesamiento específico, como son los procesadores vectoriales, cuáles son sus virtudes, sus características específicas y sus limitaciones.

En el capítulo 2, veremos las características de los sistemas multiprocesador, tanto de memoria distribuida como de memoria compartida, los clasificaremos según la clasificación propuesta por Flynn y explicaremos los entornos de desarrollo para procesamiento paralelo más utilizados.

En el capítulo 3, realizaremos una introducción al lenguaje Python detallando aquellas características que nos serán útiles en nuestros objetivos y revisaremos los módulos y herramientas desarrollados por terceros, que nos permitan la creación de la distribución propuesta.

En el capítulo 4, se describen las características generales del software de computación científica, y de forma más específica, se introducen las librerías de paso de mensajes y las librerías pertenecientes a la colección ACTS sobre las que se desarrolla este trabajo.

Los capítulos del 5 al 9 constituyen el cuerpo principal de esta memoria. En el capítulo 5, se detalla la estructura de PyACTS, compuesta por paquetes, módulos y extensiones. Para facilitar la integración de las distintas librerías de ACTS, se define la matriz PyACTS y las funciones auxiliares que permiten crear un entorno de ejecución sencillo y transparente a un usuario no iniciado en la computación paralela. Además, detallamos cuál ha sido el proceso seguido para crear esta distribución.

En los capítulos 6, 7 y 8, se presentan los módulos PyBLACS, PyPBLAS y PyScaLAPACK, respectivamente. Para cada uno de ellos, expondremos su estructura de rutinas y compararemos la sintaxis y el rendimiento obtenido con respecto a una programación tradicional en Fortran o C.

En el capítulo 9, se presenta la librería PyPnetCDF, introduciendo previamente el estándar de archivos de datos netCDF y las librerías y entornos disponibles para el acceso a este tipo de ficheros.

En el capítulo 10, se exponen dos aplicaciones que hacen uso de PyACTS: la implementación del algoritmo del gradiente conjugado y la paralelización de un software para el análisis de la variabilidad climática.

El último capítulo está dedicado a la presentación de las conclusiones obtenidas en

este trabajo y apuntaremos cuáles pueden ser las posibles tareas a realizar como líneas futuras.

El trabajo realizado ha sido parcialmente financiado por la División de Matemáticas, Informática y Ciencias Computacionales del Departamento de Energía de los Estados Unidos (bajo contrato DE-AC03-76SF00098), el Ministerio de Educación y Ciencia (TIN2005-093070-C02-02), y la Universidad de Alicante (VIGROB-020).

UNIVERSITAS
Miguel
Hernández

Capítulo 1

Arquitecturas convencionales avanzadas

1.1. Introducción

En las últimas décadas el avance de la tecnología de los microprocesadores ha sido tremendo, a la vez que se aumenta la frecuencia de trabajo, los microprocesadores son capaces de ejecutar múltiples instrucciones en un mismo ciclo de reloj. Esta evolución exige mayores prestaciones de los caminos de datos y de la memoria, de forma que los avances han de tender también a evitar los posibles cuellos de botella generados por ambos. Aparecen arquitecturas, como los multiprocesadores de memoria compartida y los multicomputadores, que suministran una mayor capacidad de los caminos de datos, que incrementan el acceso a los sistemas de almacenamiento y que consiguen que estas arquitecturas sean escalables y de bajo coste, todo esto implica la aparición de una amplia gama de aplicaciones de la computación paralela.

Tradicionalmente el desarrollo de software paralelo ha implicado un trabajo intensivo durante largos periodos de tiempo, ya sea por el problema de coordinar las diferentes tareas concurrentes, por conseguir algoritmos portátiles, por carecer de entornos estándar o por carecer de herramientas de desarrollo software. Esto provocaba que aplicaciones desarrolladas durante mucho tiempo perdieran su eficacia rápidamente al no poder adaptarse

a las nuevas arquitecturas surgidas durante el desarrollo de dichas aplicaciones. En la actualidad disponemos de las interfaces hardware y de los entornos de desarrollo estándar para poder asegurar un ciclo de vida largo a las aplicaciones paralelas, las cuales serán el método óptimo para el máximo aprovechamiento tanto de las actuales arquitecturas avanzadas de los computadores como de las tendencias futuras.

El procesamiento paralelo tiene, como objetivo principal, explotar el paralelismo inherente de las aplicaciones informáticas. Como las aplicaciones informáticas tienen diferentes características, también presentan un perfil diferente frente al paralelismo, es decir que unas aplicaciones tendrán gran paralelismo potencial y otras no tanto. Unido a este factor es necesario tener en cuenta un factor cualitativo: *Cómo se explotará el paralelismo.* Existen tres fuentes principales de paralelismo:

- *Paralelismo de control:* Se basa en que existen acciones, tareas o procesos que pueden ejecutarse más o menos independientemente sobre los recursos de cálculo disponibles.

- *Paralelismo de datos:* La explotación de este paralelismo viene dado porque ciertas aplicaciones trabajan con estructuras de datos muy regulares, estas aplicaciones repiten una misma acción sobre cada elemento de esas estructuras, ya sean vectores o matrices.

- *Paralelismo de flujo:* Esta fuente de paralelismo se puede explotar al constatar que existen aplicaciones que funcionan en modo cadena, de forma que se dispone de un flujo de datos y sobre cada dato se efectúan una serie de operaciones en cascada, los resultados de una operación servirán de operando para la siguiente operación.

En este capítulo veremos las características de los computadores *monoprocesador* de altas prestaciones, que explotan los distintos tipos de paralelismo, y cómo realizan esta explotación, ya que en función del tipo o tipos de paralelismo que se quiera utilizar, para aumentar las prestaciones de un computador, nos encontramos con diferentes arquitecturas (ver secciones 1.3, 1.4, 1.5 y 1.6). Previamente, en la sección 1.2 veremos cómo podemos clasificar las diferentes arquitecturas de computadores.

1.2. Clasificación de las arquitecturas de los computadores

Las arquitecturas de los computadores se pueden clasificar en función de su estructura o de su comportamiento. Los principales métodos de clasificación consideran el número de conjuntos de instrucciones que pueden ser procesados simultáneamente, la organización interna de los procesadores, la estructura de la conexión interprocesador, o los métodos empleados para calcular los flujos de instrucciones y datos dentro del sistema.

Una de las clasificaciones que atienden al flujo de instrucciones y datos que podemos considerar, es la creada en 1972 por Flynn, que se caracteriza por ser una de las más simples utilizada en la actualidad [45, 46]. La clasificación de Flynn (ver figura 1) se basa en la posibilidad de procesar uno o más flujos de instrucciones y/o datos. Según esto un computador puede ser capaz de procesar un flujo **S**imple de **I**nstrucciones o bien **M**últiples flujos de **I**nstrucciones. Análogamente puede ser capaz de procesar un flujo **S**imple de **D**atos o bien **M**últiples flujos de **D**atos. De esta forma pueden darse cuatro posibilidades:

SISD: Un flujo simple de instrucciones opera sobre un flujo simple de datos. Este modelo es el clásico de von Neumman que ejecuta instrucciones de forma secuencial.

SIMD: Un flujo simple de instrucciones opera sobre múltiples flujos de datos. Todos los procesadores ejecutan la misma instrucción, pero sobre distintos datos.

MIMD: Cada procesador ejecuta su propio código sobre datos distintos a los de los otros procesadores.

MISD: Un flujo múltiple de instrucciones opera sobre un flujo simple de datos. Corresponde a un modelo con poca utilidad práctica, pero que últimamente se ha utilizado en arquitecturas como los array de procesadores o en algunas redes neuronales.

En la actualidad los avances en las tecnologías han dado lugar a arquitecturas que no encajan de forma clara en esta taxonomía como por ejemplo, los procesadores vectoriales que son una mezcla de arquitectura SIMD y MISD, y las arquitecturas híbridas. Otras clasificaciones, como la de Kuck [75], tienen en cuenta más opciones, al considerar tanto el flujo de ejecución como el flujo de instrucciones, ambos flujos se consideran sobre escalares o sobre arrays, obteniendo 16 categorías frente a las 4 categorías que proporciona la

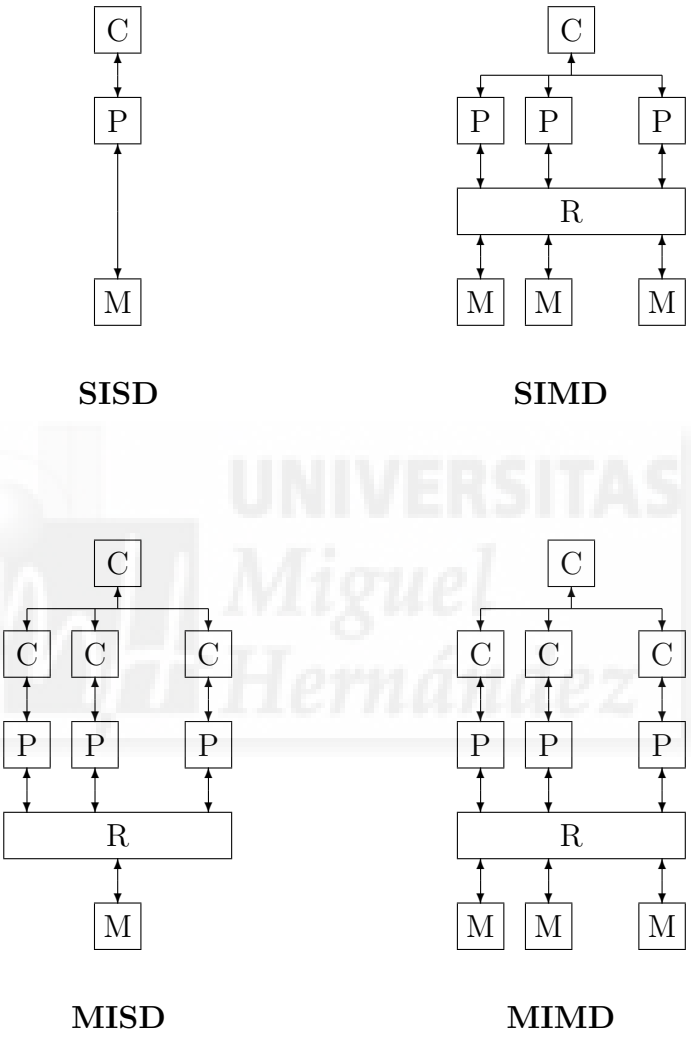


Figura 1.1: Clasificación de Flynn. C = Controlador. P = Procesador. R = Red de Interconexión. M = Memoria.

taxonomía de Flynn.

1.3. Procesadores superescalares

En esta sección veremos qué es un procesador superescalar, sus limitaciones y cómo estos procesadores consiguen aumentar el rendimiento obtenido.

Una implementación superescalar de la arquitectura de un procesador es aquella en que las instrucciones comunes pueden iniciar su ejecución simultáneamente y ejecutarse de manera independiente. Es decir disponemos de varios cauces, cada uno de los cuales consta de múltiples etapas.

La mayor parte de las operaciones, en la mayoría de aplicaciones, se realizan con cantidades escalares. Por tanto la arquitectura superescalar es la evolución de los procesadores de propósito general de altas prestaciones.

1.3.1. Limitaciones al paralelismo a nivel de instrucción

Si se supone un procesador superescalar en el cual cada etapa del cauce ha sido *ensanchada* para acoger a dos instrucciones, cada par de instrucciones se ejecuta simultáneamente y el procesador puede operar dos veces más rápido que su versión escalar. Este incremento de prestaciones no puede alcanzarse siempre, debido a que pueden existir dependencias en la secuencia de instrucciones.

La expresión *paralelismo a nivel de instrucción* se refiere al grado en que, en promedio, las instrucciones de un programa pueden ejecutarse simultáneamente. Para maximizar el paralelismo a nivel de instrucción debe usarse una combinación de optimizaciones realizadas por el compilador y de técnicas implementadas por el hardware. Las limitaciones fundamentales que ha de resolver el sistema son las siguientes:

- *Dependencia de datos verdadera:* También denominada dependencia de flujo o dependencia de escritura–lectura, se produce cuando una instrucción necesita resulta-

dos producidos por una instrucción que secuencialmente debe ejecutarse con anterioridad.

- *Dependencia relativa al procedimiento:* La presencia de bifurcaciones en una secuencia de instrucciones produce complicaciones en el cauce. Las instrucciones que siguen a una bifurcación no se pueden ejecutar hasta que se ejecute la bifurcación.
- *Conflictos en los recursos:* Un conflicto en un recurso es una competición de dos o más instrucciones por el mismo recurso al mismo tiempo.
- *Dependencia de salida:* También llamada de escritura–escritura, se produce cuando dos instrucciones vuelcan su resultado en el mismo registro, y existe la posibilidad de que la instrucción que secuencialmente es anterior finalice con posterioridad, lo cual no debe permitirse para no obtener un resultado erróneo.
- *Antidependencia:* También denominada dependencia de lectura–escritura, puede ocurrir cuando una instrucción utiliza como operando el resultado de una instrucción posterior, pero que pese a ser posterior pudiera finalizar con anterioridad, produciendo un operando erróneo.

1.3.2. Política de emisión de instrucciones

Uno de los métodos para aumentar el rendimiento en los procesadores superescalares es variar el protocolo usado para emitir las instrucciones, y no ceñirse a una emisión secuencial pura. Normalmente se utiliza el término *emisión de instrucciones* para referirse al proceso de iniciar la ejecución de instrucciones en las unidades funcionales del procesador y el término *política de emisión de instrucciones* para referirse al protocolo usado para emitir las instrucciones.

Un procesador superescalar intentará localizar instrucciones, más allá del punto de ejecución en curso, que puedan introducirse en el cauce y ejecutarse. Por tanto, tendremos tres ordenaciones importantes: orden de captación de instrucciones, orden de ejecución de instrucciones y orden en el cual las instrucciones alteran registros y posiciones de memoria. La única restricción con la cual se encuentra el procesador es, lógicamente, que el resultado debe ser correcto, y por tanto el procesador debe ser capaz de acomodar las dependencias y conflictos comentados con anterioridad.

En un procesador con política de emisión de instrucciones ordenada, como es el caso de los procesadores escalares, el procesador sólo descodificará instrucciones hasta el punto en que exista dependencia o conflicto, y no se descodifican más instrucciones hasta que el conflicto se resuelva. Por tanto, el procesador no puede buscar instrucciones más allá, las cuales, sin embargo, podrían ser independientes. La característica especial de los procesadores superescalares respecto a los escalares es la emisión desordenada.

Para permitir la emisión desordenada es necesario desacoplar las etapas de descodificación y de ejecución. Esto se hace mediante un buffer llamado *ventana de instrucciones*. El procesador va descodificando instrucciones y las coloca en la ventana de instrucciones, lo cual significa que está lista para ejecutarse. Estas instrucciones serán ejecutadas cuando sea factible, es decir, cuando los operandos estén disponibles y la unidad funcional asociada esté libre. Esta ventana de instrucciones puede ser centralizada o distribuida. Una ventana de instrucciones centralizada almacena todas las instrucciones pendientes de ejecución sin importar el tipo de instrucción de que se trate. En la ventana de instrucción distribuida, los registros de almacenamiento de instrucciones, que se denominan *puestos de reserva*, se colocan ante una determinada unidad funcional, que será en la que han de ejecutarse.

1.4. Procesadores VLIW

La aproximación VLIW (Very Long Instruction Word) consiste en incorporar varias pequeñas operaciones en una palabra de instrucción larga. Es necesario que el compilador sea capaz de generar estas instrucciones, y que además éstas sean lo suficientemente grandes, para proporcionar los bits necesarios de control de varias unidades funcionales. Por tanto, una arquitectura VLIW dispone de muchas más unidades funcionales que el procesador típico, además de un compilador que busca paralelismo para mantener a dichas unidades funcionales tan ocupadas como sea posible. El compilador debe compactar código secuencial común en palabras de instrucción largas, ya que son éstas las que hacen mejor uso de los recursos.

1.5. Procesadores supersegmentados

En un procesador supersegmentado para aumentar el rendimiento del procesador se utilizan más etapas, y de grano más fino, en el cauce de ejecución de instrucciones. Conseguimos, al tener más etapas, que un número mayor de instrucciones estén en el cauce, aumentando así el paralelismo. Tener etapas de grano más fino implica que la complejidad de cada etapa se minimiza, permitiendo el aumento de la frecuencia de reloj. Con esta técnica, denominada supersegmentación, se consiguen rendimientos similares al diseño superescalar. Esta técnica puede usarse conjuntamente con técnicas superescalares.

Una forma alternativa de enfocar un sistema supersegmentado es considerar que las funciones realizadas en cada etapa se pueden dividir en dos partes o más no solapadas, y que cada una se ejecuta en medio ciclo de reloj en el caso de dividirse en dos partes.

Comparando las arquitecturas vistas hasta ahora, se concluye que los procesadores superescalares y los VLIW son más sensibles a conflictos en los recursos que los supersegmentados. El procesador supersegmentado requiere una tecnología de transistor muy rápida para reducir el retraso del cauce de instrucción más largo, evitando que sean más lentos que los superescalares.

1.6. Procesadores vectoriales

Hasta ahora hemos visto que la utilización de procesadores superescales y supersegmentados mejora el rendimiento de un procesador, pero existen unas limitaciones al rendimiento de estos cauces. Las limitaciones se deben fundamentalmente a:

- *La frecuencia de funcionamiento del reloj:* Ésta puede incrementarse, como se realizaba para los cauces supersegmentados. Sin embargo el aumento de la frecuencia del reloj implica mayor número de etapas en el cauce. Llega un momento, en el cual, el aumento de las etapas produce una ralentización del procesador debido a las dependencias entre las instrucciones y a la sobrecarga de comunicaciones entre etapas.

- *La frecuencia de adquisición y descodificación de la instrucción:* También llamado *cuello de botella de Flynn*, se debe a la dificultad de adquirir y descodificar muchas instrucciones por ciclo de reloj. Esto impide la construcción de procesadores con frecuencia de reloj alta y con tasa alta de emisión de instrucciones.

Los procesadores de alta velocidad utilizan una memoria caché con el fin de evitar la alta latencia que supone el acceso a la memoria principal. Aunque las mejoras introducidas por las últimas arquitecturas caché han mejorado la eficiencia de la jerarquía de la memoria, no se ha evitado que esto siga siendo un problema. Podemos ver los conceptos de jerarquía de memoria en [2] o en [65].

Los procesadores vectoriales permiten realizar operaciones de alto nivel sobre vectores, Una sola instrucción típica, como es la suma de dos vectores, es equivalente, en una estructura no vectorial, a un bucle que realiza la misma operación sobre cada pareja de elementos de los vectores. Las instrucciones vectoriales tienen una serie de propiedades, las cuales resuelven gran parte de los problemas mencionados:

- El cálculo de cada resultado es independiente de los cálculos previos.
- Una instrucción vectorial lleva consigo una gran carga de trabajo, con lo que el efecto de cuello de botella de Flynn se minimiza, al ser necesario menor ancho de banda de procesador a memoria.
- El patrón de acceso a memoria de las instrucciones vectoriales es conocido.
- La sustitución de un bucle por una instrucción vectorial elimina el azar en el control de las bifurcaciones.

Por estas razones, las operaciones vectoriales pueden realizarse más rápido que la secuencia de instrucciones escalares necesaria para efectuar la misma tarea, siendo una alternativa atractiva en aplicaciones en las que su uso pueda ser frecuente.

Los procesadores vectoriales segmentan las operaciones sobre los elementos individuales del vector. Esta segmentación incluye el acceso a memoria y el cálculo de direcciones efectivas. Además, la mayoría de procesadores vectoriales admiten la ejecución simultánea de varias operaciones vectoriales.

Debemos tener en cuenta que existen dos factores que afectan al rendimiento de un programa que se ejecuta en modo vectorial. Por una parte la estructura del propio programa y por otro la capacidad del compilador. Respecto a la estructura del programa es

importante la existencia de bucles con dependencia de datos verdadera, los cuales pueden minimizarse con una buena elección de los algoritmos, así como la codificación de los mismos. Por otra parte, también es importante la capacidad del compilador, ya que dentro de ellos existe mucha diferencia en la habilidad que tienen para vectorizar un bucle vectorizable. De modo que si se quiere un rendimiento óptimo en la vectorización es aconsejable no dejar todo el trabajo al compilador.

Capítulo 2

Computadores paralelos

2.1. Introducción a los computadores paralelos

En el capítulo 1 se han visto distintas formas de aumentar el rendimiento de un sistema basado en un único procesador. Otra opción para aumentar el rendimiento del sistema es la utilización simultánea de varios procesadores. A estos sistemas se les denomina *computadores paralelos*. Normalmente cada procesador ejecuta diferentes programas simultáneamente, pero dependiendo del sistema pueden ejecutar las mismas instrucciones sobre datos diferentes. La finalidad de estos sistemas es aumentar la velocidad de operación y es obvio que con la utilización de varios procesadores, este fin es fácilmente alcanzable.

En el diseño de computadores paralelos nos encontramos con dos tendencias, por un lado los *multiprocesadores de memoria compartida* y por otro los *multicomputadores*, también llamados *multiprocesadores de memoria distribuida*.

Los multicomputadores constan esencialmente de un conjunto de procesadores, cada uno de los cuales está conectado a su memoria local. Los procesadores se comunican a través de una red de interconexión.

Los multiprocesadores de memoria compartida están constituidos por un conjunto de procesadores que comparten un espacio único de memoria, el acceso de todos los proce-

sadores a este espacio único de memoria se realiza a través de una red de interconexión.

La programación de los multicomputadores con una metodología de paso de mensajes, que se verá en la sección 2.3.4, no es una tarea fácil de realizar. Sin embargo, la programación de los sistemas multiprocesadores de memoria compartida es muy sencilla, puesto que el intercambio de datos entre los diferentes nodos del sistema es totalmente transparente al programador.

A pesar de la sencillez de programación que comportan los sistemas multiprocesador de memoria compartida, éstos tienen una limitación hardware a la hora de añadir nodos al sistema, limitación que no aparece en los sistemas multicomputadores. Un sistema multiprocesador de memoria compartida no puede incrementar el número de nodos de forma indefinida, debido a que el tiempo de acceso a la memoria incluye la latencia de acceso a la red de interconexión, y este coste se incrementa a medida que el sistema crece.

En este capítulo nos centraremos, por un lado, en las características particulares de los multiprocesadores de memoria compartida, en la sección 2.2, y en las propiedades de los multicomputadores o multiprocesadores de memoria distribuida junto con las herramientas de comunicación entre procesadores, en la sección 2.3. Además, en la sección 2.4, estudiaremos cómo se analiza la complejidad de los algoritmos paralelos, y por último, en la sección 2.5, cómo se mide el paralelismo alcanzado por un determinado algoritmo.

2.2. Multiprocesadores de memoria compartida

Los multiprocesadores de memoria compartida disponen de un espacio de memoria único para todos los procesadores, simplificando la tarea del intercambio de datos entre los procesadores. El acceso a memoria compartida ha sido implementado, tradicionalmente, usando una red de interconexión entre los procesadores y la memoria.

Veremos a continuación algunas de las redes de interconexión utilizadas en estas arquitecturas y después una clasificación de los multiprocesadores de memoria compartida en función de su arquitectura.

2.2.1. Redes de interconexión en multiprocesadores de memoria compartida

Veamos algunas de las redes más comunes de interconexión que se utilizan en los multiprocesadores de memoria compartida.

- *Red de barras cruzadas:* Las redes de barras cruzadas, o crossbar switch, permiten que cualquier procesador del sistema se conecte con cualquier otro procesador o unidad de memoria, de tal manera, que muchos procesadores pueden comunicarse simultáneamente sin contención. Es posible establecer una nueva conexión en cualquier momento siempre que los puertos de entrada y salida solicitados estén libres. Las redes de crossbar se usan en el diseño de multiprocesadores de pequeña escala pero de alto rendimiento, en el diseño de routers para redes directas, y como componentes básicos en el diseño de redes indirectas de gran escala. Un crossbar se puede definir como una red conmutada con N entradas y M salidas, que permite hasta $min(N, M)$ interconexiones punto a punto sin contención.

- *Redes multietapa:* Las redes de interconexión multietapa conectan dispositivos de entrada a dispositivos de salida a través de un conjunto de etapas de conmutadores, donde cada conmutador es una red de barras cruzadas. El número de etapas y los patrones de conexión entre etapas determinan la capacidad de encaminamiento de estas redes.

  En estos casos, un controlador central establece el camino entre la entrada y la salida. En casos en donde el número de entradas es igual al número de salidas, cada entrada puede transmitir de manera síncrona un mensaje a una salida, y cada salida recibir un mensaje de una entrada.

  Las redes multietapa, en función de la disponibilidad de caminos para establecer nuevas conexiones, se han dividido tradicionalmente en tres clases: bloqueantes, no bloqueantes y reconfigurables. En las redes bloqueantes la conexión entre un par entrada–salida libre no siempre es posible debido a conflictos entre las conexiones existentes. Típicamente, existe un único camino entre cada par entrada–salida, lo que minimiza el numero de conmutadores y los estados. En las redes no bloqueantes cualquier puerto de entrada puede conectarse a cualquier puerto de salida libre sin afectar a las conexiones ya existentes, estas redes requieren que existan múltiples caminos entre cada entrada y cada salida, lo que lleva a etapas adicionales. En las

redes reconfigurables cada puerto de entrada puede ser conectado a cada puerto de salida, sin embargo, las conexiones existentes pueden requerir de un reajuste en sus caminos. Por tanto, para realizar una nueva conexión, estas redes necesitan de la existencia de múltiples caminos entre cada entrada y cada salida, pero el número de caminos y el coste asociado es menor que en el caso de redes no bloqueantes.

2.2.2. Arquitecturas de multiprocesadores de memoria compartida

Como hemos comentado, en la arquitectura de los multiprocesadores de memoria compartida, toda la memoria está físicamente junta. Los procesadores acceden a cualquier posición de la memoria a través de la red de interconexión, proporcionando tiempos de acceso similares, independientemente de la posición accedida. A esta arquitectura se le conoce como **UMA** (*Uniform Memory Access*).

Los multiprocesadores de memoria compartida han seguido tendencias para distribuir la memoria físicamente entre los procesadores, reduciendo el tiempo de acceso a la memoria e incrementando la escalabilidad. A estos computadores paralelos se les denomina multiprocesadores de memoria compartida físicamente distribuida (DSM: Distributed Shared Memory). Son sistemas en los cuales la memoria está físicamente distribuida, pero se puede acceder a ella, usando técnicas de software o de hardware, como si fuera un modelo de memoria compartida, es decir, usando un único espacio de direccionamiento global. Los sistemas DSM combinan las ventajas de ambas aproximaciones, porque siendo máquinas de memoria distribuida, pueden ser programadas virtualmente como máquinas de memoria compartida de forma transparente para el usuario. En estos sistemas cada procesador tiene al menos un nivel de su jerarquía de memoria caché que es privada y que da lugar a un problema denominado coherencia de caché. Este problema surge cuando un bloque de memoria está presente en la memoria caché de uno o más procesadores y otro procesador modifica ese bloque de memoria. A menos que se realicen acciones especiales, estos procesadores continuarían accediendo a la copia del bloque existente en sus memorias caché, cuyo contenido no es correcto. Puede verse el funcionamiento detallado de la memoria caché así como sus jerarquías en [2] o en [65]. Las máquinas de este tipo pueden clasificarse en dos tipos:

- NUMA (Non–Uniform Memory Access): Son máquinas con un espacio de direcciones físicas estático. Esta clase se puede subdividir en varios subgrupos dependiendo del mecanismo de coherencia de caché utilizado:
  - CC–NUMA (Cache Coherent NUMA): NUMA con el sistema de coherencia de caché desarrollado en hardware, por tanto transparente al programador.
  - NC–NUMA (Non Cache coherent NUMA): NUMA sin cache coherente por hardware. Este tipo de sistemas se pueden programar utilizando un modelo de paso de mensajes o utilizando un modelo de programación de memoria compartida.
- COMA (Cache Only Memory Access): En este caso se tiene un espacio de direcciones dinámico en el cual las memorias locales son tratadas como memorias caché.

2.3. Introducción a los multicomputadores

Los multiprocesadores de memoria compartida, descritos en la sección anterior, presentan algunas desventajas, como es la necesidad de técnicas de sincronización para controlar el acceso a las variables compartidas. Además, la contención en la memoria puede reducir significativamente la velocidad del sistema y no son fácilmente expansibles para acomodar un gran número de procesadores.

Un sistema multiprocesador que soluciona los problemas indicados es aquel que únicamente dispone de memoria local y elimina toda la memoria compartida del sistema. El código para cada procesador se carga en la memoria local al igual que cualquier dato que sea necesario. Los programas todavía están divididos en diferentes partes, como en el sistema de memoria compartida, y dichas partes todavía se ejecutan concurrentemente por procesadores individuales. Cuando los procesadores necesitan acceder a la información de otro procesador, o enviar información a otro procesador, se comunican enviando mensajes. Los datos no están almacenados globalmente en el sistema, de forma que si más de un proceso requiere un dato, éste debe duplicarse y ser enviado a todos los procesadores que lo han solicitado. A estos sistemas se les denomina multicomputadores o multiprocesadores de memoria distribuida.

Idealmente, los procesos y los procesadores que ejecutan dichos procesos pueden ser vistos como entidades completamente separadas. Un problema se describirá como un conjunto de procesos que se comunican entre sí y que se hace encajar sobre una arquitectura multicomputador. El conocimiento de la estructura física y de la composición de los nodos es necesario únicamente a la hora de planificar una ejecución eficiente.

El tamaño de un proceso viene determinado por el programador y puede describirse por su granularidad, la cual se expresa como:

$$Granularidad = \frac{Tiempo\ de\ cálculo}{Tiempo\ de\ comunicación}.$$

En términos cualitativos se habla de *granularidad gruesa* si cada proceso contiene un gran número de instrucciones secuenciales que tardan un tiempo sustancial en ejecutarse. En la *granularidad fina* un proceso puede consistir en unas pocas instrucciones, en el caso extremo será de una sola instrucción. La *granularidad media* será el término medio entre ambas.

Al reducirse la granularidad, la sobrecarga de comunicación de los procesos aumenta. Normalmente se pretende minimizar la sobrecarga de comunicación, ya que supone un coste alto. Cada nodo del sistema suele tener una copia del núcleo de un sistema operativo. Este sistema operativo se encarga de la planificación de procesos y de realizar las operaciones de paso de mensajes en tiempo de ejecución. Las operaciones de encaminamiento de los mensajes suelen estar soportadas por hardware, lo que reduce la latencia de las comunicaciones. Todo el sistema suele estar controlado por un ordenador anfitrión. No todo son ventajas en los sistemas multicomputador. El código y los datos deben transferirse físicamente a la memoria local de cada nodo antes de su ejecución, y esta acción puede suponer una sobrecarga considerable. De forma similar, los resultados deben de transferirse de los nodos al sistema anfitrión. Claramente, los cálculos a realizar deben ser lo suficientemente costosos para compensar este inconveniente. Además, el programa a ejecutar debe ser intensivo en cálculo, no intensivo en operaciones de entrada–salida o de paso de mensajes. El código no puede compartirse, por tanto, si los procesos van a ejecutar el mismo código, lo que sucede frecuentemente, el código debe duplicarse en cada nodo. Los datos deben pasarse también a todos los nodos que los necesiten, lo que puede dar lugar a problemas de incoherencia. Estas arquitecturas son generalmente menos flexibles que las arquitecturas de memoria compartida. Por ejemplo, los multiprocesadores de memoria compartida pueden emular el paso de mensajes utilizando zonas compartidas

para almacenar los mensajes, mientras que los multicomputadores son muy ineficientes a la hora de emular operaciones de memoria compartida.

El uso de redes de interconexión propias y específicas provoca que tanto los multiprocesadores de memoria compartida, como los multicomputadores, sean bastante caros. Actualmente, se propone un nuevo concepto de computador paralelo constituido por las redes de estaciones de trabajo, o incluso por redes de ordenadores personales dando lugar a los denominados *clusters*. La idea es realmente simple. Las actuales estaciones de trabajo y PC's están dotados de los mismos procesadores o muy similares a los de los grandes sistemas. Por tanto, la potencia de cálculo está disponible encima de nuestra mesa de trabajo. El gran esfuerzo de investigación y desarrollo que se realizó tiempo atrás para conseguir redes de comunicación de alta velocidad, impulsado en gran medida por la necesidad de estas redes de comunicación en los grandes computadores paralelos, está dando sus frutos hoy en día. La amplia utilización de las redes de comunicación ha provocado que productos que antes se utilizaban muy poco y por tanto resultaban muy costosos, tengan hoy en día una utilización masiva, reduciéndose de este modo enormemente los costes de producción y también, por supuesto, el coste para el usuario final. Podemos encontrar clusters con redes de conexión completamente estándar y ya de baja latencia, como puede ser Fast Ethernet o fibra óptica, por ejemplo, o redes un poco más costosas y de un uso, de momento, mucho más restringido, como puede ser Gigabit–Ethernet o Myrinet.

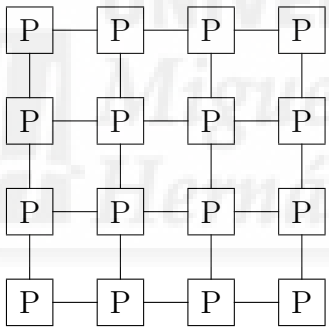
2.3.1. Redes de interconexión en multicomputadores

Las redes de interconexión usualmente utilizadas en multicomputadores pertenecen a la clase de redes directas. La mayoría de las redes implementadas tienen una topología ortogonal. Una topología se dice ortogonal si y sólo si los nodos pueden colocarse en un espacio ortogonal $n-$dimensional, y cada enlace puede ponerse de tal manera que produce un desplazamiento en una única dimensión. Las topologías ortogonales pueden a su vez clasificarse en estrictamente ortogonales y débilmente ortogonales. En una topología estrictamente ortogonal, cada nodo tiene al menos un enlace cruzando cada dimensión. En una topología débilmente ortogonal, algunos de los nodos pueden no tener enlaces en alguna dimensión. Una de las características más importantes de las topologías estrictamente

ortogonales es que el encaminamiento es muy simple. Si la topología no es estrictamente ortogonal, el encaminamiento se complica.

De las redes con topología estrictamente ortogonal podemos destacar las mallas y los hipercubos:

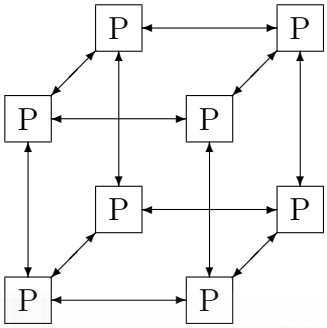
- Mallas: En esta clase caben muy distintos tipos de configuración, que pueden asimilarse a una malla, bidimensional o tridimensional, en donde los procesadores corresponderían a los cruces de la malla y las líneas de interconexión a las fibras de la misma. La idea fundamental es que cada procesador está conectado a unos pocos procesadores que son sus *vecinos*. Con relativamente pocas conexiones se consiguen diámetros reducidos. Se entiende por diámetro la máxima de las distancias entre los nodos de una red de interconexión. La optimización de este parámetro corresponde a la minimización del retraso máximo de los mensajes en la red.



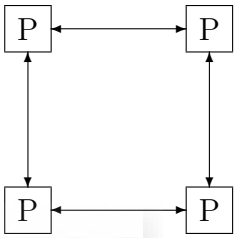
La figura anterior muestra una malla bidimensional, una de las mallas más usadas. A partir de ésta podemos obtener una variante que consiste en unir los procesadores de la última columna con los correspondientes de la primera, y proceder de forma análoga con la primera y última filas, de forma que se obtenga una estructura toroidal, reduciendo así el diámetro. También pueden incluirse conexiones diagonales. Superponiendo varias y conectando los procesadores correspondientes obtendríamos una malla tridimensional.

- Hipercubo: Este sistema de interconexión puede usarse para conectar $p = 2^n$ procesadores. Cada procesador correspondería a un vértice de un hipercubo n–dimensional y estaría conectado a otro si en el hipercubo hubiera una arista entre los

correspondientes vértices. Con un número no demasiado elevado de conexiones se consigue un diámetro reducido, aunque en cualquier caso la dimensión n del hipercubo no puede ser muy grande. La ventaja fundamental del hipercubo es que permite conectar un gran número de procesadores, 2^n, con un diámetro pequeño n y con pocas conexiones. Otra ventaja del hipercubo es que contiene otras estructuras, en particular al anillo.



Hipercubo 3-dimensional



Hipercubo 2-dimensional

2.3.2. Mecanismos de conmutación

Existen diferentes técnicas que se implementan dentro de los sistemas encargados de realizar el mecanismo por el cual los mensajes pasan a través de la red, a estos sistemas se les denomina *routers*. Estas técnicas difieren en varios aspectos. Las técnicas de conmutación determinan cuándo y cómo se conectan los conmutadores internos del router para conectar las entradas y salidas del mismo, así como cuándo los componentes del mensaje pueden transferirse a través de esos caminos creados. Estas técnicas están ligadas con los mecanismos de control de flujo, los cuales sincronizan la transferencia de unidades de información entre routers y a través de los mismos durante el proceso de envío de los mensajes a través de la red. El control de flujo está, a su vez, fuertemente acoplado con los algoritmos de manejo de buffers que deciden cómo se asignan y liberan los buffers, es decir, cómo se manejan los mensajes cuando se bloquean en la red.

Para poder comparar y contrastar diferentes alternativas de conmutación, definiremos previamente los conceptos de latencia y ancho de banda ampliamente utilizados en la terminología de redes. Se define latencia como el tiempo transcurrido entre el envío de un paquete en el origen y su recepción en el destino. Por otro lado, se define el ancho de banda como la cantidad de información que se puede enviar a través de una conexión en un tiempo dado. El ancho de banda generalmente se expresa en bits por segundo (bps) o megabits por segundo (Mbps).

Una vez definidos ambos conceptos, nos centramos en cómo influyen en el funcionamiento del router y, por tanto, en la latencia y el ancho de banda resultante. La arquitectura de un router genérico está compuesto de los siguientes componentes principales:

- *Buffers:* Son memorias de tipo FIFO (First Input First Output) que permiten almacenar mensajes en tránsito. Un buffer está asociado a un canal físico de entrada y de salida. En diseños alternativos, los buffers pueden asociarse únicamente a las entradas o a las salidas. El tamaño del buffer es un número entero de unidades de control de flujo (flits).

- *Conmutador:* Este componente es el responsable de conectar los buffers de entrada del router con los buffers de salida. Los routers de alta velocidad utilizan redes crossbar, que vimos en la sección 2.2.1, con conectividad total, mientras que implementaciones de más baja velocidad utilizan redes que no proporcionan una conectividad total entre los buffers de entrada y los de salida.

- *Unidad de encaminamiento y arbitraje:* Este componente implementa los algoritmos de encaminamiento, selecciona el enlace de salida para un mensaje entrante, programando el conmutador en función de la elección. Si varios mensajes piden de forma simultánea el mismo enlace de salida, este componente debe proporcionar un arbitraje entre ellos. Si el enlace pedido está ocupado, el mensaje debe permanecer en el buffer de entrada hasta que éste quede libre.

- *Controladores de enlaces:* El flujo de mensajes a través de los canales físicos entre routers adyacentes se implementa mediante el controlador de enlaces. Los controladores de enlaces de cada lado del canal deben coordinarse para transferir flits.

- *Interfaz del procesador:* Este componente simplemente implementa un canal físico con el procesador en lugar de con un router adyacente. Consiste en uno o más canales de inyección desde el procesador y uno o más canales de deyección hacia el

procesador. A los canales de deyección también se les denominan canales de reparto o canales de consumición.

Para analizar el rendimiento del router nos centraremos en el *retraso de encaminamiento* y en el *control de flujo*. Cuando un mensaje llega a un router, éste debe de ser examinado para determinar el canal de salida por el cual debe enviarse el mensaje. A esto se le denomina *retraso de encaminamiento*, y suele incluir el tiempo para configurar el conmutador. Una vez que se ha establecido un camino a través del router, estamos interesados en el ratio al cual podemos enviar mensajes a través del conmutador. Este ratio viene determinado por el retraso de propagación a través del conmutador o retraso intra–router, y el ratio que permite la sincronización de la transferencia de datos entre los buffers de entrada y de salida, a este retraso se le denomina *latencia de control de flujo interno*. De manera similar, al retraso a través de los enlaces físicos o retraso inter–router, se denomina *latencia del control de flujo externo*. El retraso debido al enrutamiento y los retrasos de control de flujo determinan la latencia disponible a través del conmutador y, junto con la contención de los mensajes en los enlaces, determina el rendimiento o productividad de la red.

Vamos a adentrarnos en algunos conceptos importantes para los mecanismos de conmutación:

- *Control de flujo:* El control de flujo es un protocolo asíncrono para transmitir y recibir una unidad de información. La unidad de control de flujo (flit) se refiere a aquella porción del mensaje cuya transmisión debe sincronizarse. Esta unidad se define como la menor unidad de información cuya transferencia es solicitada por el emisor y notificada por el receptor. Esta señalización petición–reconocimiento se usa para asegurar una transferencia exitosa y la disponibilidad de espacio de buffer en el receptor.

  El control de flujo ocurre a dos niveles, paquete y canal. El control de flujo del mensaje ocurre a nivel de paquete. Sin embargo, la transferencia del paquete por el canal físico que une dos routers se realiza en varios pasos. La transferencia resultante multiciclo usa un control de flujo del canal para enviar un flit a través de la conexión física.

- *Técnicas de conmutación:* Existen diferentes técnicas de conmutación, entre las más comunes está la *conmutación de circuitos* que reserva un camino físico desde el

origen hasta el destino antes de producirse la transmisión de los datos. El mejor comportamiento de esta técnica de conmutación ocurre cuando los mensajes son infrecuentes y largos. Otra técnica es la *conmutación de paquetes* donde el mensaje se divide y transmite en paquetes de longitud fija, esta técnica es ventajosa cuando los mensajes son cortos y frecuentes. En la *conmutación Virtual Cut-Through (VCT)* en lugar de esperar a recibir el paquete en su totalidad la cabecera del paquete puede ser examinada tan pronto como llega y el router puede comenzar a enviar la cabecera y los datos que le siguen tan pronto como se realice una decisión de encaminamiento y el buffer de salida esté libre. Por último, en la *conmutación segmentada* se dividen los paquetes en flits, siendo el flit la unidad de control del mensaje y por tanto se utilizan buffers más pequeños.

- *Canales virtuales:* Las técnicas de comunicación anteriores fueron descritas suponiendo que los mensajes o parte de los mensajes se almacenan a la entrada y salida de cada canal físico. Por lo tanto, una vez que un mensaje ocupa el buffer asociado a un canal, ningún otro mensaje puede acceder al canal físico. Sin embargo, un canal físico puede soportar varios canales virtuales o lógicos multiplexados sobre el mismo canal físico. El protocolo del canal físico debe ser capaz de trabajar con todos los canales virtuales presentes en el canal físico. Los canales virtuales se utilizan para mejorar la latencia de los mensajes y el rendimiento de la red.

2.3.3. Algoritmos de encaminamiento

Los algoritmos de encaminamiento establecen el camino que sigue cada mensaje o paquete. La lista de algoritmos propuestos en la literatura es casi interminable. Muchas de las propiedades de la red de interconexión son una consecuencia directa del algoritmo de encaminamiento usado. Entre estas propiedades podemos citar las siguientes:

- *Conectividad:* Habilidad de encaminar paquetes desde cualquier nodo origen a cualquier nodo destino.

- *Adaptabilidad:* Habilidad de encaminar los paquetes a través de caminos alternativos en presencia de contención o componentes defectuosos.

- *Libre de bloqueos:* Habilidad de garantizar que los paquetes no se bloquearán o se quedarán esperando en la red indefinidamente.

- *Tolerancia a fallos:* Habilidad de encaminar paquetes en presencia de componentes defectuosos. Aunque podría parecer que la tolerancia a fallos implica la adaptabilidad, esto no es necesariamente cierto. La tolerancia a fallos puede conseguirse sin adaptabilidad mediante el encaminamiento de un paquete en dos o más fases, almacenándolo en algún nodo intermedio.

Los algoritmos de encaminamiento pueden clasificarse según el lugar donde se toman las decisiones de encaminamiento. Básicamente, el camino puede establecerse bien por un controlador centralizado (encaminamiento centralizado), por el nodo origen antes de la inyección del paquete (encaminamiento de origen), o bien ser determinado de una manera distribuida mientras el paquete atraviesa la red (encaminamiento distribuido).

También son posibles esquemas híbridos, a estos esquemas híbridos se les denomina encaminamiento multifase. En el encaminamiento multifase, el nodo origen determina algunos de los nodos destinos. El camino entre estos se establece de una manera distribuida y el paquete puede ser enviado a todos los nodos destinos calculados (multicast routing) o únicamente al último de los nodos destino (unicast routing). En este caso, los nodos intermedios se usan para evitar la congestión o los fallos.

Los algoritmos de encaminamiento pueden implementarse de varias maneras, bien mediante el uso de una tabla de encaminamiento, o bien mediante la ejecución de un algoritmo de encaminamiento de acuerdo con una máquina de estados finita. En este último caso, el algoritmo puede ser determinista o adaptativo. Los algoritmos de encaminamiento deterministas siempre suministran el mismo camino entre un nodo origen y un nodo destino. Los algoritmos adaptativos usan información sobre el tráfico de la red y/o el estado de los canales para evitar la congestión o las regiones con fallos de la red.

Los algoritmos de encaminamiento pueden clasificarse, también, de acuerdo con su progresión como progresivos o con vuelta atrás. En los algoritmos de encaminamiento progresivos la cabeza siempre sigue hacia delante reservando un nuevo canal en cada operación de encaminamiento. En los algoritmos con vuelta atrás la cabecera puede retroceder hacia atrás, liberando canales reservados previamente.

A un nivel más bajo, los algoritmos de encaminamiento pueden clasificarse dependiendo de su minimalidad como aprovechables o con desencaminamientos. Los algoritmos de

encaminamiento aprovechables sólo proporcionan canales que acercan al paquete a su destino. También se les denominan mínimos. Los algoritmos con desencadenamientos pueden proporcionar además algunos canales que alejen al paquete de su destino. A estos últimos también se les denominan no mínimos.

En las redes directas, los paquetes deben atravesar varios nodos intermedios antes de alcanzar su destino. En las redes basadas en conmutadores, los paquetes atraviesan varios conmutadores antes de alcanzar su destino. Sin embargo, puede ocurrir que algunos paquetes no puedan alcanzar su destino, incluso existiendo un camino libre de fallos entre los nodos origen y destino para cada paquete. Suponiendo que existe un algoritmo de encaminamiento capaz de utilizar esos caminos, existen varias situaciones que pueden impedir la recepción del paquete, produciéndose bloqueos.

Los algoritmos utilizados deben garantizar un encaminamiento libre de bloqueos, podemos clasificarlos en:

- *Algoritmos deterministas:* Establecen el camino como una función de la dirección destino, proporcionando siempre el mismo camino entre cada par de nodos. Algunas topologías pueden descomponerse en varias dimensiones ortogonales, como es el caso de los hipercubos o mallas. En estas topologías, es fácil calcular la distancia entre el nodo actual y el nodo destino como suma de las diferencias de posiciones en todas las dimensiones. Los algoritmos de encaminamiento progresivos reducirán una de estas diferencias en cada operación de encaminamiento. El algoritmo de encaminamiento progresivo más simple consiste en reducir una de estas diferencias a cero antes de considerar la siguiente dimensión. A este algoritmo de encaminamiento se le denomina encaminamiento por dimensiones. Este algoritmo envía los paquetes cruzando las dimensiones en un orden estrictamente ascendente (o descendente), reduciendo a cero la diferencia en una dimensión antes de encaminar el paquete por la siguiente. Para redes n-dimensionales e hipercubos, el encaminamiento por dimensiones da lugar a algoritmos de encaminamiento libres de bloqueos mortales. Estos algoritmos son muy conocidos y han recibido varios nombres, como encaminamiento XY (para mallas 2D) o e-cubo (para hipercubos).

- *Algoritmos adaptativos:* Incrementan el número de caminos alternativos entre dos nodos dados. Dependiendo de si se puede utilizar cualquier camino mínimo entre un nodo origen y un nodo destino, o únicamente algunos de ellos, los algoritmos pueden clasificarse en totalmente adaptativos o parcialmente adaptativos, respecti-

vamente. Los algoritmos parcialmente adaptativos representan un compromiso entre la flexibilidad y el coste. Estos algoritmos intentan aproximarse a la flexibilidad del encaminamiento totalmente adaptativo a expensas de un moderado incremento en la complejidad con respecto al encaminamiento determinista. La mayoría de los algoritmos parcialmente adaptativos propuestos se basan en la ausencia de dependencias cíclicas entre canales para evitar los bloqueos. Algunas propuestas intentan maximizar la adaptabilidad sin incrementar los recursos necesarios para evitar los bloqueos. Otras propuestas intentan minimizar los recursos necesarios para obtener un cierto nivel de adaptabilidad. Los algoritmos totalmente adaptativos permiten la utilización de cualquier camino mínimo entre el nodo origen y el nodo destino, maximizando el rendimiento de la red.

2.3.4. Modelos de paso de mensajes PVM y MPI

Los modelos de paso de mensajes están basados en primitivas de comunicación de los procesadores mediante pares de envío–recepción. Actualmente los más utilizados son PVM (*Parallel Virtual Machine*) [56] y MPI (*Message Passing Interface*) [48].

PVM es un sistema de software que permite que un conjunto de computadoras heterogéneas sea utilizado como un único recurso computacional concurrente coherente y flexible. Las computadoras pueden ser multiprocesadores de memoria compartida o distribuida, supercomputadoras vectoriales, máquinas especializadas en gráficos, estaciones de trabajo escalares (workstations) e incluso ordenadores personales, que pueden estar interconectadas por diversos tipos de redes.

El sistema PVM está compuesto de dos partes. La primera parte es un demonio o *daemon*, llamado pvmd3, a veces abreviado pvmd, que reside en todas las computadoras que constituyen la máquina virtual. Pvmd3 está diseñado para que cualquier usuario con un login válido pueda instalar este demonio en una máquina. Cuando un usuario desea ejecutar una aplicación PVM, crea primero una máquina virtual poniendo en marcha PVM. La aplicación PVM puede empezarse entonces desde un prompt Unix en cualquiera de los *hosts*. Varios usuarios pueden configurar máquinas virtuales simultáneamente, y cada usuario puede ejecutar varias aplicaciones PVM simultáneamente.

La segunda parte del sistema es la librería de rutinas PVM. Contiene un repertorio

funcionalmente completo de primitivas necesarias para la cooperación entre las tareas de una aplicación. Esta biblioteca contiene rutinas para el paso de mensajes, para expandir procesos, coordinar tareas, y modificar la máquina virtual. Los programas, que pueden estar escritos en C, C++ o FORTRAN, acceden a PVM a través de esta librería de rutinas. Una descripción más detallada del sistema PVM puede encontrarse, por ejemplo, en [56] y [73].

MPI fue diseñada por un grupo de investigadores procedentes del mundo de la industria, académico y del gobierno, con experiencia en librerías de paso de mensajes en una amplia variedad de plataformas para servir como un estándar común -para escribir programas paralelos portables con paso de mensajes- que englobase toda la experiencia anterior en este tipo de programación.

Para su diseño, iniciado en 1992, se tuvo en cuenta los rasgos de un buen número de sistemas de paso de mensajes existentes. Actualmente está reemplazando paulatinamente a las librerías de paso de mensajes PVM. MPI permite la compilación de librerías separadas y está diseñado para poderse ejecutar en la mayoría de plataformas paralelas. Aunque MPI no tiene el concepto de máquina virtual, sí hace una abstracción de todos los recursos en términos de topología de paso de mensajes. En MPI un grupo de procesos se pueden colocar en una topología lógica de interconexión específica, a través de la cual tienen lugar las comunicaciones. En PVM no disponemos de dicha abstracción, sino que el programador es el encargado de asignar las tareas y colocarlas en grupos estableciendo explícitamente la organización de la comunicación. Una descripción más detallada del sistema MPI puede verse, por ejemplo, en [26] y [61].

2.4. Análisis de la complejidad de los algoritmos paralelos

Hasta el momento no existe un modelo único que sustente el desarrollo de la computación paralela del mismo modo que el modelo de von Neumann lo ha hecho en el campo de la computación secuencial. A lo largo de los años diversos modelos de computación paralela se han ido proponiendo como fundamento teórico de una computación paralela

de propósito general. En esta sección se dará una breve descripción de las características básicas de algunos de los modelos que han sido tradicionalmente utilizados para el análisis teórico y el diseño de algoritmos desde finales de los años 70 hasta nuestros días:

- Modelo PRAM (Parallel Random Access Machine) [58]: Históricamente ha sido el más popular. Una PRAM es una idealización de un computador paralelo. Consiste en un conjunto ilimitado de procesadores que realizan cálculos sincronizadamente, cada uno de los cuales posee su propia memoria local (un pequeño conjunto de registros), y una memoria ilimitada común a todos los procesadores.

  En una unidad de tiempo cada procesador puede hacer cualquier subconjunto de las siguientes operaciones: leer dos valores de la memoria global, realizar una de las operaciones básicas sobre esos dos valores leídos, y escribir un valor de vuelta en la memoria global. No hay comunicación explícita entre los distintos procesadores, sino que éstos comunican solamente a través de la memoria global, es decir, mediante la lectura de posiciones de memoria donde previamente ha escrito otro procesador. La complejidad de un algoritmo PRAM viene dada en términos del número de pasos (unidades de tiempo) de que consta y del número de procesadores necesario para cada uno de estos pasos. La sencillez del modelo PRAM ha hecho de él un modelo extremadamente atractivo para el análisis y diseño de algoritmos paralelos. Sin embargo, es un modelo poco realista debido fundamentalmente a la suposición de que todos los procesadores operan de forma síncrona y que la comunicación entre procesadores (a través de la memoria global) se lleva a cabo sin ningún coste. Por este motivo el modelo PRAM difícilmente puede ser implementado en arquitecturas con memoria distribuida.

- Modelo LogP [21]: Es un reflejo de las tendencias tecnológicas actuales. Es un modelo de computador paralelo de memoria distribuida en el que los procesadores comunican mediante paso de mensajes. Se basa en cuatro parámetros que modelan el ancho de banda de comunicación, el tiempo de comunicación y la eficiencia de simultanear comunicación y cálculo. Estos cuatro parámetros son los siguientes:

  - L: es una cota superior de la latencia, o retardo, en que se incurre al comunicar un mensaje (con un número reducido de palabras) desde un procesador origen a un procesador destino.
  - o: es el *overhead*, definido como el tiempo que los procesadores están ocupados en la transmisión o recepción de cada mensaje. Durante este tiempo, los

procesadores no pueden realizar ninguna otra operación.

- g: es el *gap*, definido como el mínimo intervalo de tiempo entre dos envíos o recepciones consecutivas en un mismo procesador. El recíproco de g corresponde al ancho de banda disponible por procesador.
- P: es el número de pares procesador–memoria. Se asume que las operaciones locales tardan una unidad de tiempo en ejecutarse, a esa unidad se le denomina ciclo.

El modelo LogP pretende tanto servir como base para el desarrollo de algoritmos portables y rápidos como ofrecer ideas a los diseñadores de máquinas paralelas. La elección de los parámetros se ha hecho en un intento de recoger de forma precisa las características del funcionamiento de las máquinas reales.

- Modelo BSP: Fue propuesto por Leslie G. Valiant [110] en 1990, en un intento de lograr un modelo puente entre teoría y práctica, con un punto de partida mucho más realista que la PRAM. Un computador BSP consiste en un conjunto de pares procesador–memoria, una red de comunicación que reparte mensajes punto a punto y un mecanismo para la sincronización eficiente de todos los procesadores o un conjunto de ellos. No existen facilidades adicionales que permitan combinar, replicar o difundir mensajes. Si se define la unidad de tiempo del modelo como el tiempo requerido para realizar una operación local, es decir, una de las operaciones aritméticas básicas sobre datos contenidos en la memoria local del procesador, el rendimiento de un computador BSP se caracteriza por los siguientes cuatro parámetros:

  - p: es el número de procesadores.
  - s: es la velocidad del procesador, es decir, el número de operaciones elementales que el procesador es capaz de realizar en un segundo.
  - ℓ: es el número mínimo de unidades de tiempo necesarias para llevar a cabo la sincronización de los procesadores.
  - g: se define como el cociente entre el número total de operaciones realizadas por todos los procesadores en un segundo y el número total de palabras repartidas por la red de interconexión en un segundo.

  El parámetro ℓ está relacionado con la latencia de la red, es decir, con el tiempo necesario para hacer un acceso a una memoria no local en una situación de continuo

tráfico de red. El parámetro g corresponde a la frecuencia con que se pueden hacer accesos a memoria remota. En aquellas máquinas con mayores valores del parámetro g se podrán llevar a cabo accesos a memoria local menos frecuentemente.

2.5. Medidas de paralelismo

Para evaluar las prestaciones de los algoritmos paralelos utilizaremos los parámetros que definimos a continuación: el incremento de velocidad o speed–up y la eficiencia.

El primer parámetro que definimos compara el mismo algoritmo al ser ejecutado utilizando uno o p procesadores.

Definición 2.1. Se define el *incremento de velocidad* o *speed–up* de un algoritmo paralelo como

$$S_p = \frac{\text{Tiempo de ejecución del algoritmo en un solo procesador}}{\text{Tiempo de ejecución del algoritmo en } p \text{ procesadores}}. \tag{2.1}$$

Esta definición tiene el inconveniente de que el algoritmo paralelo puede no ser el más eficiente cuando se ejecuta en secuencial. Por este motivo introducimos otra medida de paralelismo.

Definición 2.2. Se define el *incremento de velocidad* o *speed–up* de un algoritmo paralelo respecto al mejor algoritmo secuencial como el cociente

$$S'_p = \frac{\text{Tiempo de ejecución del mejor algoritmo secuencial}}{\text{Tiempo de ejecución del algoritmo en } p \text{ procesadores}}. \tag{2.2}$$

Hacemos notar, que no siempre va a ser posible determinar el mejor algoritmo secuencial en términos absolutos.

Aunque teóricamente se verifica $S'_p \leq S_p \leq p$, en la práctica podemos encontrar situaciones en las que obtengamos incrementos de velocidad superiores al número de procesadores. Así podríamos encontrarnos en esta situación cuando, por ejemplo, utilicemos

como algoritmo secuencial un algoritmo distinto al paralelo o cuando el algoritmo secuencial tenga mucha más necesidad de acceso a la memoria rápida (caché), lo cual provoca, obviamente, un retraso en la ejecución y consecuentemente un aumento del speed–up.

A partir de las medidas de paralelismo anteriores se definen dos parámetros que miden el grado de utilización de los procesadores.

Definición 2.3. Se define la *eficiencia* de un algoritmo paralelo, respecto a sí mismo, al ejecutarse en un sistema con p procesadores como

$$E_p = \frac{S_p}{p}. \tag{2.3}$$

Definición 2.4. Se llama *eficiencia* de un algoritmo paralelo respecto al mejor algoritmo secuencial al cociente

$$E'_p = \frac{S'_p}{p}. \tag{2.4}$$

El objetivo es conseguir la mayor eficiencia posible. Por otra parte, hacemos notar que los tiempos referidos en las definiciones de S_p y S'_p son tiempos reales de ejecución, de esta forma para calcular S_p, S'_p, E_p, y E'_p es necesario realizar el experimento numérico.

Para maximizar la eficiencia de un algoritmo paralelo un factor determinante, especialmente para algoritmos síncronos, es el equilibrio de la carga de trabajo entre los distintos procesadores, es decir, la distribución de las tareas entre los procesadores de forma que todos tengan una cantidad de trabajo similar. Asimismo, este reparto equitativo del trabajo ha de procurar realizarse entre todos los puntos de sincronización con el objetivo de evitar, en la medida de lo posible, que algunos procesadores se mantengan inactivos o *perezosos.* Una alternativa a la distribución equilibrada de la carga de trabajo entre los procesadores son los algoritmos asíncronos.

2.6. Plataformas de computación utilizadas

En el presente trabajo mostraremos diversas pruebas realizadas para comparar y comprobar el correcto funcionamiento de las distribuciones de software presentadas, de cada

módulo de las mismas así como de las aplicaciones que desarrollaremos en el capítulo 10. Las pruebas han sido desarrolladas en tres plataformas distintas que describimos a continuación:

- Cluster-umh: Cluster de 6 Intel Pentium 4 a 2 GHz con 1 Gbyte de memoria RAM conectados mediante un conmutador Gigabit ethernet con el sistema operativo Linux en su distribución Debian. Este recurso está localizado en el edificio Alcudia del campus de Elche de la Universidad Miguel Hernández.

- Jacquard: Es un cluster de 320 procesadores duales Opteron a 2.2 GHz y un rendimiento pico de 4.4 GFlop/s, con el sistema operativo Linux, conectados con una red InfiniBand de alta velocidad.

- Seaborg: La plataforma NERSC IBM SP RS/6000, llamada Seaborg, es una computadora de memoria distribuida con 6080 procesadores disponible para la ejecución de aplicaciones científicas. Cada procesador tiene un pico de 1.5 GFlops de rendimiento. Los procesadores están distribuidos en 380 nodos con 16 procesadores por nodo. Cada procesador tiene una memoria compartida entre 16 y 64 GBytes (4 tienen 64 GBytes; 64 tienen 32 GBytes; 312 tienen 16 GBytes). Los nodos de computación están conectados a otros nodos mediante una red conmutada de alto ancho de banda y baja latencia. Cada nodo ejecuta su propia instancia del sistema operativo AIX. El sistema de almacenamiento de disco es un sistema paralelo de entrada/salida llamado GPFS ubicado en nodos adicionales.

Los recursos Jacquard y Seaborg se encuentran en el National Energy Research Scientific Computing Center (NERSC) de EEUU.

UNIVERSITAS
Miguel
Hernández

Capítulo 3

Python

En este capítulo describiremos las principales características del lenguaje de alto nivel denominado Python, así como el uso de los módulos disponibles que han intervenido en la realización de este trabajo.

En la primera sección realizaremos una introducción del lenguaje Python describiendo los aspectos más relevantes y las principales características que lo han convertido en uno de los lenguajes de alto nivel más ampliamente utilizados.

Python dispone de una gran variedad de módulos incluidos en la distribución oficial y desarrollados por terceros. En la sección 3.2, se presentan aquellos módulos que serán de utilidad en nuestro trabajo y los describiremos detallando aquellos aspectos útiles para alcanzar nuestros objetivos.

Para conseguir estos objetivos, necesitaremos crear nuevos módulos para Python y reutilizar código de librerías de altas prestaciones. En la sección 3.3 describiremos cuales son los métodos para crear nuevos módulos y extensiones y cuáles son las herramientas disponibles para facilitar este trabajo.

3.1. Introducción a Python

El lenguaje Python [96] surge como solución ante un conjunto de problemas en el global de lenguajes existentes. Hasta el momento, la programación en *scripts* resultaba una tarea tediosa y verdaderamente complicada cuando el tamaño del *script* era considerable. Por otro lado, en la programación de librerías en C, el ciclo de edición/compilación/prueba/recompilación resulta demasiado lento en aquellas ocasiones que necesitamos pequeñas modificaciones del programa base.

Python resuelve situaciones como ésta. Su creador, Guido Van Rossum lo define como un lenguaje sencillo de utilizar pero que ofrece un lenguaje más estructurado que la programación de *scripts* con *shell*. Además, ofrece mayor comprobación a errores que lenguajes como C, y proporciona tipos intrínsecos de alto nivel tan flexibles como matrices y diccionarios. Python permite dividir nuestra aplicación en módulos que pueden ser reutilizados en otros programas. En esta línea, la distribución estándar viene acompañada de un conjunto de módulos para utilizar en nuestros programas o *scripts* tales como funciones de entrada/salida a ficheros, llamadas al sistema, *sockets* e incluso interfaces a herramientas gráficas como Tk [83].

Python es un lenguaje interpretado que puede ahorrar un tiempo considerable durante el desarrollo de una aplicación. El interprete puede actuar de forma interactiva, esto facilita la comprobación inmediata de las nuevas órdenes insertadas. Además, los programas escritos en Python son muy compactos y legibles. Normalmente, éstos son mucho más cortos que su equivalente en C por varias razones:

- Los tipos de datos de alto nivel permiten expresar operaciones complejas en un único comando.
- La agrupación de comandos se realiza mediante tabulaciones en lugar de utilizar clausulas de tipo `begin` y `end`.
- No es necesaria la declaración de variables.

Por otro lado, una característica en la que se centra esta memoria, es la extensibilidad de Python. A partir de un programa desarrollado, por ejemplo, en C, es sencillo construir un módulo para ser utilizado en el interprete de Python.

Por último, destacar que la denominación a este lenguaje de alto nivel no tiene nada que ver con ningún reptil sino más bien con la afición y devoción de su creador Guido Van Rossum al show de la BBC "El circo volador de los Monty Python".

3.1.1. Utilización del intérprete

El primer paso que debemos realizar, si todavía no lo hemos hecho, es el de instalar Python desde su distribución oficial [96]. Para iniciar el intérprete desde la línea de comando hemos de escribir:

```
python
```

La localización del ejecutable en linux suele ser `/usr/local/bin` aunque puede ser modificada en el proceso de instalación. Una vez que se ha iniciado el intérprete, aparece en pantalla el siguiente mensaje:

```
Python 2.2.3 (#1, Feb 18 2004, 16:05:37)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

El intérprete muestra los caracteres `>>>` indicando que se encuentra preparado para recibir comandos desde el teclado. El funcionamiento es muy similar a otro tipo de entornos interactivos de alto nivel como Matlab [78]. El ejemplo básico de "Hola Mundo" se ejecutaría del siguiente modo:

```
>>> print "Hello world!"
Hello world!
```

La sintaxis en las expresiones y en las variables es muy parecida a C o Java, de modo que resulta familiar a un usuario ya introducido en la programación. Con respecto a las variables, Python utiliza las variables como una referencia a un valor. Las variables no tienen tipos, pero sí los objetos a los que apuntan. De este modo, la misma variable que se refiere a un valor entero puede corresponderse posteriormente con una cadena. Una característica que influye en la productividad de este lenguaje es el hecho que no se requiera la declaración de una variable. Sin embargo, no podremos acceder a una variable que no ha sido definida previamente:

```
>>> print sindeclarartodavia
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'sindeclarartodavia' is not defined
>>>
```

Como hemos comprobado en el código anterior, en Python se generan diferentes excepciones que pueden ser gestionadas. El tratamiento de las excepciones lo introduciremos también en la presente sección.

Con respecto a las funciones, mediante la primitiva `def FunctionName (parameter, ...):` se inicia la declaración de una función. Se ha de tener en cuenta que la tabulación a partir de la primitiva `def` indica que esas líneas de código pertenecerán a la función. En el siguiente ejemplo, Python conoce el fin de la definición de la función a través de las tabulaciones respecto al margen izquierdo.

```
>>>def Suma(a,b):
    return (a+b)

Suma(3,2)
5
>>>
```

El intérprete de Python se puede ejecutar de modo interactivo tal y como lo hemos presentado hasta el momento, o bien como intérprete de un código o *script* de Python. De este modo, realizamos una distinción entre un *programa*, un *script* (generalmente más corto) y un *módulo* (un fichero diseñado para ser importado y no ser ejecutado directamente). Generalmente los códigos en Python tienen la extensión `.py` y para ejecutar un programa escrito en Python deberíamos escribir en la línea de comandos:

```
python prueba.py
```

Incluso en UNIX podríamos ejecutar el programa de forma directa, añadiendo la línea `#!/usr/bin/python` al principio del fichero y convirtiendo el archivo en ejecutable (mediante `chmod +x prueba.py`).

La agrupación de los comandos en funciones, bucles y condiciones se realiza a partir de la tabulación. En el siguiente código mostramos un ejemplo de una definición de una

función utilizada en el presente trabajo. En este ejemplo podemos observar además un bucle que agrupa una asignación (`common=....`) y una condición (`if PyACTS...`).

```
def getcommontype(array_types):
        common=array_types[0]
        for i in range(1,len(array_types)):
                bothtype=common + array_types[i]
                if PyACTS._PySLK_compatypes.has_key(bothtype):
                        common=PyACTS._PySLK_compatypes[bothtype]
                else:
                        raise PyACTS.PySLKError("TypeErr")
        PySLK_type=PyACTS._PySLK_relationtypes[common]
        return PySLK_type
```

Sin embargo, tendremos la posibilidad de poder abandonar una condición o bucle a mitad del mismo mediante el comando `break`. Este comando abandona el bucle en el que se encuentre (el más interno si hubiera varios). Además de la instrucción `for` también podemos utilizar `while` para generar un bucle condicional.

3.1.2. Listas y diccionarios

El tratamiento de listas, *tuples* y diccionarios se realiza mediante tipos propios del lenguaje, y por tanto proporciona mucha sencillez, versatilidad y velocidad en el manejo de los mismos. En Python, una lista es una colección de cero o más elementos, estos elementos podrán ser de cualquier tipo independientemente del resto. La introducción de una lista se puede realizar como se muestra a continuación.

```
FibonacciLista=[1,1,2,3,5,8]
MezclaLista=[1,2,"Hola"] # Las listas pueden contener varios tipos
OtraLista=[1,2,MezclaLista] # Las listas pueden incluir otras listas
Maslistas=[1,2,3,] # La inclusión de comas es posible
RevengeOfTheList=[] # Lista vacia
```

Observamos en el ejemplo anterior, la aparición de comentarios de ayuda al final de cada linea; estos comentarios son opcionales y permiten al futuro desarrollador o modificador del script entender mejor el funcionamiento del mismo. Para introducir un comentario, éste debe ir precedido por el caracter `#`.

Una *tuple* es similar a una lista con la diferencia que una *tuple* es inmutable, es decir, no puede ser modificada. Las *tuples* aparecen enmarcadas por paréntesis (`(...)`) en lugar de corchetes (`[...]`).

```
primeratuple=("hola","hola","adios","quetal")
segundatuple=()  # Tuple vacia
tuplesolo=(5,)   # La coma en este caso es requerida pues indica que (5,) es
                 # una tuple, y no un numero entre paréntesis (5)
```

Por otro lado, los diccionarios son objetos que permiten establecer referencias cruzadas entre claves y valores. Los valores pueden ser objetos de cualquier tipo. Un diccionario tiene la siguiente representación: las comas separan parejas de valores encerradas en corchetes {`key:value, key:value`}. A continuación mostramos un ejemplo en el uso de diccionarios.

```
>>> Telefonos={"luis":"615235723","juan":"626464729"}
>>> DiccVacio={} # Inicializa un nuevo diccionario
>>> Telefonos["luis"] # Encuentra el número de luis
'615235723'
>>> Telefonos["eva"]="645678234" # Añade un nuevo elemento
>>> print "Lista de teléfonos:",Telefonos
Lista de teléfonos: {'luis': '615235723', 'juan': '626464729',
    'eva': '645678234'}
```

En el caso que busquemos un valor inexistente para una clave determinada, salta una excepción que deberá ser tratada. Para evitar esta excepción debemos utilizar la función `dictionary.get(key,defaultValue)` que proporciona una forma segura de obtener el valor. En el caso de que no exista un valor para dicha clave, se proporcionará el valor por defecto.

```
>>> Telefonos["pep"] # Puede ocurrir una excepción
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in ?
KeyError: pepe
>>> Telefonos.get("pepe","desconocido")
'desconocido'
```

A menudo el valor por defecto utilizado es `None`. Este valor representa *nada*, similar al tratamiento que en C se le da a *Null*.

3.1.3. El concepto de módulo

Tal y como hemos visto en el intérprete de Python hasta el momento, si abandonamos el mismo y entramos de nuevo, todas las definiciones hechas (funciones y variables) se han perdido. De este modo, si queremos escribir un programa medianamente largo, es más conveniente escribirlo en un fichero de texto y después ejecutarlo como entrada al intérprete. En el caso que nuestro programa sea aún más largo, puede interesar dividirlo en diferentes ficheros para facilitar su manejo y mantenimiento. Además, si tenemos una función definida en un *script*, es probable que deseemos utilizarla en varios programas pero sin tener que copiar su definición en todos ellos.

Para realizar este tipo de tareas, Python establece las definiciones en un fichero y las utiliza bien en un *script* o en el intérprete interactivo. Este tipo de ficheros recibe el nombre de módulos, las definiciones de un módulo pueden ser importadas a otros módulos o al programa principal.

El nombre del fichero se corresponde con el nombre del módulo con el sufijo `.py`. El nombre del módulo se encuentra disponible como cadena en el valor de la propiedad `.__name__`. A continuación vamos a crear una función que almacenaremos en el fichero `fipo.py` en el mismo directorio donde ejecutemos el intérprete.

```
# módulo de los números de Fibonacci
def fib(n):    # escribe las series de Fibonacci hasta n
    a, b = 0, 1
    while b < n:
```

```
        print b,
        a, b = b, a+b

def fib2(n): # devuelve las series de Fibonacci hasta n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Si ejecutamos el intérprete de Python, podemos tener acceso a estas funciones importando el módulo del siguiente modo:

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Como vemos, al importar el módulo `fibo` tenemos acceso a las funciones que hay definidas en él. Esta modularidad implementada en Python, nos va a ser muy útil y será el principal mecanismo mediante el cual crearemos las interfaces y librerías PyACTS que conforman uno de los objetivos del presente trabajo.

Pero no todas las definiciones de un módulo han de ser accesibles desde otros, un módulo tiene su propia tabla de símbolos privados utilizada únicamente por las funciones definidas en dicho módulo. Por otro lado, para acceder a los símbolos de cada módulo es recomendable (y así lo hemos realizado en toda la documentación de PyACTS) referirse a un símbolo mediante su correspondiente módulo (`modname.itemname`).

Un módulo puede importar a otros, y generalmente estas sentencias se sitúan al comienzo del script aunque no es obligatorio. Además, también podemos seleccionar qué símbolos deseamos importar desde un módulo determinado tal y como se muestra en el ejemplo siguiente.

```
>>> from fibo import fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

De este modo, no se puede acceder a la función `fib2` y no podríamos ejecutar directamente `fib2(100)`. En el caso que queramos importar todos los símbolos definidos en el módulo, podríamos utilizar `from fibo import *`. Esto importaría todos los símbolos excepto aquellos que comenzaran por "_". Sin embargo, esta práctica no se ha realizado en el presente trabajo, puesto que la importación de todos los símbolos de un módulo puede dar lugar a colisiones en los nombres de los símbolos, es decir, puede ocurrir que importemos los símbolos (funciones, variables, objetos,...) de módulos diferentes y el nombre de alguno de ellos coincida con lo que el funcionamiento final no sería el deseado.

El proceso de búsqueda de un módulo

Cuando un módulo (por ejemplo `PyACTS`) es importado, el intérprete busca un fichero `PyACTS.py` en el directorio actual del usuario, y a continuación en la lista de directorios especificada por la variable de entorno `PYTHONPATH`. Esta lista de directorios se puede obtener desde el intérprete mediante la variable `sys.path`, que se inicializa con el directorio actual y con la lista de directorios de la variable `PYTHONPATH`. Ésto permite a los programas en Python, modificar en tiempo de ejecución el camino de búsqueda de los módulos.

Ficheros compilados

Se puede conseguir una reducción considerable en los tiempos de inicialización de programas cortos cuando se hace uso de ficheros compilados. Estos ficheros tienen la extensión `.pyc`, de modo que en el módulo citado en la sección anterior, obtendríamos `PyACTS.pyc`. Cuando existe un fichero con extensión `.pyc` se importa este fichero en lugar del `.py` que aparece en el mismo directorio siempre y cuando la fecha de modificación sea posterior.

Generalmente no es necesario realizar ninguna tarea específica para generar el archivo `.pyc`, y éste se genera cuando se interpreta el módulo por primera vez. El contenido de los archivos `.pyc` es independiente de la plataforma por lo que puede ser compartido por máquinas con distinta arquitectura.

Se ha de tener en cuenta que el hecho de utilizar archivos compilados `.pyc` hace que la lectura de las instrucciones sea más rápida y no la ejecución de las mismas.

Por otro lado, es posible tener un módulo con su archivo `.pyc` sin su correspondiente archivo `.py`. Esto permite distribuir un módulo sin tener que distribuir su código fuente. Sin embargo, no es éste el objetivo de nuestro trabajo puesto que pensamos que nuestra herramienta puede ser útil a la comunidad científica y se requiere el código fuente en el caso de querer realizar pequeñas modificaciones enfocadas a cada aplicación científica.

Módulos estándar

La distribución de Python, proporciona una librería de módulos estándar descritos en [33]. Algunos módulos son internos al intérprete; éstos proporcionan el acceso a las operaciones que no forman parte del núcleo del lenguaje pero que por eficiencia, o por realizar llamadas a primitivas del sistema operativo se han incluido en el mismo. Por ejemplo, el módulo `sys` se encuentra disponible en todos los intérpretes de cualquier plataforma, las variables `sys.ps1` y `sys.ps2` definen las cadenas utilizadas como indicativos de órdenes en la línea de comandos pero únicamente en el modo interactivo.

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Hola!'
Hola!
C>
```

3.1.4. El concepto de paquete

Los paquetes son una forma de estructurar el espacio de nombres en los módulos de Python como si de objetos se tratase. Por ejemplo, el módulo con nombre `A.B` designa al submódulo `B` en un paquete llamado `A`. De este modo, los autores de diferentes paquetes no han de preocuparse por el nombre que utilizan en sus módulos siempre y cuando el nombre del paquete sea distinto al resto.

En el capítulo 5 describiremos en mayor profundidad la distribución PyACTS, sin embargo, podemos adelantar en esta sección que para conseguir nuestro objetivo hemos creado un paquete (llamado `PyACTS`) que dispone de distintos módulos y cuya estructura mostramos a continuación. Deseamos resaltar la tipología que hemos utilizado al hablar de la distribución PyACTS como nombre de una distribución de software, y `PyACTS` como nombre de un paquete de Python incluido en dicha distribución.

```
PyACTS/
      __init__.py
      PyBLACS.py
      PyPBLAS.py
      PyScaLAPACK.py
      PyScaLAPACK_Tools.py
      pyscalapack.so
```

De este modo, podremos acceder, por ejemplo, a cualquiera de las funciones de la librería PyPBLAS del siguiente modo: `PyACTS.PyPBLAS.pvaxpy(alpha,x,y)`. En esta distribución, observamos que existe un archivo llamado `__init__.py`. Este archivo es necesario para tratar el directorio como paquete contenedor de módulos y nos permite ejecutar e inicializar determinados parámetros y funciones del módulo. En el paquete `PyACTS`, utilizaremos este archivo para inicializar y asignar los principales parámetros de la configuración de malla por defecto, como son : `icread`, `irread`, `nb`, `mb`, ...

Hasta el momento, hemos visto el manejo de los módulos desde Python, y hemos considerado en todo momento que estos módulos son archivos con extensión `.py` o `.pyc`. Sin embargo, podemos observar en el listado del párrafo anterior un archivo con extensión `.so`. Este archivo también será tratado como un módulo desde Python pero tiene una peculiaridad bien distinta puesto que se trata de una librería compartida de funciones

escritas en FORTRAN o C y a la que vamos a acceder desde Python. El proceso de diseño y generación de estas librerías lo explicamos en la siguiente sección.

3.1.5. Extendiendo y embebiendo Python

Una de las principales características de Python, nombrada en la sección 3.1, es la extensibilidad de Python, esto es, que a partir de un código ya escrito en C, podremos crear las interfaces adecuadas para poder acceder al mismo desde Python. A través de estas extensiones de Python, podemos crear nuevos tipos de objetos y llamar a funciones de librerías en C. Para poder utilizar estas extensiones, mediante `Python API` (*Application Programmers Interface*) se define un conjunto de funciones, macros y variables que proporciona el acceso a la mayor parte de las características de Python. Esta `Python API` se incorpora a nuestro código fuente en C incluyendo la cabecera "`Python.h`" en el mismo.

Durante esta sección, trataremos de describir las principales características de este proceso de extensión. Nos centramos principalmente en esta característica de Python, puesto que para la consecución de las librerías PyACTS hemos tenido que extender las librerías disponibles en la colección ACTS [16] y hacerlas accesibles desde Python.

Un ejemplo sencillo

Para mostrar mejor el proceso de creación de una extensión, vamos a crear un módulo llamado "`spam`" que ejecute una llamada a una función del sistema. No importa la complejidad de la operación sino el hecho que esta operación se realice desde una librería en C. Nuestro objetivo será conseguir una rutina que se pueda utilizar desde Python del siguiente modo:

```
>>> import spam
>>> status = spam.system("ls -l")
```

Comenzaremos creando el archivo `spammodule.c`. No es obligatorio pero generalmente los archivos en C que contienen la funcionalidad del módulo reciben el sufijo `module.c` a

partir del nombre del módulo.

La primera línea de este archivo puede ser:

```
#include <Python.h>
```

Esta línea incluye la API de Python. Por conveniencia, todos los símbolos definidos en la cabecera `Python.h` tienen el prefijo `Py` o `PY`. A continuación, añadiremos en nuestro fichero la función que queremos que se ejecute cuando en Python introduzcamos la expresión `spam.system(string)`.

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;
    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

En este ejemplo, se realiza una traslación de los parámetros de Python (por ejemplo, la expresión "ls -l") a los argumentos pasados a la función C. La función C de este ejemplo siempre tiene dos argumentos, convencionalmente llamados `self` y `args`. El argumento `self` sólo se utiliza cuando la función C implementa métodos internos, no cuando implementa funciones. En este ejemplo, `self` será siempre un puntero con valor `NULL`, puesto que estamos definiendo una función y no un método.

Por otro lado, el argumento `args` es un puntero a un objeto *tuple* de Python que contiene los argumentos. Cada *item* de la *tuple* corresponde a un argumento en la llamada realizada. La función `PyArg_ParseTuple()` de la API de Python comprueba los tipos de los argumentos y los convierte a valores en C. Ésta utiliza unas cadenas de referencia para determinar cuales son los tipos requeridos en los argumentos para poder así convertirlos a su valor en C.

Por último, la función `PyArg_ParseTuple()` devuelve `true` si todos los argumentos tienen el tipo adecuado y sus componentes han sido almacenadas en las variables cuyas

direcciones fueron pasadas como parámetros. Esta función devuelve falso en el caso de haber algún argumento clasificado como no válido, en este caso salta la correspondiente excepción y la llamada a esta función devuelve `NULL` inmediatamente.

Para el tratamiento de las excepciones que se pueden originar durante los pasos mencionados anteriormente, en Python se han definido un conjunto de funciones (`PyErr_`) que permiten tratarlas y procesarlas antes de enviar esta excepción al nivel superior, es decir, al intérprete de Python. Preferimos no extendernos en el tratamiento de las extensiones en esta sección y mostrar una visión más general del proceso de extensión. En el caso que se desee obtener más información del mismo, recomendamos la lectura del punto 1.2 de la referencia [30].

Volviendo al código mostrado anteriormente, observamos que si el tipo de los argumentos son correctos, el programa seguirá su ejecución (`if (!PyArg_ParseTuple(args, "s", &command))...`). Por otro lado, el valor de la cadena se copia a la variable local `command`, siendo ésta un puntero declarado como `const char *command`.

En la siguiente línea se realiza la llamada a la función de sistema de Unix, pasando como argumento el puntero obtenido de `PyArg_ParseTuple()`:

```
sts = system(command);
```

La ejecución desde Python de `spam.system()` debe devolver el valor de `sts` como un objeto de Python. Para conseguir esto hemos de utilizar la función `Py_BuildValue()`, que a partir de una cadena indicativa del formato y de un numero arbitrario de valores en C, devuelve un nuevo objeto Python. En este caso, mediante el comando `return Py_BuildValue("i", sts);` devolvemos un objeto de tipo entero. En aquellos casos en los que la función C no devuelva ningún argumento (por ejemplo, una función que devuelve `void`) necesitaremos una correspondencia en Python implementada con `Py_RETURN_NONE`.

```
Py_INCREF(Py_None);
return Py_None;
```

Las funciones de inicialización y la tabla de los métodos del módulo

Para indicar a Python cómo ha de llamar a la función en C llamada `spam_system()`, tendremos que definirlo en la tabla de métodos. Esta tabla tiene el siguiente aspecto:

```
static PyMethodDef SpamMethods[] = {
    ...
    {"system",  spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

En este ejemplo hemos de tener en cuenta varios aspectos importantes. El primer parámetro representa el nombre de la función dentro del módulo (por eso se llama a `spam.system`). El segundo parámetro indica el nombre de la función en C (en este caso, `spam_system`). El tercer parámetro `METH_VARARGS` es un flag que indica al intérprete de Python la forma en la que llamar a la función C. Generalmente este valor es `METH_VARARGS` o `METH_VARARGS | METH_KEYWORDS`. Cuando sólo se usa la primera forma, la función esperará que los parámetros pasados tengan el formato adecuado. En el caso de indicar también `METH_KEYWORDS` significa que en los argumentos pasados aparecen funciones (por lo que habría que utilizar la función `PyArg_ParseTupleAndKeywords()`).

La tabla de métodos debe indicarse en la función de inicialización del módulo. La función de inicialización ha de llamarse `initname()`, donde `name()` es el nombre del módulo.

```
PyMODINIT_FUNC initspam(void)
{
   (void) Py_InitModule("spam", SpamMethods);
}
```

Observemos que `PyMODINIT_FUNC` declara la función con tipo de retorno vacío. Cuando el intérprete de Python importa el módulo `spam` la primera vez, se ejecuta `initspam()`. Éste a su vez ejecuta `Py_InitModule()`, el cual crea un objeto 'de tipo módulo' que será insertado en el diccionario `sys.modules` con la clave `spam`, e inserta la función

en el nuevo módulo a través de la tabla de métodos definida en el segundo argumento. `Py_InitModule()` devuelve un puntero al objeto módulo creado. La rutina se aborta si el módulo no se pudo iniciar satisfactoriamente.

Hasta el momento, en esta sección hemos visto cómo realizar una extensión a Python, es decir, utilizar desde Python funciones, objetos y rutinas definidas en librerías de C. Como ya hemos comentado, este ha sido uno de los puntos clave para conseguir las librerías `PyACTS`. Sin embargo, en este trabajo veremos cómo también hemos necesitado embeber el intérprete de Python (ver la sección 5.4).

En el caso de embeber a Python, estamos construyendo una aplicación (generalmente en C) que integra al intérprete de Python para adaptarlo o modificarlo con una finalidad determinada. En este caso, la función `initspam()` no se ejecuta de forma automática a menos que aparezca en la tabla `_PyImport_Inittab`. La forma más sencilla es inicializar los módulos llamando directamente a la función `initspam()` después de haber inicializado Python mediante `Py_Initialize()`.

```
int main(int argc, char *argv[])
{
    /* Pass argv[0] to the Python intérpreter */
    Py_SetProgramName(argv[0]);

    /* Initialize the Python intérpreter.  Required. */
    Py_Initialize();

    /* Add a static module */
    initspam();
}
```

Compilación y enlazado

Por último, hemos de realizar dos pasos más para obtener la extensión: compilar y enlazar con las librerías de Python. Si utilizamos la carga dinámica, los detalles dependen del estilo de carga dinámica que utilice el sistema (ver capítulos 3 y 4 de [30]). En el caso

que no podamos utilizar carga dinámica, o si queremos conseguir que el módulo sea parte permanente del sistema, tendremos que cambiar la configuración y regenerar el intérprete.

Este proceso de compilación y enlazado se puede realizar de forma manual, o bien mediante el módulo `dist` de Python. Mediante este módulo, será el propio intérprete de Python, el encargado de compilar y enlazar las fuentes para generar el archivo de librerías dinámicas (es decir, la extensión propiamente dicha). Este proceso lo describiremos más detalladamente en la siguiente sección.

3.1.6. Instalación de módulos

A pesar de que la librería de Python es muy extensa y cubre muchas necesidades, a menudo necesitamos nuevas funcionalidades desarrolladas por terceros (éste es el caso de los módulos de la librería PyACTS).

A partir de la versión 2.0 de Python, se ha incluido el módulo `distutils` que permite y facilita el proceso de instalación de módulos de terceras personas en la distribución de Python. En este apartado describiremos brevemente el proceso clásico de instalación de un módulo desarrollado por un tercero. Por otro lado, en el apartado siguiente veremos el proceso de instalación desde el otro lado, es decir, si somos nosotros los que hemos desarrollado un módulo y queremos ponerlo a disposición de los usuarios de Python, la utilidad `distutils` nos permite crear el módulo de una manera más guiada y normalizada.

En el mejor de los casos, puede que la persona que haya desarrollado el módulo haya incluido una versión específica para la plataforma donde la deseamos instalar. Por ejemplo, el módulo desarrollado puede ser un ejecutable para un usuario de Windows, un paquete RPM para un usuario de Linux basado en paquetes RPM (Red Hat, SuSE, Mandrake, y otros), o bien un paquete basado en Debian. En este caso no es necesario ningún proceso de compilación de forma que la instalación es realmente sencilla.

Pero éste no suele ser el caso (y no lo es en la distribución PyACTS). Generalmente será necesario acceder a las fuentes de la distribución del módulo para compilarlo e instalarlo en la plataforma que se desee. Este punto será el que trataremos en este apartado.

En el caso que descarguemos la distribución de las fuentes de un módulo, podemos comprobar rápidamente si ésta ha sido generada utilizando `distutils`. Generalmente, la

distribución ha de tener un archivo llamado `setup.py` y un fichero `README.txt` en el cual se explica que el proceso de construcción e instalación del módulo consiste básicamente en ejecutar el siguiente comando:

```
python setup.py install
```

Generalmente, si tenemos una distribución estándar, este comando basta para instalar el módulo. De todos modos, puede interesarnos dividir el proceso en dos partes: construcción del módulo a partir de las fuentes e instalación del mismo en el intérprete de Python.

```
python setup.py build
python setup.py install
```

El proceso de "construcción" se encarga de componer el módulo a partir de las fuentes. Por defecto, éste se construye en la dirección de la distribución aunque se le puede indicar donde crear la distribución mediante el comando :

```
python setup.py build --build-base=/tmp/pybuild/foo-1.0
```

Este proceso genera una serie de directorios dentro del directorio `build` que tienen el siguiente aspecto:

```
--- build/ --- lib/
or
--- build/ --- lib.<plat>/
               temp.<plat>/
```

donde `plat` se refiere a la plataforma o sistema operativo y a la versión de Python. En nuestro caso (distribución PyACTS), el directorio `build` tiene el aspecto del segundo caso, donde el directorio `temp.<plat>` contiene los ficheros temporales resultado del proceso de compilación y el directorio `lib.<plat>` contiene el módulo Python (incluyendo archivos de Python `.py` y las extensiones).

Después de generar el módulo, podemos indicar que lo instale en el intérprete de Python. En el caso de no indicar ningún directorio específico (`python setup.py install`),

Python establece una ubicación por defecto que depende de la plataforma en la cual esté instalado. En el caso de las pruebas realizadas, es decir en Unix, la ubicación de los módulos de Python es `/usr/local/lib/python<ver>/site-packages` donde `ver` indica la versión de Python.

Preferimos no extendernos ni detallar casos particulares en la instalación de módulos de terceros en Python. En el caso que el lector desee obtener más información se recomienda la lectura de la distribución oficial [31].

3.1.7. Distribución de módulos

En este apartado describiremos las utilidades para la distribución de Python, también conocida como `distutils`, desde el punto de vista del desarrollador de módulos. Veremos como `distutils` consigue que la distribución de nuevos módulos y extensiones sea más sencilla para el resto de la comunidad usuaria de Python. En nuestro trabajo, hemos debido tener en cuenta estos mecanismos, no directamente para desarrollar las extensiones pero sí para luego hacer que la utilización de las mismas sea lo más extensa posible por la comunidad científica. Estos detalles serán ampliados cuando describamos la creación de la distribución PyACTS en la sección 5.7.

La utilización de `distutils` permite tanto a desarrolladores como a administradores generar e instalar módulos de Python, respectivamente. En nuestro caso, como desarrolladores de PyACTS tenemos las siguientes responsabilidades:

- Escribir un *script* `setup.py`.
- Opcionalmente podemos escribir un archivo de configuración.
- Crear una distribución de las fuentes.
- Opcionalmente podemos crear una o mas distribuciones ya construidas para plataformas específicas.

Detallaremos cada una de estas tareas de una forma breve en esta sección.

Un ejemplo básico

El *script* `setup.py` suele ser bastante sencillo, ya que está escrito en Python y no hay límites arbitrarios de qué podemos hacer en él. Sin embargo, debemos ser cuidadosos en las operaciones desarrolladas en él, por ejemplo, en el caso de *scripts* de auto-configuración, el `setup.py` se suele ejecutar múltiples veces y ésto puede cargar el sistema. Supongamos que queremos distribuir nuestro módulo básico llamado `pyacts` que contiene un archivo `pyacts.py`, entonces nuestro *script* tendrá el siguiente aspecto:

```
from distutils.core import setup
setup(name='pyacts',
      version='1.0',
      py_modules=['pyacts'],
      )
```

En este código podemos apreciar que la utilidad que utilizamos de `distutils` es la función `setup()`. En esta función indicamos los parámetros con su correspondiente clave: datos del paquete, indicando el nombre y la versión; módulo(s) del paquete, indicando su nombre y no el nombre del archivo.

Si deseamos crear una distribución para este módulo, podemos hacerlo a partir del código mostrado anteriormente y guardándolo como un archivo `setup.py`, para a continuación ejecutar:

```
python setup.py sdist
```

Este comando creará un archivo (tipo `tar` en Unix, o `zip` en windows) que contiene el archivo de configuración `setup.py` y el módulo a distribuir `pyacts.py`. El nombre del archivo creado será `pyacts-1.0.tar.gz` (o `.zip`), y se descomprimirá en el directorio `pyacts-1.0`. Si un usuario deseara instalarse el módulo `pyacts`, deberá descargarse la distribución, descomprimir el fichero, y en el directorio mencionado ejecutar:

```
python setup.py install
```

Suponiendo una distribución de módulos como la descrita al comienzo del apartado 3.1.4, esta instrucción copiaría la carpeta `PyACTS` con los ficheros `.py` en su interior en

el directorio correspondiente a los módulos desarrollados por terceros en el intérprete de Python.

En este ejemplo hemos mostrado los conceptos fundamentales de `distutils`. De este modo, tanto desarrolladores e instaladores tienen la misma intefaz (`setup.py`), aunque la diferencia se centra en el tratamiento que hacen cada uno de ellos a través de los comandos adicionales (`install`, `dist`, ...).

Por otro lado, podemos facilitar aún más la tarea a los usuarios si creamos una o mas distribuciones ya compiladas. Por ejemplo, si queremos facilitar la instalación para los usuarios de Windows, podemos crear un ejecutable del siguiente modo:

```
python setup.py bdist_wininst
```

De este modo se crea un archivo `pyacts-1.0.win32.exe` en el directorio actual. También es útil los formatos de distribución como `RPM`, implementados mediante el parámetro `bdist_rpm`, o para solaris `bdist_pkgtool` o plataformas HP-UX `bdist_sdux`. Si deseamos obtener más información acerca de los formatos disponibles, podemos hacerlo mediante el siguiente comando:

```
python setup.py bdist --help-formats
```

Hasta el momento hemos visto cuales son los fundamentos para realizar la distribución de paquetes mediante `distutils`. Sin embargo, la complejidad del presente trabajo (el paquete o librería PyACTS) hace que para distribuir el mismo debamos completar el *script* de configuración con información adicional. Esta información la trataremos en el siguiente punto.

Edición del archivo de configuración

Antes de proceder con las configuraciones especiales del archivo `setup.py`, es conveniente conocer el significado de cada término utilizado en la herramienta `distutils`.

- **Módulo**: La unidad básica de la reusabilidad del código Python, es decir, un bloque de código importado por otro código. Dentro de los módulos podemos tener tres tipos: módulos puros de Python, módulos de extensión, y paquetes.

- **Módulo puro de Python**: Módulo escrito únicamente en Python en un fichero `.py` (y sus posibles asociados `.pyc` y `.pyo`).
- **Módulo de extensión**: Módulo escrito con un lenguaje de bajo nivel (C/C++) para Python o Java para Jython. Normalmente está contenido en un fichero precompilado (por ejemplo, una librería compartida `pyscalapack.so` en nuestro caso) para extensiones de Python en Linux o un archivo `.dll` para distribuciones en Windows.
- **Paquete**: Un módulo que contiene otros módulos, típicamente contenidos todos en un directorio con el archivo `__init__.py`.
- **Módulo de distribución**: Conjunto de módulos de Python distribuidos juntos como un recurso conjunto para ser instalado en masa. Algunos ejemplos de este tipo son Numeric Python [4], PyXML [32] y en este caso PyACTS.

Una vez descritos algunos conceptos previos, procedemos a describir las principales características del archivo de configuración `setup.py`. Este archivo es el núcleo principal en el proceso de construcción, distribución e instalación. Creemos conveniente explicar los distintos parámetros de `setup.py` con ejemplos. Además, si el ejemplo utilizado es el archivo de configuración utilizado en el trabajo presentado, conseguiremos dos propósitos: explicar el funcionamiento general del script `setup.py` y aplicar estas utilidades a la distribución del paquete PyACTS.

Por otro lado, las sucesivas versiones que del paquete PyACTS se vayan desarrollando podrán incorporar modificaciones en su archivo `setup.py`. A pesar de todo, pensamos que las partes importantes del mismo no se modificarán sino que se añadirían nuevas funcionalidades. Iremos explicando línea a línea las principales funciones del archivo `setup.py` distribuido en el paquete PyACTS.

Las primeras líneas las hemos destinado como firma del archivo de configuración así como del resto del paquete, debido a que el caracter "#" es el identificador de los comentarios, el archivo `setup.py` comenzará del siguiente modo:

```
#-----------------------------------------------------------------------
# File      : setup.py
# Programmer: Vicente Galiano Ibarra  vgaliano@umh.es
# Version of: October 1, 2006, Elche (Alicante)
# Package:    PyACTS ; http://www.pyacts.org
#-----------------------------------------------------------------------
```

A continuación en el *script* aparecen unas líneas cuya finalidad no es la descripción del paquete PyACTS, sino que comprueban el sistema de librerías del usuario y la versión del intérprete de Python instalado. En el caso que éste fuera anterior a la versión 2.0 informaríamos al usuario de la imposibilidad de instalar el mismo debido a que `distutils` se encuentra disponible a partir de la versión 2.0 de Python.

```
import sys, os
from distutils.core import setup, Extension

lib = os.path.join(os.path.join(sys.prefix, 'lib'),
'python'+sys.version[:3])
site_packages = os.path.join(lib, 'site-packages')

if not hasattr(sys, 'version_info') or
sys.version_info < (2,0,0,'alpha',0):
    raise SystemExit, "Python 2.0 or later required to build PyACTS."
```

Podemos comprobar cómo la última línea provoca la salida de la ejecución del *script* de instalación sin llevarse ésta a cabo.

Al igual que hemos visto en el apartado 3.1.7, en nuestro archivo de configuración llamamos a la función `setup.py` proporcionada en las `distutils`. Sin embargo en este caso hemos añadido algunas líneas por causas no tan triviales o sencillas:

```
setup (name = "PyACTS",
       version = "1.0.0",
       description = "Python Interface to ACTS Collection.
       First Version has ScaLAPACK, PBLAS and BLACS interface.",
       author = "L.A. Drummond, V. Galiano, J. Penades and V. Migallon",
       author_email = "vgaliano@umh.es",
       url = "http://www.pyacts.org",
       package_dir = { 'PyACTS':'LIB/PyACTS'},
       packages = ["PyACTS"],
       ext_modules =[module_pyscalapack]
       )
```

Para conseguir una mayor documentación e identificación del paquete le hemos añadido claves adicionales como `description`, `author`, `author_email` y `url`. Sin embargo, podemos comprobar la nueva clave definida como `packages`. Esta opción le indica a `distutils` el proceso de instalación de todos los módulos puros encontrados en cada paquete de la lista de paquetes mencionada. En nuestro caso, únicamente distribuimos el paquete PyACTS, sin embargo en la misma distribución podemos llegar a tener diversos paquetes (por ejemplo, PyACTS y PyPNetCDF en una misma distribución). Ha de haber una correspondencia entre el nombre de los paquetes y los directorios en el sistema de ficheros, de este modo si indicamos el paquete `PyACTS` implica la existencia de un fichero `PyACTS/__init__.py`. En el caso de utilizar una convención diferente no representa un problema puesto que podemos hacerlo (y lo hacemos) mediante la clave `package_dir`. Mediante el diccionario `'PyACTS':'LIB/PyACTS'` estamos indicando a `distutils` que la distribución de PyACTS se encuentra dentro del directorio `LIB` a partir del directorio raíz de la distribución (donde se encuentra `setup.py`).

En resumen, la clave `package` se utiliza para únicamente módulos puros de Python. Sin embargo, es frecuente encontrarse con módulos que utilizan extensiones (ver la sección 3.2) y por tanto ya no tenemos distribuciones con sólo código Python. En este caso, describir el paquete mediante `distutils` es más complicado porque hemos de especificar nombres de extensiones, ficheros fuente, y los requerimientos de los compiladores (directorios de archivos de inclusión, librerías a enlazar, etc.).

De este modo, en la rutina `setup()`, le indicamos la clave `ext_modules` como una lista de extensiones. En nuestro caso, y tal y como hemos visto en el código anterior, tenemos definida una lista de extensiones con un único elemento que hemos llamado `module_pyscalapack`. La definición de esta extensión según `distutils` se muestra a continuación:

```
module_pyscalapack = Extension( 'PyACTS.pyscalapack',
    ['SRC/fortranobject.c','SRC/pyscalapackmodule.c'],
    libraries = libraries_list,
    library_dirs=library_dirs_list,
    include_dirs = include_dirs_list
#   ,define_macros=[('F2PY_REPORT_ATEXIT', '1')]
    )
```

La clase `Extension` es importada desde `distutils.core` (ver primeras líneas del script) y maneja gran parte de la flexibilidad de Python en el manejo de las extensiones. El primer argumento del constructor `Extension` es siempre el nombre de la extensión (en este caso `PyACTS.pyscalapack`).

El segundo parámetro es una lista con las fuentes de la extensión. `distutils` únicamente soporta los lenguajes C, C++. Además de éstos, podemos incluir a ficheros de definición de interfaces SWIG (ver apartado 3.3.1) de tipo `.i` de modo que la clave `build_ext` conoce como manejar estas interfaces que se han llegado a convertir en un estándar para Python.

Además de los dos parámetros vistos, existen tres parámetros opcionales que nos ayudarán a especificar los directorios de búsqueda de los archivos de inclusión o definir las macros de preprocesado: `include_dirs`, `define_macros`, y `undef_macros`.

En nuestro caso hemos creado una lista de ubicaciones de los *includes* necesarios para la instalación y que el instalador deberá modificar para la plataforma específica. Como vemos en la siguiente figura, para no complicar las tareas del instalador, en la propia distribución adjuntamos los *includes* necesarios por lo que no habría que modificar esta línea.

```
include_dirs_list = ['./SRC']
```

Por otro lado, podemos definir macros de preproceso con las claves `define_macros` y `undef_macros` como una *tuple* de parejas "`nombre:valor`" donde `nombre` es el nombre de la macro a definir y `value` su correspondiente valor (puede ser una cadena o `None`). En el caso del paquete PyACTS, podemos comprobar que está comentada la línea donde se realiza la declaración (`# define_macros=[('F2PY_REPORT_ATEXIT', '1')]`. Esta línea no la incluimos en la versión final de la distribución PyACTS, sin embargo hemos querido mostrarla en este apartado puesto que establece una macro de preproceso que nos será muy útil con la herramienta F2PY que describiremos en la sección 3.3.2.

Por último, mediante `distutils` podemos especificar las librerías con las que podemos enlazar nuestra extensión e indicar los directorios donde se encuentran estas librerías mediante la clave `library_dirs_list`. Hemos creado una lista de las distintas librerías, y se ha de tener en cuenta que, por ejemplo, si la librería que indicamos en `libraries_list` es `scalapack` entonces el nombre de la librería que busca el intérprete para enlazarla es `libscalapack.a`, es decir habría que añadirle el prefijo `lib` y la extensión `.a` a cada una

de las librerías indicadas.

```
library_dirs_list=['./BUILD','/usr/local/BLACS/BLACS_PYTHON/LIB',
                   '/usr/local/mpich/lib','/usr/lib',
                   '/usr/local/SCALAPACK/SCALAPACK/'
                    ]
libraries_list = [
             'scalapack',
             'blacsF77init_MPI-LINUX-0',
             'blacs_MPI-LINUX-0',
             'blacsF77init_MPI-LINUX-0',
             'blacsCinit_MPI-LINUX-0',
             'blacs_MPI-LINUX-0',
             'blacsCinit_MPI-LINUX-0',
             'blas',
             'mpich',
             'g2c']
```

Las librerías enlazadas pertenecen a las librerías de MPI, BLACS, PBLAS y ScaLAPACK que veremos y describiremos más detenidamente en los apartados 4.3.1, 4.3.3, 4.3.4 y 4.3.5 respectivamente.

La mayor parte de la configuración del *script* explicado en este apartado se ha basado en el script `setup.py` utilizado en Cluster-umh (plataforma descrita en la sección 2.6). Sin embargo, también hemos instalado nuestro paquete en máquinas con arquitecturas diferentes (por ejemplo, en Seaborg, que también describiremos en la sección 2.6). En el caso de Seaborg, se trata de un multiprocesador IBM SP2 y por tanto tendrá un sistema operativo y una arquitectura diferente. Esto implica que los nombres de las librerías a enlazar serán diferentes al Cluster-umh.

A continuación mostramos el código que hace referencia a las librerías en el *script* de instalación para Seaborg:

```
library_dirs_list=[
         './BUILD',
         '/u8/vgaliano/LIBS',
         '/usr/lpp/ppe.poe/lib/',
```

```
         '/usr/common/usg/python/2.2.2/lib/pyMPI1.3',
]
libraries_list = [
        'scalapack_SP',
        'redist_SP',
        'tools_SP',
        'pblas_SP',
        'blacsF77init_MPI-SP-0',
        'blacs_MPI-SP-0',
        'blacsF77init_MPI-SP-0',
        'blacsCinit_MPI-SP-0',
        'blacs_MPI-SP-0',
        'blacsCinit_MPI-SP-0',
        'blas',
        'mpi_r',
        'pympi'
    ]
```

Como podemos comprobar, la lista `libraries_list` no se corresponde con la lista mostrada para el Cluster-umh, sin embargo el conjunto de interfaces que conforman la librería son los mismos por lo que el proceso de instalación se lleva a cabo de manera satisfactoria.

Por otro lado, hemos de tener en cuenta que los compiladores no son los mismos en arquitecturas diferentes y cada uno de ellos puede generar resultados distintos. Esto precisamente ocurre para el caso de Seaborg. En esta plataforma hemos necesitado definir una macro en el archivo `setup.py` para que el compilador no añada a cada una de las funciones el sufijo "_". Para conseguirlo, hemos añadido en la definición de la extensión la línea `define_macros=[('NO_APPEND_FORTRAN', '1')]`.

3.2. Módulos y paquetes

En la sección 3.1, hemos hecho un breve recorrido sobre las principales características del lenguaje Python, así como de las herramientas y utilidades que incorpora para crear extensiones, distribuirlas e instalarlas en las distintas plataformas en las que Python se encuentra disponible.

Tal y como comentamos en el apartado 3.1.6, además de los módulos que incorpora la distribución estándar, existen módulos desarrollados por terceros que incorporan funcionalidades que no están cubiertas por los módulos estándar. Uno de nuestros objetivos es proporcionar un paquete para Python que nos permita la utilización de las librerías pertenecientes a la colección ACTS (más explícitamente las librerías BLACS, PBLAS y ScaLAPACK). Sin embargo, para conseguir un módulo cómodo e integrado completamente con el conjunto de herramientas disponibles para Python, hemos necesitado hacer uso de módulos desarrollados por terceros y que serán utilizados por las fuentes de PyACTS.

En esta sección describiremos cuales han sido los módulos de terceros utilizados y las principales funcionalidades que han sido de mayor utilidad en el desarrollo de nuestra librería.

3.2.1. Numeric Python

Las extensiones llamadas `Numeric Python` [4] (o en su modo abreviado `Numpy`) son un conjunto de extensiones al lenguaje de alto nivel Python que permite a los programadores manipular eficientemente elevados volúmenes de datos organizados en matrices. Estas matrices pueden tener cualquier número de dimensiones: en el caso de una dimensión, el aspecto de los arrays es muy similar al de las listas en Python; con dos dimensiones las matrices son similares al tratamiento de éstas en el álgebra lineal.

La necesidad de estas extensiones se basa en el requerimiento de manejar grandes cantidades de datos en Python con estructuras diferentes a las listas, tuples o clases, puesto que éstas no son eficientes y requieren mucho espacio en estos casos. Las rutinas disponibles en `Numpy` o `Numeric` proporcionan las operaciones más comunes que se suelen realizar en el tratamiento de matrices mediante una interfaz sencilla y bien documentada.

Los propios autores reconocen que para ofrecer esta similitud se han basado en la facilidad que incorporan otros lenguajes en el tratamiento de matrices tales como Basis, Matlab, Fortran, S/S+, y otros.

A continuación, realizaremos un breve resumen de las principales características de `Numpy` detallando aquellas funcionalidades utilizadas en la extensión PyACTS.

Una visión general

En este apartado, pretendemos proporcionar una visión a grandes rasgos de las extensiones `Numpy` a partir de las definiciones de los principales componentes de la misma.

Las `Numpy` consisten en un módulo principal `Numeric.py` en el que se definen dos nuevos tipos de objetos y un conjunto de funciones que permiten la manipulación de dichos objetos. Estos objetos son los llamados `multiarray` y funciones universales (`ufunct`). Estos objetos de Python se llaman formalmente "multiarray" aunque generalmente se hace referencia a ellos como "array". Realizamos esta puntualización, porque estos objetos son diferentes a los definidos en el módulo de la distribución estándar de Python llamado "array" poco usados de forma general.

Los objetos "array" de `Numpy` generalmente son colecciones homogéneas de un elevado volumen de números. Todos los números pertenecientes a un `multiarray` tienen el mismo tipo (precisión doble, coma flotante, etc.). Estas matrices no pueden estar vacías y su tamaño es inmutable. Como peculiaridad, las operaciones en estas matrices se realizan elemento a elemento y no conforme al álgebra clásica. Por ejemplo, si se definen dos matrices `a` y `b` del mismo tamaño, la operación `c=a*b` resultará en una nueva matriz del mismo tamaño donde sus elementos son el resultado del producto de los elementos de las matrices ($c_{ij} = a_{ij} * b_{ij}$).

Cada matriz de tipo `array` tiene las siguientes propiedades:

- El tamaño `size` se corresponde con el numero total de elementos y éste no cambia en el tiempo de vida de la matriz.
- La propiedad `shape` se corresponde con las dimensiones de la matriz. Esta propiedad puede cambiar durante la vida de la matriz. En términos de Python, la propiedad `shape` es un elemento de tipo tuple con un entero para cada dimensión.

- La propiedad rango (`rank`) de una matriz se refiere al numero de dimensiones en las que está definida. Esta propiedad también podrá cambiar en el tiempo de vida de la matriz.
- El tipo de dato (`typecode`) de una matriz se corresponde con un caracter indicativo del tipo de elementos que contiene. Este parámetro será de importancia cuando tengamos que relacionar las funciones proporcionadas en las rutinas PyACTS con los objetos `array` definidos en `Numpy`.
- La propiedad `itemsize` se refiere al número de bytes utilizados para almacenar un único elemento de la matriz.

Estas propiedades son directamente accesibles desde el intérprete de Python tal y como mostramos en el siguiente ejemplo:

```
>>> from Numeric import *
>>> vector1 = array([1,2,3,4,5])
>>> print vector1
[1 2 3 4 5]
>>> matrix1 = array(([0,1,4],[1,3,5]))
>>> print matrix1
[[0 1 4]
[1 3 5]]
>>> print vector1.shape, matrix1.shape, matrix1.rank
(5,) (2,3) 2
>>> print vector1 + vector1
[ 2 4 6 8 10]
>>> print matrix1 * matrix1
[[0 1 16]
[1 9 25]]
```

Fundamentos del tratamiento de matrices

Pretendemos en este apartado mostrar ejemplos claros y sencillos donde se muestren las principales características en el tratamiento de las matrices mediante el paquete `Numpy`.

Los arrays `Numpy` poseen dos características basadas en el comportamiento intuitivo de las operaciones que las hacen muy interesantes:

- Por defecto, las operaciones en las matrices se realizan elemento a elemento. Por ejemplo, en el caso de las multiplicaciones tiene la lógica adecuada pero en el caso de la multiplicación con el operador `*` se realizará elemento a elemento, muy diferente a la multiplicación de dos matrices vista en el álgebra lineal. Para operaciones que no se desea que se realicen elemento a elemento, `Numpy` ofrece funciones que las realizan de una manera sencilla.
- Algunas funciones devuelven matrices cuyo contenido hace referencia a partes de otras matrices de datos. Este comportamiento lo mostraremos más adelante con ejemplos ilustrativos.

Tenemos diferentes formas de crear un `array`, la forma más básica es utilizar la función `array` del siguiente modo:

```
>>> a = array([1.2, 3.5, -1])
>>> print a
[ 1.2 3.5 -1. ]
```

De este modo, la función `array(numbers, typecode=None, savespace=0)` requiere de tres argumentos: el primero de ellos se corresponde con los valores de la matriz; el segundo y el tercero son opcionales y se refieren al tipo de datos en el que están representados y al espacio de guardado de esos datos. Esta última opción la introduciremos más adelante en este apartado. En el caso que estos parámetros sean omitidos, Python designará un tipo en el cual esos elementos pueden ser clasificados. Pero en otros casos, puede ser interesante especificar de una forma explicita el tipo de datos que queremos que se utilice en un `array`:

```
>>> x,y,z = 1,2,3
>>> a = array([x,y,z]) # son de tipo entero
>>> print a
[1 2 3]
>>> a = array([x,y,z], Float) # se indica otro tipo por defecto
>>> print a
[ 1. 2. 3.]
```

En este ejemplo apreciamos como las valores `[1 2 3]` son clasificados como enteros si no se especifica lo contrario o bien, especificando el tipo `Float` podemos representarlos en formato de coma flotante.

En este punto es interesante que nos detengamos en los posibles valores que se pueden especificar en el campo `typecode`, puesto que a partir del tipo de datos que contenga un `array` elegiremos la función de las librerías PyACTS que sea más adecuada. Por tanto, los tipos de datos disponibles, y sus identificadores son :

1. Corresponde con un caracter ASCII el valor `Character`.
2. Tipo Numérico utilizado para almacenar valores entre 0 y 255, se corresponde con `UnsignedInt8`.
3. Enteros con signo podremos utilizar `Int`, `Int0`, `Int8`, `Int16`, `Int32` en algunas plataformas y `Int64` y `Int128` en otras, dependiendo de las características de la plataforma.
4. Números con coma flotante están definidos como `Float`, `Float0`, `Float8`, `Float16`, `Float32`, `Float64` y en algunas plataformas además se puede utilizar `Float128`.
5. Los números complejos se representan como `Complex`, `Complex0`, `Complex8`, `Complex16`, `Complex32`, `Complex64` y `Complex128`.
6. Un tipo no numérico definido es `PyObject`. Las matrices con este tipo de datos contienen referencias que pueden apuntar a cualquier tipo de datos de Python.

Las versiones donde no se les indica número a los tipos (`Int`, `Float`, `Complex`) corresponden a los tipos por defecto en Python, es decir, a enteros largos, números en coma flotante doble y tipos complejos.

Respecto a las dimensiones de una matriz, éstas se pueden modificar mediante comandos como `reshape()`. De este modo, se trata a la matriz como una sucesión de valores y se les asignan las dimensiones indicadas:

```
>>> a = array([1,2,3,4,5,6,7,8])
>>> print a
[1 2 3 4 5 6 7 8]
>>> b = reshape(a, (2,4)) # 2*4 == 8
```

```
>>> print b
[[1 2 3 4]
[5 6 7 8]]
```

Además de este comando, el módulo `Numeric` incorpora diversas funciones que actuan sobre variables de tipo `array` modificando sus propiedades.

Por otro lado, también podemos crear variables en el tiempo de ejecución del intérprete, para ello se proporcionan rutinas que han sido frecuentemente utilizadas en el paquete PyACTS. Mostraremos de forma muy breve las principales rutinas utilizadas:

- `zeros()` y `ones()` : Crea matrices con las dimensiones indicadas conteniendo ceros o unos en todos sus elemento.

```
>>> z = zeros((3,3))
>>> print z
[[0 0 0]
[0 0 0]
[0 0 0]]
>>> o = ones((2,3), Float)
>>> print o
[[ 1. 1. 1.]
[ 1. 1. 1.]]
```

- `arrayrange()`: Similar a la función `range` de Python pero ésta crea un `array` en lugar de una lista.

- `fromfunction()`: Crea una matriz en la que sus elementos se obtienen de una función utilizando como variables sus índices.

```
>>> def dist(x,y):
... return (x-5)**2+(y-5)**2 # distance from point (5,5) squared
...
>>> m = fromfunction(dist, (10,10))
```

Operaciones con matrices

Como ya hemos comentado, muchas de las operaciones en las matrices se realizan elemento a elemento, además de esto, se pueden realizar operaciones correctas desde un punto de vista lógico pero no desde un punto de vista matemático. Tal es el caso de la suma de un entero con una matriz, esa operación no es válida en el álgebra lineal, pero se puede realizar en el paquete `Numpy`.

```
>>> print a
[1 2 3]
>>> print a * 3
[3 6 9]
>>> print a + 3
[4 5 6]
```

Las operaciones a realizar dependen de si éstas son con números o matrices. Si una de ellas es un número, se aplica a todos los elementos de la matriz. Esto ocurre también con un conjunto de funciones como el seno y el coseno.

```
>>> print sin(a)
[ 0.84147098 0.90929743 0.14112001]
```

Cuando los dos elementos son matrices, se crea una nueva matriz con el resultado de dicha operacion siempre que no exista algun problema en el cálculo.

```
>>> print a
[1 2 3]
>>> b = array([4,5,6,7]) # Este vector tiene 4 elementos
>>> print a + b
Traceback (innermost last):
File ‘‘<stdin>‘‘, line 1, in ?
ArrayError: frames are not aligned
```

Como hemos comentado, en estas operaciones Python crea una nueva matriz con el resultado de la operación. Sin embargo, se pueden realizar operaciones *in situ*, es decir, en los mismos datos de la operación mediante los símbolos `+=`, `-=`,`*=`, y `/=`. En este caso hay que tener cuidado con el tipo de datos puesto que no puede cambiar al realizar la operación.

Acceso a las matrices

El acceso a los datos en objetos de tipo `array` es muy similar al de otros lenguajes como Matlab. Dependiendo de las dimensiones de una matriz, podremos especificar de forma opcional todas o algunas de sus dimensiones. El siguiente ejemplo ilustra algunos accesos a elementos de la matriz creada.

```
>>> a = arrayrange(9)
>>> a.shape = (3,3)
>>> print a
[[0 1 2]
[3 4 5]
[6 7 8]]
>>> print a[0] # Se obtiene la primera fila, no el primer elemento
[0 1 2]
>>> print a[1] # Se obtiene la segunda fila
[3 4 5]
>>> print a[1,0]
3
```

Por otro lado, en Python se pueden fraccionar las porciones de la matriz a la que se accede (*slicing*). El operador `[:]` es el utilizado para realizar este acceso fraccionado tal y como mostramos en el siguiente ejemplo.

```
>>> a = reshape(arrayrange(9),(3,3))
>>> print a
[[0 1 2]
[3 4 5]
[6 7 8]]
>>> print a[:,1]
[1 4 7]
```

Como podemos comprobar en este ejemplo, al hacer uso de `[:]` accedemos a todos los valores de esa dimensión. De este modo, en el caso de que `A` fuera una matriz de tres dimensiones el conjunto de datos accedidos sería el mismo en `A[1] == A[1,:] == A[1,:,:]`.

En definitiva, la distribución `Numpy` proporciona a Python un amplio conjunto de funcionalidades en el tratamiento de matrices de una manera intuitiva y eficiente. En esta introducción no se ha profundizado en exceso y se deja a voluntad del lector ampliar los conocimientos de la misma mediante su documentación oficial [4]. De este modo, se puede obtener más información de las rutinas denominadas `ufuncs`, los pseudo-indices, diversas funciones aplicables a las matrices y de los métodos de las matrices `array` que no se han descrito en el presente documento.

Creación de una extensión con Numpy

Como se ha comprobado en las secciones anteriores, la distribución `Numpy` proporciona un manejo eficiente y cómodo a un elevado volumen de datos distribuidos en matrices. Siendo la distribución `Numpy` el estándar de hecho en el tratamiento de matrices en Python, lo más lógico será que la distribución PyACTS haga uso de `Numpy` en lo referente al manejo de matrices. En definitiva, los objetos contenedores de los datos que usaremos en la distribución PyACTS serán de tipo `array`. Por tanto debemos conocer cual es el mecanismo por el que podremos crear las interfaces adecuadas entre los objetos `array` en Python y los datos en memoria que serán pasados a las rutinas incluidas en la colección ACTS.

Dicho de otro modo, tenemos la necesidad de que el paquete PyACTS trabaje con datos de tipo `array`. El tipo `array` de `NumPy` tiene la ventaja de usar la misma disposición de datos que se utiliza en C y Fortran. En esta sección describiremos cómo preparar nuestra extensión para trabajar con matrices `NumPy`.

Como primer paso, el archivo de extensión debe contener una línea donde se incluya al archivo de definiciones `arrayobject.h`, a continuación de la línea donde se incluye el archivo `Python.h`. El archivo `arrayobject.h` está incluido en la distribución de `Numpy` y dependiendo de dónde esté instalado deberemos indicárselo al compilador mediante los parámetros adecuados tal y como vimos en el apartado 3.1.7. Además de incluir este archivo, el archivo de extensión deberá llamar a la función de inicialización `import_array()` después de haber llamado a la función `Py_InitModule()`. Esta llamada comprueba que el tipo `array` ha sido importado.

Las matrices `array` de `Numpy` están definidas mediante la estructura `PyArrayObject`

la cual está caracterizada por los siguientes elementos:

- `int nd`.
  Indica el número de dimensiones de la matriz
- `int *dimensions`
  Puntero a un *array* de `nd` dimensiones que describe el número de elementos en cada dimensión distribuidos de la siguiente forma:
  `a.shape==(dimensions[0], dimensions[1], ..., dimensions[nd-1])`
- `int *strides`
  Puntero a una matriz de `nd` enteros en el que se indica el desplazamiento entre dos elementos sucesivos en cada dimensión. Esta propiedad es especialmente útil cuando se trabaja con matrices no continuas en memoria.
- `char *data`
  Puntero al primer elemento del `array`.

La dirección de un elemento puede ser calculada a partir de sus índices y de la información proporcionada por `strides`. Por ejemplo, el elemento `[i, j]` de una matriz de dos dimensiones se encuentra en la dirección `data + i*array->strides[0] + j*array->strides[1]`. Se ha de tener en cuenta que el desplazamiento (`strides`) está expresado en bytes y no en unidades de almacenamiento. La ventaja principal en este tipo de tratamiento es que no necesita conocer el tipo de datos que contiene la matriz.

En lo referente a los tipos de datos, éstos se encuentran definidos en `arrayobject.h` mediante un valor constante y se representan en la tabla 3.1.

Se ha de tener en cuenta que los tipos expresados en la tabla 3.1 son tipos de C, y no tipos de Python. Por ejemplo, un entero en Python se corresponde con una variable de tipo doble en C.

Un detalle importante en las matrices `Numpy` es la continuidad de las mismas. En `Numpy` se definen como matrices continuas aquellas que ocupan un sólo bloque en memoria y mantienen el mismo orden que una matriz en C. Este detalle es muy importante en la obtención de buenos tiempos del paquete `PyACTS` puesto que las matrices utilizadas deben ser continuas. Aquellas matrices que han sido creadas desde funciones específicas incluidas en `Numpy` son siempre continuas, pero matrices resultantes del indexado desde otras no lo son. La principal ventaja de las matrices continuas es la facilidad de manejo en C,

Constante	Tipo de dato del elemento
PyArray_CHAR	char
PyArray_UBYTE	unsigned char
PyArray_SBYTE	signed char
PyArray_SHORT	short
PyArray_INT	int
PyArray_LONG	long
PyArray_FLOAT	float
PyArray_DOUBLE	double
PyArray_CFLOAT	float[2]
PyArray_CDOUBLE	double[2]
PyArray_OBJECT	PyObject *

Tabla 3.1: Relación de tipos de datos

puesto que el puntero a los datos de la matriz se utiliza del mismo modo que en C y los valores de `stride` no son necesarios. De este modo, antes de llamar a una función de C o FORTRAN (como las incluidas en la colección ACTS) se ha de comprobar la continuidad de sus datos mediante la función de conversión `PyArray_ContiguousFromObject()`.

A continuación se muestra un ejemplo básico en el que integraremos las librerías `Numpy` en nuestro módulo desarrollado en C. En este ejemplo, se calcula la suma de los elementos de la diagonal en una matriz de dos dimensiones.

```
static PyObject *
trace(PyObject *self, PyObject *args)
{
    PyArrayObject *array;
    double sum;
    int i, n;
if (!PyArg_ParseTuple(args, "O!",&PyArray_Type, &array))
        return NULL;
if (array->nd != 2 || array->descr->type_num != PyArray_DOUBLE)
    {
    PyErr_SetString(PyExc_ValueError,
                "array must be two-dimensional and of type float");
    return NULL;
```

```
    }
    n = array->dimensions[0];
    if (n > array->dimensions[1])
        n = array->dimensions[1];
    sum = 0.;
    for (i = 0; i < n; i++)
        sum += *(double *)(array->data + i*array->strides[0]
                + i*array->strides[1]);
return PyFloat_FromDouble(sum);
}
```

El ejemplo en este caso requiere que la matriz sea de tipo doble, sin embargo `Numpy` ofrece funciones que aceptan secuencias de objetos arbitrarios como entrada y devuelven una matriz equivalente con el tipo indicado.

```
PyObject * PyArray_ContiguousFromObject(PyObject *object,
    int type_num,
    int min_dimensions,
    int max_dimensions);
```

En esta función, el primer argumento se corresponde con la secuencia de la cual queremos obtener los datos. En el segundo (`type_num`) indicaremos el tipo de la matriz resultante. Los dos últimos argumentos permiten especificar el numero de dimensiones de la matriz resultante. De este modo, mediante esta función se puede garantizar que las matrices utilizadas en los cálculos de los módulos en C son contiguas puesto que si no lo eran las convierte y si ya lo eran no se produce una perdida considerable al llamar a esta rutina.

Utilizando esta rutina, el ejemplo mostrado anteriormente quedará del siguiente modo:

```
static PyObject *
trace(PyObject *self, PyObject *args)
{
    PyObject *input;
    PyArrayObject *array;
    double sum;
```

```
    int i, n;
    if (!PyArg_ParseTuple(args, "O", &input))
        return NULL;
    array = (PyArrayObject *)
    PyArray_ContiguousFromObject(input, PyArray_DOUBLE, 2, 2);
    if (array == NULL)
        return NULL;
    n = array->dimensions[0];
    if (n > array->dimensions[1])
        n = array->dimensions[1];
    sum = 0.;
    for (i = 0; i < n; i++)
    sum += *(double *)(array->data + i*array->strides[0]
            + i*array->strides[1]);
    Py_DECREF(array);
return PyFloat_FromDouble(sum);
}
```

Hasta el momento, los datos de las matrices utilizadas en los módulos en C provenían de niveles superiores, es decir, desde Python. Sin embargo, puede ser interesante crear datos desde C para devolverlos a Python. Esto implica que `Numpy` debe proporcionar herramientas suficientes desde C para crear este tipo de objetos. Con este fin se ha creado la siguiente función:

```
PyObject * PyArray_FromDims(int n_dimensions,
        int dimensions[n_dimensions],
        int type_num);
```

En su primer argumento se indica el número de dimensiones, en el segundo la longitud de cada una de ellas y en el tercero el tipo de datos (ver tabla 3.1).

Finalmente, para devolver las matrices desde C hacia niveles superiores (Python en este caso), se incluyen facilidades en la distribución `Numpy` como la función `PyObject * PyArray_Return(PyArrayObject *array)` que devuelve una matriz con una o mas dimensiones o en su caso un escalar con el formato adecuado.

3.2.2. ScientificPython

`ScientificPython` [70] es una colección de módulos de Python muy útiles en la computación científica. La mayoría de sus módulos realizan tareas comunes en muchas áreas científicas, otros pertenecen a dominios específicos y por tanto únicamente tienen interés en ese campo específico de la investigación. La mayor parte de los módulos hacen uso del paquete `Numpy` (visto en el apartado 3.2.1).

En este apartado se describen brevemente los módulos que componen la distribución `ScientificPython` resaltando aquellos que sean utilizados en el paquete PyACTS. Los módulos que forman parte de la distribución son los siguientes:

- `BSP`: Contiene constructores para implementar la paralelización de cálculos a través del modelo síncrono paralelo BSP detallado en las referencias [109] y [10]. Las paralelizaciones requieren la utilización de librerías de bajo nivel como BSP o MPI [26]. La distribución PyACTS se ha basado en la utilización de MPI por su amplia estandarización e implementación en múltiples plataformas.
- `DictWithDefault`: Las instancias de esta clase actúan de forma similar a los diccionarios de Python (ver apartado 3.1.2) excepto que además devuelven un valor por defecto si no existe valor asociado a la clave.
- `Functions`: Este módulo incorpora los siguientes submódulos.
  - `Derivatives`: Proporciona diferenciación automática para funciones con cualquier número de variables y hasta cualquier orden. Una instancia de la clase `DerivVar` representa el valor de una función y los valores de sus derivadas parciales con respecto a una lista de variables.
  - `FindRoot`: Encuentra la raíz de una función comprendida entre dos valores `[lox,hix]`. Este módulo está basado en el algoritmo de Newton-Raphson [74].
  - `Interpolation`: Una función de interpolacion de n variables con m dimensiones está definida por una matriz de valores $m + n$ dimensional y n vectores que definen los valores correspondientes a los puntos de la malla. A partir de estos valores, se proporciona la función que los interpola.
  - `LeastSquares`: Obtiene las raíces no lineales utilizando el algoritmo descrito en [77] y las derivadas automáticas.

- `Polynomial`: Las instancias de esta clase representan polinomios de cualquier orden y cualquier número de variables.
- `Rational`: Instancias de esta clase representan funciones de una variable que pueden ser evaluadas como tales.
- `Romberg`: Devuelve la integral de una función de una variable sobre un intervalo utilizando para ello la descomposición por trapezoides.

- `Geometry`: Este subpaquete contiene clases que permiten el manejo de objetos y conceptos geométricos. Los conceptos geométricos son vectores y tensores, transformaciones y rotaciones. Por otro lado, incorpora clases con objetos geométricos elementales como esferas y planos. Este subpaquete incorpora los siguientes módulos.

  - `Objects3D`: Proporciona las clases de objetos geométricos en 3D básicos.
  - `Quaternion`: Incorpora el uso de *quaternion* [62]
  - `TensorAnalysis`: Permite el análisis de campos tensores y vectores con sus correspondientes derivadas.
  - `Transformation`: Permite realizar transformaciones lineales en el espacio de las tres dimensiones.

- `IO`: En este subpaquete se incluyen utilidades para el manejo de la entrada/salida de datos en las aplicaciones de Python.

  - `ArrayIO`: Contiene funciones para la entrada/salida de matrices numéricas en una o dos dimensiones desde ficheros de texto.
  - `FortranFormat`: Proporciona rutinas para la lectura de ficheros de textos formateados por Fortran.
  - `NetCDF`: Permite la entrada/salida de datos mediante el estándar NetCDF [91]. Éste formato y esta herramienta la usaremos en la tercera parte del trabajo (apartado 9.2) como referencia en la funcionalidad de las rutinas a implementar.
  - `PDB` : Este módulo proporciona diversas clases que representan los archivos del Banco de Datos de Proteínas (PDB).
  - `TextFile` : Contiene función de lectura y escritura sobre texto pero permitiendo que éstos estén comprimidos en diversos formatos como `.gz` *(gzip/gunzip)* y `.bz2` *(bzip2)*.

- `MPI`: Este submódulo permite implementar el paso de mensajes mediante el estándar MPI, creando un comunicador con el submódulo `Scientific.MPI.world`. A partir de este submódulo y de la herramienta PyMPI que veremos en el apartado 3.2.3, se define el entorno en el que el paquete PyACTS va a ser ejecutado. En el capítulo 5.4 se describe el intérprete en paralelo de Python que se distribuye junto con el paquete PyACTS y que está basado en este submódulo de `ScientificPython`.
- `NumberDict`: Una instancia de esta clase actúa como una matriz de números con índices generales (no necesariamente números).
- `Physics`:: Incorpora utilidades científicas en el ámbito de la física a traves de dos submódulos.
  - `PhysicalQuantities`
  - `Potential`
- `Signals`: Proporciona modelos de señales para su análisis en procesos estocásticos.
- `Statistics`: Ofrece las principales funciones útiles en cálculos estadísticos.
- `Threading`: Proporciona una forma sencilla de organizar las tareas en los sistemas multiprocesadores con memoria compartida.
- `TkWidgets`: Proporciona acceso a la librería gráfica Tk.
- `Visualization`:Este subpaquete proporciona diversas herramientas para la visualización de objetos 3D a través de herramientas como VRML, VMD, VPython.
- `Indexing`: Proporciona las herramientas adecuadas para crear algoritmicamente los índices de las matrices.

3.2.3. pyMPI

Hasta el momento se han introducido módulos y paquetes necesarios y útiles para la obtención de PyACTS, sin embargo se ha de mencionar que el intérprete de Python trabaja en modo monoprocesador de forma secuencial. Sin embargo, para poder conseguir una herramienta escalable, se necesita de un mecanismo que proporcione un entorno

multiproceso asíncrono. Esta facilidad la proporciona pyMPI [79] y se describe en este apartado junto con sus principales características. De este modo podemos decir que la extensión pyMPI está diseñada para proporcionar operaciones paralelas en Python en una arquitectura distribuida mediante MPI.

Además de proporcionar una interfaz completa desde Python a las funciones avanzadas de MPI y a los comunicadores, pyMPI también incluye una interfaz simplificada que consigue que la programación sea más sencilla. Una de las formas más sencillas de utilizar pyMPI es de forma interactiva. Una vez inicializado pyMPI en la arquitectura correspondiente (de la forma normal en cada sistema con mpirun, prun, poe, etc, ...), se obtiene la *shell* de comandos de entrada en Python (`>>>`). En este momento, se está ejecutando Python en modo multiprocesador tal y como se muestra a continuación.

```
% mpirun -np 3 pyMPI
>>> import mpi
>>> print 'Soy el proceso', mpi.rank,' de un total de ',mpi.size
Soy el proceso 0 de un total de 3
Soy el proceso 2 de un total de 3
Soy el proceso 1 de un total de 3
>>>
```

Se puede comprobar en este ejemplo cómo se lanzan tres procesos en ejecución asíncrona puesto que cada uno escribe su valor identificativo sin orden aparente. Los atributos `size` y `rank` del módulo importado `mpi` se corresponden con el número de procesos ejecutándose y con el identificador de cada uno de ellos, respectivamente.

La extensión pyMPI incorpora las funcionalidades de MPI a través de unas interfaces cómodas. En este sentido, podemos utilizar operaciones de sincronización mediante las rutinas `mpi.barrier` y `mpi.bcast`. Si en lugar de realizar una difusión, se desea realizar una recolección, la rutina `y=mpi.reduce(x,OPERACION)` permite realizar una recolección de los datos en función de una determinada `OPERACION` que puede ser de varios tipos y que mostramos en la tabla 3.2.

Por otro lado, se proporcionan operaciones de intercambio de mensajes punto a punto, pudiendo ser bloqueantes o no bloqueantes. Si se desea enviar información utilizaremos la rutina `send` indicando el valor y el destinatario (`mpi.send(msg,3)`). Si deseamos recibir un valor, éste se obtendrá de la llamada a la función de recepción (`msg, status =`

Clave	**Operación**	**Ejemplo**
BAND	AND Booleano	`y = mpi.reduce(x,mpi.BAND)`
BOR	OR Booleano	`y = mpi.reduce(x,mpi.BOR)`
BXOR	XOR Booleano	`y = mpi.reduce(x,mpi.LXOR)`
LAND	AND Lógico	`y = mpi.reduce(x,mpi.LAND)`
LOR	OR Lógico	`y = mpi.reduce(x,mpi.LOR)`
LXOR	XOR Lógico	`y = mpi.reduce(x,mpi.LXOR)`
MAX	Máximo	`y = mpi.reduce(x,mpi.MAX)`
MAXLOC	Primer máximo	`y,rank = mpi.reduce(x,mpi.MAXLOC)`
MIN	Mínimo	`y = mpi.reduce(x,mpi.MIN)`
MINLOC	Primer mínimo	`y,rank = mpi.reduce(x,mpi.MINLOC)`
PROD	Producto	`y = mpi.reduce(x,mpi.PROD)`
SUM	Suma	`y = mpi.reduce(x,mpi.SUM)`

Tabla 3.2: Operaciones de recolección en pyMPI

`mpi.recv()`). El valor de `status` hace referencia a la salida de dicha función y su valor de retorno es 0 si la operación se ha realizado con éxito.

En resumen, podemos concluir que pyMPI es una extensión de Python que nos permite implementar la computación paralela en un sistema distribuido mediante la librería de paso de mensajes MPI. Se ha de tener en cuenta que la extensión PyACTS no utiliza directamente las funciones vistas en este apartado y únicamente utilizaremos pyMPI para inicializar y ejecutar el entorno MPI y el intérprete de Python en los distintos procesos (ejecutando, por ejemplo, `mpirun -np 4 pyMPI ejemploacts.py`).

Concluyendo este apartado, se desea destacar que pyMPI es la extensión con estas funcionalidades más utilizada en la programación con Python. Sin embargo el proceso de instalación es ciertamente complejo y proporciona las interfaces a MPI a un nivel demasiado bajo. Por estas razones se ha implementado un intérprete de Python que inicializa MPI en distintos procesos a partir del módulo `Scientific.MPI` descrito en el apartado 3.2.2. Este intérprete se encuentra incorporado en la distribución de PyACTS con el fin de facilitar la utilización del paquete PyACTS en una arquitectura distribuida.

3.3. Creación de extensiones y módulos

Tal y como se ha comentado en el apartado 3.1.5, a partir de un código ya escrito en C, es posible utilizar este código creando las interfaces adecuadas para acceder a las rutinas del código C desde un *script* de Python. El proceso de creación de estas interfaces es complejo, y representa una tarea larga y ciertamente repetitiva cuando el número de rutinas incluidas en el código C es elevado. En el presente trabajo, el elevado numero de rutinas incluidas en la colección ACTS para las que se ha creado la interfaz (BLACS, PBLAS y ScaLAPACK), ha motivado la utilización de herramientas de automatización en la creación de estas interfaces.

En esta sección se presentan las dos principales herramientas que se han utilizado para crear las interfaces a las rutinas desde C. Por un lado, para poder acceder a las rutinas incluidas en las librerías BLACS, PBLAS y ScaLAPACK se ha utilizado F2PY que describiremos en el apartado 3.3.2. Para implementar las interfaces a la librería PnetCDF se ha utilizado la herramienta SWIG que será descrita a continuación.

3.3.1. SWIG

Se puede definir SWIG [7] como una herramienta sencilla para construir programas interactivos de C, C++ con lenguajes interpretados como Tcl, Perl y Python. SWIG ha sido desarrollado originalmente en la División de Física Teórica en el Laboratorio Nacional de los Álamos para construir interfaces a un conjunto elevado de simulaciones. En un entorno donde el volumen de los datos es elevado, la arquitectura de las máquinas es compleja y se producen cambios continuos en el código fuente, se hace necesario aislar los problemas que se desean resolver de toda la problemática asociada a las herramientas utilizadas para resolverlos.

En sus inicios, SWIG fue desarrollada para aplicaciones científicas, sin embargo se ha convertido en una herramienta de propósito general para multitud de aplicaciones en las que se desea acceder a rutinas de C desde lenguajes interpretados. De este modo, SWIG se ha creado como una herramienta de libre distribución en la que los usuarios han sugerido sus aportaciones obteniendo cada vez una herramienta más cómoda y precisa.

Se presenta en esta sección un breve resumen de la utilización de SWIG únicamente para Python puesto que el resto de lenguajes (Tcl y Perl) no han sido utilizados en el presente trabajo.

SWIG es un compilador que permite crear el conjunto de interfaces a los lenguajes de alto nivel a partir de un archivo de interfaces definido en el formato específico de SWIG. A partir de las declaraciones contenidas en este archivo de interfaces (`.i`), SWIG genera el código necesario para la interfaz y además documenta el código creado. El proceso funcional de SWIG queda descrito en la figura 3.1.

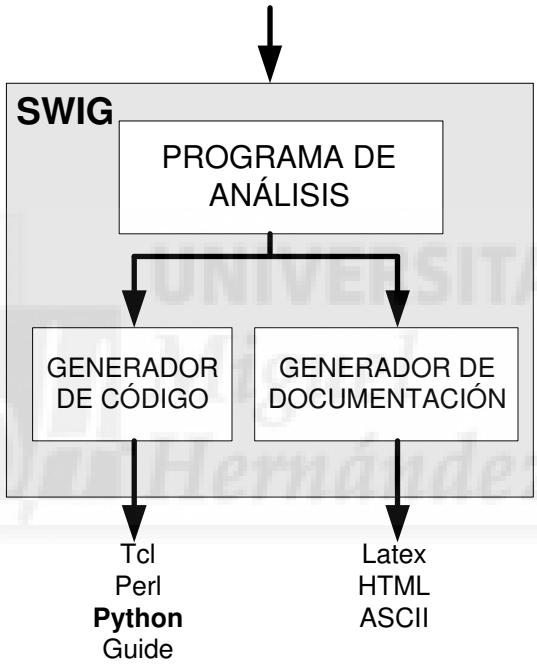


Figura 3.1: Funcionalidad de SWIG

Para ilustrar el proceso mediante el cual SWIG crea las interfaces adecuadas a una rutina de C, se muestra a continuación un ejemplo básico del proceso a seguir. A continuación se muestra el código C de las funciones a las que se desea crear la interfaz.

```
/* Fichero: ejemplo.c */
double Mivariable = 3.0;

/* Calcula el factorial de n*/
```

```
int fact(int n) {
   if (n <= 1) return 1;
   else return n*fact(n-1);
}

/* Calcula la división entera entre n y m*/
int mi_mod(int n, int m) {
return(n % m);
}
```

El primer paso que se debe realizar es crear el archivo de definición de interfaces (`.i`) que tendría el siguiente aspecto:

```
 /* File : ejemplo.i */
%module ejemplo
%{
/* Introduce cabeceras y declaraciones adicionales*/
%}
extern double Mivariable;
extern int fact(int);
extern int mi_mod(int n, int m);
```

Este archivo contiene las declaraciones en formato ANSI C de las funciones y variables a las que se desea crear la interfaz. La directiva `%module` define el nombre del módulo creado. El bloque `%{ ... %}` permite añadir codigo adicional en C al módulo creado.

Una vez escrito el archivo de definición de interfaces, generamos el módulo para Python mediante el siguiente comando:

```
unix > swig -python example.i
Generating wrappers for Python
unix > gcc -c ejemplo.c ejemplo_wrap.c -I/usr/local/include/python2.2
unix > ld -shared ejemplo.o ejemplo_wrap.o -o ejemplomodule.so
unix > python
Python 2.2.3 (#1, Feb 18 2004, 16:05:37)
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> import ejemplo
>>> ejemplo.fact(4)
24
>>> ejemplo.mi_mod(23,7)
2
>>> ejemplo.cvar.Mivariable + 4.5
7.5
```

Además de generar las interfaces para la importación en los lenguajes de alto nivel, SWIG también genera la documentación del mismo. Aplicando esta función al ejemplo mostrado tendríamos el siguiente resultado:

```
example_wrap.c
[ Module : example, Package : example ]

$My_variable
    [ Global : double My_variable ]

fact(n);
    [ returns int ]
my_mod(n,m);
    [ returns int ]
```

Como se ha visto en este ejemplo, el proceso de creación de un módulo para Python utilizando SWIG puede llegar a ser muy sencillo. Sin embargo, en el presente trabajo esta herramienta ha sido utilizada para generar una primera versión de las interfaces a la librería PnetCDF [20]. Por tanto, se han necesitado utilizar en mayor profundidad características de SWIG que no han sido descritas en este apartado (como son el tratamiento de punteros, la importación de objetos definidos en C a Python, etc) pero que serán analizados en mayor profundidad en el capítulo 9.4.

3.3.2. F2PY

La herramienta F2PY (*Fortran to Python*) [85] también ha sido utilizada en el presente trabajo. Como ya se ha comentado, F2PY ha sido utilizado para generar las interfaces a las rutinas incluidas en las librerías BLACS, PBLAS y ScaLAPACK. La razón por la que se ha utilizado F2PY en lugar de SWIG se debe a que F2PY esta especializado en generar interfaces a códigos desarrollados en Fortran (como describiremos más detenidamente a continuación).

Una apreciación personal del autor de este trabajo, es que F2PY es una herramienta más potente que SWIG a pesar de ser menos citada o utilizada por la comunidad científica. De hecho, la distribución oficial de Python incluye un módulo de SWIG y no ocurre lo mismo con F2PY. En esta sección realizaremos un breve repaso por las funcionalidades que incluye F2PY, para finalmente hacer una comparación con la herramienta SWIG. Se ha de recordar que ambas herramientas han sido utilizadas en el presente trabajo debido a las condiciones preliminares que los módulos a importar exigían.

Como se ha comentado ya, el objetivo principal de F2PY es proporcionar un generador de interfaces para poder importar los módulos desarrollados en el lenguaje Fortran al lenguaje interpretado Python. Además de Fortran, F2PY también puede ser utilizado para generar las interfaces a rutinas escritas en C (igual que SWIG), convirtiendo a F2PY en una herramienta muy versátil. F2PY es un paquete de Python que facilita la creación de extensiones que permitan realizar las siguientes tareas:

- Realizar llamadas a subrutinas de Fortran 77/90/95 del mismo modo que en C.
- Acceder a bloques de datos Fortran 77 COMMON y Fortran 90/95.

Tres formas de crear la interfaz

Para generar las interfaces a las subrutinas de Fortran mediante F2PY se han de realizar los siguientes pasos:

- Crear el archivo "de firmas" que contiene las descripciones de las interfaces a las funciones de Fortran o C, también llamado *firmas* de las funciones.

- Opcionalmente, los archivos de firmas creados pueden ser editados para optimizar las interfaces.
- F2PY lee un archivo de firmas y escribe el código fuente del módulo Python C/API a partir de las definiciones descritas en el apartado 3.1.5.
- F2PY compila todas las fuentes y genera el módulo conteniendo las rutinas en Fortran o C y las interfaces creadas. F2PY utiliza `scipy_distutils` [43] que proporciona los compiladores Fortran 77/90/95 para las plataformas Gnu, Intel, Sun Fortre, Absoft, NAG, Compaq, etc.

Dependiendo de las situaciones particulares, estos pasos pueden ser más o menos complejos e incluso ser omitidos o combinados con otros. Estos pasos se pueden llevar a cabo de tres formas diferentes, las cuales serán descritas a continuación utilizando el siguiente código como ejemplo para generar la interfaz para acceder desde Python.

```
C FILE: FIB1.F
      SUBROUTINE FIB(A,N)
C
C     CALCULA LOS PRIMEROS N NUMEROS DE FIBONACCI
C
      INTEGER N
      REAL*8 A(N)
      DO I=1,N
         IF (I.EQ.1) THEN
            A(I) = 0.0D0
         ELSEIF (I.EQ.2) THEN
            A(I) = 1.0D0
         ELSE
            A(I) = A(I-1) + A(I-2)
         ENDIF
      ENDDO
      END
C END FILE FIB1.F
```

La forma más rápida de generar la interfaz es ejecutando el siguiente comando:

```
f2py -c fib1.f -m fib1
```

Este comando construye la extensión `fib1.so` en el directorio actual mediante el parámetro `-c` utilizado. Podremos acceder a esta subrutina desde Python del siguiente modo:

```
>>> import Numeric
>>> import fib1
>>> print fib1.fib.__doc__
fib - Function signature:
  fib(a,[n])
Required arguments:
  a : input rank-1 array('d') with bounds (n)
Optional arguments:
  n := len(a) input int

>>> a=Numeric.zeros(8,'d')
>>> fib1.fib(a)
>>> print a
[  0.   1.   1.   2.   3.   5.   8.  13.]
```

Se ha de notar que F2PY detecta que el segundo argumento de la función (`n`) representa las dimensiones de la matriz `n`, por lo que será opcional ya que puede ser obtenido como una propiedad de la matriz `Numeric`. Además, se observa cómo se pasa como parámetro a la interfaz creada un objeto de tipo `Numeric` (visto en el apartado 3.2.1). De este modo, se observa que la herramienta F2PY integra `Numeric` de una forma sencilla. Este será un punto importante por el cual consideramos F2PY como una herramienta mejor que SWIG.

Cuando una matriz `Numeric` es contigua en Fortran (definiremos este concepto más adelante) y su tipo se corresponde con un tipo de Fortran, entonces el puntero C de la matriz es directamente pasado a Fortran. En otro caso, F2PY realiza una copia contigua (con el tipo adecuado) de la matriz de entrada y pasa el puntero de C de la copia a la función C. Como resultado, es posible que se realicen cambios en la matriz de entrada pero que no tienen efectos en el argumento de entrada. A continuación se muestra un ejemplo en el que debido a la diferencia en el tipo de dato (coma flotante o entero), se realiza una copia y el resultado no se almacena en argumento de entrada.

```
>>> a=Numeric.ones(8,'i')
>>> fib1.fib(a)
>>> print a
[1 1 1 1 1 1 1 1]
```

Este no era el resultado esperado puesto que en la función de Fortran se había definido la matriz `A` como `REAL*8`.

La utilización de `fib1.fib` en Python es muy similar a utilizar `FIB` en Fortran. Sin embargo, la utilización *in situ* de los argumentos de salida en Python indica un pobre estilo en la seguridad puesto que no se realizan comprobaciones de los argumentos. Cuando se utiliza Fortran o C, los compiladores descubren los errores de tipo durante la compilación, pero en Python los tipos han de ser comprobados en el tiempo de ejecución. Por tanto, si se utilizan argumentos *in situ* será más difícil realizar las comprobaciones necesarias antes de devolver la matriz como parámetro de salida.

Una forma mas elegante de crear las interfaces a Python desde las funciones de Fortran mediante F2PY se realiza creando un fichero de firmas (de forma similar a como se hacía en SWIG). F2PY puede crear un archivo de firmas ejecutando el siguiente comando:

```
f2py fib1.f -m fib2 -h fib1.pyf
```

El archivo de firmas se guarda como `fib1.pyf` (indicado por el flag `-h`) y su contenido será el siguiente:

```
python module fib2 ! in
    interface  ! in :fib2
        subroutine fib(a,n) ! in :fib2:fib1.f
            real*8 dimension(n) :: a
            integer optional,check(len(a)>=n),depend(a) :: n=len(a)
        end subroutine fib
    end interface
end python module fib2
```

Este archivo generado puede ser editado y modificado. Podemos indicar y modificar la definición en el archivo de firmas indicando que la matriz `a` sea únicamente un parámetro

de salida y que ésta se genere a partir del valor de `n`. De este modo, el archivo de firmas modificado será el siguiente:

```
python module fib2
    interface
        subroutine fib(a,n)
            real*8 dimension(n),intent(out),depend(n) :: a
            integer intent(in) :: n
        end subroutine fib
    end interface
end python module fib2
```

Finalmente, el módulo se construye mediante el comando:

```
f2py -c fib2.pyf fib1.f
```

La tercera forma de construir el módulo ha sido la utilizada en el trabajo PyACTS y la documentación de F2PY la califica como "rápida y elegante". En el caso que sea posible modificar los códigos en Fortran, la generación de un archivo de firmas puede ser opcional. La indicación de los atributos de F2PY (`intent(in)`,`intent(out)`,...) puede ser insertada directamente en el código fuente de Fortran utilizando las directivas F2PY. Las directivas F2PY son líneas de comentarios especiales (que comienzan por `Cf2py`) que son ignoradas por los compiladores Fortran pero interpretadas por F2PY.

A continuación mostramos la forma en la que se introducen las directivas F2PY dentro del código Fortran:

```
C FILE: FIB3.F
      SUBROUTINE FIB(A,N)
C     CALCULA LOS PRIMEROS N NUMEROS DE FIBONACCI
C
      INTEGER N
      REAL*8 A(N)
Cf2py intent(in) n
Cf2py intent(out) a
Cf2py depend(n) a
      DO I=1,N
```

```
         IF (I.EQ.1) THEN
            A(I) = 0.0D0
         ELSEIF (I.EQ.2) THEN
            A(I) = 1.0D0
         ELSE
            A(I) = A(I-1) + A(I-2)
         ENDIF
      ENDDO
      END
C END FILE FIB3.F
```

Mediante el siguiente comando se generará la extensión:

```
f2py -c -m fib3 fib3.f
```

De este modo podremos utilizar la extensión desde Python con una interfaz adaptada a nuestras necesidades:

```
>>> import fib3
>>> print fib3.fib.__doc__
fib - Function signature:
  a = fib(n)
Required arguments:
  n : input int
Return objects:
  a : rank-1 array('d') with bounds (n)
>>> print fib3.fib(8)
[  0.   1.   1.   2.   3.   5.   8.  13.]
```

El archivo de firmas

Las especificaciones de la sintaxis para los archivos de firmas (`.pyf`) están basadas en las de Fortran 90/95. F2PY introduce algunas especificaciones adicionales que ayudan en el

diseño de la interfaz entre Python y Fortran. Los archivos de firmas pueden contener código Fortran por lo que los archivos Fortran también pueden ser considerados "de firmas".

En general, los contenidos de los archivos de firmas son sensibles a las mayúsculas, por lo que se ha de ser muy cuidadoso utilizándolas y evitando las minúsculas si se decide utilizar las mayúsculas de forma continua.

Un archivo de firmas suele contener uno (es lo recomendado) o mas módulos de Python (definido con `python module`). Por tanto, el nombre del archivo en C que contiene las interfaces y que F2PY genera automáticamente será `<modulename>module.c`). Cada `python module` posee la siguiente estructura:

```
python module <modulename>
  [<usercode statement>]...
  [
  interface
    <usercode statement>
    <Fortran block data signatures>
    <Fortran/C routine signatures>
  end [interface]
  ]...
  [
  interface
    module <F90 modulename>
      [<F90 module data type declarations>]
      [<F90 module routine signatures>]
    end [module [<F90 modulename>]]
  end [interface]
  ]...
end [python module [<modulename>]]
```

Donde los corchetes (`[ ...]`) indican que es una parte opcional y los puntos suspensivos (`...`) que puede haber más repeticiones de esa parte de código. Se ha de tener en cuenta que este archivo de firmas corresponde con un archivo genérico para crear las interfaces a Fortran 77/90/95 y a C.

El archivo de firmas específico para crear las interfaces desde código en Fortran tendrá la siguiente estructura:

```
[<typespec>] function | subroutine <routine name> \
              [ ( [<arguments>] ) ] [ result ( <entityname> ) ]
  [<argument/variable type declarations>]
  [<argument/variable attribute statements>]
  [<use statements>]
  [<common block statements>]
  [<other statements>]
end [ function | subroutine [<routine name>] ]
```

A partir de este archivo de firmas, F2PY generará el archivo de extensión (que contiene las interfaces) con el siguiente contenido:

```
def <routine name>(<required arguments>[,<optional arguments>]):
     ...
     return <return variables>
```

Por otro lado, la "firma" de un bloque de datos tiene el siguiente aspecto:

```
block data [ <block data name> ]
  [<variable type declarations>]
  [<variable attribute statements>]
  [<use statements>]
  [<common block statements>]
  [<include statements>]
end [ block data [<block data name>] ]
```

Como se puede observar, en los dos últimos ejemplos se ha insertado una declaración de tipos (`variable type declarations`) que tendrá el siguiente aspecto:

```
<typespec> [ [<attrspec>] :: ] <entitydecl>
```

Mediante `typespec` indicaremos el tipo de la variable de entrada, este valor puede ser cualquiera entre `byte`, `character`, `complex`, `real`, `double complex`, `double precision`, `integer` y `logical`. El resto de atributos (`attrspec` y `entitydecl`) puede tomar diversos valores que no se indican en este trabajo y se referencian en el manual de F2PY [85].

En el manual referenciado anteriormente se indican los diferentes atributos que pueden tener las declaraciones en la "firma" mostrada anteriormente. Sin embargo, durante la creación de las interfaces a las rutinas de la colección ACTS se ha necesitado profundizar en los siguientes atributos de F2PY:

- `optional`
  El argumento se considera opcional. El valor por defecto puede ser especificado por `init_expr` en una expresión en C válida.

- `dimension(<arrayspec>)`
  La correspondiente variable se considera como un *array* de dimensiones `arrayspec`.

- `intent(<intentspec>)`
  Indica la "intención" del correspondiente argumento donde `intentspec` se corresponde con los siguientes valores separados por comas:

  - `in`
    Se considera el argumento únicamente como entrada. Esto implica que el valor del argumento es pasado a la función Fortran/C y no se espera que la función modifique el valor.
  - `inout`
    Se considera el argumento como de entrada/salida o un argumento de salida in situ. Las variables calificadas como argumentos `inout` únicamente pueden ser matrices de tipo `Numeric` "continuas" tanto en Fortran como en C. El tema de la continuidad de las matrices `Numeric` fue tratado en el apartado 3.2.1. No se recomenda utilizar `inout` sino utilizar los atributos `intent(in,out)`, `intent(inplace)`.
  - `inplace`
    El argumento se considera como un argumento de entrada/salida o *in situ*. Los argumentos de este tipo han de ser matrices de tipo `Numeric` continuas con el tamaño adecuado. En el caso que las matrices no fueran continuas se debería indicar otro atributo o bien convertir la matriz a continua. La utilización de `intent(inplace)` no se recomienda en su uso general, puesto que no es válida la utilización en el caso de listas, o *tuples*.
  - `out`
    El argumento se considera como una variable de salida y la añade a la lista

`<returnedvariables>`. La utilización de este atributo establece la variable como oculta en la entrada `intent(hide)` a menos que se indique que también es un argumento de entrada con `intent(in)`.

- `hide`
  Se extrae el argumento de la lista de argumentos requeridos y de la de opcionales. Se suele utilizar `intent(hide)` cuando también se utiliza sólo `intent(out)` o cuando `<init_expr>` determina completamente el valor del argumento como en el siguiente ejemplo:

  ```
  integer intent(hide),depend(a) :: n = len(a)
  real intent(in),dimension(n) :: a
  ```

- `c`
  El argumento se trata como un escalar o una matriz de C. No es necesario utilizar este atributo para matrices unidimensionales, no importa si la función a la que accedemos es de Fortran o C puesto que para matrices unidimensionales el aspecto de las matrices es el mismo.
- `cache`
  El argumento se trata como una parte desechable de memoria donde no se realizan comprobaciones de continuidad en Fortran o C.
- `copy`
  Asegura que los contenidos de `intent(in)` se conserven copiando los datos en una matriz temporal.
- `overwrite`
  Los contenidos de `intent(in)` pueden ser alterados por la función Fortran/C.
- `out=<new name>`
  Reemplaza el nombre del argumento de retorno en la cadena `__doc__`
- `callback`
  Este atributo nos permite pasar una función de Python como parámetro para ser utilizada por una función de Fortran.
- `aux`
  Define una variable auxiliar en C en la interfaz F2PY generada. Este tipo de variables es muy útil para salvar valores de parámetros y utilizarlos en las funciones de inicialización de otras variables. Queremos hacer notar que `intent(aux)` implica `intent(c)`.

Con respecto a este conjunto de atributos descrito, es importante resaltar las siguientes reglas:

- Si no se especifica `intent(in |inout |out |hide)` se asume `intent(in)`.
- `intent(in,inout)` es similar a `intent(in)`.
- `intent(in,hide)` o `intent(inout,hide)` es similar a `intent(hide)`.
- `intent(out)` es similar a `intent(out,hide)` a menos que se especifique `intent(in)` o `intent(inout)`.
- Si se indica `intent(copy)` o `intent(overwrite)`, se introduce un argumento opcional llamado `overwrite_<argument name>` y un valor por defecto a 0 o 1, respectivamente.
- `intent(inout,inplace)` es similar a `intent(inplace)`.
- `intent(in,inplace)` es similar a `intent(inplace)`.

- `check([<C-booleanexpr>])`
  Realiza una comprobación de consistencia con los argumentos evaluando la expresión `<C-booleanexpr>`. Si `<C-booleanexpr>` devuelve 0, se genera una excepción. Si no se utiliza `check(..)`, F2PY genera unas comprobaciones estándar automáticamente (por ejemplo, en el caso de una matriz comprueba el tamaño y dimensiones). Si deseamos deshabilitar cualquier tipo de comprobaciones, hemos de utilizar `check()`

- `depend([<names>])`
  Indica que el correspondiente argumento depende del valor de otras variables de la lista `<names>`.

- `external`
  El correspondiente argumento es una función proporcionada por el usuario.

Las directivas utilizadas en el archivo de firma F2PY son similares a las del código fuente en Fortran 77/90. De este modo, se puede evitar crear un archivo de firmas intermedio e indicar los atributos directamente sobre el código fuente. De hecho, éste ha sido el mecanismo utilizado para generar las interfaces a BLACS, PBLAS y ScaLAPACK, es decir, se han modificado los archivos fuentes para generar un archivo de firmas completo (`.pyf`).

La incorporación de una directiva dentro del código Fortran 77/90 se realiza del siguiente modo:

```
<comment char>f2py ...
```

En este caso, <`comment char` >`f2py` se corresponde con los caracteres que indican comentarios en Fortran, tales como "c","C","*","!","#". Lo que sigue en esta cadena es ignorado por el compilador de Fortran pero interpretado por F2PY.

Como se ha podido comprobar en ejemplos anteriores, en el archivo de firmas se utiliza el lenguaje C para realizar comprobaciones en los siguientes bloques:

- <`init_expr`> para la inicialización de variables.
- <`C-booleanexpr`> para el atributo de comprobación.
- <`arrayspec`> para el atributo de dimensiones.

Estos bloques pueden contener expresiones en C que hagan uso de algunas funciones C estándar (como `rank`, `shape`, `size`,...) o bien funciones definidas en archivos de inclusión como `math.h` y `Python.h`.

Tratamiento de variables

Las funciones Fortran que se han importado a Python, tienen un atributo `_cpointer` que contiene la referencia al puntero C de la correspondiente función.

En el caso de argumentos escalares típicos de Python (`integer, float, complex number`), se realiza una adaptación de tipos en los que es posible una pérdida de información, por ejemplo al pasar de coma flotante a entero. En este caso, F2PY no genera ninguna excepción.

Por otro lado, las interfaces generadas para formatos de matrices utilizan funciones que convierten las secuencias en objetos de tipo `Numeric`. Una excepción a esta conversión se produce en el caso de especificar la propiedad `intent(inout)`, en este caso el argumento ha de ser continuo y tener el tipo adecuado.

Generalmente, las matrices de tipo `Numeric` utilizadas como argumentos en las interfaces F2PY son pasadas directamente a las funciones Fortran/C. Este será el modo de proceder en las interfaces creadas para BLACS, PBLAS y ScaLAPACK, puesto que los tiempos serán muy inferiores a tener que realizar una copia del mismo para proporcionar la continuidad de la matriz.

Existen dos tipos de continuidad en las matrices de tipo `Numeric`:

- Continuidad Fortran cuando el dato se almacena por columnas.
- Continuidad en C cuando el dato se almacena por filas.

En el caso de matrices de una dimensión, ambas continuidades coinciden. Por ejemplo, una matriz A de dimensiones 2×2 es Fortran-continua si sus elementos se almacenan en memoria con el orden `A[0,0] A[1,0] A[0,1] A[1,1]` y C-continua si se almacenan `A[0,0] A[0,1] A[1,0] A[1,1]`.

Para comprobar si una matriz es C-continua, se puede utilizar el método `.iscontiguous()` en un objeto `Numeric`. Por otro lado, para comprobar la continuidad en Fortran, en todos los módulos generados por F2PY se incluye la función `has_column_major_storage(<array>)` que es equivalente pero más eficiente que realizar `Numeric.transpose(<array>).iscontiguous()`.

Generalmente no es necesario preocuparse de cómo se almacena una matriz en memoria, puesto que las interfaces asumen uno u otro orden dependiendo si la función enlazada es C o Fortran y F2PY asigna el orden automáticamente. Sin embargo, se proporciona una función que copia la matriz en el orden adecuado sólo cuando sea absolutamente necesario (debido al coste de memoria y computacional que exige).

Para transformar matrices de entrada al orden de almacenamiento en Fortran (por columnas) antes de pasarlas a las rutinas Fortran, se puede utilizar la función proporcionada en todos los módulos generados por F2PY: `as_column_major_storage(<array>)`.

Utilización de F2PY

F2PY puede ser utilizado desde la linea de comandos (`f2py`) o bien como módulo de Python (`f2py2e`). Se describen a continuación los dos modos de funcionamiento y sus

principales características.

Mediante el comando `f2py` podremos utilizarlo, a su vez, de tres modos diferentes distinguidos por los parámetros `-c` y `-h`.

1. Si se desea analizar los códigos Fortran y generar el archivos de firmas, se ha de utilizar:

```
f2py -h <filename.pyf> <options> <fortran files>   \
  [[ only: <fortran functions>  : ]                \
   [ skip: <fortran functions>  : ]]...            \
  [<fortran files> ...]
```

   En este caso, cada fichero Fortran puede contener más de una rutina y no es necesario realizar la interfaz a todas las rutinas de Fortran. Si se desea excluir alguna puede ser indicado con `skip`. Por otro lado, si se indica un nombre de fichero de salida (por ejemplo, `filename.pyf`), las firmas se almacenarán en dicho archivo.

2. Para generar el archivo de definición del módulo de extensión:

```
f2py <options> <fortran files>      \
  [[ only: <fortran functions>  : ]    \
   [ skip: <fortran functions>  : ]]... \
  [<fortran files> ...]
```

   Mediante este comando, se genera el archivo `<modulename>module.c` en el directorio de trabajo donde `<modulename>` se corresponde con el nombre del módulo creado (por ejemplo, si se define al módulo como `<scalapack>`, el archivo se llamará `pyscalapackmodule.c`. Además de generarse el archivo anterior también se generará `<filename.pyf>` que podrá ser utilizado para generar de nuevo el módulo de extensión sin necesidad de procesar de nuevo los archivos Fortran.

3. Para construir el módulo de extensión:

```
f2py -c <options> <fortran files>       \
  [[ only: <fortran functions>  : ]     \
   [ skip: <fortran functions>  : ]]... \
  [ <fortran/c source files> ] [ <.o, .a, .so files> ]
```

Esta forma es la más directa y rápida y realiza todo el proceso a partir de un único comando. Si los archivos Fortran contienen las directivas de F2PY, se generará el archivo de definición de las interfaces a Python `<modulename>module.c` que será compilado junto con todas las rutinas C y Fortran. Finalmente, todos los objetos y las librerías se enlazan en un archivo de extensión `<modulename>.so` que se almacena en el directorio de trabajo.

Esta ha sido la forma más utilizada en el trabajo que presentamos. Se han introducido las directivas F2PY en cada archivo Fortran de definición de las funciones a las que deseamos crear la interfaz. A continuación se realiza el proceso de construir el módulo, que genera automáticamente el archivo de firmas total (`.pyf`) y el archivo de definición de las interfaces (`<modulename>module.c`). Además, en este paso se admite la utilización de un conjunto de parámetros que han resultado muy útiles en el trabajo presentado. Algunos de estos parámetros son los siguientes:

- `--f77exec=<path>` y `--f90exec=<path>`: Indicamos la ubicación del compilador a utilizar.
- `--f77flags=<string>` y `--f90flags=<string>`: Especificamos los *flags* a utilizar en el compilador Fortran 77/90.
- `--arch=<string>`, `--noarch`: Opciones de optimización a partir de la arquitectura del sistema.
- `-l<libname>`: Utiliza la librería `<libname>` en el enlazado de los objetos.
- `-D<macro>[=<defn=1>]`: Define la macro `<macro>` como `<defn>`.
- `-I<dir>`: Añade el directorio `<dir>` al listado de directorios que contienen archivos de inclusión.
- `-L<dir>`: Añade el directorio `<dir>` al listado de directorios que contienen archivos de librerías.

Estas opciones han sido de mucha utilidad en la realización de las librerías PyACTS cuando se han utilizado compiladores distintos a `gcc`. Para la creación de las librerías PyACTS se ha debido tener en cuenta las macros que establecen los nombres de los símbolos en las librerías y si se añade o no el carácter "_" a cada símbolo. En este caso, hemos tenido que utilizar las siguientes macros:

```
-DPREPEND_FORTRAN
```

`-DNO_APPEND_FORTRAN`
`-DUPPERCASE_FORTRAN`

Además, en la fase de prueba del rendimiento de las librerías, resulta interesante conocer el funcionamiento de las interfaces y cuál es la pérdida de rendimiento con respecto a la ejecución en un entorno puramente de Fortran. Para ello hemos utilizado la macro `-DF2PY_REPORT_ATEXIT`; los resultados serán expuestos en los capítulos finales del presente trabajo.

En los tres modos de utilización de `f2py` es posible configurar el proceso mediante parámetros adicionales:

- `<modulename>`
  Establece el nombre de la extensión. Su valor por defecto es `untitled`
- `--[no-]lower`
  Convierte a minusculas (o no), los identificadores de los ficheros de Fortran. Por defecto, se asume `--lower`
- `--build-dir <dirname>`
  Todos los ficheros temporales y los resultantes de la ejecución de `f2py` serán creados en `<dirname>`.

Sin embargo, F2PY puede ser utilizado desde el intérprete de comandos de Python mediante `f2py2e`. En este apartado se comenta brevemente esta opción aunque no ha sido utilizada para el desarrollo presentado. A continuación se muestra un ejemplo de utilización de `f2py2e` similar a `f2py -m fib3 fib3.f` pero desde el intérprete de comandos de Python:

```
>>> import f2py2e
>>> r=f2py2e.run_main(['-m','scalar','docs/usersguide/scalar.f'])
Reading fortran codes...
        Reading file 'docs/usersguide/scalar.f'
Post-processing...
        Block: scalar
        Block: FOO
Building modules...
```

```
        Building module "scalar"...
        Wrote C/API module "scalar" to file "./scalarmodule.c"
>>> print r
{'scalar':
   {'h': ['/home/users/pearu/src_cvs/f2py2e/src/fortranobject.h'],
 'csrc': ['./scalarmodule.c',
   '/home/users/pearu/src_cvs/f2py2e/src/fortranobject.c']}
   }
```

Como se puede comprobar, mediante la llamada a la rutina `f2py2e.run_main` se realiza un proceso similar al descrito anteriormente, salvo la peculiaridad que no podremos compilar el archivo con esta orden y necesitaremos utilizar el correspondiente comando de Python:

```
compile(source, modulename='untitled', extra_args='', verbose=1 ,
          source_fn=None)
```

En definitiva, se ha tratado en este apartado de mostrar las principales características de una herramienta que nos ha permitido crear una interfaz (*wrapper*) adaptada y optimizada a cada una de las rutinas incluidas en las librerías BLACS, PBLAS, ScaLAPACK y PNetCDF siendo en total un número elevado. En el caso de realizar la edición manual de los archivos de extensión, el proyecto se hubiera considerado inabordable. Sin embargo, mediante la correcta utilización de esta herramienta y la definición adecuada en cada caso del archivo de firmas, podemos finalmente obtener una extensión que nos permite acceder a las librerías antes mencionadas desde el intérprete de Python.

Capítulo 4

Software de computación científica

Este capítulo se centra en el software de computación científica. En una primera introducción, se proporciona una visión general de este tipo de software y de las técnicas empleadas en la computación de altas prestaciones (*HPC, High Performance Computing*). En la sección 4.2 se describe la problemática específica del software de computación científica exponiendo brevemente las principales fuentes de complejidad asociadas. Se definen en esta sección, diversos criterios de calidad deseables en cualquier componente software de computación científica y se describen algunas soluciones propuestas para conseguir dichos objetivos. En la sección 4.3 se realiza un recorrido por el software disponible actualmente, detallando en el apartado 4.3.1, el software disponible para el paso de mensajes en plataformas de memoria distribuida. En el apartado 4.3.2, se presenta la colección ACTS y en apartados posteriores describiremos las librerías de dicha colección utilizadas este trabajo, tales como librerías para las comunicaciones en el álgebra lineal (apartado 4.3.3), para la implementación de operaciones relativas al álgebra lineal (apartado 4.3.4) y para la resolución de ecuaciones lineales y problemas de valores propios en matrices densas (apartado 4.3.5).

4.1. Introducción

Al hablar de computación científica nos referimos a la necesidad de computación, esto es, cálculos mediante un computador, que surge en cualquier disciplina de la ciencia o la ingeniería. Esta necesidad no es en absoluto trivial y en muchos casos conlleva gran dificultad, lo cual ha llevado a la aparición de otra disciplina en sí misma como es la Ciencia e Ingeniería Computacional. Esta rama abarca muy diversos aspectos y es inherentemente interdisciplinar ya que en ella confluyen otras muchas disciplinas como el análisis numérico, el análisis matemático, la física, la matemática, la geometría, la teoría de grafos y la meteorología entre otras muchas.

Desde la aparición de los primeros computadores, una de sus principales aplicaciones ha sido la realización de cálculos para la resolución de problemas matemáticos planteados por científicos e ingenieros. Prueba de ello es que el primer lenguaje de programación de alto nivel fuera Fortran, que apareció en los años 50 y fue diseñado específicamente para describir cálculos matemáticos de forma sencilla. Desde entonces, la computación científica ha sufrido continuos cambios hasta considerarse una disciplina en sí misma.

Generalmente, los científicos e ingenieros han debido conformarse con plantear problemas simplificados que pudieran ser resueltos con los computadores disponibles en cada momento. A medida que la potencia de cálculo ha ido aumentando, se han podido obtener soluciones cada vez más fiables y precisas. Actualmente, los computadores han alcanzado una capacidad suficiente para un amplio espectro de aplicaciones, sin embargo, otras aplicaciones siguen demandando más potencia de cálculo.

La computación científica está presente en muchos ámbitos y, por tanto, las aplicaciones pueden ser muy diferentes según sea el sistema físico que se estudie, ya sea la atmósfera terrestre, un circuito integrado o una simulación celular. En todos los casos, hay un elemento común que es la realización de un modelo del sistema físico, casi siempre formulado mediante ecuaciones matemáticas, que pueden ser ecuaciones diferenciales, ecuaciones integrales, ecuaciones matriciales o de otro tipo. En esta tesis nos centraremos en las aplicaciones que manejan matrices densas y cuyos datos han de ser tratados y analizados mediante un entorno cómodo por programadores que desconocen los fundamentos de la computación paralela. Bien puede ser el caso de aplicaciones meteorológicas, o geológicas donde el tamaño de los datos a tratar es elevado y esto motiva una resolución del problema mediante herramientas de alto rendimiento.

El término computación de altas prestaciones se refiere al conjunto de técnicas que intentan mejorar las prestaciones de las implementaciones de los algoritmos mediante enfoques que tienen en cuenta las características de la arquitectura de los computadores actuales. Básicamente incluye dos conceptos fundamentales, por un lado maximizar el aprovechamiento de las jerarquías de memoria, y por otro, utilizar la computación paralela. El primer concepto se basa en que los computadores actuales cuentan con una memoria formada al menos por dos niveles, cuyos tiempos de acceso difieren sustancialmente. Para obtener buenas prestaciones en un programa, los accesos realizados tanto a los datos como a las propias instrucciones del programa, han de localizarse en la medida de lo posible en los niveles superiores de la jerarquía. Para conseguirlo, se utilizan una serie de técnicas como son el desenrollado de bucles, el bloqueo de algoritmos, etc [22].

La computación paralela es un elemento cada vez más importante para las simulaciones numéricas de gran escala. Si bien la potencia de cálculo (secuencial) por unidad de coste se incrementa año a año, también crece la demanda por hacer cálculos más grandes y rápidos. La computación paralela permite realizar la misma computación científica en una fracción de tiempo de ejecución secuencial, y por tanto, obtiene la solución más rápidamente y posibilita la resolución de problemas de dimensión mayor, superando al mismo tiempo la limitación de la capacidad de la memoria.

En los últimos años, el hardware paralelo ha madurado mucho, las redes de interconexión son más rápidas y baratas, permitiendo construir clusters de alta capacidad con componentes de uso común que desafían la potencia de las máquinas paralelas de gama alta. Por tanto, hay disponibles gran cantidad de recursos de cálculo paralelo, los cuales han de ser explotados por software paralelo.

Las características y los tipos de plataformas paralelas ya fueron descritas en el capítulo 2. En este capítulo trataremos de describir el conjunto de herramientas software que nos permiten obtener una computación de altas prestaciones en nuestra arquitectura paralela.

4.2. Desarrollo de software

El creciente número de publicaciones dedicadas al software computacional de finalidad científica es un claro indicador de la importancia del problema. En esta sección se

describen, de forma breve, las características de las aplicaciones científicas y se enumeran las cualidades que debe tener una librería para ser considerada una herramienta adecuada en la computación científica.

4.2.1. Computación científica

El software para la simulación numérica de fenómenos físicos (por ejemplo, la herramienta PyClimate [98]) no se fabrica en serie sino que es desarrollado por los propios expertos del área, que son los que poseen el conocimiento del problema necesario para poder modelarlo. En el límite, el experto, diseñador de software, programador, testeador y usuario final son la misma persona. Esto es especialmente cierto cuando el propósito del software es validar algoritmos o métodos novedosos, o investigar fenómenos físicos no comprendidos completamente.

Por otro lado, los programadores de software científico normalmente son expertos en algoritmos numéricos o en modelado matemático pero no tienen formación informática o en ingeniería del software.

En algunos campos, los códigos computacionales son típicamente programas grandes (varios cientos de miles de líneas de código) desarrollados por grupos de investigadores durante un largo periodo de tiempo. Estos códigos y librerías constituyen una enorme inversión que difícilmente va a ser abandonada en favor de nuevas reimplementaciones con técnicas más modernas, a pesar de las ventajas que esto pudiera conllevar para su futuro desarrollo.

El software de computación científica presenta algunas particularidades respecto del software de otras áreas. Dos de los principales objetivos en el desarrollo del software computacional son la portabilidad y la eficiencia.

La portabilidad además de conveniente, es muchas veces necesaria ya que el ciclo de vida de los códigos científicos puede ser de varias décadas mientras que la duración de los computadores se reduce a años. Por tanto, ha de ser factible trasladar los códigos a las nuevas generaciones de computadores. La eficiencia, tanto en términos de memoria como de tiempo de ejecución, es un factor de calidad importante en las simulaciones numéricas, mucho más que en otro tipo de aplicaciones. El tamaño del problema se puede

ampliar tanto como se desee, y generalmente, esto lleva a una mejor aproximación a la solución. Desde el punto de vista práctico, una aplicación muy eficiente proporciona mayor funcionalidad, ya que es posible abordar problemas más grandes o más interesantes.

La noción de eficiencia en computación científica es diferente a la complejidad computacional utilizada habitualmente en informática para caracterizar los algoritmos, la cual se suele considerar únicamente en términos asintóticos. En computación científica, los detalles de implementación de bajo nivel pueden suponer diferencias en las prestaciones de un orden de magnitud o más.

Un factor relacionado con la eficiencia es el espacio de memoria necesario. Los programas de computación científica típicamente crean y procesan grandes cantidades de datos. La capacidad de memoria disponible puede ser un factor limitador para el tamaño de los problemas que se pueden tratar.

Otro aspecto problemático del software numérico es el testeo y la validación, ya que los errores suelen ser difíciles de detectar. En primer término, es necesario tener mucha experiencia para decidir si el resultado de un programa es erróneo o no, y en ese caso saber si se trata de un error de programación, de un parámetro mal elegido o un algoritmo inapropiado. Por otro lado, los programas típicamente manejan grandes cantidades de datos, lo cual puede hacer difícil detectar variables erróneas o funcionamientos incorrectos, sobre todo si el problema no se reproduce en tamaños de problema menores. En ese caso es necesario utilizar herramientas gráficas para visualizar los resultados.

4.2.2. Criterios de calidad

En este apartado se enumeran propiedades que miden la calidad de los componentes software desde la perspectiva específica de la computación científica. Desde el punto de vista del usuario, los siguientes aspectos son importantes en un componente software:

- **Correción**. Esta propiedad se refiere a la capacidad del componente de proporcionar un resultado correcto. En este apartado surgen cuestiones como qué garantía hay de que el resultado es correcto, para qué datos de entrada, cómo ha sido comprobada la corrección, etc. Aspectos relacionados son la precisión y la estabilidad numérica.

- **Robustez**. Un componente robusto es aquel que funciona siempre que los datos de entrada son válidos y detecta los casos en que los datos de entrada no son válidos. Cuantas más comprobaciones se hagan en los puntos de entrada del componente para detectar errores de uso, más robusto será el componente. Otro factor a tener en cuenta es el tratamiento que se hace de los errores, si simplemente se aborta la ejecución, o se da información detallada del error para que el usuario pueda corregirlo.

- **Eficiencia**. Como se ha comentado varias veces, la eficiencia es un factor muy importante y engloba diversos aspectos como la eficiencia de uso de memoria (uso de espacio de trabajo, liberación apropiada de la memoria) y las prestaciones en la ejecución (uso de algoritmos eficientes, optimización en la implementación). En el caso de componentes paralelos, se puede hablar también de eficiencia paralela y, en ese caso, una propiedad importante es la escalabilidad, es decir, que las prestaciones no se degraden sustancialmente si se incrementa tanto el tamaño del problema como el número de procesadores.

- **Modularidad y granularidad**. Estas propiedades tienen que ver con la estructura interna de los componentes que proporcionan funcionalidad de alto nivel, y si se refiere a la posibilidad de reutilizar partes de más bajo nivel.

4.3. Descripción del software disponible

En esta sección se incluyen las descripciones de las principales librerías de computación científica utilizadas en el presente trabajo. Todas ellas son de dominio público y están disponibles a través de Internet en las direcciones que indicaremos en cada una de ellas. Se presentan por tanto en esta sección, el conjunto de librerías que centran esta tesis, y que permiten el paso de mensajes, la realización de operaciones del álgebra lineal en entornos de alto rendimiento, la resolución de sistemas lineales y obtención de valores propios entre otras funciones.

4.3.1. Librerías de paso de mensajes

En la programación paralela, dos paradigmas han madurado y se han convertido en soluciones claras y válidas: la memoria compartida y el paso de mensajes. Particularmente, para memoria distribuida, el paso de mensajes es la técnica más popular en la implementación de aplicaciones paralelas. En este apartado, revisaremos cual es el estado de las dos principales librerías que implementan el intercambio de mensajes: PVM y MPI.

PVM: Paralel Virtual Machine

PVM [56] consiste en un software y un conjunto de librerías, que permiten establecer una colección de uno o mas sistemas de computación, con el fin de poder integrar dichos sistemas en un esquema de una sola máquina virtual (PVM: Parallel Virtual Machine - Máquina virtual en paralelo).

La cantidad de plataformas en las que se puede ejecutar PVM es muy amplia, por ejemplo PVM opera sobre diferentes plataformas de UNIX y también de Windows. PVM puede establecerse en cualquier esquema de red heterogénea, sobre todo en el ambiente de Internet y bajo el protocolo IP. PVM brinda rutinas en lenguajes C y Fortran para procesos relativas al paso de mensajes asíncronos y para el control de los procesos. El paralelismo ofrecido por PVM es escalable, además de otras muchas ventajas que explicaremos en este apartado.

A pesar de que PVM no es un estándar, es sumamente popular para realizar y desarrollar aplicaciones científicas complejas que requieren un esquema de programación en paralelo. Las principales ventajas de PVM son las siguientes:

- Portabilidad: Es probablemente la librería de paso de mensajes más portátil que existe.
- Paralelismo escalable: PVM permite definir cuantos procesadores puede utilizar una aplicación, en caso de que falten procesadores, PVM realiza el trabajo en menos procesadores, recurriendo a técnicas de procesamiento concurrente.
- Tolerancia a fallos.

- De fácil instalación y uso.
- Popular: Es una de las librerías de paso de mensajes más fáciles y óptimas.
- Flexible: Su flexibilidad se refleja en la facilidad de modificar y configurar o definir la máquina virtual. Además se realiza un control arbitrario de dependencia de estructuras, donde la aplicación decide dónde y cuándo producir o terminar las tareas, y qué máquinas se agregan o se eliminan de la máquina virtual en paralelo, o cuáles tareas se pueden comunicar y/o sincronizar con otras.

Sin embargo, posee determinadas características que se pueden considerar negativas en la computación paralela. Estas características son :

- Como PVM es un esquema heterogéneo de computadoras, dependiendo de la capacidad de procesamiento de las computadoras vinculadas al esquema de la máquina virtual, su desarrollo puede verse mermado o incrementado, según sea el poder de cómputo de las computadoras anfitrionas.
- Cuenta con un esquema no estandarizado, es decir, PVM no es un estándar (como lo es MPI).
- Es algo deficiente en cuanto al paso de mensajes se refiere.

Los componentes de PVM son el demonio PVM y las librerías PVM que se encargan de las siguientes tareas:

- Comunicaciones.
- Control de procesos.
- La interfaz de programación para el usuario.

El programa demonio PVM es un proceso de UNIX que revisa la operación de los procesos generados en una aplicación de PVM, y que coordina su ejecución y su comunicación. En la versión 3.0 de PVM, el demonio se inicializa con el comando `pvmd3`, el cual se ejecuta en la máquina destinada a tener el demonio maestro. Se desea destacar que solo se puede inicializar un demonio por máquina. La forma de inicializar el demonio de PVM es la siguiente:

```
% pvmd3 hostfile &
```

donde `hostfile` es el archivo que contiene la información de las computadoras anfitrionas que se incorporan al esquema de la máquina virtual. En cuanto se inicializa el demonio maestro de PVM, éste a su vez inicializa los demonios de todas las máquinas anfitrionas del esquema de la máquina virtual. Cada demonio mantiene una tabla de configuración y de información de procesos para la máquina virtual. Los demonios reciben los procesos de comunicación entre usuarios y mandan llamar las librerías locales de la máquina en donde se encuentran, por lo que el software de PVM forzosamente debe de encontrarse instalado en todas las máquinas que pertenezcan al esquema de la máquina virtual.

Por otro lado, las librerías de PVM son las siguientes:

- `libpvm3.a` : Esta librería brinda una gran cantidad de rutinas escritas en lenguaje C. Esta librería siempre es requerida.
- `libfpvm3.a`: Librería adicional que se requiere en caso de que la aplicación escrita en PVM utilice código Fortran.
- `libgpvm.a`: Librería requerida en caso de usar grupos dinámicos.

Las citadas librerías contienen subrutinas sencillas que son incluidas en el código de aplicación escrito en PVM. Proveen las herramientas suficientes como para realizar acciones tales como el iniciado y borrado de procesos, el empaquetamiento, desempaquetamiento y recepción de mensajes, sincronización vía barrera y configuración dinámica de la máquina virtual.

A pesar de la mayor difusión de MPI frente a PVM, en la totalidad de los sistemas con los que hemos trabajado en esta tesis, hemos querido comentar la existencia de PVM pero desde el conocimiento que MPI es el estándar más extendido para aplicaciones de memoria distribuida con paso de mensajes.

MPI: Message Passing Interface

El paso de mensajes es una tarea ampliamente usada en ciertas clases de máquinas paralelas, especialmente aquellas que cuentan con memoria distribuida. Aunque existen

muchas variaciones, el concepto básico en el proceso de comunicación mediante mensajes es bien entendido. En los ultimo 10 años, se ha logrado un progreso substancial en convertir aplicaciones significativas hacia este tipo de tareas. Más recientemente, diferentes sistemas han demostrado que un sistema de paso de mensajes puede ser implementado eficientemente y con un alto grado de portabilidad.

Al diseñarse MPI [48], se tomaron en cuenta las características más atractivas de los sistemas existentes para el paso de mensajes, en vez de seleccionar sólo uno de ellos y adoptarlo como el estándar. Resultando así, en una fuerte influencia para MPI los trabajos hechos por IBM, INTEL NX/2, Express, nCUBE's Vernex, p4 y PARMACS. Otras contribuciones importantes provienen de Zipcode, Chimp, PVM, Chameleon y PICL.

La meta de MPI o *Message Passing Interface* (interfaz de paso de mensajes), es el desarrollar un estándar (que sea ampliamente usado) para escribir programas que implementen el paso de mensajes. Por lo cual la interfaz intenta establecer para esto un estándar práctico, portátil, eficiente y flexible.

El esfuerzo para estandarizar MPI involucró cerca de 60 personas de 40 organizaciones diferentes principalmente de U.S.A. y Europa. La mayoría de los vendedores de computadoras concurrentes estaban involucrados con MPI, así como con investigadores de diferentes universidades, laboratorios del gobierno e industrias. El proceso de estandarización comenzó en el taller de estándares para el paso de mensajes en un ambiente con memoria distribuida, patrocinado por el Centro de Investigación en Computación Paralela en Williamsbur, Virginia (abril 29-30 de 1992). Se llegó a una propuesta preliminar conocida como MPI1, enfocada principalmente en comunicaciones punto a punto sin incluir rutinas para comunicación colectiva y no presentaba tareas seguras. El estándar final para MPI fue presentado en la conferencia de Supercomputación en noviembre de 1993, constituyéndose así el foro MPI.

En un ambiente de comunicación con memoria distribuida en el cual las rutinas de nivel mas alto y/o las abstracciones son construidas sobre rutinas de paso de mensajes de nivel bajo, los beneficios de la estandarización son muy notorios. La principal ventaja al establecer un estándar para el paso de mensajes es la portabilidad y su facilidad de uso.

MPI es un sistema complejo, que se ha enfocado hacia la consecución de los siguientes objetivos:

- Diseñar una interfaz de programación aplicable (no necesariamente para compiladores o sistemas que implementan una librería).

- Permitir una comunicación eficiente: Evitando el copiar de memoria a memoria y permitiendo (donde sea posible) la sobreposición de computación y comunicación, además de aligerar la comunicación con el procesador.
- Permitir implementaciones que puedan ser utilizadas en un ambiente heterogéneo.
- Permitir enlaces convenientes en C y Fortran 77 para la interfaz.
- Asumir una interfaz de comunicación segura: El usuario no debe lidiar con fallos de comunicación. Tales fallos son controlados por el subsistema de comunicación interior.
- Definir una interfaz que no sea muy diferente a las actuales, tales como PVM, NX, Express, p4, etc., y proveer de extensiones para permitir mayor flexibilidad.
- Definir una interfaz que pueda ser implementada en diferentes plataformas, sin cambios significativos en el software ni en las funciones internas de comunicación.
- La semántica de la interfaz debe ser independiente del lenguaje.
- La interfaz debe ser diseñada para producir tareas seguras.

En el modelo de programación MPI, un cómputo comprende de uno o más procesos comunicados a través de llamadas a rutinas de librería para mandar (`send`) y recibir (`receive`) mensajes entre procesos. En la mayoría de las implementaciones de MPI, se crea un conjunto fijo de procesos al inicializar el programa, y un proceso es creado por cada tarea. Sin embargo, estos procesos pueden ejecutar diferentes programas. De ahí que, el modelo de programación MPI es algunas veces denominado como MIMD para distinguirlo del modelo SIMD, en el cual cada procesador ejecuta el mismo programa.

Debido a que el número de procesos en un cómputo de MPI es normalmente fijo, se puede enfatizar en el uso de los mecanismos para comunicar datos entre procesos. Los procesos pueden utilizar operaciones de comunicación punto a punto para mandar mensajes de un proceso a otro, estas operaciones pueden ser usadas para implementar comunicaciones locales y no estructuradas. Un grupo de procesos puede llamar colectivamente operaciones de comunicación para realizar tareas globales tales como broadcast, etc. La habilidad de MPI para probar mensajes da como resultado el soportar comunicaciones asíncronas. Probablemente una de las características más importantes de MPI es el soporte para la programación modular. Un mecanismo llamado comunicador permite al programador de

MPI definir módulos que encapsulan estructuras internas de comunicación (estos módulos pueden ser combinados secuencialmente y paralelamente).

Aunque MPI es un sistema complejo y multifacético, podemos resolver un amplio rango de problemas usando seis de sus funciones, estas funciones inician un comunicador (`mpi_init`) y lo finalizan (`mpi_finalize`), identifican procesos (`mpi_comm_size, mpi_comm_rank`), además de enviar (`mpi_send`) y recibir mensajes (`mpi_receive`).

MPI también provee un conjunto especializado de funciones colectivas de comunicación que realizan operaciones globales mostradas en la figura 4.1: (1) `mpi_bcast`, (2) `mpi_scatter,` (3) `mpi_send / mpi_receive`, (4) `mpi_reduce_all`, y (5) `mpi_gather`.

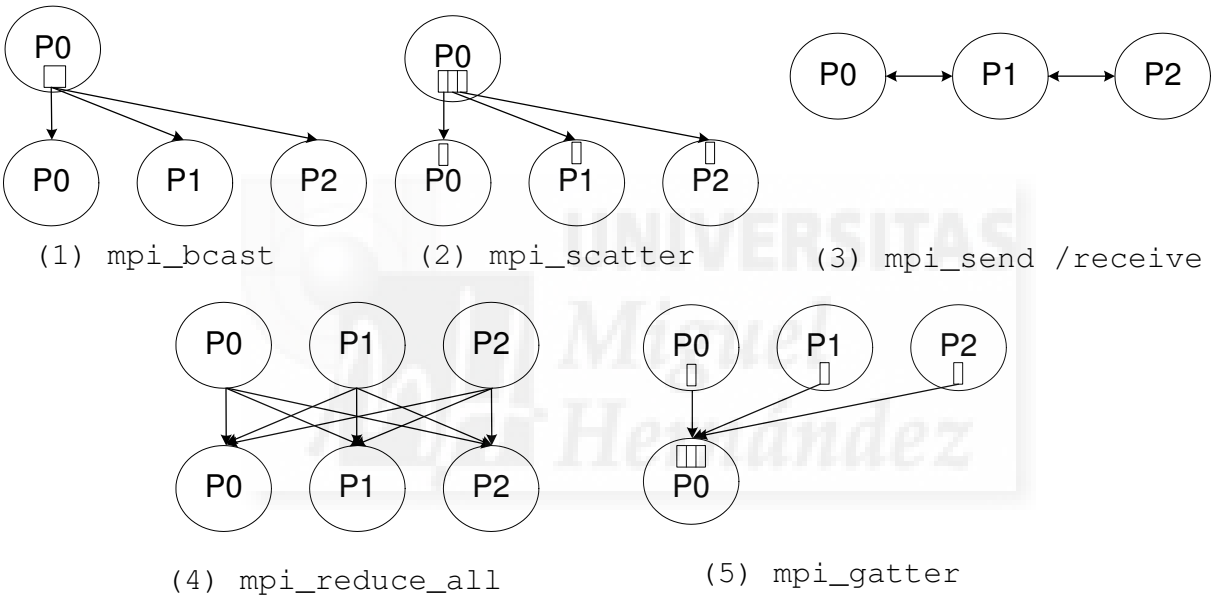


Figura 4.1: Operaciones Globales en MPI

4.3.2. La colección ACTS

En este apartado pretendemos introducir la colección ACTS [16] describiendo sus principales características y componentes. Podemos definir la colección ACTS como un conjunto de herramientas software muy útiles para los desarrolladores de programas en plataformas paralelas y difiere de otras herramientas para la computación paralela en el hecho que enfoca su funcionalidad en los niveles inferiores de una aplicación, proporcionando librerías que podrán ser utilizadas desde distintos lenguajes (C, Fortran 77/90, C++, etc.). La mayor parte de las herramientas han sido desarrolladas por universidades y laboratorios del Departamento de Energía de Estados Unidos [81].

La mayoría de las librerías incluidas en la colección ACTS están programadas en C (BLACS, PBLAS, Hypre, PETSc), otras en C++ (OPT++) e incluso en Fortran (ScaLAPACK). Estas librerías están diseñadas para su ejecución en procesadores paralelos utilizando el estándar MPI (ver apartado 4.3.1) para realizar la comunicación entre procesos.

La colección ACTS no es un conjunto cerrado de herramientas y se permite la adición de nuevas herramientas siempre que éstas cumplan con unos criterios mínimos para la computación científica [42]. Las funcionalidades de las librerías pertenecientes a la colección ACTS se pueden clasificar en cuatro grupos:

- *Numéricas*: Implementan métodos numéricos de resolución de sistemas densos y dispersos, en este grupo incluimos: Aztec [102, 108], Hipre [44], OPT++ [71, 99], PETSc [6], ScaLAPACK [80], SLEPc [68], SUNDIALS [15], SuperLU [59] y TAO [9, 97].

- *Entornos de trabajo*: Proporcionan la infraestructura necesaria que permite manejar la complejidad de la programación paralela (distribución de matrices, comunicación de información del entorno, etc.) pero no implementan métodos numéricos. Este grupo de herramientas está formado por Global Arrays [63], y Overture [67].

- *Soporte para la ejecución de aplicaciones*: Agrupa utilidades en el nivel de aplicación como puede ser el análisis del rendimiento y la visualización remota. Librerías pertenecientes a este bloque son CUMULUS [84], GLOBUS [49, 50] y TAO [103].

- *Soporte para el desarrollador*: Proporcionan la infraestructura y las herramientas que ayudan en el desarrollo de una aplicación de este tipo y mejoran el rendimiento

de éstas. Una librería perteneciente a este tipo es ATLAS [66].

Este conjunto de librerías enumerado anteriormente, ha sido desarrollado de forma independiente entre ellas y generalmente bajo la supervisión del Departamento de Energía de Estados Unidos. El hecho de difundir el conjunto de herramientas dentro del proyecto ACTS, permite una mayor difusión de estas herramientas y ha posibilitado, en este caso el desarrollo del trabajo presentado en esta tesis para poder integrarlas todas dentro de un mismo entorno de programación bajo Python.

Las librerías susceptibles de ser incluidas dentro de esta colección deberían cumplir determinados prerrequisitos enumerados y descritos en el apartado 4.2.2. En los próximos apartados de esta sección, describiremos aquellas librerías pertenecientes a la colección ACTS a las que se les ha creado una interfaz para su acceso desde Python. Una vez introducidas estas herramientas, describiremos en capítulos siguientes las librerías desarrolladas y su integración dentro de la distribución que hemos querido denominar PyACTS.

4.3.3. BLACS

Las librerías BLACS (Basic Linear Algebra Communication Subprograms) [25, 28, 111] proporcionan la misma facilidad de uso y portabilidad en las comunicaciones necesarias para cálculos del álgebra lineal en sistemas con memoria distribuida que las librerías BLAS [23, 24] proporcionan para el álgebra lineal.

La idea de concentrar las rutinas más utilizadas en computación en un núcleo altamente optimizado ha sido desarrollado y probado con éxito en la librería LAPACK [3, 29]. Conviene reseñar que la librería LAPACK es un conjunto para el álgebra lineal en sistemas secuenciales y de memoria compartida.

Las motivaciones del proyecto ScaLAPACK [17, 19, 27] eran las de conseguir unas librerías similares a LAPACK pero en sistemas paralelos con memoria distribuida. Los comienzos en el desarrollo de ScaLAPACK fueron rápidos pero pronto se hizo evidente la necesidad de un núcleo de rutinas destinadas a las comunicaciones. De este modo surge la necesidad de la librería BLACS.

Mediante estos dos núcleos de rutinas, el software para álgebra lineal en matrices densas en plataformas MIMD puede consistir en una serie de llamadas a BLAS y a las ru-

tinas de comunicación BLACS. Ambos paquetes han sido optimizados de forma particular para cada plataforma, obteniendo un buen rendimiento con relativamente poco esfuerzo. Además, ambas librerías se encuentran disponibles en un amplio conjunto de plataformas, en las que la modificación del código para adaptarlas a cada una de ellas será mínima.

Como ya hemos comentado en el apartado 4.3.1, existen disponibles diferentes librerías que proporcionan las interfaces de intercambio de mensajes para múltiples plataformas (PICL [57], PVM y más recientemente MPI). Estas tres librerías están destinadas para un ámbito más general y sus interfaces no son tan sencillas para el álgebra lineal como sería deseable.

Sin embargo, desde la aparición de BLACS, han ido surgiendo nuevas interfaces y métodos especializados a un campo específico. Por ejemplo, las librerías BLACS están escritas en un nivel donde la manipulación de matrices en la computación del álgebra lineal es natural y conveniente. Los principales objetivos de BLACS son:

- *Facilidad de programación.* Tanto como sea posible, BLACS simplifica el paso de mensajes reduciendo los errores de programación.

- *Facilidad de utilización.* Tanto para niveles superiores como para programadores.

- *Portabilidad.* BLACS debe proporcionar una interfaz utilizable en un amplio conjunto de sistemas paralelos, incluyendo sistemas heterogéneos.

En la terminología utilizada en BLACS, un proceso se define como un hilo de ejecución que incluye como mínimo una pila, registros y memoria. Múltiples procesos pueden compartir un procesador. De este modo, el termino procesador se refiere más a un recurso hardware.

En BLACS, cada proceso es tratado como si fuera un procesador: el proceso debe existir durante el tiempo de vida de la ejecución y sólo tendrá efecto en la ejecución de los otros procesos mediante el intercambio de mensajes. En estos términos, el proceso se convierte en una magnitud importante en la definición de BLACS. De este modo describiremos en los siguientes puntos las principales características de BLACS:

Comunicaciones basadas en matrices

Muchas distribuciones pueden ser clasificadas en función de las operaciones disponibles en matrices unidimensionales, que se corresponde con la representación de un vector en el álgebra lineal. Sin embargo, en los problemas del álgebra lineal, es más natural expresar las operaciones en términos de matrices de dos dimensiones. En computación, una matriz se representa mediante un cadena de dos dimensiones (*2D array*) y con este tipo de variables operará la librería BLACS.

BLACS reconoce las dos clases de matrices más comunes para el álgebra lineal de matrices densas. La primera de estas clases consiste en un matriz rectangular general, la cual se almacena en la máquina mediante una cadena 2D con `M` filas y `N` columnas, y con un factor (`LDA`, *leading dimension*) que determina la distancia entre dos elementos sucesivos de una fila en memoria. Un detalle importante a tener en cuenta es el hecho que BLACS asume que la matriz se encuentra ordenada por columnas en memoria, de ahí la importancia del parámetro `LDA`.

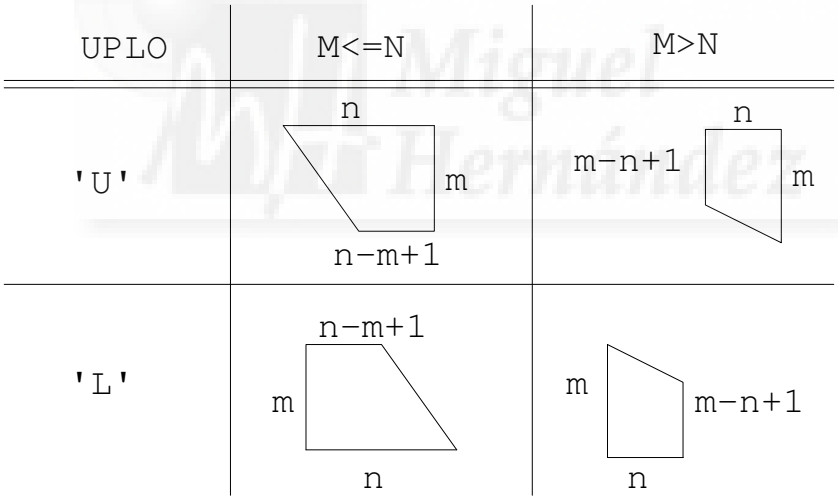


Figura 4.2: Aspecto de las matrices en función de los parámetros `M`, `N` y `UPLO`

La segunda clase de matrices reconocidas por BLACS son las matrices trapezoidales. Este tipo de matrices están definidas por `M`, `N` y `LDA` pero también por el parámetro `UPLO` (que nos indica si la matriz es trapezoidal superior o inferior) y por el parámetro `DIAG` que determina si la diagonal necesita ser comunicada al resto de procesos. Las matrices triangulares son una subclase de las trapezoidales, y por tanto pueden ser gestionadas por

BLACS.

De este modo, una matriz trapezoidal puede ser definida por los parámetros `M`, `N` y `UPLO` (ver figura 4.2). El empaquetado de las matrices se gestiona de forma interna en BLACS, permitiendo concentrarse al usuario en la matriz y no en cómo los datos son organizados en la memoria del sistema.

Malla de procesos y operaciones permitidas

Los N_p procesos de una tarea en paralelo suelen estar identificados mediante una etiqueta 0,1,..., N_{p-1}. Por razones que describiremos a continuación, resulta más adecuado transformar este vector de procesos de una dimensión, en una malla de procesos de dos dimensiones de `P` filas y `Q` columnas, donde $\texttt{P} \cdot \texttt{Q} = N_g \leq N_p$.

De este modo, un proceso puede ser descrito por sus coordenadas en esta malla de procesos (indicado por `(p,q)`, donde $\texttt{0} \leq \texttt{p} < \texttt{P}$, y $\texttt{0} \leq \texttt{q} < \texttt{Q}$). Un ejemplo de este tipo de configuración se representa en la figura 4.3.

	0	1	2	3
0	0	1	2	3
1	4	5	6	7

Figura 4.3: 8 procesos configurados en una malla 2×4

Una operación que incluye a más de un emisor o más de un receptor se denomina *de ámbito*. Todos aquellos procesos que participan en una operación de este tipo se dice que pertenecen al ámbito de la operación.

En un sistema que utiliza una matriz lineal de procesos, el ámbito natural incluye a todos los procesos. Utilizando una malla de dos dimensiones, tenemos 3 ámbitos naturales, tal y como se muestra en la tabla 4.1.

Estas agrupaciones de procesos tienen un interés particular en los programas de computación de álgebra lineal. A partir de la distribución de las matrices 2D y de la configuración

Ámbito	Significado
Fila	Todos los procesos de una fila
Columna	Todos los procesos de una columna
All	Todos los procesos de la malla

Tabla 4.1: Ámbitos en una malla de procesos 2D

de la malla de procesos, un proceso puede obtener toda una fila a través del ámbito de fila.

De este modo, las filas y columnas de la malla de procesos incorporan nuevos niveles adicionales de paralelismo al programador pero que no están exentos de las limitaciones hardware intrínsecas al sistema paralelo que estemos utilizando. Por ejemplo, si la malla de procesadores se encuentran conectadas vía Ethernet, podemos observar que la única ganancia se obtiene en la facilidad de programación. En este caso, no será posible distribuir simultáneamente los datos a una fila o columna y por ello una malla de procesos 1D obtiene mejor rendimiento. Afortunadamente, la mayoría de redes de interconexión modernas incorporan una malla 2D de comunicaciones y de este modo estos niveles de paralelismo pueden ser aprovechados.

Contextos

En BLACS, cada malla de procesos está identificada en un contexto. Un contexto puede describirse como un propio universo para el paso de mensajes. Esto significa que en una malla puede perfectamente establecer comunicaciones incluso si otra (posiblemente superpuesta) se encuentra también comunicando. De este modo, podemos decir que los términos *malla* y *contexto* definen la misma cosa.

Los contextos son utilizadas por las rutinas cuando este parámetro es requerido y funcionan de forma segura sin preocupación acerca de otros procesos distribuidos en la misma máquina.

Un ejemplo de uso del contexto puede ser una definición de una malla de procesos 2D donde la mayoría de procesos en ejecución tienen cabida en dicha malla. Sin embargo,

una porción del código ha de ser accesible por los procesos en una configuración de malla 1D, y además se puede presentar el caso que deseemos compartir información únicamente con los vecinos más cercanos. En este caso cada proceso forma parte de tres contextos: la malla 2D, la malla 1D y una pequeña malla que contiene al proceso y a sus vecinos más cercanos.

Por tanto, un contexto permite:

- Crear grupos arbitrarios de procesos.
- Crear un número indeterminado de mallas solapadas o discontinuas.
- Aislar cada malla de procesos de forma que no interfiera con el resto.

En BLACS, existen dos rutinas de creación de mallas (`BLACS_GRIDINIT` y `BLACS_GRIDMAP`) que crean una malla en un contexto determinado. Estas rutinas devuelven el contexto que define la malla, el cual es un simple entero. Mediante este identificador podemos determinar la malla o contexto a la que estamos haciendo referencia.

Un contexto consume recursos, y por lo tanto es recomendable liberar los contextos si no se necesitan. Para realizar esta liberación utilizaremos la rutina `BLACS_GRIDEXIT`. Si deseamos liberar la totalidad de los recursos BLACS, debemos llamar a la rutina `BLACS_EXIT`, y de este modo todos los recursos y contextos serán liberados.

Comunicaciones sin identificador

Una de las características que diferencian a BLACS de otros sistemas de intercambio de mensajes, es que no es necesario especificar un identificador de mensaje (`msgid`). Este identificador suele ser un entero que permite a un proceso la distinción entre diferentes mensajes recibidos. La generación de estos identificadores puede llegar a ser problemática, y un error común es el de utilizar un mismo `msgid` en un bucle, de este modo es posible que tengamos en recepción múltiples mensajes con el mismo identificador. Este tipo de programación provoca errores que pueden proporcionar resultados erróneos en los tiempos e incluso finalizar la ejecución del programa.

Para añadir comodidad en la utilización de BLACS, se decidió que BLACS generara de forma interna los `msgid` requeridos. Estos identificadores generados tienen determinadas

propiedades: en primer lugar, no han de estar relacionados con otros mensajes que tengan el mismo destino; por otro lado, el algoritmo de generación del ID debería utilizar únicamente información local para obtener un buen rendimiento. Además, deberemos permitir el funcionamiento de BLACS junto con otras plataformas de comunicación.

Para conseguir estos objetivos se ha decidido adoptar dos restricciones en las comunicaciones. La primera de ellas es que un proceso que reciba un mensaje debe conocer las coordenadas del proceso fuente. La segunda restricción establece que la comunicación entre dos procesos es estrictamente ordenada. Esto significa que si el proceso `(0,0)` envía dos mensajes a `(0,1)`, entonces `(0,1)` debe recibirlos en el mismo orden que fueron enviados.

Finalmente, para que BLACS pueda coexistir con otras distribuciones de comunicaciones, BLACS debe permitir al usuario especificar el rango de los `msgid` que BLACS puede utilizar.

Niveles de bloqueo

Para poder comprender los niveles de bloqueo utilizados en BLACS, es necesario establecer comunicaciones seguras entre los procesos. Una operación de comunicación tiene varios recursos vinculados a la misma, pero tiene mayor importancia la memoria intermedia (de aquí en adelante *buffer*) del usuario.

Los niveles de bloqueo indican al usuario la relación entre la respuesta a la ejecución de una rutina y la disponibilidad del *buffer*. Por ejemplo, si el usuario espera una recepción, necesita conocer el momento en el que el dato es recibido y almacenado en el *buffer*.

BLACS define tres niveles de bloqueo: no bloqueante, localmente bloqueante y globalmente bloqueante. Estos niveles se describen muy brevemente a continuación:

- *Comunicaciones no bloqueantes*. La respuesta de la rutina de comunicación implica únicamente que la solicitud de recepción del mensaje ha sido ejecutada. Es responsabilidad del usuario comprobar si la operación ha sido completada.
- *Comunicaciones localmente bloqueantes*. Aplicable en operaciones de envío pero no de recepción. La respuesta a este envío, implica que el *buffer* esta disponible para su reutilización.

v	**Significado**
I	Dato entero a ser comunicado
S	Dato de precisión simple a ser comunicado
D	Dato de precisión doble a ser comunicado
C	Dato de tipo complejo de precisión simple a ser comunicado
Z	Dato de tipo complejo de precisión doble a ser comunicado

Tabla 4.2: Relación entre la nomenclatura y los tipos de datos

- *Comunicaciones globalmente bloqueantes.* La respuesta de este tipo de operaciones implica que el *buffer* está disponible para su reutilización. Esta operación puede no completarse a menos que su operación complementaria sea ejecutada (un envío no se completa si otro proceso no ejecuta la recepción). BLACS proporciona comunicaciones globalmente bloqueantes en la recepción punto a punto, difusiones y las combinaciones de éstas. El envío punto a punto es localmente bloqueante.

Nomenclatura

BLACS clasifica las rutinas en función del tipo de comunicaciones y a partir de esta clasificación da nombre a cada rutina:

- *Punto a punto y difusión.* Los nombres de las rutinas de comunicación tienen la forma `vXXYY2D`, donde la letra `v` indica el tipo de dato utilizado (ver tabla 4.2), `XX` se reemplaza según el formato de la matriz mediante lo indicado en la tabla 4.3, e `YY` se reemplaza por el tipo de comunicación (ver tabla 4.4).

- *Combinaciones.* El aspecto en la nomenclatura es `vGZZZ2D`, donde la letra `v` indica el tipo de dato utilizado, y `ZZZ` indica el tipo de operación a realizar. Los tipos de operación se detallan en la tabla 4.5.

- *Rutinas de soporte.* Este tipo de rutinas proporcionan diversas funciones de mantenimiento y no tienen un alto grado de estandarización en su nomenclatura. Generalmente todas ellas comienzan por `BLACS_` más una descripción de su funcionalidad.

XX	Significado
GE	El dato a ser enviado está almacenado en una matriz rectangular de tipo general
TR	El dato a ser enviado está almacenado en una matriz trapezoidal

Tabla 4.3: Tipos de matrices utilizadas en BLACS

YY	Significado
SD	Envío. Un proceso envía a otro
RV	Recepción. Un proceso recibe de otro
BS	Difusión. Un proceso comienza la difusión en un ámbito determinado
BS	Recolección. Un proceso recibe datos de todos los procesos de un ámbito determinado. Se corresponde con el proceso inverso a la difusión

Tabla 4.4: Valores y significados en la nomenclatura de comunicaciones punto a punto y de difusión

ZZZ	Significado
AMX	Las entradas de la matriz resultante tendrá los mayores valores absolutos en esa posición.
AMN	Las entradas de la matriz resultante tendrá los menores valores absolutos en esa posición.

Tabla 4.5: Valores y significados en la nomenclatura de las operaciones de combinaciones

4.3.4. PBLAS

Se considera a PBLAS [18] una librería con la misma funcionalidad que la librería BLAS [114] pero son escalables y desarrolladas para su ejecución en una arquitectura paralela de memoria distribuida. En definitiva, se considera a PBLAS la versión paralela de BLAS. Esta librería BLAS (*Basic Linear Algebra Subprograms*) está compuesta por rutinas que proporcionan un alto rendimiento en operaciones entre matrices y vectores. BLAS (y por tanto PBLAS) está clasificada en tres niveles atendiendo a la complejidad de las operaciones de las rutinas. El nivel 1 proporciona operaciones de tipo escalar-vector y vector-vector, en el nivel 2 se definen operaciones entre vectores y matrices y finalmente en el nivel 3 se implementan las operaciones entre matrices. Tanto BLAS como PBLAS son rutinas eficientes, portables y ampliamente disponibles para una variedad de plataformas y sistemas. De este modo, BLAS ha sido utilizada en el desarrollo de otras librerías del álgebra lineal como LAPACK, del mismo modo que PBLAS ha sido utilizada para el desarrollo de librerías como ScaLAPACK que describiremos en el siguiente apartado.

Las librerías PBLAS están muy relacionadas con ScaLAPACK y de hecho se distribuyen como parte de ScaLAPACK. Debido a la estrecha relación y a la similitud en la definición de interfaces, variables y descriptores preferimos realizar una descripción más detallada que dedicaremos a ScaLAPACK en el apartado 4.3.5.

Por otro lado, deseamos destacar la librería ATLAS (*Automatically Tuned Linear Algebra Software*) [112, 113, 114, 115] como un paquete con la misma funcionalidad de BLAS pero cuyo propósito es el de proporcionar software optimizado para una plataforma específica. La versión actual de ATLAS proporciona una interfaz completa de BLAS (tanto para C como para Fortran 77) y un pequeño conjunto de las rutinas incluidas en LAPACK.

ATLAS es una librería de código fuente abierto y utilizada en muchos procesos científicos y como parte integrada en multitud de programas comerciales como Maple, Matlab, Mathematica u Octave. Además, ATLAS se encuentra incluido en sistemas operativos como Debian Linux, FreeBSD, Mac OS 10, Scyld Beowulf y SuSE Linux.

En `www.netlib.org/atlas` se puede encontrar toda la documentación disponible en lo referente a ATLAS, desde la descripción de sus interfaces, una sección de preguntas y soluciones así como un repositorio para controlar las diferentes versiones de ATLAS.

4.3.5. ScaLAPACK

ScaLAPACK [17] es una librería que engloba rutinas de alto rendimiento para el álgebra lineal para sistemas de memoria distribuida para la ejecución MIMD mediante el intercambio de mensajes en sistemas que soporten PVM [56] y/o MPI [48, 104]. ScaLAPACK se considera una continuación del proyecto LAPACK, en el cual se diseñó un software análogo para estaciones de trabajo, computadoras vectoriales y sistemas paralelos con memoria compartida. Ambas librerías contienen rutinas para la resolución de sistemas de ecuaciones lineales, problemas de mínimos cuadrados, y resolución de valores propios. El nombre ScaLAPACK se corresponde con la descripción de librería LAPACK escalable (***Sca****lable* ***LAPACK***).

El objetivo en ambos proyectos es obtener unas librerías con las características de eficiencia (funcionar tan rápidamente como sea posible con los recursos disponibles), escalabilidad (con respecto al tamaño y al número de procesadores), robustez (delimitación de los márgenes de error), portabilidad (a través de todas las máquinas paralelas más extendidas y utilizadas), flexibilidad (los usuarios pueden construir nuevas rutinas a partir de componentes bien diseñados), y facilidad de empleo (haciendo que la interfaz a LAPACK y a ScaLAPACK resulte tan similar como sea posible). Muchas de estas metas, particularmente la de portabilidad, se consiguen desarrollando y promoviendo estándares, especialmente diseñados para rutinas de bajo nivel en la comunicación y el cómputo.

El diseño de estas librerías se considera adecuado puesto que se ha conseguido limitar la mayoría de las dependencias de la máquina a dos bibliotecas estándares llamadas BLAS vistas en el apartado 4.3.4, y las rutinas para las comunicaciones en el álgebra lineal básica BLACS, vistas en el apartado 4.3.3. De este modo, LAPACK funcionaría en cualquier máquina donde estuviera disponible BLAS, y ScaLAPACK funcionaría en cualquier máquina donde estuvieran disponibles BLAS y BLACS.

Actualmente, la librería está escrita en Fortran 77 (a excepción de algunas rutinas auxiliares del cálculo de valores propios en matrices simétricas, escritas en C) en un estilo múltiple de programación MIMD utilizando el intercambio de mensajes para la comunicación entre los procesos.

Del mismo modo que LAPACK, las rutinas de ScaLAPACK están basadas en algoritmos de partición por bloques para minimizar la frecuencia de movimiento de datos entre los diferentes niveles de la jerarquía de memoria. Los bloques fundamentales de los que

hace uso la librería ScaLAPACK son la versión distribuida de la librería BLAS, esto es, la librería PBLAS y un conjunto de rutinas para las comunicaciones en el álgebra lineal englobadas en la librería BLACS. En las rutinas de ScaLAPACK, la mayor parte de las comunicaciones entre los procesos tiene lugar a través de las rutinas de PBLAS, de este modo, el código de los niveles superiores tiene un aspecto similar a su versión secuencial LAPACK.

Las rutinas proporcionadas están orientadas para el tratamiento de matrices densas y en banda, pero no para matrices dispersas. En la figura 4.4 se muestran los diferentes niveles de software utilizados por ScaLAPACK. Se puede observar la diferencia entre el software que se ejecuta en modo local (indicado en la parte inferior de la línea discontinua) y el software que se ejecuta en modo global a través de rutinas síncronas en paralelo, cuyos argumentos son matrices y vectores distribuidos entre los procesos que intervienen en la malla. Cada uno de los componentes mostrados en la figura, han sido introducidos y descritos en los apartados previos.

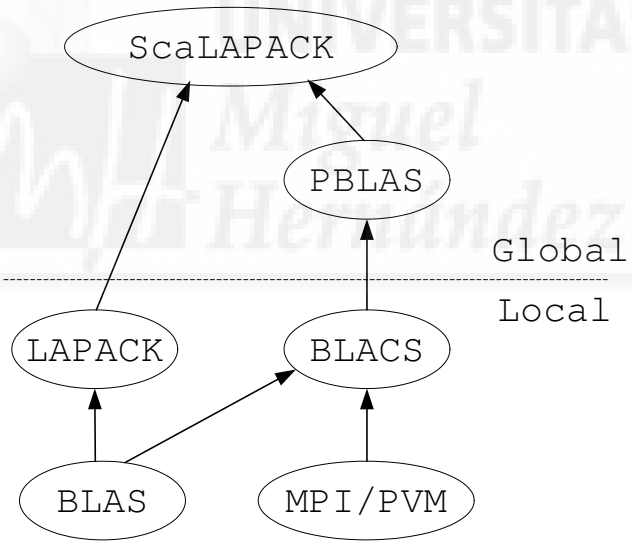


Figura 4.4: Jerarquía de software en ScaLAPACK

Para la ejecución de una rutina de ScaLAPACK, es necesario realizar una serie de pasos en el orden establecido para obtener una respuesta adecuada. A continuación enumeramos estos pasos que serán descritos con más detalle:

1. Inicializar la malla de procesos.

2. Distribuir los datos entre los procesos de la malla.

3. Realizar la llamada a la rutina de ScaLAPACK.

4. Liberar la malla de procesos.

Por tanto, el primer paso para la ejecución de una rutina de ScaLAPACK, es el de inicializar la malla de procesos. Con esta finalidad se incluye la rutina `SL_INIT`. Esta rutina inicializa una malla de tamaño `nprow` × `npcol` (notación descrita en 4.3.3) utilizando un orden por filas de los procesos y asignando un identificador de contexto a esta malla. A partir de esta inicialización, cada proceso puede obtener su identificación dentro de la malla (`myrow,mycol`) con la rutina `BLACS_GRIDINFO`. Un ejemplo de utilización de estas rutinas se puede apreciar en la figura 4.5.

```
CALL SL_INIT( ICTXT, NPROW, NPCOL )
CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
```

Figura 4.5: Ejemplo de inicialización de una malla de procesos

Describiremos de forma breve los parámetros utilizados en la llamada a `SL_INIT`:

- `ICTXT`: Parámetro de salida. Entero global que indica el contexto BLACS que identifica a la malla creada.
- `NPROW`: Parámetro de entrada. Entero global que indica el número de filas en la malla de procesos que se crea.
- `NPCOL`: Parámetro de entrada. Entero global que indica el número de columnas en la malla de procesos que se crea.

Los parámetros descritos ya han sido introducidos en el apartado 4.3.3, si se desea una mayor profundidad en cada uno de ellos, se recomienda la lectura del manual de usuario de BLACS [28].

El segundo paso a realizar consiste en la distribución de los datos entre los procesos de la malla. Queremos incidir en el hecho de que es responsabilidad del usuario realizar una correcta distribución de los datos previa a la ejecución de la correspondiente rutina de ScaLAPACK. Por tanto, no se incluye ninguna rutina en esta librería que distribuya los datos y únicamente se publican algunos ejemplos de cómo realizar esta distribución.

Sin embargo, en el manual de usuario [14] se explican de forma detallada las bases de la distribución de datos cíclica 2D utilizada en ScaLAPACK y que describiremos de forma breve a continuación.

Cada matriz global es distribuida entre los procesos que pertenecen a la malla a partir de un descriptor de dicha matriz. El descriptor asociado a una matriz es un vector de 9 elementos, y generalmente se le nombra en el código con `DESC`. Se asume que todos los vectores y matrices globales se encuentran distribuidos mediante la distribución cíclica por bloques que explicaremos posteriormente, y cada uno de estos vectores o matrices tendrá asociado su descriptor correspondiente. Añadimos el sufijo "`_A`" a cada uno de los parámetros que explicaremos a continuación, para indicar que hacen referencia a la matriz `A`. En el caso que se haga referencia a estos parámetros sin utilizar un sufijo estaremos haciendo referencia a este parámetro de una manera global.

Generalmente, una matriz distribuida `A` se encuentra definida por sus dimensiones `M_A` $\times$ `N_A`, el tamaño de los bloques utilizados para su descomposición `MB_A` $\times$ `NB_A`, las coordenadas del proceso que tiene en su memoria local la primera entrada de la matriz (`RSRC_A`, `CSRC_A`), y el contexto BLACS `ICTXT_A` en el cual está definida la matriz. Finalmente, el parámetro `LLD_A` indica la distancia entre dos elementos contiguos en la misma fila, puesto que el orden utilizado en ScaLAPACK (al igual que en Fortran) es por columnas. El significado de los elementos del descriptor `DESC_A` se muestra en la tabla 4.6.

En el manual del usuario de ScaLAPACK [14] se detallan algunos ejemplos que aclaran la distribución cíclica utilizada y el valor de cada uno de los elementos del descriptor. Sin embargo, a continuación detallaremos un ejemplo que utilizaremos en apartados posteriores. En este caso, supongamos que deseamos distribuir en una malla de 4 procesos 2×2, por tanto, `nprow=2` y `npcol=2`. El aspecto de la matriz se presenta en la figura 4.6, donde se aprecia que es una matriz de tamaño 8×8.

Para construir el descriptor de esta matriz podremos utilizar la rutina `DESCINIT` proporcionada en ScaLAPACK, los parámetros necesarios en la rutina son los descritos en la tabla 4.6 y el código de ejemplo de su utilización se presenta a continuación:

```
CALL DESCINIT(DESC_A,M_A,N_A,MB_A,NB_A,RSRC_A,CSRC_A,ICTXT_A,MXLLDA,INFO)
```

Donde `DESC_A` es un parámetro de salida que se ha definido previamente como un vector de enteros de tamaño igual a 9. En el ejemplo mostrado en la figura 4.6, el tamaño de la

DESC_()	Nombre	Ámbito	Definición
1	DTYPE_A	Global	Tipo de descriptor para matrices densas. DTYPE_A=1
2	ICTXT_A	Global	Contexto BLACS en el cual está distribuida la matriz A
3	M_A	Global	Número de filas en la matriz global A
4	N_A	Global	Número de columnas en la matriz global A
5	MB_A	Global	Tamaño de bloque usado para distribuir las filas de la matriz A
6	NB_A	Global	Tamaño de bloque usado para distribuir las columnas de la matriz A
7	RSRC_A	Global	Indicador del número de fila del proceso en el que la primera columna de la matriz A se ha distribuido
8	CSRC_A	Global	Indicador del número de columna del proceso en el que la primera columna de la matriz A se ha distribuido
9	LLD_A	Local	Distancia entre dos elementos contiguos de la misma fila. LLD_A $\geq$ MAX(1,$LOC_R(M_A)$) (donde $LOC_R()$ y $LOC_C()$ se utiliza para definir el número de filas o columnas de la matriz global que tiene almacenado cada proceso en local)

Tabla 4.6: Contenido del descriptor para matrices densas

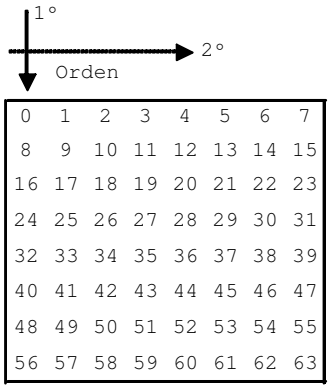


Figura 4.6: Datos de la matriz A de ejemplo

matriz vendrá definido por `M_A=8` y `N_A=8`. Para mostrar de una forma gráfica y cómoda en este ejemplo la distribución cíclica en dos dimensiones elegimos un tamaño de bloque 2×2 y por tanto `MB_A=2` y `NB_A=2`. Generalmente, el proceso identificado como `(0,0)` suele ser el que tiene los primeros datos de la primera columna de A, y por tanto configuraremos `RSRC_A=0` y `CSRC_A=0`. El valor de `ICTXT_A` dependerá del obtenido al inicializar la malla en el paso previo. Por último, `MXLLDA` se corresponde con el número de filas de la matriz local resultante. El valor de `MXLLDA` depende de varios parámetros, puesto que no sólo depende del número de filas de la matriz sino también del número de columnas de la malla de procesos. De este modo `MXLLDA` puede ser calculado mediante la fórmula `MXLLDA` $\geq$ `MAX(1,`$LOC_R(M_A)$`)` y se obtiene que `MXLLDA=4`. Se ha de tener en cuenta que este dato es local, es decir, puede diferir entre distintos procesos aunque no haya ocurrido así en este ejemplo. Supongamos que a partir de la matriz A representada en la figura 4.6 utilizamos un tamaño de bloque 3×3. En este caso, y de forma similar a como se realiza la distribución en la figura 4.7, el proceso `(0,0)` tendría un valor `MXLLDA` igual a 6, mientras que en el proceso `(0,1)`, `MXLLDA` vale 2.

Además del descriptor calculado anteriormente, deberemos distribuir los datos. Generalmente, el proceso identificado como `(0,0)` suele ser el encargado de leer estos datos desde un fichero o fuente para distribuirlos al resto de procesos utilizando la librería BLACS. En la distribución de ScaLAPACK no se incluyen de forma interna a las librerías ninguna rutina auxiliar para distribuir estos datos, sin embargo, en su manual se incluyen algunas rutinas como `PDLAREAD` que pueden servir de ejemplo al programador.

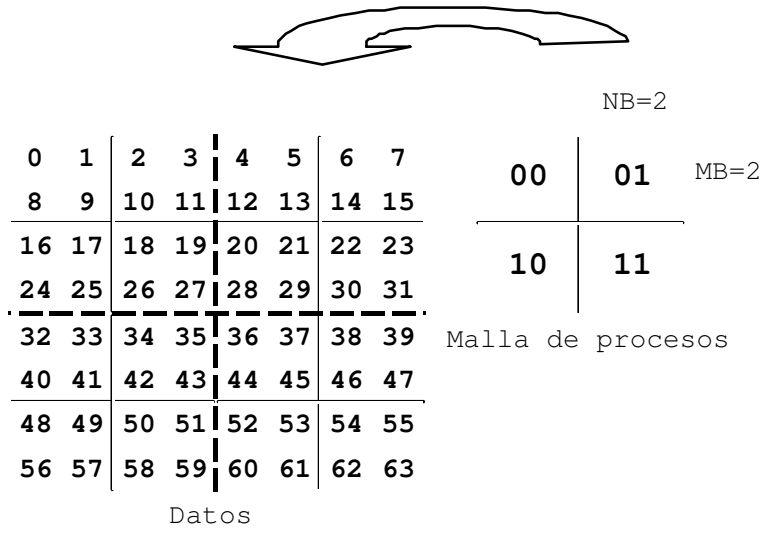


Figura 4.7: Ejemplo de distribución cíclica en dos dimensiones

En la figura 4.7 se puede apreciar la distribución realizada a partir de la matriz `A` y con una configuración de malla de procesos 2×2 y tamaño de bloque 2×2. Un detalle que deseamos destacar es el orden de los datos en memoria, ya que la programación de ScaLAPACK está realizada generalmente en Fortran y el orden de las matrices en Fortran se realiza por columnas. Por ejemplo, el proceso `(0,0)` tiene en memoria los datos representados en la figura 4.8 y ordenados por columnas (`[0,8,32,40,1,9,33,41,5,13,37,45]`).

El tercer paso a realizar consiste en la propia llamada a la rutina de ScaLAPACK. En este punto se asume que los datos han sido ya distribuidos entre todos los procesos pertenecientes a la malla. Generalmente, tanto los parámetros de entrada como de salida suelen encontrarse distribuidos conforme a la malla establecida y si queremos obtener los datos de una matriz de salida deberemos recolectarlos de nuevo en un único proceso. En este caso, la librería ScaLAPACK sí incluye algunas rutinas para imprimir en pantalla una matriz distribuida (`PvLAPRNT`, donde `v` representa el tipo de la variable conforme a la asignación vista en la tabla 4.2) pero no así para guardarla en un fichero por lo que el usuario debería programar dicha rutina.

El cuarto y último paso consiste en liberar la malla de procesos una vez hayamos ejecutado las rutinas de ScaLAPACK. Para liberar una malla de procesos específica se

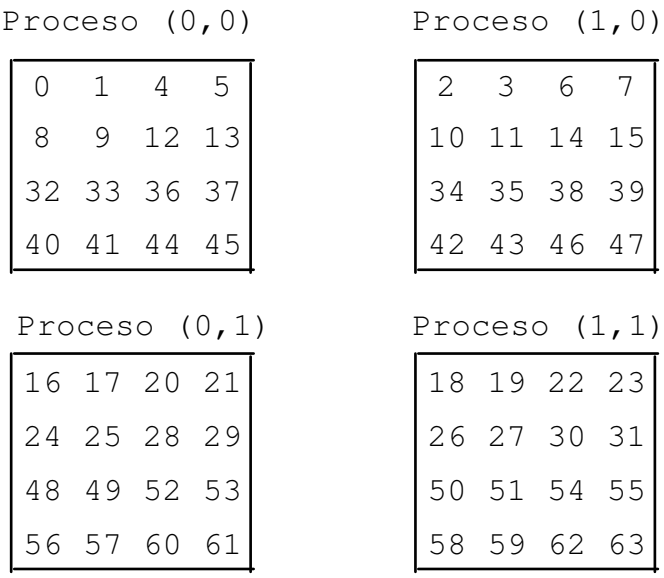


Figura 4.8: Asignación de datos conforme la distribución cíclica del ejemplo

recomienda utilizar `BLACS_GRIDEXIT`, indicando mediante su único parámetro `ICTXT` el identificador de contexto de dicha malla. Si se han finalizado todos los cálculos y se desea liberar el entorno BLACS, podemos ejecutar `BLACS_EXIT` sin necesidad de ningún parámetro adicional.

Este cuarto paso se suele implementar en el siguiente código:

```
CALL BLACS_GRIDEXIT( ICTXT )
CALL BLACS_EXIT( 0 )
```

Una vez revisado el proceso necesario para la ejecución de una rutina de ScaLAPACK, describiremos el conjunto de rutinas disponibles en ScaLAPACK clasificadas de acuerdo a su funcionalidad. Las rutinas de ScaLAPACK se dividen en tres categorías:

1. Rutinas *Driver*. Cada una de estas rutinas resuelve un problema completo, como por ejemplo la resolución de un sistema de ecuaciones o el cálculo de los valores propios de una matriz.

2. Rutinas Computacionales. Cada una de estas rutinas realiza una tarea computacional diferente, por ejemplo, una factorización *LU* o la reducción de una matriz real

simétrica a su forma tridiagonal. Cada rutina driver llama a una secuencia de rutinas computacionales.

3. Rutinas Auxiliares. Pueden ser clasificadas del siguiente modo.

   a) Rutinas que implementan subtareas de algoritmos en bloques.

   b) Rutinas que realizan funciones a bajo nivel, por ejemplo, el escalado de una matriz, la norma de una matriz, o la generación de matrices de datos temporales.

En la documentación oficial de ScaLAPACK [80] se encuentran documentadas las rutinas driver y las rutinas computacionales, pero no las rutinas auxiliares. Por otro lado, las rutinas englobadas en PBLAS, BLAS, BLACS, y LAPACK no se consideran parte de ScaLAPACK aunque las rutinas de ésta ultima realicen llamadas frecuentemente a las librerías mencionadas.

Cada rutina de ScaLAPACK suele tener su equivalente LAPACK y su nombre será el mismo que el de la rutina LAPACK pero con el prefijo "`P`". Todas las rutinas driver y computacionales tienen el siguiente aspecto:"`PXYZZZ`". La segunda letra tiene el mismo significado que el visto en la tabla 4.2 en el apartado de BLACS. De este modo, para una misma funcionalidad (por ejemplo `PvGESV`) tendremos una rutina para cada tipo de dato (`PSGESV, PDGESV, PCGESV` y `PZGESV`)

Las siguientes dos letras, `YY`, indican el tipo de matriz. La mayoría de los códigos de dos letras utilizados se aplican en matrices reales y complejas, sin embargo algunos nombres de rutinas difieren para el caso real y el complejo. Los valores posibles de `YY` se muestran en la tabla 4.7.

Las últimas tres letras `ZZZ` indican la operación realizada. Su significado se detalla a continuación de manera más amplia.

1. Rutinas Driver.

   a) **Ecuaciones lineales.**
   Se proporcionan dos tipos de rutinas para la resolución de estos sistemas:

   - Una rutina sencilla (acabada en `SV`) que resuelve el sistema

$$AX = B,$$

YY	**Significado**
DB	General en banda (dominante diagonalmente)
DT	General tridiagonal (dominante diagonalmente)
GB	General en banda
GE	General (no simétrica, rectangular)
GG	Matrices Generales
HE	Hermítica (compleja)
OR	Ortogonal (real)
PB	Simétrica o hermítica en bandas definida positiva
PO	Simétrica o hermítica definida positiva
PT	Simétrica o hermítica tridiagonal definida positiva
ST	Simétrica tridiagonal (real)
SY	Simétrica
TR	Triangular
TZ	Trapezoidal
UN	Unitaria (compleja)

Tabla 4.7: Tipos de matriz en la nomenclatura de rutinas de ScaLAPACK

donde $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times nrhs}$ para rutinas de tipo real (`PS` y `PD`), o bien, $A \in \mathbb{C}^{m \times n}, B \in \mathbb{C}^{n \times nrhs}$ para rutinas de tipo complejo (`PC` y `PZ`). La solución se obtiene factorizando A y sobreescribiendo B con la solución X.

- Una rutina avanzada (acabada en `SVX`) que también resuelve el sistema descrito anteriormente, pero que proporciona las siguientes funcionalidades:
  - resuelve $A^T X = B$ o $A^H X = B$ (a menos que A sea simétrica o hermítica).
  - estima el número de condición de A y comprueba la singularidad de A.
  - Ajusta la solución y la comprueba en ambos sentidos para la corrección de errores de redondeo.
  - Equilibra el sistema si A está mal escalada.

En ambos casos, la matriz B puede tener diferentes columnas, es decir, el sistema tiene múltiples soluciones, una para cada columna. En la tabla 4.8 se muestran los nombres de las rutinas de resolución de ecuaciones lineales, clasificadas por funcionalidad, tipo de datos y tipo de matriz.

b) **Problemas de mínimos cuadrados**.
Un problema LLS (*linear least squares*) de este tipo se puede definir cómo:

$$min \parallel b - Ax \parallel_2$$

donde A es una matriz $m \times n$, b es un vector de m elementos y x es el vector solución de n elementos.

En los casos más generales, $m \geq n$ y $rango(A) = n$. En este caso, la solución al problema es única y se debe encontrar la solución mínima cuadrada a un sistema de ecuaciones lineales sobredeterminado.

Si $m < n$ y $rango(A) = m$, entonces hay un número infinito de soluciones que satisfaces $b - Ax = 0$. En este caso, suele ser más útil encontrar la solución única que minimiza $\parallel x \parallel_2$ y el problema se convierte en encontrar una solución mínima a un sistema de ecuaciones indeterminado.

La rutina `PvGELS` resuelve este tipo de sistemas, asumiendo que $rango(A) = min(m, n)$, es decir, A tiene rango completo y por tanto encuentra la solución mínima de un sistema sobredeterminado si $m > n$, y una solución de norma

Tipo de matriz y Almacenamiento	Operación	Precisión Simple		Precisión Doble	
		Real	Complejo	Real	Complejo
General (pivotación parcial)	sencilla avanzada	PSGESV PSGESVX	PCGESV PCGESVX	PDGESV PDGESVX	PZGESV PZGESVX
General en bandas (pivotación parcial)	sencilla	PSGBSV	PCGBSV	PDGBSV	PZGBSV
General en bandas (sin pivotación)	sencilla	PSDBSV	PCDBSV	PDDBSV	PZDBSV
General Tridiagonal (sin pivotación)	sencilla	PSDTSV	PCDTSV	PDDTSV	PZDTSV
Simétrica/Hermítica definida positiva	sencilla avanzada	PSPOSV PSPOSVX	PCPOSV PCPOSVX	PDPOSV PDPOSVX	PZPOSV PZPOSVX
Simétrica/Hermítica definida positiva en bandas	sencilla	PSPBSV	PCPBSV	PDPBSV	PZPBSV
Simétrica/Hermítica definida positiva tridiagonal	sencilla	PSPTSV	PCPTSV	PDPTSV	PZPTSV

Tabla 4.8: Rutinas driver para resolución de ecuaciones lineales

Operación	Precisión Simple		Precisión Doble	
Resolución LLS utilizando factorización QR y LQ	PSGELS	PCGELS	PDGELS	PZGELS

Tabla 4.9: Rutinas driver para problemas de mínimos cuadrados

mínima a un sistema indeterminado cuando $m < n$. La rutina `PvGELS` utiliza una factorización QR o LQ de A y también permite realizarla con A^T o con A^H en el caso de matrices complejas. En la tabla 4.9 se muestran las rutinas disponibles para la resolución de problemas de mínimos cuadrados.

c) **Obtención de valores propios y descomposición en valores singulares.**

- Problemas de valores propios en matrices simétricas.
  En este tipo de problemas (SEP, *symmetric eigenvalue problem*) se debe encontrar los valores propios λ y los correspondientes vectores propios $z \neq 0$ que cumplan

$$Az = \lambda z,\ A = A^T,\ A \in \mathbb{R}^{m \times n}$$

  Para el problema de valores propios hermíticos se debe cumplir

$$Az = \lambda z,\ A = A^H, A \in \mathbb{C}^{m \times n}$$

  Siendo λ real en ambos casos. Cuando todos los valores propios han sido calculados podemos decir que

$$A = Z \Lambda Z^T,$$

  donde Λ es una matriz diagonal con los elementos de la misma obtenidos de λ, y Z es una matriz ortogonal cuyas columnas son los vectores propios. Este tipo de factorización se conoce como factorización espectral. En la librería ScaLAPACK, se proporcionan dos rutinas que solucionan este tipo de problemas:

  - Una rutina sencilla (acabada en `EV`) que calcula todos los valores propios y opcionalmente los vectores propios de la matriz A, siendo ésta simétrica y hermítica.
  - Una rutina avanzada (acabada en `EVX`) que calcula todos o un conjunto de los valores propios y opcionalmente los correspondientes vectores propios.

- Descomposición en valores singulares.
  La descomposición SVD (*singular value decomposition*) de una matriz A de tamaño $m \times n$ se puede expresar como

$$A = U \Sigma V^T,\ A \in \mathbb{R}^{m \times n},$$

Tipo de Problema	Operación	Precisión Simple		Precisión Doble	
		Real	Complejo	Real	Complejo
SEP	sencilla	PSSYEV		PDSYEV	
	avanzada	PSSYEVX	PCHEEVX	PDSYEVX	PZHEEVX
SVD	vectores/valores singulares	PSGESVD		PDGESVD	

Tabla 4.10: Rutinas driver para problemas de valores propios y singulares

o bien

$$A = U\Sigma V^H,\ A \in \mathbb{C}^{m\times n},$$

donde U y V son matrices ortogonales (unitarios) y Σ es una matriz $m \times n$ diagonal real con sus elementos σ_i de modo que

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{min(m,n)}.$$

Los valores y vectores singulares cumplen la siguiente relación

$$Av_i = \sigma_i u_i$$

y

$$A^T u_i = \sigma_i v_i,\ A \in \mathbb{R}^{m\times n};\ A^H u_i = \sigma_i v_i,\ A \in \mathbb{C}^{m\times n}.$$

En este tipo de problemas, la rutina `PvGESVD` realiza la descomposición en valores singulares para una matriz general no simétrica. De este modo, si A es una matriz $m \times n$ donde $m > n$, entonces únicamente se obtienen las primeras n columnas de U y Σ es una matriz de tamaño $n \times n$. Las rutinas incluidas actualmente en ScaLAPACK para resolver problemas de este tipo de muestran en la tabla 4.10. Actualmente no se proporcionan las rutinas para el caso complejo por lo que las celdas correspondientes de la tabla 4.10 aparecen en blanco.

d) **Obtención de valores propios en matrices simétricas definidas**. Este tipo de rutinas GSEP (*generalized symmetric definite eigenproblems*) se pueden definir como avanzadas y obtienen todos los valores propios y opcionalmente los vectores propios de los siguientes tipos de problemas:

1) $Az = \lambda Bz$

Tipo de Problema	Operación	Precisión Simple		Precisión Doble	
		Real	Complejo	Real	Complejo
GSEP	avanzada	PSSYGVX	PCHEGVX	PDSYGVX	PZHEGVX

Tabla 4.11: Rutinas driver para problemas de valores propios en matrices simétricas o hermíticas

2) $ABz = \lambda z$

3) $BAz = \lambda z$

donde A y B son matrices simétricas o hermíticas y B es definida positiva. Para todos estos problemas, $\lambda \in \mathbb{R}$. Cuando A y B son simétricas, las matrices Z de los vectores propios computados cumplen $Z^T AZ = \Lambda$ (para problemas de tipo 1 y 3) o $Z^{-1}A^{-T} = I$ (para problemas de tipo 2), donde Λ es una matriz diagonal con los valores propios en la diagonal. Z también cumple que $Z^T BZ = I$ (para problemas de tipo 1 y 2) o $Z^T B^{-1} Z = I$ (para problemas de tipo 3). Si A y B son hermíticas, las matrices Z satisfacen que $Z^H AZ = \Lambda$ (para problemas de tipo 1 y 3) o $Z^{-1}A^{-H} = \Lambda$ (para problemas de tipo 2), donde Λ es una matriz diagonal con los valores propios en la diagonal. Z también cumple que $Z^H BZ = I$ (tipo 1 y 2) o $Z^H B^{-1} Z = I$.

Las rutinas que implementan la obtención de valores propios en matrices simétricas definidas se muestran en la tabla 4.11.

2. Rutinas Computacionales.

Este conjunto de rutinas realiza una tarea específica pero no resuelve ningún problema completo. Describiremos el conjunto de rutinas computacionales incluidas en ScaLAPACK, pero cuya implementación difiere de su versión secuencial LAPACK, debido fundamentalmente a una mejor explotación del paralelismo. Las rutinas computacionales se dividen en los siguientes grupos funcionales:

a) **Ecuaciones lineales**.

Si recordamos la notación estándar para un sistema de ecuaciones lineales:

$$Ax = B,$$

donde $A \in \mathbb{R}^{m \times n}$ es la matriz de coeficientes, $B \in \mathbb{R}^{n \times nrhs}$ es la matriz de términos independientes, y x representa las soluciones al sistema. En este sis-

tema generalmente se asume que A es una matriz cuadrada de dimensiones $n \times n$, aunque algunas rutinas permiten matrices rectangulares.

Si A es una matriz triangular superior o inferior, se puede resolver mediante sustituciones consecutivas. De este modo, la solución se puede obtener después de haber factorizado la matriz A como producto de matrices triangulares.

La forma de la factorización depende de las propiedades de la matriz A. ScaLAPACK proporciona rutinas para los siguientes tipos de matrices:

- Matrices generales (factorización LU con pivotación parcial):

  $$A = PLU,$$

  donde P es una matriz de permutación, L es una matriz triangular inferior con unos en su diagonal principal, y U es una matriz triangular superior.
- Matrices simétricas y hermíticas definidas positivas (factorización de Cholesky):

  $$A = U^T U \text{ o } A = LL^T \text{ (en el caso de matrices simétricas),}$$
  $$A = U^H U \text{ o } A = LL^H \text{ (en el caso de matrices hermíticas),}$$

  donde U es una matriz triangular superior y L es triangular inferior.
- Matrices generales en bandas (factorización LU con pivotación parcial):
  Si A es una matriz $m \times n$ con *bwl* diagonales inferiores y *bwu* diagonales superiores, la factorización de A se puede expresar como

  $$A = PLUQ,$$

  donde P y Q son matrices permutación y U y L son matrices en bandas triangulares superior e inferior respectivamente.
- Matrices generales en bandas del tipo diagonalmente dominantes (factorización LU sin pivotación):
  A es una matriz del tipo diagonalmente dominante si se conoce *a priori* que no se requiere pivotación para obtener estabilidad en la factorización LU.
  Si A es una matriz $m \times n$ con *bwl* diagonales inferiores y *bwu* diagonales superiores, la factorización se puede obtener como:

  $$A = PLUP^T,$$

  donde P es una matriz de permutación y L y U son matrices triangulares en bandas inferior y superior respectivamente.

- Matrices en bandas simétricas y hermíticas definidas positivas (factorización de Cholesky):

  $A = PU^TUP^T$ o $A = PLL^TP^T$ (en el caso de matrices simétricas),
  $A = PU^HUP^T$ o $A = PLL^HP^T$ (en el caso de matrices hermíticas),

  donde P es una matriz de permutación y L y U son matrices en banda triangular inferior y superior respectivamente.
- Matrices tridiagonales simétricas y hermíticas definidas positivas (factorización LDL^T):

  $A = PU^TDUP^T$ o $A = PLDL^TP^T$ (en el caso de matrices simétricas),
  $A = PU^HDUP^T$ o $A = PLDL^HP^T$ (en el caso de matrices hermíticas),

  donde P es una matriz de permutación y L y U son matrices bidiagonales inferior y superior respectivamente.

Mientras que el uso principal de una factorización es la resolución de un sistema de ecuaciones, se proporcionan en ScaLAPACK diversas rutinas que implementan diferentes pasos en la factorización y que varían en función del tipo de matriz y del esquema de almacenamiento utilizado:

- `PxyyTRF`: factoriza una matriz.
- `PxyyTRS`: utiliza la factorización para resolver el sistema mediante sustituciones.
- `PxyyCON`: estima el número de condición $\kappa(A) = \|A\| \, \|A^{-1}\|$.
- `PxyyRFS`: calcula los límites del error en la solución obtenida.
- `PxyyTRI`: utiliza la factorización para calcular A^{-1}.
- `PxyyEQU`: calcula los factores de escalado para equilibrar A.

Se ha de tener en cuenta que algunas rutinas computacionales dependen de los resultados obtenidos en otras:

- `PxyyTRF`: trabaja sobre una matriz equilibrada producida por `PxyyEQU`.
- `PxyyTRS`: requiere la factorización devuelta por `PxyyTRF`.
- `PxyyCON`: requiere la norma de la matriz original A y la factorización devuelta por `PxyyTRF`.
- `PxyyRFS`: requiere las matrices originales A y B, la factorización devuelta por `PxyyTRF` y la solución devuelta por `PxyyTRS`.
- `PxyyTRI`: requiere la factorización devuelta por `PxyyTRF`.

En la tabla 4.12 mostramos el conjunto de rutinas computacionales relativas a sistemas de ecuaciones lineales clasificadas en función del tipo de dato con el que operan y con el tipo de almacenamiento de la matriz.

b) **Problemas de mínimos cuadrados y factorizaciones ortogonales**.
ScaLAPACK proporciona un conjunto de rutinas para factorizar una matriz A rectangular $m \times n$, como el producto de una matriz ortogonal (o unitaria en el caso complejo) y una matriz triangular.

Una matriz real Q es ortogonal si $Q^T Q = I$; una matriz compleja Q es unitaria si $Q^H Q = I$. Las matrices ortogonales y unitarias tienen importante propiedad que mantienen invariante la norma dos de un vector:

$$\|x\|_2 = \|Qx\|_2, \text{ si } Q \text{ es ortogonal o unitaria.}$$

Como resultado, permiten mantener la estabilidad numérica ya que no se amplifican los errores de redondeo. Las factorizaciones ortogonales se utilizan en la solución de problemas de mínimos cuadrados, o también como paso previo en problemas de valores singulares.

En la tabla 4.13 se muestran las rutinas incluidas en ScaLAPACK que implementan la factorización ortogonal y la generación de la matriz Q para cada tipo de matriz y esquema de almacenamiento. A continuación, describiremos de forma breve la funcionalidad de cada uno de los grupos de rutinas mostrados en dicha tabla.

- Factorización QR.
  Es la factorización más común y conocida. Esta definida por

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}, si\ m \geq n,$$

  donde R es una matriz triangular superior $n \times n$ y Q es una matriz $m \times m$ ortogonal (o unitaria). Si A tiene rango completo n, entonces R es no singular. Por otro lado, es conveniente escribir la factorización como

$$A = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R \\ 0 \end{pmatrix},$$

  que se reduce a

$$A = Q_1 R,$$

Tipo matriz y Almacen.	Operación	Precisión Simple		Precisión Doble	
		Real	Complejo	Real	Complejo
general	factorizar	PSGETRF	PCGETRF	PDGETRF	PZGETRF
piv. parcial	resolver fact.	PSGETRS	PCGETRS	PDGETRS	PZGETRS
	est. num. cond.	PSGECON	PCGECON	PDGECON	PZGECON
	lím. errores	PSGERFS	PCGERFS	PDGERFS	PZGERFS
	invertir fact.	PSGETRI	PCGETRI	PDGETRI	PZGETRI
	equilibrar	PSGEEQU	PCGEEQU	PDGEEQU	PZGEEQU
gen. bandas	factorizar	PSGBTRF	PCGBTRF	PDGBTRF	PZGBTRF
piv. parcial	resolver fact.	PSGBTRS	PCGBTRS	PDGBTRS	PZGBTRS
gen. bandas	factorizar	PSDBTRF	PCDBTRF	PDDBTRF	PZDBTRF
sin piv.	resolver fact.	PSDBTRS	PCDBTRS	PDDBTRS	PZDBTRS
gen. tridiag.	factorizar	PSDTTRF	PCDTTRF	PDDTTRF	PZDTTRF
sin piv.	resolver fact.	PSDTTRS	PCDTTRS	PDDTTRS	PZDTTRS
simétrica /	factorizar	PSPOTRF	PCPOTRF	PDPOTRF	PZPOTRF
hermítica	resolver fact.	PSPOTRS	PCPOTRS	PDPOTRS	PZPOTRS
def. positiva	est. num. cond.	PSPOCON	PCPOCON	PDPOCON	PZPOCON
	lím. errores	PSPORFS	PCPORFS	PDPORFS	PZPORFS
	invertir fact.	PSPOTRI	PCPOTRI	PDPOTRI	PZPOTRI
	equilibrar	PSPOEQU	PCPOEQU	PDPOEQU	PZPOEQU
simétrica /	factorizar	PSPBTRF	PCPBTRF	PDPBTRF	PZPBTRF
hermítica	resolver fact.	PSPBTRS	PCPBTRS	PDPBTRS	PZPBTRS
def. positiva en bandas					
simétrica /	factorizar	PSPTTRF	PCPTTRF	PDPTTRF	PZPTTRF
hermítica	resolver fact.	PSPTTRS	PCPTTRS	PDPTTRS	PZPTTRS
def. positiva tridiagonal					

Tabla 4.12: Rutinas computacionales para ecuaciones lineales

Tipo fact. y matriz	Operación	Precisión Simple		Precisión Doble	
		Real	Complejo	Real	Complejo
QR, general	fact. con pivot.	PSGEQPF	PCGEQPF	PDGEQPF	PZGEQPF
	fact. sin pivot.	PSGEQRF	PCGEQRF	PDGEQRF	PZGEQRF
	generar Q	PSORGQR	PCUNGQR	PDORGQR	PZUNGQR
	mult. por Q	PSORMQR	PCUNMQR	PDORMQR	PZUNMQR
LQ, general	fact. sin pivot.	PSGELQF	PCGELQF	PDGELQF	PZGELQF
	generar Q	PSORGLQ	PCUNGLQ	PDORGLQ	PZUNGLQ
	mult. por Q	PSORMLQ	PCUNMLQ	PDORMLQ	PZUNMLQ
QL, general	fact. sin pivot.	PSGEQLF	PCGEQLF	PDGEQLF	PZGEQLF
	generar Q	PSORGQL	PCUNGQL	PDORGQL	PZUNGQL
	mult. por Q	PSORMQL	PCUNMQL	PDORMQL	PZUNMQL
RQ, general	fact. sin pivot.	PSGERQF	PCGERQF	PDGERQF	PZGERQF
	generar Q	PSORGRQ	PCUNGRQ	PDORGRQ	PZUNGRQ
	mult. por Q	PSORMRQ	PCUNMRQ	PDORMRQ	PZUNMRQ
RZ, trapezoidal	fact. sin pivot.	PSTZRZF	PCTZRZF	PDTZRZF	PZTZRZF
	mult. por Z	PSORMRZ	PCUNMRZ	PDORMRZ	PZUNMRZ

Tabla 4.13: Rutinas computacionales para factorizaciones ortogonales

donde Q_1 consiste en las primeras n columnas de Q, y Q_2 las restantes $m-n$ columnas.

Si $m < n$, R es trapezoidal, y la factorización se puede escribir como

$$A = Q \begin{pmatrix} R_1 & R_2 \end{pmatrix}, \ si\ m < n,$$

donde R_1 es triangular superior y R_2 es rectangular.

La rutina `PxGEQRF` obtiene la factorización QR. Esta factorización puede utilizarse para resolver un problema de mínimos cuadrados cuando $m \geq n$ y A tiene rango completo del siguiente modo:

$$\|b - Ax\|_2 = \left\|Q^T b - Q^T A x\right\|_2 = \left\| \begin{matrix} c_1 - Rx \\ c_2 \end{matrix} \right\|_2,$$

$$\text{donde } c = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} Q_1^T b \\ Q_2^T b \end{pmatrix} = Q^T b;$$

c se obtiene mediante `PxORMQR` (o `PxUNMQR` para el caso complejo), y c_1 consiste en sus primeros n elementos. De este modo, x es la solución del sistema triangular superior

$$Rx = c_1,$$

que puede calcularse mediante la rutina `PxTRTRS`. El vector residual r es

$$r = b - Ax = Q \begin{pmatrix} 0 \\ c_2 \end{pmatrix},$$

y puede calcularse mediante `PxORMQR` (o `PxUNMQR`).

- Factorización LQ.
  La factorización LQ esta definida como

$$A = \begin{pmatrix} L & 0 \end{pmatrix} Q = \begin{pmatrix} L & 0 \end{pmatrix} \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = LQ_1, \ si\ m \leq n,$$

donde L es una matriz $m \times m$ triangular inferior, Q es una matriz $n \times n$ ortogonal (o unitaria), y Q_1 contiene las primeras m filas de Q y Q_2 las $n-m$ filas restantes.

La factorización LQ de A es esencialmente la misma que la factorización QR de A^T (A^H si A es compleja), ya que

$$A = \begin{pmatrix} L & 0 \end{pmatrix} Q \Leftrightarrow A^T = Q^T \begin{pmatrix} L^T \\ 0 \end{pmatrix}$$

La factorización LQ se puede utilizar para encontrar una solución con norma mínima para un sistema indeterminado de ecuaciones lineales $Ax = b$, donde A es una matriz $m \times n$ con $m < n$ y rango m. La solución se obtiene a partir de

$$x = Q^T \begin{pmatrix} L^{-1}b \\ 0 \end{pmatrix},$$

que puede calcularse mediante las llamadas a `PxTRTRS` y `PxORMLQ`.

- Factorización QR con pivotación por columnas.
  Para resolver un problema de mínimos cuadrados cuando A no tiene un rango completo o se desconoce el rango de A, podemos realizar una factorización QR con pivotación por columnas o una descomposición de valores singulares.

  La factorización QR con pivotación por columnas está definida por

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} P^T, \ m \geq n,$$

  donde Q y R han sido definidos en los puntos anteriores y P es una matriz de permutación, elegida de modo que

$$|r_{11}| \geq |r_{22}| \geq \ldots \geq |r_{nn}|,$$

  de modo que para cada k,

$$|r_{kk}| \geq \|R_{k:j,j}\|_2 \text{ para } j = k+1, \ldots, n.$$

  La rutina `PxGEQPF` obtiene la factorización QR con pivotación por columnas sin tratar de determinar el rango de A. La matriz Q se representa del mismo modo que en la llamada a `PxGEQRF`, y las rutinas `PxORGQR` y `PxORMQR` trabajan con la matriz Q del mismo modo que en los puntos anteriores.

- Factorización ortogonal completa.
  La factorización QR con pivotación por columnas no permite el cálculo de una solución con norma mínima a un problema de mínimos cuadrados deficiente en su rango a menos que $R_{12} = 0$. Sin embargo, aplicando transformaciones ortogonales, desde la derecha a la matriz trapezoidal superior $\begin{pmatrix} R_{11} & R_{12} \end{pmatrix}$, utilizando la rutina `PxTZRZF`, R_{12} puede ser eliminada:

$$\begin{pmatrix} R_{11} & R_{12} \end{pmatrix} Z = \begin{pmatrix} T_{11} & 0 \end{pmatrix}.$$

Esta expresión proporciona la factorización ortogonal completa

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z^T,$$

donde la solución con norma mínima puede ser obtenida como

$$x = PZ \begin{pmatrix} T_{11}^{-1} \hat{c}_1 \\ 0 \end{pmatrix}.$$

La matriz Z no se proporciona de forma explícita sino como producto de sus reflectores elementales. Las rutinas que se proporcionan para trabajar con la matriz Z son `PxORMRZ` (o `PxUNMRZ`).

c) **Factorizaciones ortogonales generalizadas**.

- Factorización QR generalizada.
  La factorización ortogonal generalizad GQR (*generalized QR*) de una matriz A de tamaño $n \times m$ y una matriz B de tamaño $n \times p$ está definida por la siguiente pareja de factorizaciones
  $$A = QR \text{ y } B = QTZ,$$
  donde Q y Z son matrices ortogonales (o unitarias en el caso complejo) de tamaño $n \times n$ y $p \times p$, respectivamente. Se puede obtener más detalle de estas factorizaciones en [14].

  La rutina `PxGEQRF` obtiene la factorización GQR calculando primero la factorización QR de A y despues la factorización RQ de $Q^T B$.
- Factorización RQ generalizada.
  La factorizació ortogonal generalizada GRQ (*generalized RQ*) de una matriz A de tamaño $m \times n$ y una matriz B de tamaño $p \times n$ están definidas por la siguiente pareja de factorizaciones
  $$A = RQ \text{ y } B = ZTQ,$$
  donde Q y Z son matrices ortogonales (o unitarias en el caso complejo) de tamaño $n \times n$ y $p \times p$ respectivamente. Se puede obtener más detalle de estas factorizaciones en [14].

d) **Problemas de valores propios en matrices simétricas**.
Sea A una matriz simétrica real (o hermítica en el caso complejo). Se denomina al escalar λ como valor propio y a un vector columna z el correspondiente vector

Tipo de Problema	Operación	Precisión Simple		Precisión Doble	
		Real	Complejo	Real	Complejo
densa simétrica (o hermítica)	reducción tridiagonal	PSSYTRD	PCHETRD	PDSYTRD	PZHETRD
ortogonal (/ unitaria)	matriz multi. después reducc. por PxSYTRD	PSORMTR	PCUNMTR	PDORMTR	PZUNMTR

Tabla 4.14: Rutinas computacionales para problemas de valores y vectores propios en matrices simétricas o hermíticas

propio si $Az = \lambda z$. λ es siempre un valor real si A es simétrica o hermítica. Las tareas básicas en este tipo de rutinas se centran en obtener los valores de λ y, opcionalmente, los vectores z de una matriz dada.

Los cálculos computacionales se realizan en dos etapas:

1) La matriz simétrica real o compleja hermítica A se reduce a su forma tridiagonal T. Si A es real simétrica, la descomposición es $A = QTQ^T$ donde Q es ortogonal y T simétrica tridiagonal real. Si A es compleja hermítica, la descomposición de A es $A = QTQ^H$ donde Q es unitaria y T es una matriz simétrica tridiagonal real.
2) Se calculan los valores y vectores propios de la matriz simétrica tridiagonal T. Si se calculan todos los valores y vectores propios, se realiza un proceso equivalente a factorizar T como $T = S\Lambda S^T$, donde S es una matriz ortogonal y Λ es diagonal. Los elementos de la diagonal de Λ son los valores propios de T y también los de A, y las columnas de S son los vectores propios de T. Los vectores propios de A son las columnas $Z = QS$, de modo que $A = Z\Lambda Z^T$ (o $A = Z\Lambda Z^H$ si A es compleja hermítica).

Las rutinas que implementan el cálculo de valores y vectores propios pueden consultarse en la tabla 4.14

e) **Problemas de valores propios en matrices no simétricas**.
Sea A una matriz cuadrada $n \times n$. Denominamos al escalar λ como valor propio y al vector no nulo v como vector propio por la derecha si cumplen que $Av = \lambda v$. Un vector no nulo u que cumple que $u^H A = \lambda u^H$ se llama vector propio por la izquierda. La primera tarea básica de las rutinas descritas en este apartado es

calcular, para una matriz A, todos los n valores de λ y, si se desea, sus vectores asociados u o v.

La segunda tarea básica es calcular la factorización Schur de una matriz A. Si A es compleja, entonces su factorización Schur es $A = ZTZ^H$, donde Z es unitaria y T es triangular superior. Si A es real, su factorización Schur es $A = ZTZ^T$. donde Z es ortogonal, y T es casi-triangular superior (bloques 1×1 y 2×2 en su diagonal). Las columnas de Z se llaman vectores Schur de A. Los valores propios de A aparecen en la diagonal de T. Estas dos tareas se pueden realizar en dos fases:

1) La matriz general A se reduce a la forma superior de Hessenberg H con ceros por debajo de la primera subdiagonal. La reducción se puede escribir como $A = QHQ^T$ donde Q es ortogonal si A es real, o $A = QHQ^H$ donde Q es unitaria si A es compleja. La reducción se realiza mediante la rutina `PxGEHRD`, que representa Q. La rutina `PxORMHR` (o en el caso complejo `PxUNMHR`) implementa la multiplicación de Q por otra matriz sin obtener de forma explicita Q.
2) La matriz triangular superior de Hessenberg H se reduce a su forma de Schur T mediante la factorización $H = STS^T$ (si H es real) o $H = STS^H$ (si H es compleja). La matriz S puede ser obtenida y también puede ser multiplicada por Q determinada en la fase anterior, para proporcionar $Z = QS$, es decir, los vectores Schur de A. Los valores propios se obtienen de la diagonal de T. Estas funcionalidades están proporcionadas por la rutina `PxLAHQR`.

En la tabla 4.15 se muestra el conjunto de rutinas disponibles que implementan la obtención de los valores propios mediante los pasos descritos.

f) **Descomposición en valores singulares**.
Sea A una matriz real $m \times n$, la descomposición en valores singulares SVD de A se corresponde con la factorización $A = U\Sigma V^T$, donde U y V son ortogonales y $\Sigma = diag(\sigma_1, \ldots, \sigma_r)$, siendo $r = min(m, n)$, donde $\sigma_1 \geq \ldots \geq \sigma_r \geq 0$. Si A es compleja, la descomposición SVD es $A = U\Sigma V^H$, donde U y V son unitarias y Σ es similar a la definida anteriormente, con sus elementos en la diagonal reales. Los valores σ_i se denominan valores singulares, las primeras r columnas de V son los vectores singulares por la derecha y las primeras r columnas de U son los vectores singulares por la izquierda.

Tipo de Problema	Operación	Precisión Simple		Precisión Doble	
		Real	Complejo	Real	Complejo
general ortogonal o unitaria	Hessenberg reducción multip. matriz	PSGEHRD PSORMHR	PCGEHRD PCUNMHR	PDGEHRD PDORMHR	PZGEHRD PZUNMHR
Hessenberg	valores propios y descomp. Schur	PSLAHQR			PDLAHQR

Tabla 4.15: Rutinas computacionales para problemas de valores y vectores propios en matrices no simétricas

Tipo de Problema	Operación	Precisión Simple		Precisión Doble	
		Real	Complejo	Real	Complejo
general	reducción bidiagonal	PSGEBRD	PCGEBRD	PDGEBRD	PZGEBRD
ortogonal o unitaria	multip. matriz	PSORMBR	PCUNMBR	PDORMBR	PZUNMBR

Tabla 4.16: Rutinas computacionales para la descomposición en valores singulares

Las rutinas que proporcionan estas funcionalidades se muestran en la tabla 4.16, y los cálculos se implementan en dos etapas:

1) La matriz A se reduce a su forma bidiagonal: $A = U_1 B V_1^T$ si A es real ($A = U_1 B V_1^H$ si A es compleja), donde U_1 y V_1 son ortogonales (unitarias en el caso complejo), y B es real y bidiagonal superior si $m \geq n$ y bidiagonal inferior si $m < n$.
2) La descompisición SVD de la matriz B se calcula mediante $B = U_2 \Sigma V_2^T$, donde U_2 y V_2 son ortogonales y Σ es diagonal tal y como se ha descrito anteriormente. Los vectores singulares de A son $U = U_1 U_2$ y $V = V_1 V_2$.

La reducción a la forma bidiagonal la realiza la rutina `PcGEBRD` y la descomposición SVD se realiza mediante la rutina de LAPACK `xBDSQR`.

g) **Problemas de valores propios en matrices simétricas definidas**.

Este grupo de rutinas atacan el problema de obtener la solución a los problemas de

	Tipo	Factorización	Reducción	Obtención de vect. propios
1.	$Az = \lambda Bz$	$B = LL^T$	$C = L^{-1}AL^{-T}$	$z = L^{-T}y$
		$B = U^TU$	$C = U^{-T}AU^{-1}$	$z = U^{-1}y$
2.	$ABz = \lambda z$	$B = LL^T$	$C = L^TAL$	$z = L^{-T}y$
		$B = U^TU$	$C = U^TAU$	$z = U^{-1}y$
3.	$BAz = \lambda z$	$B = LL^T$	$C = L^TAL$	$z = Ly$
		$B = U^TU$	$C = UAU^T$	$z = U^Ty$

Tabla 4.17: Reducción de problemas de valores propios simétricos generalizados a problemas estándar

Tipo de Problema	Operación	Precisión Simple		Precisión Doble	
		Real	Complejo	Real	Complejo
simétrica /hermítica	reducción	PSSYGST	PCHEGST	PDSYGST	PZHEGST

Tabla 4.18: Rutinas computacionales para problemas de valores propios simétricos generalizados a problemas estándar

valores propios $Az = \lambda Bz$, $ABz = \lambda z$, y $BAz = \lambda z$, donde A y B son matrices reales simétricas o complejas hermíticas y B es definida positiva. Cada uno de estos problemas puede ser reducido a un problema de valores propios simétrico utilizando la factorización de Cholesky de B como $B = LL^T$ o $B = U^TU$ (LL^H o U^HU en el caso hermítico).

Con $B = LL^T$, tenemos que

$$Az = \lambda Bz \Rightarrow (L^{-1}AL^{-T})(L^Tz) = \lambda(L^Tz).$$

De este modo, los valores propios de $Az = \lambda Bz$ son aquellos donde $Cy = \lambda y$, donde C es una matriz simétrica $C = L^{-1}AL^{-T}$ e $y = L^Tz$. En el caso complejo, C es hermítica donde $C = L^{-1}AL^{-H}$ e $y = L^Hz$. La tabla 4.17 muestra los tres tipos de problemas que pueden ser reducidos a un problema estándar de la forma $Cy = \lambda y$, y cómo los vectores propios z del problema original pueden ser obtenidos a partir de los vectores propios y del problema reducido. En la tabla 4.18 se muestran la rutinas de ScaLAPACK que implementan estas operaciones.

Capítulo 5

PyACTS

Tal y como se ha visto en el apartado 4.3.2, la colección ACTS engloba un conjunto de herramientas muy útiles para la computación de alto rendimiento. Una vez introducido el lenguaje interpretado Python y las principales librerías de computación en sistemas paralelos, introduciremos el paquete de Python que centra este trabajo y que hemos denominado PyACTS [34, 36, 37, 38, 41, 40, 53, 55]. En la primera sección de este capítulo realizaremos una declaración de intenciones y trataremos cuáles son las bases de PyACTS y el porqué de los primeros módulos que se han incorporado en el mismo. La segunda sección, definiremos su estructura, detallando los principales objetos y clases que se han definido mostrando ejemplos que aclaran su utilización. Uno de estos objetos definidos (la matriz PyACTS) se detalla de forma más profunda en la tercera sección. Por otro lado, la distribución PyACTS se puede utilizar con diferentes implementaciones de MPI sobre Python; este punto será tratado en la sección 5.4. En la sección 5.5 describiremos las funciones auxiliares incluidas en la distribución PyACTS, que permiten facilitar y automatizar muchos de sus procesos. En las dos secciones siguientes describimos el proceso de creación de la distribución PyACTS, que abarca desde la implementación de los principales módulos `.py` (sección 5.6) hasta la creación de la librería de objetos compartidos `.so` (sección 5.7). En la sección 5.8 se analizan algunas decisiones de diseño tomadas durante el proceso de creación de la distribución PyACTS. Por último, en la sección 5.9 se realiza un análisis de PyACTS destacando las principales conclusiones obtenidas durante este capítulo.

5.1. Introducción

Como ya se ha comentado en el apartado 4.3.2, la colección ACTS es un conjunto de herramientas que ayuda a los programadores a crear aplicaciones en entornos de alto rendimiento. ACTS es un proyecto paraguas que engloba un conjunto de librerías diseñadas para sistemas paralelos de memoria distribuida y comunicaciones mediante paso de mensajes.

En múltiples áreas de conocimiento de la ingeniería o la investigación científica, los programadores suelen ser ingenieros, geólogos, biólogos, meteorólogos con unas necesidades para desarrollar una aplicación o una simulación pero cuyos conocimientos de programación suelen generalmente básicos. Además, en muchas de estas aplicaciones, el elevado volumen de datos con el que deben trabajar obliga a buscar soluciones escalables que puedan ser ejecutadas en plataformas de memoria distribuida. Por tanto, estos usuarios deben hacer uso de librerías de computación paralela pero la utilización de estas librerías desde lenguajes como C o Fortran requiere de ciertos conocimientos de los que no disponen. En definitiva, el investigador consume sus recursos y su tiempo en aprender los fundamentos de las librerías paralelas en lugar de centrarse en desarrollar una aplicación que simplemente haga uso de ellos de una manera cómoda y transparente.

En este trabajo, proponemos PyACTS como un conjunto de módulos que pueden ser importados a Python (introducido en el capítulo 3) y que permiten hacer uso de algunas de las rutinas incluidas en las librerías que engloba ACTS desde este lenguaje de programación. Esto permite aprovechar las características del lenguaje Python (interpretado, facilidad, interactividad, ...) junto con las de las librerías de alto rendimiento ACTS.

El objetivo por tanto, es crear un paquete de distribución denominado PyACTS que incorpore el nivel intermedio entre dos niveles de programación: por un lado las librerías de bajo nivel incluidas en ACTS (ScaLAPACK, Hypre, SuperLU, ...), y por otro lado el intérprete de Python. Este objetivo a priori parece alcanzable puesto que tal y como se indica en el apartado 3.1.5, una de las características de este lenguaje es la extensibilidad de Python, es decir, poder añadir nuevas funciones y objetos a partir de rutinas desarrolladas en otros lenguajes (C, Fortran, o Java en otros).

De este modo, un usuario de Python podrá hacer uso de librerías de alto rendimiento que hasta el momento únicamente están disponibles en entornos de más bajo nivel como C o Fortran, obteniendo una rápida curva de aprendizaje en la utilización de herramientas

de alto rendimiento. Una de las consideraciones previas en este trabajo, era la penalización introducida por la interfaz intermedia que introduciríamos entre el usuario y las librerías de computación. Durante los capítulos de análisis de cada uno de estos módulos se tratará de cuantificar la penalización introducida por los interfaces de alto nivel.

5.2. Estructura

Si la colección ACTS es un proyecto que engloba un conjunto de librerías de alto rendimiento, nuestro objetivo es conseguir un paquete PyACTS que incorpore diferentes módulos que hagan uso de cada una de las librerías incluidas en ACTS, pero desde una interfaz a Python y estableciendo una estructura de objetos que nos permita interactuar entre las diferentes librerías. Esto es, poder utilizar con un mismo conjunto de datos una o varias librerías incluidas en ACTS en un mismo código de forma transparente al usuario. A continuación, detallamos las diferentes partes de PyACTS, las cuales pueden verse de forma gráfica en la figura 5.1.

- ***Intérprete de Python paralelo (MIMD)***

  La distribución oficial de Python no está orientada para la ejecución de diferentes procesos en paralelo sino más bien para aplicaciones secuenciales. Sin embargo, para conseguir un intérprete en paralelo de Python que inicialice las librerías de paso de mensajes existen distribuciones o módulos de Python que incorporan esta funcionalidad. Por un lado pyMPI (descrito en el apartado 3.2.3) y por otro lado, en la distribución de ScientificPython (apartado 3.2.2) se incluye un módulo que incorpora las facilidades de MPI a través de las correspondientes interfaces. La diferencia entre cada intérprete y la solución que hemos adoptado al respecto, se trata en el apartado 5.4.

- ***Paquete PyACTS***

  Nos referimos al término "paquete" porque queremos destacar que se trata de un conjunto de *scripts* (`.py`) y librerías compiladas como objetos compartidos (`.so`) englobados dentro de la carpeta `PyACTS`. Al incluir, por ejemplo, el archivo `PyScaLAPACK.py` dentro de la carpeta `PyACTS`, podremos importar el archivo con

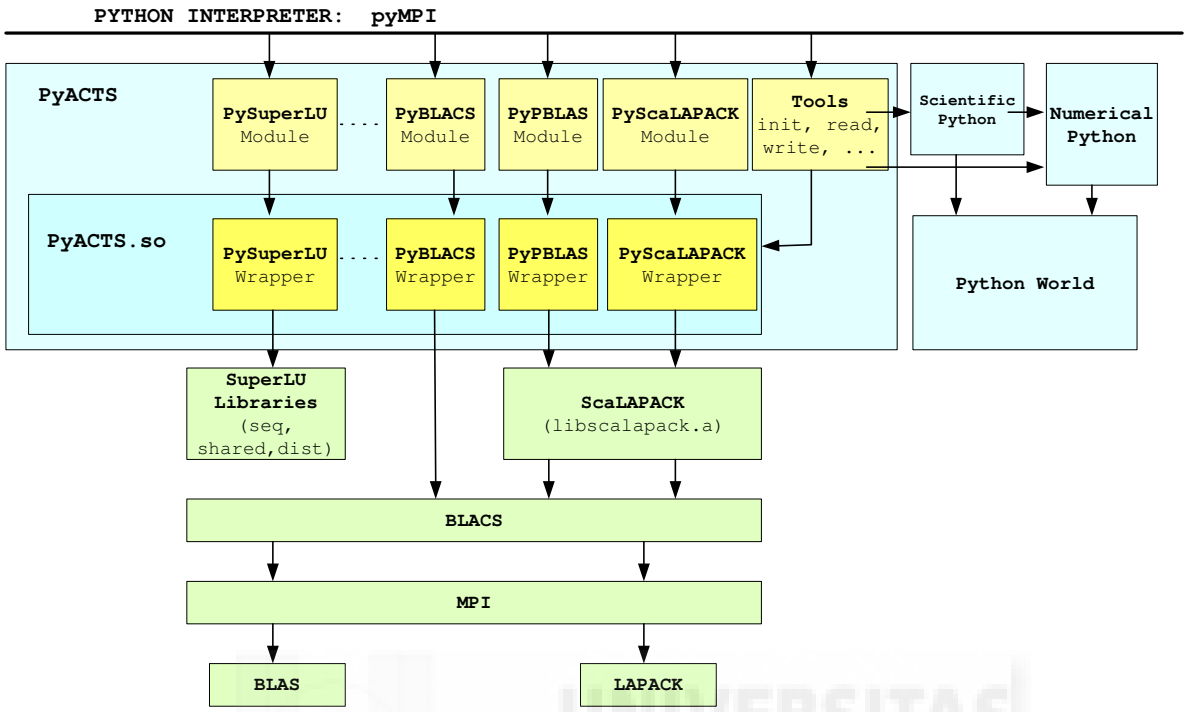


Figura 5.1: Vista conceptual de PyACTS

el direccionamiento `PyACTS.PyScaLAPACK`. Por tanto, para seleccionar el módulo correspondiente deberemos primero indicar el paquete en el que se encuentra el módulo. Por otro lado, dentro del directorio se encuentra el archivo `__init__.py`. Este archivo inicializa las variables de entorno necesarias en el módulo PyACTS y éstas se cargarían en el momento en el que se importara cualquier módulo del paquete. Además, en este archivo se definen las funciones auxiliares (que se describirán en la sección 5.5) comunes a ACTS como pueden ser `PyACTS.Num2PyACTS` o `PyACTS.gridinit()`.

- ***Módulos de cada librería (PyBLACS, PyScaLAPACK, PyPBLAS, ...)***

  En función de la estructura definida por las distribuciones a niveles inferiores de BLACS, PBLAS y ScaLAPACK, se ha decidido mantener esta estructura y por tanto diferenciar varios módulos dentro de PyACTS. De este modo, si un usuario desea hacer uso de una función específica (por ejemplo, `pdgemm`, $\alpha AB+\beta C$) que se encuentra en una librería específica (`PBLAS`, en este caso), deberemos importar el módulo correspondiente para poder hacer la llamada a dicha rutina (`PyPBLAS.pvgemm`).

En el apartado 5.6 se introduce la estructura básica de estos archivos y en los capítulos 6, 7 y 8 describiremos con más detalle cada uno de los módulos `PyBLACS`, `PyPBLAS` y `PyScaLAPACK` respectivamente, detallando las rutinas que contiene cada uno de ellos, su rendimiento y mostrando ejemplos de uso que expongan la ganancia en facilidad de utilización.

- ***Módulo de herramientas auxiliares***

  Para proporcionar una interfaz más sencilla y cómoda al usuario, se deben proporcionar un grupo de rutinas que resuelvan las siguientes tareas:

  - **Servicios básicos**: Un conjunto de rutinas que permitan crear, destruir, consultar, copiar y modificar los objetos y variables utilizados en el entorno PyACTS.
  - **Servicios de entrada/salida**: Para proporcionar mayor versatilidad al paquete se incluye un conjunto de rutinas que permitan leer desde diferentes formatos de datos y guardarlos como variables utilizables por PyACTS. Deseamos destacar que esta funcionalidad no se incluye en las distribuciones de la colección ACTS, y que al incluirlas como parte del paquete PyACTS se desea proporcionar mayor facilidad en el manejo de las rutinas.
  - **Verificación y validación**: Para la correcta ejecución de las rutinas incluidas en la colección ACTS se debe seguir una serie de pasos de forma ordenada (ver apartado 4.3.5). En el caso de no realizar una correcta ejecución de estos pasos, la llamada a las rutinas de niveles inferiores devolvería un error en tiempo de ejecución. Para evitar esto, antes de pasar el control a los niveles inferiores se debe comprobar que todos los parámetros son adecuados y se encuentran correctamente definidos. Para realizar estas verificaciones se ha creado un nuevo tipo de error a partir del manejo que hace Python de los mismos. La forma en la que PyACTS maneja las validaciones de los datos se detalla con mayor profundidad en el apartado 5.5.
  - **Conversión de datos**: Si se desea integrar la colección ACTS como un paquete Python, se debe proporcionar rutinas que permitan la conversión de diferentes tipos de datos (Numeric, NetCDF o bien SuperLU, PETSc, ...) a matrices PyACTS. El concepto de la matriz PyACTS se detallará en el apartado 5.3.

    Deseamos destacar que se dispone de dos niveles de rutinas auxiliares, es decir, aquellas rutinas comunes al paquete PyACTS que se definen en el archivo

__init__.py comentado anteriormente. Sin embargo, si por ejemplo llamamos a la rutina `PyACTS.Num2PyACTS(a,ACTS_lib=1)`, el parámetro `ACTS_lib` indica que deseamos realizar una conversión a una matriz PyACTS con el formato de distribución cíclica 2D definida para ScaLAPACK. A partir del indicador `ACTS_lib` se discrimina entre diferentes formatos y se llama a la rutina auxiliar del módulo correspondiente. En este caso, las rutinas se encuentran definidas en el archivo `PyScaLAPACK_Tools.py` que conforman el segundo nivel de rutinas auxiliares.

- ***Archivo de librerías compartidas*** `PyACTS.so`

  En este único archivo se encuentran las interfaces C creadas para poder acceder a las rutinas de ScaLAPACK, BLACS y PBLAS desde Python. Estas interfaces han sido creadas conforme a la documentación descrita en la sección 3.3 y utilizando programas de automatización como F2PY (apartado 3.3.2) o SWIG (apartado 3.3.1), principalmente. Será en este archivo donde se define cada una de las interfaces a las librerías de niveles inferiores y donde deberemos realizar comprobaciones previas para verificar la continuidad y validez de los datos obtenidos de niveles superiores. Este archivo `PyACTS.so` es directamente importable desde el intérprete de comandos Python mediante una simple ejecución "`import PyACTS.PyACTS`" (puesto que el primer `PyACTS` indica el nombre del paquete y el segundo el nombre del archivo o módulo que se encuentra en el directorio de dicho paquete). Se puede decir que en este archivo se proporciona la *importabilidad* a Python, sin embargo, el nivel superior (archivos `PyPBLAS.py, PyBLACS.py y PyScaLAPACK.py)` proporciona el contexto, las automatizaciones y verificaciones descritas anteriormente.

Un detalle importante a tener en cuenta en este caso, se refiere al uso eficaz de los objetos en cuanto a requerimientos de memoria. Como se observa en la figura 5.1, todas las interfaces y librerías de alto rendimiento se encuentran agrupadas dentro del archivo `PyACTS.so`, sin embargo si sólo hiciéramos uso de librerías para el tratamiento de matrices densas (ScaLAPACK, PBLAS y BLACS), no sería necesario cargar en memoria las interfaces y librerías de SuperLU. Es por ello que creemos conveniente separar en archivos de librerías compartidas aquellos módulos que no tienen una relación interna entre sus librerías.

En resumen, debemos evitar generar librerías compartidas de tamaños excesivamente elevados. Sin embargo, un pequeño detalle de gran importancia nos obliga a volver a la situación original de un único archivo de librería compartida. Se ha de

tener en cuenta que si creamos un archivo `PyScaLAPACK.so` que agrupe interfaces y rutinas de ScaLAPACK, PBLAS, BLACS, también las rutinas de MPI deberán ser incluidas en dicho archivo. Por otro lado, si creamos un archivo `PySuperLU.so` deberíamos añadirle del mismo modo los interfaces y rutinas de SuperLU y además las rutinas de MPI. Por tanto, se repiten las rutinas de la interfaz de paso de mensajes MPI. Se puede considerar un inconveniente o un mal recurso el hecho de que la misma rutina resida en memoria por duplicado en objetos diferentes, pero además de ese detalle, no obtendríamos una correcta interacción entre las rutinas de los dos archivos puesto que al utilizar diferentes rutinas MPI (en diferentes posiciones de memoria), se tratarían como mundos MPI diferentes (`MPI_world`), impidiendo el funcionamiento esperado entre ambas.

- ***Librerías ACTS***

  A pesar de incluirse dentro del archivo `PyACTS.so`, queremos diferenciar este conjunto de librerías en un nivel inferior. En el último paso de generar el archivo de librerías compartidas `PyACTS.so`, incluiremos las librerías a las que se hace referencia en las interfaces (`libscalapack.a`, `libblacs.a`, `libblas.a`, `libmpich.a`,...). Estas librerías son las mismas que las utilizadas en la programación clásica mediante Fortran o C, por lo que el rendimiento de las mismas en ese nivel, a priori debería ser el mismo independientemente de la capa superior que haga uso de ella.

  Un detalle significativo en el proceso final de enlazado de librerías reside en la importancia de la nomenclatura de cada una de las rutinas. En el caso que una rutina (perteneciente a cualquiera de las librerías) hiciera referencia a otra rutina que no se ha incluido en la librería compartida, se generaría un error en tiempo de ejecución en el momento que deseamos importar la rutina. Debido a la complejidad residente en la librería `PyACTS.so` creada, dedicaremos la sección 5.7 para detallar el proceso de creación y diseño que se ha seguido.

En resumen, en la figura 5.1 podemos apreciar la diferenciación que se realiza entre los diferentes módulos (`PyBLACS`, `PyPBLAS`, `PyScaLAPACK`, `PySuperLU`, entre otros) y a su vez la integración de todos ellos dentro de un mismo paquete denominado `PyACTS`, donde los conceptos de módulo y paquete han sido introducidos en el apartado 3.1.3 y 3.1.4, respectivamente. A la vez, observamos cómo el núcleo de las operaciones se ha de realizar en los niveles inferiores por lo que teóricamente el rendimiento de esta herramienta debería ser muy similar o, como mucho, ligeramente inferior debido a la penalización introducida en los niveles intermedios.

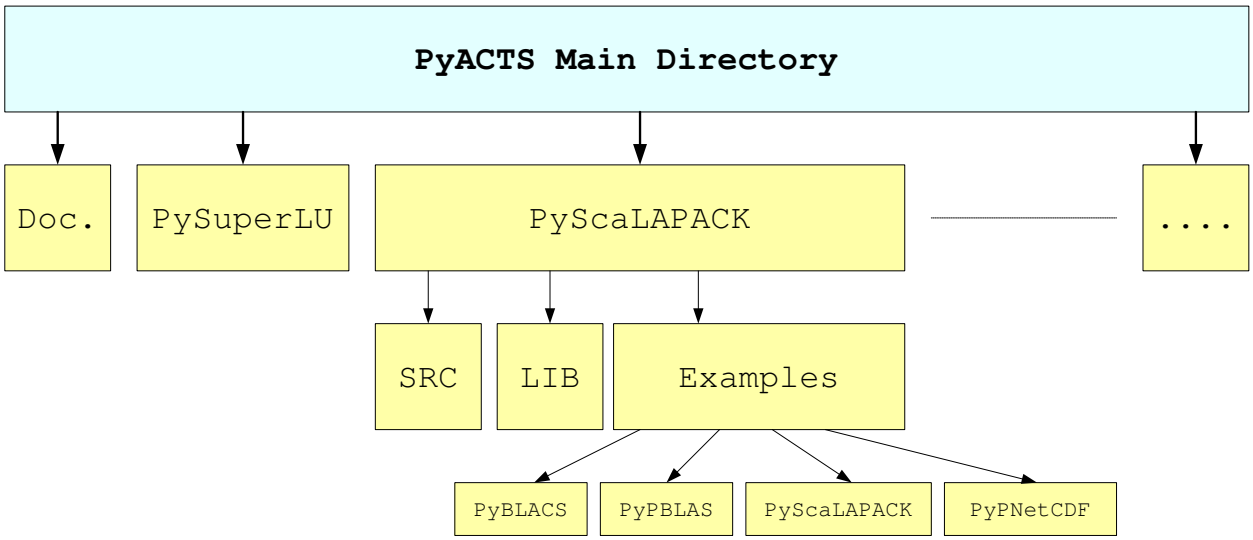


Figura 5.2: Sistema de archivos y directorios en la distribución PyACTS

La vista conceptual tiene una importante relación con la distribución del paquete en archivos y directorios. Este sistema de archivos se representa en la figura 5.2. En ella observamos que existe una carpeta raíz denominada PyACTS, y dentro de ella diferentes carpetas identificadas por cada librería de la colección ACTS. De este modo, por ejemplo, las interfaces de PyScaLAPACK residen en un archivo `.so` diferente a las interfaces de PySuperLU. A su vez, en el directorio raíz reside un archivo de configuración único `setup.py` mediante el cual se puede configurar y seleccionar los paquetes a instalar. Los detalles de configuración de PyACTS y el proceso de instalación serán tratados en la sección 5.7 de este capítulo.

Dentro de cada paquete, se observa una estructura de carpetas que deberá ser homogénea en todos ellos. Detallamos a continuación esta estructura:

- Carpeta `SRC`

  Contiene las fuentes de las interfaces de más bajo nivel, conocidas como *wrapers* (descritas en el apartado 3.3). Estas interfaces están escritas en C y utilizan archivos de inclusión (`.h`) que se encuentran en la misma carpeta. El aspecto de estos archivos sigue la definición de creación de interfaces marcada por Python [30], y debido al elevado número de rutinas (por ejemplo, los módulos PyBLACS, PyPBLAS y PyScaLAPACK suman un total de 223) a las que se han de crear las interfaces, se han utilizado las herramientas SWIG y F2PY descritas en el apartado 3.3.1 y 3.3.2

respectivamente.

- Carpeta `LIB`

  Contiene las fuentes en Python de cada uno de los módulos creados (`PyBLACS.py`, `PyPBLAS.PY`, `PyScaLAPACK.py`, entre otros). Tal y como se aprecia en la figura 5.1, estas interfaces se corresponden con el nivel superior y ofrecen un aspecto más sencillo y cómodo al usuario final.

- Carpeta `EXAMPLES`

  En esta carpeta agruparemos diferentes ejemplos de utilización de las rutinas de cada paquete. La carpeta de ejemplos correspondiente al paquete PyScaLAPACK está organizada, a su vez, en subcarpetas dependiendo de los diferentes módulos (PyBLACS, PyPBLAS y PyScaLAPACK) a los que hace referencia el objeto.

En este punto llama la atención un detalle. Tal y como se ha indicado, dentro de la carpeta PyScaLAPACK se encuentran las fuentes en C y en Python de los módulos PyBLACS, PyPBLAS y PyScaLAPACK cuando lo lógico hubiera sido establecer diferentes carpetas puesto que se tratan como módulos diferentes dentro del mismo paquete. Sin embargo, en función de la estructura de librerías de ScaLAPACK (vista en el apartado 4.3.5), para poder hacer uso de ScaLAPACK necesitamos enlazar sus rutinas con rutinas de PBLAS y éstas a su vez hacen uso de BLACS. Por tanto, para no repetir las rutinas como módulos diferentes, se han integrado todas como un mismo archivo de librerías compartidas (interfaces a Python de bajo nivel, es decir, un mismo archivo `.so`), mientras que las interfaces a Python de alto nivel sí se encuentran separadas. Es por este motivo por el cual, se integran los módulos PyBLACS, PyPBLAS y PyScaLAPACK dentro de la carpeta `PyScaLAPACK`.

5.3. La matriz PyACTS

Para facilitar la integración de los diferentes módulos dentro del mismo paquete y la incorporación de nuevos módulos que integren nuevas librerías numéricas, hemos definido una nueva clase de objeto en Python. A esta nueva clase la hemos querido denominar

"matriz PyACTS". Por tanto, en el resto de este trabajo identificaremos con este nombre a la matriz que hemos definido para su utilización en el paquete PyACTS.

En este sentido, desde el punto de vista de la programación orientada a objetos definimos el objeto matriz PyACTS con las siguientes propiedades:

- `data`: Contiene a su vez una matriz de tipo `Numeric Python` introducida en el apartado 3.2.1. Se ha de tener en cuenta que el contenido de esta matriz tiene carácter local y por tanto cada proceso almacena en esta matriz la parte de datos de la matriz global que le corresponde. El contenido de esta matriz depende de la propiedad `lib` que se describe a continuación.

- `lib`: Un entero que identifica la librería ACTS para la cual están preparados los datos contenidos en la propiedad `data`. Se ha de tener en cuenta que diferentes librerías pertenecientes a ACTS, pueden tener diferente definición de datos. Por ejemplo, los datos no se almacenan en memoria del mismo modo si estamos ante un problema de álgebra con matrices densas o con matrices dispersas. Debido a que en este trabajo de tesis se presenta el proyecto ACTS y por tanto se desarrollan los primeros módulos de dicho paquete, hemos establecido un identificador para cada distribución de la colección ACTS. Los valores establecidos son los siguientes

  - `lib=1`: Los datos contenidos en la propiedad `data` siguen una distribución cíclica en dos dimensiones utilizando los parámetros definidos en el sistema (`MB`, `NB`, `nprow`, `nprow`). Este tipo de distribución es la utilizada en librerías como ScaLAPACK y PBLAS.

  - `lib=501`: Los datos se encuentran distribuidos conforme a una distribución por bandas utilizada en ScaLAPACK y que se detalla en mayor profundidad en el apartado 8.2.1.

  - `lib=502`: Los datos se encuentran distribuidos conforme a una distribución denominada *right-hand side vector* utilizada en ScaLAPACK de forma general para las matrices de términos independientes junto con las matrices en bandas y que se detallan en el apartado 8.2.1.

  - `lib=`x: Del mismo modo que para ScaLAPACK, PBLAS o BLACS, a medida que incorporemos nuevas librerías al paquete PyACTS, se establecerán nuevos valores válidos para esta propiedad.

- `desc`: Corresponde con una matriz de tamaño fijo denominada descriptor. Generalmente, el descriptor se utiliza en este tipo de librerías para que los procesos almacenen atributos globales de la variable como por ejemplo el tamaño global. De modo especifico para el caso de la librería PyScaLAPACK, esta propiedad sigue las mismas convenciones que ScaLAPACK y ya fueron descritas en el apartado 4.3.5.

Para mostrar un ejemplo de una forma visual, debemos recordar el ejemplo de la distribución cíclica 2D descrito en el apartado 4.3.5. Recordamos que en este ejemplo, para poder ejecutar una rutina de ScaLAPACK o PBLAS, previo a la ejecución de dicha rutina, el usuario debe distribuir los datos entre los procesos a partir de la configuración de la malla de procesos `nprow` × `npcol` y del tamaño de bloque definido `MB` × `NB`. Aunque en el entorno clásico de programación (Fortran o C) no se proporcionan rutinas para realizar esta distribución considerándose tarea del programador, en el paquete PyACTS hemos considerado adecuado proporcionar herramientas que permitan realizar este tipo de distribuciones de forma transparente al usuario.

A continuación, mostramos un ejemplo en la donde se expone la conversión de una matriz de tipo `Numpy` a una matriz PyACTS y posteriormente se muestran las propiedades de la matriz PyACTS a. En este ejemplo observamos cómo sólo un procesador, aquel con la variable boolean `PyACTS.iread` verdadera será el origen de los datos, mientras en el resto se define una variable con contenido vacío (`None` en Python es un valor similar a Null en otros lenguajes de programación como C).

```
1  import PyACTS
2  import Numeric
3  n=8
4  PyACTS.gridinit(nb=2)
5  ACTS_lib=1                              # ScaLAPACK library
6  if PyACTS.iread==1:
7        a=Numeric.reshape(range(n*n),[n,n])
8  else:
9        a=None
10 a=PyACTS.Num2PyACTS(a,ACTS_lib)     # convert array to PyACTS array
11 print "PyACTS Array Properties in [",PyACTS.myrow,",",PyACTS.mycol,"]"
12 print "          lib=",a.lib
13 print "          desc=",a.desc
14 print "          data=",a.data
```

La función `Num2PyACTS` será la encargada de convertir una variable de tipo `Numpy` (sólo en el proceso donde `PyACTS.iread=1`) en una matriz PyACTS global, cuyo contenido de datos ya se encuentra distribuido conforme a la distribución cíclica 2D configurada en el entorno BLACS. Una vez ejecutada la llamada a la rutina de conversión, ésta devuelve un objeto de tipo matriz PyACTS, con las propiedades descritas anteriormente. A continuación, mostramos el resultado obtenido de la ejecución de este código en 4 procesos. Podemos observar cómo cada proceso está identificado dentro de la malla de procesos mediante los valores (`myrow`, `mycol`), y cada uno de ellos tiene definida la variable `a`. En cada proceso, `a` tiene las tres propiedades descritas, donde el descriptor y el indicador de la librería son iguales, sin embargo la propiedad `data` es diferente en cada proceso y atiende a la distribución cíclica 2D descrita en el apartado 4.3.5.

```
vgaliano@rho$ mpirun -np 4 mpipython objetoPyACTS.py
PyACTS Array Properties in [ 0 , 0 ]
       lib= 1
       desc= [1 0 8 8 2 2 0 0 4]
       data= [ 0  8 32 40  1  9 33 41 4 12 36 44  5 13 37 45]
PyACTS Array Properties in [ 1 , 1 ]
       lib= 1
       desc= [1 0 8 8 2 2 0 0 4]
       data= [18 26 50 58 19 27 51 59 22 30 54 62 23 31 55 63]
PyACTS Array Properties in [ 0 , 1 ]
       lib= 1
       desc= [1 0 8 8 2 2 0 0 4]
       data= [ 2 10 34 42  3 11 35 43  6 14 38 46  7 15 39 47]
PyACTS Array Properties in [ 1 , 0 ]
       lib= 1
       desc= [1 0 8 8 2 2 0 0 4]
       data= [16 24 48 56 17 25 49 57 20 28 52 60 21 29 53 61]
```

Como se ha podido comprobar en este ejemplo, en un número reducido de líneas, hemos inicializado la malla de procesos, hemos creado una matriz en un proceso y la hemos distribuido entre todos los procesos que intervienen en la malla de una manera sencilla y cómoda para el programador. De este modo, mediante la definición de la matriz PyACTS podemos manejar las variables de una manera cómoda para que sus datos sean tratados en niveles inferiores por las rutinas incluidas en la colección ACTS.

5.4. mpipython o pyMPI

Durante la introducción a Python (capítulo 3) se introdujo éste como un lenguaje interpretado de alto nivel y de carácter secuencial, esto significa que el intérprete de Python no está implementado para su ejecución en múltiples procesos en paralelo. Sin embargo, una de las características de Python es su extensibilidad gracias a que su código fuente se encuentra abierto y disponible para posteriores modificaciones.

Por tanto, para poder ejecutar un código escrito en Python en paralelo en varios procesos de memoria distribuida tendremos que adaptar el intérprete de Python a un sistema paralelo que incorpore el intercambio de mensajes entre los procesos. En los apartados 3.2.2 y 3.2.3 se introdujo ScientificPython y pyMPI respectivamente. Ambos partían del intérprete de Python y extendían la funcionalidad de comunicación mediante paso de mensajes a través del estándar MPI. Tanto ScientificPython como pyMPI inicializan el entorno MPI cuando se inicializa el intérprete de Python, consiguiendo una ejecución en paralelo en el numero de procesos indicados mediante el parámetro correspondiente (por ejemplo, `mpirun -np 4 pyMPI`). En el caso de la distribución de ScientificPython, el intérprete de Python se encuentra en uno de sus módulos, y su versión ejecutable se denomina mpipython.

Un usuario puede estar interesado en un intérprete de Python que implemente la funcionalidad MPI, sin embargo puede que no necesite del resto de funcionalidades y rutinas que incorpora el paquete ScientificPython. En este sentido, hemos aislado una distribución de mpipython mucho más sencilla y que se distribuye como herramienta adicional del paquete PyACTS. De este modo, si un usuario quiere hacer uso de PyACTS, tiene dos opciones: utilizar pyMPI o utilizar mpipython que proporcionamos y que está basado en el paquete ScientificPython.

Por tanto, observamos como actualmente existen dos implementaciones del intérprete de Python que implementan la funcionalidad del estándar de paso de mensajes MPI. En este punto, es lógico el planteamiento de la siguiente pregunta: si existen dos implementaciones, ¿cual de ellas deberemos utilizar? ¿mpipython o pyMPI?

Durante el desarrollo del presente trabajo, han sido múltiples las veces en las que nos hemos planteado esta cuestión. En los siguientes puntos trataremos de clarificar las ventajas e inconvenientes de cada opción y obtendremos conclusiones al respecto:

- pyMPI es una herramienta más utilizada y conocida que mpipython y en algunos de los sistemas de computación (por ejemplo Seaborg), ya se encuentra instalado en el sistema por los propios administradores.

- pyMPI proporciona mayor versatilidad en cuanto a las funciones y objetos que se pueden manejar del estándar MPI. Sin embargo, para la ejecución de PyACTS no accedemos a las funciones de MPI desde su interfaz de Python sino desde niveles superiores por lo que no son necesarias tantas funcionalidades si tratamos pyMPI como una herramienta para ejecutar rutinas de PyACTS.

- Debido a esa mayor versatilidad, pyMPI es más complejo y pesado (en cuanto a requisitos de memoria y espacio en el sistema de archivos, 283 Kbytes). La distribución mpipython es mucho más ligera (10 Kbytes) y sencilla en su instalación y ofrece todas las funcionalidades necesarias requeridas para poder ejecutar PyACTS correctamente.

- En función de esa mayor sencillez, hemos conseguido instalar mpipython en todas las plataformas probadas, mientras que con pyMPI hemos encontrado bastantes dificultades en algunas plataformas durante el proceso de instalación.

- Un punto importante a tener en cuenta es el rendimiento obtenido en función de uno u otro intérprete de Python. En la figura 5.3 se muestran algunas comparaciones entre los tiempos de ejecución obtenidos al ejecutar rutinas de PyPBLAS y de PyScaLAPACK con pyMPI y con mpipython.

  En las figuras 5.3(a) y 5.3(b) se ha realizado una llamada a las rutinas `pvaxpy` (nivel 1: $x+\alpha y,\ \alpha \in \mathbb{R},\ x, y \in \mathbb{R}^n$), y `pvgemm` (nivel 3: $\alpha AB+\beta C,\ \alpha, \beta \in \mathbb{R},\ A, B, C \in \mathbb{R}^{n\times n}$) respectivamente. En ambas no se detecta una diferencia significativa entre los tiempos de ejecución obtenidos al utilizar pyMPI y mpipython para diferentes configuraciones de mallas de procesos. Cabe mencionar que para configuraciones de malla de procesos 1×1 y 2×1 y a partir de un tamaño de los vectores de $6 \cdot 10^7$ elementos no se ha podido realizar la ejecución debido a que el sistema no dispone de la suficiente memoria para almacenar los elementos. A medida que aumenta el numero de procesos, se dispone de mayor memoria distribuida para almacenar vectores de tamaño superior.

  En las figuras 5.4(a) y 5.4(b) se muestran los resultados obtenidos de las pruebas realizadas con las rutinas de PyScaLAPACK `pvgesv` (resolución de un sistema de

ecuaciones) y `pvgesvd` (descomposición en valores y vectores singulares de una matriz). Al igual que ocurre con las rutinas de PyPBLAS, no se observa una diferencia notable en la ejecución de las rutinas utilizando uno u otro intérprete de Python para ninguna de sus configuraciones de procesos.

Estas pruebas mostradas confirman que la función principal de pyMPI o mpipython es la de inicializar el entorno MPI en todos los procesos implicados dejando las funciones de paso de mensajes a niveles superiores como PBLAS.

En resumen, podemos concluir que ambas herramientas son válidas para su utilización con el paquete PyACTS y la elección entre ambas puede depender de factores como la sencillez y facilidad de instalación (donde elegiríamos mpipython), o la funcionalidad adicional de MPI desde Python (para la que habría que utilizar pyMPI).

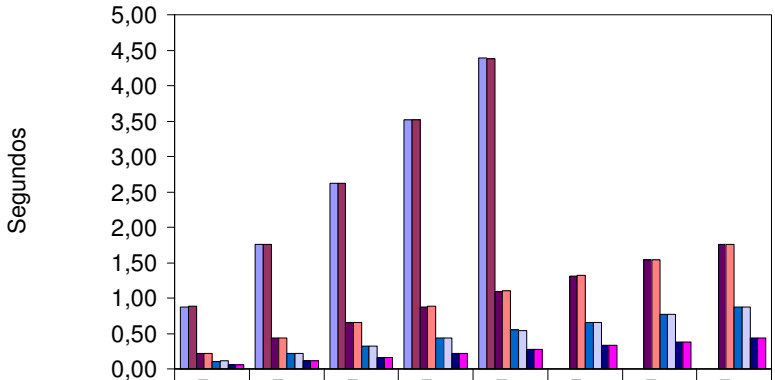
5.5. Funciones auxiliares

Tal y como se ha descrito hasta el momento, PyACTS permite acceder a las rutinas incluidas en las librerías ACTS desde el lenguaje interpretado Python. Sin embargo, para que sea un paquete completo deberemos proporcionar rutinas que realicen y verifiquen los pasos necesarios para la correcta ejecución de las rutinas incluidas en la colección ACTS. De este modo, dentro del paquete PyACTS, utilizaremos un conjunto de rutinas que nos serán útiles para inicializar, consultar y liberar los recursos necesarios para poder ejecutar las rutinas incluidas en el paquete tanto en PyBLACS, PyPBLAS y PyScaLAPACK.

A continuación se comentan las principales rutinas auxiliares incluidas en el paquete PyACTS, sin embargo, si se desea mayor información de las mismas, una descripción más detallada de los parámetros de entrada y salida, así como ejemplos de utilización, se recomienda consultar el manual de usuario de PyACTS que acompaña a este trabajo [34].

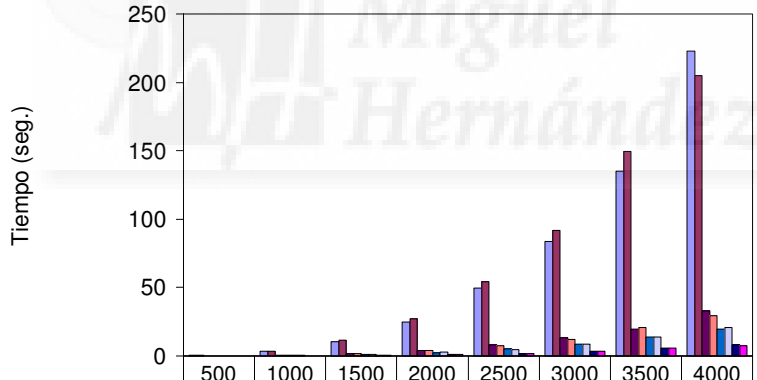
Estas funciones auxiliares pueden ser agrupadas atendiendo a su funcionalidad en los siguientes grupos:

- Inicialización y liberación de recursos.
- Verificación de variables PyACTS.



	1,E+07	2,E+07	3,E+07	4,E+07	5,E+07	6,E+07	7,E+07	8,E+07
mpipython 1X1	0,872	1,759	2,617	3,519	4,395			
pyMPI 1X1	0,880	1,755	2,617	3,513	4,376			
mpipython 4X1	0,220	0,435	0,653	0,878	1,088	1,306	1,536	1,755
pyMPI 4X1	0,220	0,440	0,660	0,880	1,100	1,317	1,536	1,755
mpipython 8X1	0,109	0,220	0,327	0,439	0,549	0,653	0,768	0,871
pyMPI 8X1	0,110	0,219	0,326	0,439	0,544	0,658	0,768	0,878
mpipython 16X1	0,055	0,110	0,163	0,220	0,272	0,329	0,384	0,436
pyMPI 16X1	0,055	0,110	0,163	0,220	0,274	0,329	0,384	0,435

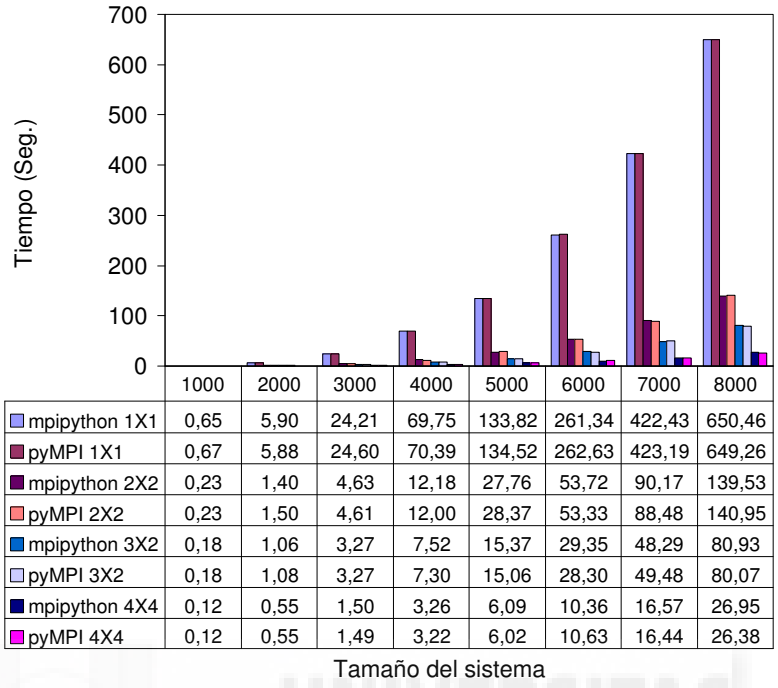
(a) Rutina pvaxpy de Nivel 1



	500	1000	1500	2000	2500	3000	3500	4000
mpipython 1X1	0,40	3,51	10,66	24,84	49,43	83,71	135,37	222,60
pyMPI 1X1	0,44	3,19	11,79	27,10	54,49	91,53	149,61	204,83
mpipython 2X2	0,05	0,59	1,72	3,97	7,91	13,05	19,58	32,85
pyMPI 2X2	0,05	0,50	1,70	3,82	7,24	12,35	20,77	29,43
mpipython 3X2	0,04	0,32	1,10	2,59	5,07	8,64	13,63	19,62
pyMPI 3X2	0,07	0,35	1,20	2,68	4,85	8,94	13,74	20,59
mpipython 4X4	0,03	0,13	0,44	1,03	1,91	3,64	5,51	7,87
pyMPI 4X4	0,03	0,13	0,46	1,00	1,99	3,57	5,54	7,73

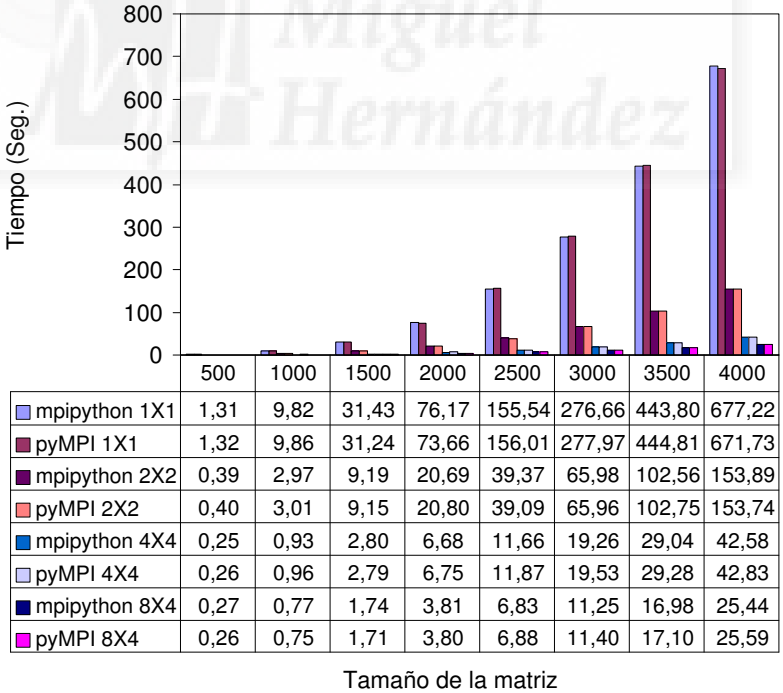
(b) Rutina pvgemm de Nivel 3

Figura 5.3: Comparación entre mpipython y pyMPI para PyPBLAS



	1000	2000	3000	4000	5000	6000	7000	8000
mpipython 1X1	0,65	5,90	24,21	69,75	133,82	261,34	422,43	650,46
pyMPI 1X1	0,67	5,88	24,60	70,39	134,52	262,63	423,19	649,26
mpipython 2X2	0,23	1,40	4,63	12,18	27,76	53,72	90,17	139,53
pyMPI 2X2	0,23	1,50	4,61	12,00	28,37	53,33	88,48	140,95
mpipython 3X2	0,18	1,06	3,27	7,52	15,37	29,35	48,29	80,93
pyMPI 3X2	0,18	1,08	3,27	7,30	15,06	28,30	49,48	80,07
mpipython 4X4	0,12	0,55	1,50	3,26	6,09	10,36	16,57	26,95
pyMPI 4X4	0,12	0,55	1,49	3,22	6,02	10,63	16,44	26,38

(a) Rutina pvgesv



	500	1000	1500	2000	2500	3000	3500	4000
mpipython 1X1	1,31	9,82	31,43	76,17	155,54	276,66	443,80	677,22
pyMPI 1X1	1,32	9,86	31,24	73,66	156,01	277,97	444,81	671,73
mpipython 2X2	0,39	2,97	9,19	20,69	39,37	65,98	102,56	153,89
pyMPI 2X2	0,40	3,01	9,15	20,80	39,09	65,96	102,75	153,74
mpipython 4X4	0,25	0,93	2,80	6,68	11,66	19,26	29,04	42,58
pyMPI 4X4	0,26	0,96	2,79	6,75	11,87	19,53	29,28	42,83
mpipython 8X4	0,27	0,77	1,74	3,81	6,83	11,25	16,98	25,44
pyMPI 8X4	0,26	0,75	1,71	3,80	6,88	11,40	17,10	25,59

(b) Rutina pvgesvd

Figura 5.4: Comparación entre mpipython y pyMPI para PyScaLAPACK

- Consulta y generación de variables PyACTS.
- Conversión de formatos PyACTS.

En los siguientes apartados damos una breve descripción de las rutinas englobadas en cada uno de estos grupos.

5.5.1. Inicialización y liberación

Estas rutinas han de ser ejecutadas para inicializar o liberar recursos y deberán estar presentes de forma anterior y posterior (respectivamente) a la ejecución de las rutinas de los módulos PyBLACS, PyPBLAS y PyScaLAPACK.

- `gridinit([file_config='',mb,nb,nprow,npcol])`

  Esta rutina se encarga de inicializar la malla de procesos en función de los parámetros de entrada indicados. Éstos son opcionales y en su ausencia, se opta por configurar una malla lo mas cuadrada posible a partir del número de procesos con los que se ejecuta `mpirun -np numprocs`, donde `numprocs` hace referencia al número de procesos que ejecutan el script en paralelo. A continuación mostramos algunos ejemplos de configuraciones automáticas a partir de un número de procesos:

  - Si `numprocs=4` entonces la malla de procesos es 2×2.
  - Si `numprocs=5` entonces la malla de procesos es 2×2.
  - Si `numprocs=6` entonces la malla de procesos es 3×2.
  - Si `numprocs=7` entonces la malla de procesos es 3×2.

  Por otro lado, es posible que el usuario desee una configuración de la malla diferente a la obtenida por defecto. En ese caso, podrá indicarse mediante el uso de los parámetros adicionales `nprow` y `npcol`. Mediante estos parámetros se fuerza a que la configuración de la malla tenga las dimensiones indicadas mediante uno o ambos parámetros. A continuación, mostramos algunos ejemplos de configuración de la malla de procesos.

  - `numprocs=4` y `gridinit(nprow=1)`: 1×4.

- `numprocs=4` y `gridinit(nprow=2)`: 2×2.
- `numprocs=4` y `gridinit(nprow=4)`: 4×1.
- `numprocs=5` y `gridinit(nprow=3)`: 3×1.
- `numprocs=5` y `gridinit(nprow=3,npcol=2)`: Informa de un error.

Los parámetros de configuración de la malla se corresponden con los descritos en el apartado 4.3.3 en el que se hace referencia a la librería BLACS. A pesar de no tener parámetros de salida, esta función establece los valores de las siguientes variables en el modulo PyACTS y que serán utilizadas por el conjunto de rutinas incluidas en el mismo: `PyACTS.mb`, `PyACTS.nb`, `PyACTS.nprow`, `PyACTS.npcol`, `PyACTS.myrow`, `PyACTS.mycol`, `PyACTS.ictxt`. Dichos parámetros guardan estrecha relación con los parámetros descritos en el apartado 4.3.5.

- `gridexit()`

  Esta rutina se encarga de liberar los recursos y la malla de procesos una vez realizados los cálculos. Esta rutina es muy sencilla y no posee ningún parámetro de entrada ni de salida. Se ha de tener en cuenta que una vez liberados los recursos no podremos realizar nuevas llamadas a las rutinas de comunicación de datos de PyBLACS, o rutinas de PyPBLAS o PyScaLAPACK a no ser que volvamos a inicializar una malla mediante `gridinit`.

5.5.2. Verificación de variables PyACTS

De forma previa a la ejecución de la propia rutina de PyBLACS, PyPBLAS o PyScaLAPACK, se debe comprobar que los parámetros utilizados tienen el tipo adecuado para su ejecución. En este apartado, incluimos aquellas rutinas que comprueban si los parámetros o variables a utilizar por las rutinas tienen el formato adecuado, que generalmente debería ser una matriz o escalar PyACTS.

- `IsACTSlib(ACTS_var)`

  Devuelve una expresión booleana que indica si la variable pasada como parámetro, `ACTS_var`, tiene el formato PyACTS válido.

5.5.3. Consulta y generación de variables PyACTS

Una vez definido el formato de la matriz PyACTS, puede ser necesario consultar o generar este tipo de variables a partir de una determinada información. A continuación incluimos un conjunto de rutinas que creemos útiles para el tratamiento de las variables PyACTS.

- `readACTSdesc(ACTSArray)`

  Devuelve el descriptor de la variable pasada como parámetro `ACTSArray`.

- `desc=ACTSgetdesc(a[,ACTS_lib,ia,ja])`

  Esta rutina crea un descriptor `desc` de la matriz global `a` de tipo `Numeric` pasada como parámetro para la librería indicada por el valor `ACTS_lib`. Este descriptor podrá ser utilizado para construir la matriz PyACTS o bien para crear una nueva matriz distribuida a partir de su descriptor.

- `desc=ACTSmakedesc(m,n[,ACTS_lib,ia,ja])`

  Mientras que la rutina `ACTSgetdesc` creaba un descriptor a partir de una matriz de tipo `Numeric` dada, la rutina `ACTSmakedesc` obtendrá un descriptor a partir del tamaño de la matriz indicado mediante los parámetros de entrada `m` y `n`.

  Se ha de tener en cuenta que estas dos últimas rutinas son muy similares. Así, las siguientes llamadas devolverían el mismo resultado, siendo `a` una matriz de tipo `Numeric`.

```
1 desc=ACTSgetdesc(a)
2 desc=ACTSmakedesc(a.shape)
```

  En la primera línea se devuelve el descriptor de una matriz pasada por parámetro. En la línea 2, pasamos las dimensiones a la rutina que en este caso, pertenecen a la matriz `a`.

- `m,n=getdims(x)`

  La rutina `getdims` obtiene la pareja de dimensiones de una matriz que reside en un único proceso (aquel con `PyACTS.iread==1`) y la distribuye a todos los procesos de una malla. Esta rutina proporciona una forma sencilla de poder conocer el valor de una determinada matriz que todavía no ha sido distribuida ni creado su

descriptor.De este modo, todos los procesos pueden obtener de una forma sencilla el tamaño de la matriz a distribuir:

```
1 #Send and receive the dims of the array
2 m, n = getdims(a)
```

- `mb, nb, nprow, npcol = readconfig( iam, nprocs, file_config, mb0, nb0, nprow0, npcol0)`
  La rutina `readconfig` obtiene la configuración de la malla de procesos desde un archivo de configuración que indicaremos mediante `file_config` o bien la calcula a partir del número de procesos disponibles. En el caso que la calcule de forma automática, se buscará una matriz de procesos lo más cuadrada posible y que agrupe el mayor número posible de procesos con los que se ha iniciado la ejecución, tal y como se ha explicado en la rutina `gridinit`.

5.5.4. Conversión de variables PyACTS

- `a_acts=Num2PyACTS(a,ACTS_lib,[bwu,bwl,piv])`

  Esta rutina será muy utilizada en la mayor parte de los ejemplos de los capítulos posteriores en los que introduciremos ejemplos de PyBLACS, PyPBLAS y PyScaLAPACK. La rutina `Num2PyACTS` devuelve una variable de tipo matriz PyACTS a partir de una variable de tipo `Numeric` y del indicador de la librería ACTS que deseamos aplicar.

  Se ha de tener en cuenta que esta función, de forma interna, realiza las siguientes acciones:

  1. A partir de la matriz `Numeric`, calcula y distribuye sus dimensiones desde el proceso origen (aquel con `PyACTS.iread=1`) al resto de nodos de la malla de procesos.
  2. Cada proceso crea las matrices locales de las dimensiones adecuadas conforme a la distribución utilizada para recibir y almacenar los datos.
  3. Se crea el descriptor en cada proceso.

4. Se distribuyen los datos de acuerdo a ese descriptor.
5. Se crea la matriz PyACTS (instancia del objeto definido como `ACTSArray`) que contiene las tres propiedades: `ACTS_lib`, `data` y `desc`.

Los parámetros opcionales `bwu,bwl,piv` se utilizan para la conversión de matrices `Numeric` a matrices PyACTS en bandas, cuya distribución es diferente a la cíclica 2D y será descrita en el apartado 8.2.1.

- `a=PyACTS2Num(a_acts)`

  Esta rutina se puede considerar la inversa de la rutina `Num2PyACTS` y también será utilizada en la mayor parte de los ejemplos de los capítulos posteriores. La rutina `PyACTS2Num` devuelve una variable de tipo `Numeric` a partir de una variable de tipo `PyACTS`.

  Se ha de tener en cuenta que esta función, de forma interna, realiza las siguientes acciones:

  1. A partir de la variable PyACTS y por tanto de su descriptor, se crea en el proceso con `PyACTS.iread=1` una matriz de las dimensiones de la matriz total.
  2. Todos los procesos envían al proceso `PyACTS.iread=1` los datos y éste los copia en sus índices correspondientes.

- `a_acts=Txt2PyACTS(file_in,ACTS_lib)`

  En esta rutina, el proceso `PyACTS.iread=1` realiza la lectura de los datos desde el fichero `file_in` y los distribuye conforme a una distribución indicada por `ACTS_lib`. Debido a la gran variedad de formatos aplicables a los datos en el fichero de texto, en esta rutina hemos utilizado los formatos de fichero utilizados por la librería `Numpy` [4].

  Se ha de tener en cuenta que esta función, de forma interna, realiza las siguientes acciones:

  1. El proceso origen (aquel con `PyACTS.iread=1`) lee el fichero y lo almacena en memoria en una matriz `Numeric`.
  2. A partir de la matriz `Numeric`, calcula y distribuye sus dimensiones desde el proceso origen (aquel con `PyACTS.iread=1`) al resto de nodos de la malla de procesos.

3. Cada proceso crea las matrices locales de las dimensiones adecuadas conforme a la distribución utilizada.
4. Se crea el descriptor en cada proceso.
5. Se distribuyen los datos de acuerdo a ese descriptor.
6. Se crea la matriz PyACTS.

- `PyACTS2Txt(a_acts,file_out)`

  Esta rutina se puede considerar la inversa de la rutina `Txt2PyACTS` y realiza la escritura en un único fichero de una matriz PyACTS que se encuentra distribuida entre los elementos de una malla de procesos.

  Se ha de tener en cuenta que esta función, de forma interna, realiza las siguientes acciones:

  1. A partir de la variable PyACTS y por tanto de su descriptor, se crea en el proceso con `PyACTS.iread=1` una matriz de las dimensiones de la matriz total.
  2. Todos los procesos envían al proceso `PyACTS.iread=1` los datos y éste los copia en sus índices correspondientes.
  3. A partir de la variable `Numeric` se realiza la escritura a un fichero mediante las rutinas que dichos objetos proporcionan.

- `a_acts=Rand2PyACTS(m,n,ACTS_lib)`

  Esta rutina crea una variable de tipo PyACTS donde los datos creados son aleatorios y el tamaño global de la misma es $m \times n$. Se ha de tener en cuenta, que en esta rutina la creación de las datos aleatorios se realiza en cada proceso, por lo que no se realiza distribución de datos. La única distribución que se realiza es el tamaño global de la matriz a crear, $m \times n$. A partir de este tamaño, cada proceso calculará el tamaño de su matriz local que dependerá de su configuración dentro de la malla de procesos.

  De forma interna, se realizan las siguientes acciones:

  1. Se crea el descriptor en cada proceso.
  2. Se distribuye el tamaño de la matriz a crear.
  3. Cada proceso crea las matrices locales aleatorias de las dimensiones calculadas en función de la configuración de la malla de procesos.

4. Se crea la matriz PyACTS con sus tres propiedades: `ACTS_lib`, `data` y `desc`.

- `alpha_acts=Scal2PyACTS(alpha,ACTS_lib)`

  Esta rutina crea una variable de tipo PyACTS donde el dato (que es un escalar) reside en la propiedad `data`. La utilidad de está función se centra en automatizar la distribución del escalar en todos los procesos que pertenecen a la configuración de la malla de procesos. Ejemplos en la utilización de esta rutina se pueden encontrar en los capítulos posteriores PyBLACS, PyPBLAS y PyScaLAPACK.

- `a_acts=PNetCDF2PyACTS(a_pnetcdf,ACTS_lib)`

  Para poder hacer uso de esta rutina es necesario tener instalado el paquete PyPnetCDF que será presentado en el capítulo 9. Una vez instalado este paquete correctamente, podremos hacer que interactúe con el paquete PyACTS mediante ésta y otras rutinas que explicaremos a continuación. La principal finalidad de esta rutina es la de obtener los datos desde una instancia de un objeto PyPnetCDF para conseguir una variable de tipo PyACTS que nos permita utilizarla en las rutinas PyBLACS, PyPBLAS y PyScaLAPACK.

  La utilización de esta rutina trata de ser muy sencilla, sin embargo para poder comprender su funcionamiento será necesario también entender el funcionamiento básico del paquete PyPnetCDF. Debido a que el estándar netCDF se utiliza para almacenar matrices densas y no otro tipo de matrices como en bandas o dispersas, la conversión únicamente se realiza para `ACTS_lib=1`, es decir, para una distribución cíclica 2D.

  El proceso que se sigue en esta rutina será el siguiente:

  1. Cada proceso lee los parámetros de interés de la instancia PyPnetCDF para crear el descriptor.
  2. Se calcula el tamaño de las matrices locales conforme a una distribución cíclica 2D, a la configuración de la malla de procesos y al tamaño de la matriz global, obtenido de las dimensiones de la variable PyPnetCDF.
  3. Cada proceso lee de forma independiente los datos situados en los índices correspondientes en función de la distribución cíclica 2D.
  4. Una vez leídos los datos, se les realiza una traspuesta a la matriz local para almacenar los datos por columnas. Conviene recordar que en Fortran el orden de las matrices utilizado es por columnas y en C por filas.

- `a_netcdf=PyACTS2PNetCDF(a_acts,a_netcdf)`

  Del mismo modo que en la rutina anterior, para poder hacer uso de esta rutina es necesario tener instalado el paquete PyPnetCDF. La principal finalidad de esta rutina es la de almacenar los datos desde una instancia de una matriz PyACTS a una instancia de tipo PyPnetCDF, para posteriormente guardar los datos. La utilización de esta rutina trata de ser muy sencilla, sin embargo para poder comprender su funcionamiento será necesario también entender el funcionamiento básico del paquete PyPnetCDF. A grandes rasgos, esta rutina realiza los siguientes pasos:

  1. Cada proceso realiza una traspuesta a la matriz local para obtener el orden de los datos por filas (conforme a C).
  2. Cada proceso escribe por bloques los datos en un único fichero de destino netCDF utilizando la librería PnetCDF (que se trata en el apartado 9.3) e indicando la posición y tamaño del bloque en la matriz global.

5.6. Creación de los módulos

En la sección 5.2 se introdujo la estructura básica de PyACTS y en la figura 5.1 se representaban los módulos de PyACTS como la interfaz de las librerías ACTS con el programador o usuario de Python. De este modo, el usuario de PyACTS puede utilizar cada uno de los módulos PyBLACS, PyPBLAS o PyScaLAPACK importándolos de forma independiente.

En cada uno de los módulos `PyBLACS.py`, `PyPBLAS.py` y `PyScaLAPACK.py` se definen un conjunto de rutinas que se describirán más detenidamente en los capítulos 6, 7 y 8, respectivamente. Sin embargo, en todos ellos se definen las rutinas mediante una estructura similar que describiremos a continuación mediante la definición de una de ellas, en este caso, `pvgesv` perteneciente a `PyScaLAPACK.py`.

```
1 import PyScaLAPACK_Tools as PySLK_tools #Scalapack Tools
2 import pyscalapack #Shared Objects file
3 from PyACTS import *
4 import Numeric
```

```
5
6  def  pvgesv(ACTS_a,ACTS_b,ia=1,ja=1,ib=1,jb=1):
7          """
8  b,info= PySLK.pvgesv(a,b)
9  PVGESV computes the solution to a (real/complex) system of linear
10 equations
11
12                          sub( A ) * X = sub( B ),
13
14 where sub( A ) = A(IA:IA+N-1,JA:JA+N-1) is an N-by-N distributed
15 matrix and X and sub( B ) = B(IB:IB+N-1,JB:JB+NRHS-1) are
16 N-by-NRHS distributed matrices.
17
18  The LU decomposition with partial pivoting and row interchanges is
19 used to factor sub( A ) as sub( A ) = P * L * U, where P is a permu-
20 tation matrix, L is unit lower triangular, and U is upper triangular.
21 L and U are stored in sub( A ). The factored form of sub( A ) is then
22 used to solve the system of equations sub( A ) * X = sub( B ).
23  """
24    try:
25      ACTS_vars_in=[ACTS_a,ACTS_b]
26      ACTS_inc_in=[ia,ja,ib,jb]
27      ACTS_Error=PySLK_tools.checkerrors(ACTS_vars_in,ACTS_inc_in)
28      if ACTS_Error<>"":
29        raise PySLKError(ACTS_Error)
30
31      PySLKtype=PySLK_tools.getcommontype([ACTS_a.data.typecode(),
32                ACTS_b.data.typecode()])
33
34      if PySLKtype=='s':
35        PvScaLAPACK=pyscalapack.psgesv
36      elif PySLKtype=='d':
37        PvScaLAPACK=pyscalapack.pdgesv
38      elif PySLKtype=='c':
39        PvScaLAPACK=pyscalapack.pcgesv
40      elif PySLKtype=='z':
41        PvScaLAPACK=pyscalapack.pzgesv
```

```
42        n=ACTS_a.desc[PyACTS.m_]
43        nrhs=ACTS_b.desc[PyACTS.n_]
44        info=0
45        dim_ipiv=ACTS_a.desc[PyACTS.lld_]+ACTS_a.desc[PyACTS.mb_]
46        ipiv=Numeric.zeros((dim_ipiv,1),'i')
47        ACTS_a.data,ipiv,ACTS_b.data,info=PvScaLAPACK(n,nrhs,ACTS_a.data,
48          ia,ja,ACTS_a.desc,ipiv,ACTS_b.data,ib,jb,ACTS_b.desc,info)
49    except PySLKError,Error:
50        ACTS_b.data,info=[],-1
51    return ACTS_b,info
```

En el inicio de cada módulo, se realiza la importación de otros módulos como las rutinas auxiliares definidas para PyScaLAPACK (línea 1), la librería de objetos compartidos `.so` que se explica en el apartado 5.7 (línea 2), el propio módulo PyACTS y las funciones definidas en el archivo `__init__.py` (línea 3) o las funciones de la librería Numeric descrita en el apartado 3.2.1 (línea 4).

Desde la línea 6 hasta la línea 51 se realiza la definición de la rutina `pvgesv`. En la línea 6 se define el nombre de la rutina (o método del módulo PyScaLAPACK) y los parámetros de entrada a dicha rutina. Aquellos parámetros de entrada que tienen asignado un valor mediante el signo "`=`" se corresponden con los parámetros opcionales y no será necesario indicarlos a menos que el usuario desee utilizar un valor diferente al asignado por defecto. Se observa la sencillez en la declaración de una rutina en Python, puesto que en la cabecera no se especifica el tipo de dato de las variables `ACTS_a` y `ACTS_b`.

En la la línea 7 y en la línea 23 observamos tres comillas que delimitan el texto de ayuda situado entre ambas líneas. Mediante estos delimitadores se indica al intérprete de Python el texto que debe mostrar en el caso que el usuario solicite ayuda mediante el comando `help(PyScaLAPACK.pvgesv)`.

En la línea 24 se utiliza la directiva de Python `try` para tratar los posibles errores que pudieran ocurrir en la línea 49. Como hemos comentado durante el presente capítulo, antes de realizar una llamada a una rutina de una librería de nivel inferior, debemos comprobar que los parámetros pasados tienen el formato adecuado. De este modo, en la línea 25 y 26 "empaquetamos" las variables introducidas en la rutina y las verificamos mediante el método `checkerrors()` definido en el módulo de rutinas auxiliares. En PyACTS, hemos definido una clase para tratar los posibles errores que pueden ocurrir y tratarlos o informar al usuario de dichos errores. Esta nueva clase se denomina `PySLKError`, y en el caso de saltar una excepción de este tipo se informa mediante un mensaje de acuerdo al error que

haya ocurrido. El diccionario de errores comprobados por la rutina `checkerrors()` y que son remitidos al usuario es el siguiente:

```
Mssg_Errors={
   "GridErr":" PyScaLAPACK grid has not been initialized. \n Please try to
         Initialize with 'gridinit()'",
   "ACTSArrayErr":" Input Arrays are not PyACTSArray Types.\n Please try to
         use convert functions.\n Read the convert functions section in the
         PyScaLAPACK Reference\n",
   "ACTSSLKArrayErr":" Input Arrays are not PyScaLAPACKArray Types.\n Please
         try to use convert functions.\n Read the convert functions section
         in the PyScaLAPACK Reference\n",
  "TypeErr":" Invalid Array Types for this operation.\n Convert the input
       data in the same Numeric type.\n Read the convert functions section
       in the PyScaLAPACK Reference\n",
  "ACTSNoGrid":"",
  "ACTSNoNetCDF_NoMatchSize":"Global shapes of data don't match in
         PNetCDF Variable and PyACTS Variable. Make global shapes be same.",
  "ACTSNoNetCDF_ProblemSize":"There has been a problem with local size of
       PyACTS and PNetCDF arrays. Please Make sure you are using correct
       dimensions."
  }
```

De este modo, si un usuario intenta ejecutar la rutina `pvgesv` sin haber inicializado la malla de procesos, se aborta la ejecución de la rutina y será informado mediante la excepción `GridErr`. Por el contrario, si los parámetros y el entorno paralelo es correcto, la rutina `checkerrors()` devolverá una cadena vacía y por tanto no saltará ninguna excepción.

En la línea 31 realizamos una llamada a la rutina `getcommontype` que devuelve el tipo de datos común a los pasados como variables de entrada. Esta rutina devuelve un caracter identificador del tipo de dato que será el utilizado para asignar cual de las rutinas dependientes del tipo de dato utilizaremos en niveles inferiores (líneas 34 a 41).

En este ejemplo, desde la línea 42 a la línea 46, se preparan los parámetros de entrada a la rutina de nivel inferior a partir de las matrices PyACTS. Además, se reservan y definen los espacios de memoria necesarios para trabajo de la rutina o para implementar la pivotación del algoritmo (lineas 45 y 46).

En las líneas 47 y 48 se realiza la llamada a la rutina de ScaLAPACK correspondiente al tipo de dato utilizado en las matrices PyACTS. Observamos que de los cuatro parámetros de salida obtenidos únicamente se devuelven en la línea 51, aquellos parámetros con datos útiles, en este caso, la matriz PyACTS que contiene la solución del sistema, `ACTS_b` y un entero `info` que identifica el resultado de la ejecución.

Esta estructura descrita se repite para la mayor parte de métodos implementados en PyBLACS, PyPBLAS y PyScaLAPACK y nos ha permitido crear una interfaz más sencilla al programador de Python que automatiza, verifica y simplifica muchas de las tareas necesarias en la programación de aplicaciones de alto rendimiento con las librerías de la colección ACTS.

5.7. Creación de la librería de objetos

En la sección 5.2 hemos descrito el aspecto y estructura del paquete PyACTS. Como parte de esta estructura se ha nombrado al archivo de librería de objetos compartidos `PyACTS.so`. En la presente sección, se trata de analizar y describir el proceso mediante el cual ha sido creada esta librería. Consideramos que la complejidad del mismo es elevada puesto que proporciona la interfaz para más de 425 rutinas escritas en C o Fortran.

Para cada una de estas rutinas, se ha de crear una serie de rutinas y funciones descritas en la sección 3.3. Sin embargo, el elevado número de rutinas que se desean incorporar al paquete PyACTS obliga a utilizar algún tipo de herramienta que permita automatizar y facilitar este proceso. En el citado apartado, se han descrito dos de las herramientas disponibles para crear este tipo de interfaces: SWIG y F2PY. En el presente apartado, describiremos cuál ha sido este proceso y cuándo hemos utilizado una u otra herramienta en función de sus características. En el presente trabajo se presentan dos paquetes principales, el primero de ellos, que hemos denominado PyACTS y se introduce en este capítulo. El segundo es el paquete PyPnetCDF y se introduce en el capítulo 9. Para cada uno de estos paquetes hemos utilizado una herramienta de automatización en la creación de interfaces diferente. En el caso de PyACTS, la herramienta utilizada ha sido F2PY, sin embargo para PyPnetCDF se prefirió utilizar SWIG por las razones que serán descritas en el capítulo dedicado a este paquete.

Por tanto, introduciremos en los siguientes apartados los pasos que hemos seguido para crear la librería de objetos compartidos resultante `PyACTS.so`.

5.7.1. Definición de las interfaces

Debido al elevado tamaño (3,4 MBytes de archivo C y 85000 líneas de código, aproximadamente) y complejidad del archivo de interfaces (distribuido como pyscalapackmodule.c) se ha preferido utilizar la herramienta F2PY descrita en el apartado 3.3.2. En dicho apartado se han descrito diversas formas de definir las interfaces de las funciones a las que se les desea crear una interfaz desde Python.

En el presente apartado describiremos cuál ha sido el proceso inicial que hemos seguido, el estado actual de estas interfaces y detallamos la forma adecuada de modificar o crear nuevas interfaces a rutinas incluidas en este paquete.

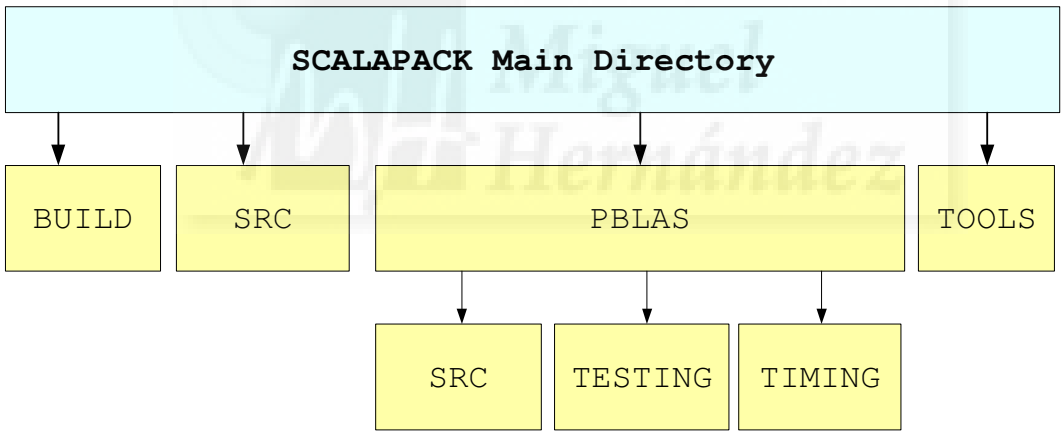


Figura 5.5: Sistema de directorios en la distribución ScaLAPACK

Para explicar el proceso de una forma ordenada, debemos recordar cual es el aspecto de la distribución ScaLAPACK. Se ha de tener en cuenta que las librerías ScaLAPACK, PBLAS y BLACS han sido las primeras en incorporarse a esta distribución de modo que nos centraremos en ellas para explicar el proceso. De este modo, recordamos que la distribución ScaLAPACK tiene la estructura de archivos y directorios mostrados en la figura 5.5. En esta estructura se observa que en el directorio `SCALAPACK/SRC` se encuentran

las fuentes de las rutinas de ScaLAPACK escritas en Fortran. Por otro lado, dentro de la carpeta `PBLAS` encontramos las fuentes de PBLAS (escritas en C), así como directorios con rutinas de prueba para PBLAS (`PBLAS/TESTING`, `PBLAS/TIMING`), y rutinas que implementan funcionalidades útiles para ScaLAPACK (`TOOLS`).

El proceso que se ha seguido para generar el archivo final ha sido el siguiente:

1. **Edición de cada rutina de BLACS, PBLAS y ScaLAPACK**

   A partir de los automatismos creados en F2PY y gracias a las facilidades que éste incorpora, editaremos cada uno de los archivos de cada rutina e introduciremos una cabecera que entenderá únicamente F2PY mientras que los compiladores de Fortran lo interpretan como comentarios.

   A modo de muestra, expondremos cómo hemos creado la interfaz para la rutina `PDGESV` incluida en ScaLAPACK. Para ello, editamos el archivo `pdgesv.f` en el que se encuentra la definición del código fuente de esta rutina. Como se observa bajo estas líneas, el archivo comienza con la declaración de la rutina, sin embargo se han añadido tres líneas que comienzan con la directiva de F2PY "`Cf2py`". En estas líneas se indica cuáles de los parámetros son de entrada, de salida o ambos, atendiendo a las diferentes clasificaciones descritas en el apartado 3.3.2.

```
1  SUBROUTINE PDGESV( N, NRHS, A, IA, JA, DESCA, IPIV, B,
2      $                 IB, JB, DESCB, INFO )
3  Cf2py intent(in)  N, NRHS, A, IA, JA, DESCA, IPIV, B,
4  Cf2py intent(in)  IB, JB, DESCB, INFO
5  Cf2py intent(out) A,IPIV,B,INFO
6
7  *
8  *  -- ScaLAPACK routine (version 1.7) --
9  *     University of Tennessee, Knoxville, Oak Ridge Nat.Lab.,
10 *     and University of California, Berkeley.
11 *     May 1, 1997
12 *
13 *     .. Scalar Arguments ..
14       INTEGER            IA, IB, INFO, JA, JB, N, NRHS
15 *     ..
16 *     .. Array Arguments ..
17       INTEGER            DESCA( * ), DESCB( * ), IPIV( * )
18       DOUBLE PRECISION   A( * ), B( * )
```

```
19 *        ..
```

Por tanto, esta operación de edición y modificación de las fuentes hay que llevarla a cabo con todas las rutinas de BLACS, PBLAS y ScaLAPACK a las que se desea crear una interfaz para Python. Se ha de observar que es una ventaja realizar este método puesto que nos permite comprobar y verificar de una forma más cómoda y directa las interfaces creadas con la propia definición de la rutina. Además, el propio F2PY interpreta y genera las interfaces basadas en la directiva introducida (por ejemplo, `Cf2py intent(in)`) y en la definición de tipos, puesto que no se trata del mismo modo la interfaz a un escalar que a una matriz.

2. **Generación del archivo de firmas** `pyscalapack.pyf`

   Una vez definidos de forma individual las interfaces a cada una de las rutinas, podremos generar el archivo de firmas en el que se refleja la definición de las interfaces que posteriormente creará F2PY. Para obtener este archivo de firmas, se ha creado un archivo `Makefile` que automatiza todo el proceso de generación y compilación de la interfaz. A continuación, mostramos la parte del código que inicia F2PY para que lea las fuentes situadas en los directorios indicados y genere el archivo de firmas `pyscalapack.pyf`.

```
PYSCALAPACKNAME=pyscalapack
F2PY= f2py
F2PYFLAGS     = -h --fcompiler=Gnu

$(F2PY) $(F2PYFLAGS) --build-dir $(PYSCALAPACKBUILD)
        $(BLACSSRCdir)/*.f  $(SCALAPACKSRCdir)/*.f
        $(PBLASSRCdir)/*.f -m $(PYSCALAPACKNAME)
```

El resultado que nos resulta más interesante en este punto, se obtiene en la carpeta de trabajo indicada mediante la variable `$(PYSCALAPACKBUILD)`. En esta carpeta se habrá generado el archivo `pyscalapack.pyf` y contiene la definición de todas las interfaces en una sintaxis comprensible por F2PY. De este modo, ya tenemos la definición de todas las interfaces en un único archivo que podrá ser utilizado para generar a su vez el código C que implementa dichas interfaces. Se ha de tener en cuenta, que una vez generado este archivo ya no será necesario realizar en futuras ocasiones el paso anterior y podremos realizar las modificaciones

directamente en este archivo puesto que podemos considerarlo como un punto donde se encuentran centralizadas todas las definiciones de las rutinas a las que hay que crear la interfaz.

Para el ejemplo de la rutina `PDGESV`, mostrado anteriormente, obtendríamos la siguiente definición:

```
subroutine pdgesv(n,nrhs,a,ia,ja,desca,ipiv,b,ib,jb,descb,info)
     integer intent(in) :: n
     integer intent(in) :: nrhs
     double precision dimension(*),intent(out,in) :: a
     integer intent(in) :: ia
     integer intent(in) :: ja
     integer dimension(*),intent(in) :: desca
     integer dimension(*),intent(out,in) :: ipiv
     double precision dimension(*),intent(out,in) :: b
     integer intent(in) :: ib
     integer intent(in) :: jb
     integer dimension(*),intent(in) :: descb
     integer intent(out,in) :: info
end subroutine pdgesv
```

Como se puede observar, se definen tanto variables de entrada (`n, nrhs, ia, ja, ...`), como variables de entrada y salida (`a, b` y `c`) que se suelen corresponder con las matrices de datos sobre las que se va a realizar el cálculo. Un detalle en el que se debe incidir, es el hecho que no se declaran variables únicamente de salida. La razón principal para no declarar variables exclusivamente de salida es que hemos de pasarle a la rutina Fortran las direcciones y espacios de memoria suficientes y necesarios para realizar los cálculos, de modo que no se realicen fallos de segmentación de memoria y sobreescrituras entre varias variables. De hecho, será en la interfaz creada mediante los scripts de Python (`PyBLACS.py, PyPBLAS.py, ...`) donde se creen las variables y espacios de memoria necesarios para la ejecución de cada rutina, liberando al usuario de esta tarea.

3. **Generación del archivo de interfaces** `pyscalapackmodule.c`

   En el paso anterior, habíamos obtenido el archivo de firmas. Sin embargo, en ese mismo proceso podríamos también haber obtenido el archivo de interfaces

pyscalapackmodule.c si en vez del parámetro -h hubiéramos utilizado -c. Se ha destacado como paso independiente debido a que a partir del archivo de firmas podremos volver a generar el archivo de interfaces sin necesidad de volver a leer de nuevo todas las rutinas que se habían modificado en el paso 1. De este modo, si realizáramos algún cambio en algún atributo de una determinada rutina, es más recomendable modificar el archivo de firmas y volver a generar el archivo de interfaces que modificar directamente el archivo de interfaces. Para generar el archivo de interfaces a partir de las definiciones que se encuentran en el archivo de firmas, deberíamos ejecutar este comando muy similar al anterior:

```
PYSCALAPACKNAME=pyscalapack
F2PY= f2py
F2PYFLAGS     = -c --fcompiler=Gnu

$(F2PY) $(F2PYFLAGS) --build-dir $(PYSCALAPACKBUILD)
        $(PYSCALAPACKBUILD)/$(PYSCALAPACKNAME).pyf
        -m $(PYSCALAPACKNAME)
```

Esta ejecución generará un nuevo archivo de interfaces pyscalapackmodule.c implementando los objetos y métodos adecuados según las definiciones vistas en la documentación de Python y descritas en el apartado 3.3. Por ejemplo, volviendo a la rutina que hemos utilizado para ilustrar el proceso, mostramos a continuación las referencias y las interfaces que se definen en pyscalapackmodule.c en relación a la rutina mostrada anteriormente:

- **Definición de la rutina externa perteneciente a la colección ACTS**.

```
extern void  F_FUNC(pdgesv,PDGESV)(int*,int*,double*,int*,
             int*,int*,int*,double*,int*,int*,int*,int*);
```

- **Rutina de verificación, conversión de parámetros y llamada a la rutina**.

  Esta rutina es la que integra los procesos de verificación de los parámetros pasados mediante Python y en el caso de que alguno de ellos no fuera adecuado informaría al intérprete de Python sin proseguir con la ejecución de dicha rutina. De este modo, todas aquellas comprobaciones y conversiones necesarias que

se realizarían para una correcta llamada de la rutina a niveles inferiores serían la principal penalización en cuestión de rendimiento al utilizar estas interfaces. Es por ello, que si preparamos los datos de forma adecuada (por ejemplo, con matrices contiguas en memoria) la penalización sufrida en la verificación y conversión de parámetros es muy pequeña.

A continuación exponemos parte del código de la rutina creada que implementará la interfaz a la rutina de ScaLAPACK vista como ejemplo:

```
1  static PyObject *f2py_rout_pyscalapack_pdgesv(
2                  const PyObject *capi_self,
3                  PyObject *capi_args,
4                  PyObject *capi_keywds,
5                  void (*f2py_func)(int*,int*,double*,int*,int*,
6                        int*,int*,double*,int*,int*,int*,int*))
7   {
8    PyObject * volatile capi_buildvalue = NULL;
9    volatile int f2py_success = 1;
10 /*decl*/
11   int n = 0;
12   PyObject *n_capi = Py_None;
13   int nrhs = 0;
14   PyObject *nrhs_capi = Py_None;
15   double *a = NULL;
16   int a_Dims[1] = {-1};
17   const int a_Rank = 1;
18   PyArrayObject *capi_a_tmp = NULL;
19   int capi_a_intent = 0;
20   PyObject *a_capi = Py_None;
21
22   ...
23
24  /* Processing variable a */
25   ;
26   capi_a_intent |= F2PY_INTENT_OUT|F2PY_INTENT_IN;
27   capi_a_tmp = array_from_pyobj(PyArray_DOUBLE,a_Dims,
28                                 a_Rank,capi_a_intent,a_capi);
29   if (capi_a_tmp == NULL) {
30     if (!PyErr_Occurred())
```

```
31          PyErr_SetString(pyscalapack_error,
32                "failed in converting 3rd argument `a' of
33                 pyscalapack.pdgesv to C/Fortran array" );
34      } else {
35        a = (double *)(capi_a_tmp->data);
36
37
38        ...
39
40
41        (*f2py_func)(&n,&nrhs,a,&ia,&ja,desca,ipiv,b,&ib,&jb,
42                     descb,&info);
43       ...
44
45     /*freemem*/
46    #ifdef F2PY_REPORT_ATEXIT
47    f2py_stop_clock();
48    #endif
49      return capi_buildvalue;
50    }
```

El código de esta rutina es más amplio pero se han querido destacar las principales partes del mismo que se suelen repetir para el resto de rutinas. Al principio de dicho código (líneas 1 a 6), se observa la definición de dicha rutina utilizando punteros al propio objeto y a los parámetros que se le pasan. A continuación, observamos la declaración de las variables a utilizar dentro de la rutina desde la línea 8 hasta la línea 23.

En este código, hemos adjuntado fundamentalmente las variables que se utilizarán para el tratamiento de la matriz `a`. Mediante los caracteres "..." se ha querido indicar la existencia de código que no se ha ilustrado para evitar un ejemplo demasiado extenso. A continuación de la identificación `/* Processing variable a */` (línea 24), se realiza el tratamiento de la variable `a` extrayendo la dirección de memoria de los datos a partir del objeto `Numeric Array` que le ha sido pasado. Observamos además en la línea 26, cómo ha sido definida la variable `a` como de entrada y salida mediante las constantes `F2PY_INTENT_OUT|F2PY_INTENT_IN` . Una vez "preparada" la variable mediante los punteros correspondientes, se procede a la llamada de la rutina mediante

el comando `*f2py_func` (línea 41) puesto que esta variable es un puntero de la función que se le pasa como variable y que en este caso se trata de `pdgesv`.

Una vez finalizada la ejecución de la rutina `pdgesv`, se procede a liberar la memoria temporal consumida y se tratan los parámetros de salida para proporcionar una respuesta adecuada al entorno Python (líneas 46 a 49).

4. **Inclusión en la tabla de métodos del módulo**.

   El vector nombrado como `static FortranDataDef f2py_routine_defs[]` está compuesto por una estructura de definiciones de funciones donde se relacionan el nombre que la rutina tiene para los niveles superiores (en Python), y el nombre de la rutina que lanza su ejecución (visto en el punto anterior). De este modo y siguiendo con el ejemplo de la rutina `pdgesv`, dentro de esta tabla de definiciones, encontraríamos:

```
    ...doc_f2py_rout_pyscalapack_pdgesvd},
{"pdgesv",-1,{{-1}},0,(char  *)F_FUNC(pdgesv,PDGESV),
       (f2py_init_func)f2py_rout_pyscalapack_pdgesv,
        doc_f2py_rout_pyscalapack_pdgesv},
{"pdgesvx",...
```

5. **Documentación del módulo y ayuda al programador**.

   Dentro de la rutina principal `initpyscalapack(void)` que se ejecuta cuando se importa el módulo en Python, se llevan a cabo diversas acciones para comprobar los enlaces a las librerías. En el caso de que algún objeto o rutina no estuviera disponible en la libreria compartida `.so`, Python informaría del error y se abortaría la importación de dicho módulo. Dentro de esta rutina también se define un vector que servirá de ayuda al usuario si éste quisiera ver su ayuda mediante el comando `>>>help(pyscalapack)`. Mediante este comando se muestran todas las rutinas disponibles dentro del módulo y con cada uno de los parámetros necesarios. Para el caso de la rutina de ejemplo observamos que su descripción aparece del siguiente modo:

```
s = PyString_FromString("This module 'pyscalapack' contains "
".\nFunctions:\n"
"  blacs_abort_(ictxt,errornumber)\n"
...
```

```
"  a,ipiv,b,info = pdgesv(n,nrhs,a,ia,ja,desca,ipiv,b,ib,jb,descb,info)\n"
...
".");
PyDict_SetItemString(d, "__doc__", s);
```

5.7.2. Generación del archivo de instalación `setup.py`

En los pasos anteriores hemos comprobado la utilidad de la herramienta F2PY para crear las interfaces de forma automática. Además, F2PY completa el proceso creando la librería `pyscalapack.so` compilando el archivo de interfaces del punto anterior y enlazándolo a las librerías correspondientes (BLACS, PBLAS, etc.). Sin embargo, esta última utilidad no va a ser explotada por el paquete PyACTS puesto que entonces, sería necesario tener instalado F2PY para poder instalar PyACTS.

Si nuestro objetivo es conseguir una herramienta de computación de altas prestaciones lo más amigable y sencilla posible sin pérdida en el rendimiento, también debería ser así su proceso de instalación. Por este motivo, y conforme a los apartados 3.1.6 y 3.1.7 en los que se establece cuál ha de ser el proceso de distribución y posterior instalación de un módulo de Python, hemos desarrollado un archivo de instalación `setup.py` que es capaz de gestionar el proceso de construcción e instalación de PyACTS mediante un sencillo comando del tipo `python setup.py install`.

Por tanto, a partir de los archivos de interfaces creados, tendremos que indicar a Python mediante el archivo `setup.py` los pasos necesarios para realizar el compilado y enlazado y la ubicación de las librerías necesarias en el módulo. A continuación describiremos cada una de las partes de este *script* de Python que realiza la compilación e instalación del módulo:

- **Importación de los módulos necesarios para la distribución**.

  En una primera parte del *script* debemos importar los módulos `sys` y `os` para comprobar la versión de Python, y también se importa rutinas del módulo `distutils` para realizar la instalación de la distribución.

```
import sys, os
```

```
from distutils.core import setup, Extension
```

- **Especificación de los directorios y los ficheros de librerías**.

  En el proceso final de enlazado de las librerías para generar `PyACTS.so`, Python necesita conocer la ubicación y los nombres de las librerías que contienen todos los símbolos y rutinas de las que se hace uso. Para ello, se crean dos listas `library_dirs_list` y `library_list` que contienen la ubicación y el nombre de los archivos de librerías necesarios.

```
library_dirs_list=[
    './BUILD','/home/vgaliano/tesis/hpc/BLACS/LIB',
'/home/vgaliano/tesis/hpc/mpich-1.2.6/lib','/usr/lib',
'/home/vgaliano/tesis/hpc/SCALAPACK',
'/home/vgaliano/tesis/hpc/ATLAS/lib/Linux_P4SSE2',
'/home/vgaliano/tesis/hpc/blas'
]
libraries_list = [
    'scalapack','blacsF77init_MPI-LINUX-0',
    'blacs_MPI-LINUX-0','blacsF77init_MPI-LINUX-0',
    'blacsCinit_MPI-LINUX-0','blacs_MPI-LINUX-0',
    'blacsCinit_MPI-LINUX-0','blas','mpich',
    'atlas','g2c']
```

  Se ha de tener en cuenta que cada nombre de librería indicado en este archivo hace referencia al nombre del fichero con el prefijo "`lib`" y la extensión "`.a`". Por ejemplo, en el caso de la librería `scalapack` indicada en la lista de librerías, el compilador buscará el archivo `libscalapack.a` en cualquiera de los directorios enumerados en `library_dirs_list` .

- **Especificación de los directorios de los archivos de inclusión**.

  Esta propiedad no será necesaria modificarla puesto que los archivos necesarios (generalmente de extensión "`.h`") se encuentran en la subcarpeta indicada.

```
include_dirs_list = ['./SRC']
```

- **Definición de la extensión**.

  Mediante un único archivo podremos crear diferentes módulos `.so`. En este caso, únicamente se ha de crear `pyscalapack.so`. Deseamos destacar que el archivo de librerías compartidas `PyACTS.so` indicado en la figura 5.1 se corresponde realmente con el archivo `pyscalapack.so` en la distribución actual de PyACTS. Si deseáramos crear nuevas librerías (por ejemplo, `superlu.so`) se debería repetir una estructura similar a la siguiente:

```
    module_pyscalapack = Extension( 'PyACTS.pyscalapack',
['SRC/fortranobject.c','SRC/pyscalapackmodule.c'],
libraries = libraries_list,
library_dirs=library_dirs_list,
include_dirs = include_dirs_list
)
```

  Se aprecia en esta definición la llamada al archivo de interfaces descrito en el punto anterior y que será compilado y enlazado con el resto de archivos y librerías.

- **Definición de la rutina de instalación**.

  Una vez definidos los objetos que han de ser instalados, el archivo ejecuta la instalación mediante la rutina `setup` incluida en el módulo importado `distutils`.

```
setup (name = "PyACTS",
       version = "1.0.0",
       description = "Python Interface to ACTS Collection.
        First Version has ScaLAPACK, PBLAS and
        BLACS interface.",
       author = "Vicente Galiano, Jose Penadés,
                 Violeta Migallón and Tony Drummond",
       author_email = "vgaliano@umh.es",
       url = "http://www.pyacts.org",
       package_dir = { 'PyACTS':'LIB/PyACTS'},
       packages = ["PyACTS"],
       ext_modules =[module_pyscalapack]
       )
```

La rutina `setup` procede con el copiado de los archivos `.py` y la compilación y enlazado de fuentes en este caso. En función del parámetro utilizado con el *script* se procede a su construcción o bien a su instalación utilizando `python setup.py build` o bien `python setup.py install`, respectivamente. Este archivo de instalación ha sido presentado previamente y analizado en mayor profundidad en el apartado 3.1.7.

5.8. Comportamiento de PyACTS

Durante los apartados anteriores del presente capítulo, hemos descrito en un primer momento, el paquete PyACTS como un conjunto de módulos importables desde Python que permiten el acceso a las rutinas accesibles en la colección ACTS. Además, hemos definido una estructura en dicho paquete así como un tipo de datos que permiten la interacción entre diferentes rutinas de la colección ACTS (PyScaLAPACK, PyBLACS, ...) y entre otras librerías disponibles en Python (Numeric, ScientificPython, ...). Por otro lado, hemos dotado a dicho paquete de un conjunto de rutinas que facilitan la inicialización, configurando algunos parámetros de forma automática y asumiendo determinadas configuraciones como adecuadas.

Durante el desarrollo del paquete PyACTS, se han tomado determinadas decisiones de diseño que pueden ser influyentes en la óptima ejecución de nuestras aplicaciones. Estas decisiones se refieren a parámetros que se han establecido como predeterminados pero que pueden ser modificados por el usuario de PyACTS si lo considera apropiado.

Por ejemplo, supongamos que se desea sumar dos vectores mediante una operación del tipo $\alpha x + y$, que se corresponde con el método `PyPBLAS.pvaxpy`. Si el usuario hiciera la programación más sencilla y automatizada, tendría el siguiente aspecto:

```
1 from PyACTS import *
2 import PyACTS.PyPBLAS as PyPBLAS
3 ACTS_lib=1                          # ScaLAPACK ID
4 PyACTS.gridinit()                   # inicialización de la malla
5 alpha=Scal2PyACTS(2,ACTS_lib)       # Convierte escalar a matriz PyACTS
6 a=Txt2PyACTS("data_a.txt",ACTS_lib) # Lee de un fichero Txt
7 b=Txt2PyACTS("data_b.txt",ACTS_lib) # y almacena en matriz PyACTS
```

```
8  b=PyPBLAS.pvaxpy(alpha,a,b)      # Llama a rutina de PBLAS de nivel 1
9  PyACTS2Txt(b,"data_result.txt")  # Escribe los resultados en fichero
10 PyACTS.gridexit()
```

En este código podemos observar que en un primer momento se importa el paquete `PyACTS` y el módulo `PyPBLAS` para poder tener acceso a las rutinas que contiene. En la línea 4, se inicializa la malla de procesos mediante el comando `gridinit()`. Tal y como se dijo en el apartado 5.5.1, si no indicamos ningún parámetro, la malla se configurará tan cuadrada como sea posible. De este modo, si ejecutamos este código en 4 nodos (por ejemplo mediante el comando `mpirun -np 4 ejemplo_pvaxpy.py`), obtendremos una configuración de la malla de procesos de aspecto 2×2. De este modo, al realizar la distribución cíclica 2D mediante el comando `Txt2PyACTS`, los procesos de la segunda columna no recibirán ningún dato y no participarán prácticamente en la operación puesto que el cálculo sólo se realizará en los procesos de la primera columna. Por tanto, la ejecución del programa no fallará pero realmente sólo intervienen dos procesos en el cálculo, con lo que se pierde una eficiencia importante.

Este detalle se puede solucionar de una forma sencilla si se hubiera configurado una malla de procesos adecuada a este problema específico. Por ejemplo, mediante una inicialización de la malla con el comando `gridinit(nprow=4,npcol=1)` se habría configurado una malla de aspecto 4×1 por lo que se hubieran distribuido los vectores entre los cuatro procesos y todos hubieran intervenido en la operación solicitada.

Con este ejemplo hemos pretendido mostrar que para facilitar la programación al usuario final se han asumido determinados parámetros y opciones cómo predeterminadas, pero que en determinadas ocasiones pueden no ser óptimas ni recomendables.

Ante la variedad de configuración y parámetros que hemos establecido, se pueden plantear las siguientes preguntas:

- ¿Cual es el tamaño de bloque ($MB \times NB$) más adecuado para la distribución de datos?
- ¿Cual es el tamaño de bloque ($MB \times NB$) más adecuado para la ejecución de una rutina de PyPBLAS o de PyScaLAPACK?
- ¿Cual es la configuración de la malla de procesos más adecuada para la distribución de datos?

- ¿Cual es la configuración de la malla de procesos más adecuada para la ejecución de una rutina de PyPBLAS o de PyScaLAPACK?

En la implementación y automatización de PyACTS se han tenido en cuenta estas cuestiones y hemos tomado los siguientes parámetros y configuraciones por defecto:

- El tamaño de bloque por defecto que se utilizará en las rutinas de PyACTS será $MB \times NB = 64 \times 64$. Esto implica que tanto la distribución y recolección de los datos, como las llamadas a las rutinas de PyBLACS, PyPBLAS y PyScaLAPACK utilizarán este tamaño de bloque por defecto.

- Como ya se ha comentado en puntos previos, tomamos una configuración de la malla de procesos tan cuadrada como sea posible.

Ambas decisiones no son casuales y responden a la información obtenida de la documentación de las librerías BLACS, PBLAS y ScaLAPACK y han sido confirmadas mediante unas pruebas que se mostrarán en el capítulo 8. En los capítulos 6, 7 y 8 introduciremos los paquetes PyBLACS, PyPBLAS y PyScaLAPACK, respectivamente. Una vez introducidos, podremos realizar pruebas de ejecución con los mismos para confirmar que los parámetros por defecto elegidos son adecuados. Las pruebas realizadas y las conclusiones que se obtienen al respecto se describirán en la sección 8.4.

5.9. Conclusiones

En el presente capítulo se ha tratado de presentar la distribución PyACTS como un paquete de Python que permite la ejecución y depuración de librerías de altas prestaciones incluidas en la colección ACTS. En función de la estructura de PyACTS descrita en la figura 5.1, se observan sus grupos funcionales agrupados en módulos de Python. Se consigue un entorno de trabajo sencillo, que bajo el lenguaje interpretado Python, nos permite ejecutar rutinas de altas prestaciones en plataformas de memoria distribuida. La dificultad implícita en las librerías actuales de computación científica en plataformas de este tipo, se oculta mediante un lenguaje interpretado con una sintaxis clara y sencilla,

y mediante una distribución propuesta que nos permite acceder a las rutinas como si de métodos de un objeto se tratase.

Por otro lado, la definición de la matriz PyACTS nos permite unificar el tipo de datos que las rutinas de PyACTS utilizarán como parámetro. A partir de la definición PyACTS, se proporcionan rutinas para la conversión de estas desde y hacia otros tipos de datos, por ejemplo, matrices locales Numpy (`Num2PyACTS/PyACTS2Num`), ficheros de texto (`Txt2PyACTS/PyACTS2Txt`) o ficheros netCDF (`PNetCDF2PyACTS/PyACTS2PNetCDF`). Además de rutinas de conversión de tipos, se proporcionan rutinas auxiliares que permiten la inicialización y liberación del entorno de ejecución, la verificación de los formatos PyACTS o la consulta de parámetros del entorno PyACTS. En resumen, se proporcionan las herramientas suficientes para permitir crear un entorno de ejecución en una plataforma de memoria distribuida mediante un intérprete paralelo de Python .

Precisamente, en la sección 5.4 se planteó qué intérprete de Python es el adecuado para la ejecución de PyACTS. Actualmente existen múltiples distribuciones que embeben la funcionalidad de MPI en el intérprete de Python, además es posible la aparición de actualizaciones o nuevas versiones que incluyan nuevas funcionalidades. Sin embargo, hemos comprobado que únicamente se necesita de una implementación que inicialice el entorno MPI, sin necesidad de acceder a las rutinas de MPI, puesto que ésto lo realizamos mediante el módulo PyBLACS y las funciones auxiliares de PyACTS. El rendimiento de PyACTS obtenido utilizando dos implementaciones diferentes de un intérprete paralelo de Python (mpipython y pyMPI) es el mismo según ha quedado reflejado en las figuras 5.3 y 5.4.

A partir de la estructura de PyACTS descrita en la sección 5.2, hemos descrito los módulos principales (agrupados en ficheros `.py`), y se ha detallado el proceso de creación del fichero de interfaces `pyscalapackmodule.c`. El elevado volumen de rutinas y la complejidad de cada una de ellas, nos ha obligado a utilizar una herramienta de generación de interfaces. Se ha tratado de describir este proceso de una manera breve y concisa para permitir la generación de interfaces a nuevas rutinas que se deseen incorporar. Para permitir al programador final la instalación cómoda de PyACTS, hemos mostrado el archivo de instalación configurado que permite una instalación automatizada siempre que se hayan instalado los requisitos de forma previa, y el archivo `setup.py` tenga configurados sus parámetros correctamente. De este modo, hemos conseguido personalizar la instalación de PyACTS en un único punto: el archivo de instalación.

Por último, en PyACTS hemos definido algunos parámetros y configuraciones por

defecto. En la sección 5.8 se han planteado diferentes situaciones en las que las elecciones adoptadas pueden ser apropiadas y otras en las que no. La argumentación en la elección de los parámetros se realizará en la sección 8.4, una vez se hayan presentado todos los módulos que componen PyACTS.

En definitiva, observamos en PyACTS una mejora en la usabilidad de las herramientas de altas prestaciones, permitiendo que éstas sean más accesibles por desarrolladores o investigadores con requisitos computacionales muy elevados, pero que no disponen de los conocimientos ni recursos necesarios para utilizar las herramientas y entornos tradicionales.

UNIVERSITAS
Miguel
Hernández

Capítulo 6

PyBLACS

En este capítulo introduciremos el paquete PyBLACS [36] como parte de la distribución PyACTS. En el apartado 4.3.3 ya se introdujo la librería BLACS y su funcionalidad dentro de una arquitectura de procesadores de memoria distribuida. En la sección 6.1 se introduce PyBLACS y su relación con el resto de paquetes, y mostraremos el conjunto de rutinas disponibles en PyBLACS. Cada grupo funcional de rutinas se describe más detalladamente en las secciones 6.2, 6.3, 6.4, 6.5 y 6.6. En cada una de estas secciones se muestran las pruebas de rendimiento realizadas para evaluar el comportamiento de la librería. Por último, en la sección 6.7 obtendremos conclusiones a partir de los ejemplos y pruebas mostrados en las secciones anteriores.

6.1. Introducción

Dentro de la distribución PyACTS se incluye el paquete PyBLACS que nos proporciona las rutinas de comunicaciones necesarias para realizar las tareas en el álgebra lineal cuando se realizan cálculos en un sistema de memoria distribuida. Se ha de tener en cuenta que a partir de la estructura de la librería ScaLAPACK representada en la figura 4.4, las librerías BLACS son necesarias para la ejecución de las rutinas pertenecientes a ScaLAPACK. Por tanto, puede resultar muy útil e interesante el crear las interfaces

para acceder a las rutinas proporcionadas en los niveles intermedios representados. De este modo, además del módulo PyScaLAPACK creamos las interfaces PyBLACS y PyPBLAS que también proporcionan el acceso a las rutinas incluidas en BLACS y PBLAS, respectivamente, desde el lenguaje Python.

Para describir las diferentes rutinas incluidas en PyBLACS haremos uso de su funcionalidad. En función de ésta podemos agruparlas del siguiente modo:

- Inicialización
- Destrucción
- Informativas
- Envío
- Recepción
- Difusión
- Operaciones combinadas

Esta clasificación se ha basado en la clasificación realizada para la librería BLACS (apartado 4.3.3). Se ha de tener en cuenta que los nombres de las rutinas en la librería BLACS pertenecientes a los grupos de envío, recepción, difusión u operaciones combinadas, dependen del tipo de dato que utilicen. Por ejemplo, la rutina de envío de una matriz de tipo general depende de si el tipo de dato es entero (`IGESD2D`), real o coma flotante (`SGESD2D`), real de doble precisión (`DGESD2D`), complejo (`CGESD2D`) o complejo de doble precisión (`ZGESD2D`). Sin embargo, en PyBLACS ya no será necesario realizar una distinción del tipo de dato utilizado para saber qué rutina utilizar. En PyBLACS existe una sola rutina para cada funcionalidad, y en función del tipo de dato que contiene, PyBLACS realiza la llamada a la rutina de BLACS que corresponda. De este modo, se consigue mayor transparencia y sencillez para los usuarios de PyBLACS.

En los siguientes apartados, describiremos las rutinas de cada uno de los grupos funcionales y mostraremos las pruebas realizadas indicativas de la penalización introducida por las interfaces desarrolladas. A continuación, mostramos de una forma breve (que será más detallada en las siguientes secciones), las rutinas de PyBLACS pertenecientes a cada uno de estos grupos.

- Inicialización
  - `iam,nprocs=pinfo()`: Obtiene identidad y número de procesos (MPI).
  - `iam,nprocs = setup(nprocs)`: Obtiene identidad y número de procesos (PVM).
  - `val = get(what,[ictxt])`: Obtiene valores internos de las BLACS.
  - `ictxt = gridinit(nprow,npcol[,order])`: Inicializa la malla con tamaño indicado.
  - `ictxt = gridmap(ictxt,usermap,ldumap,nprow,npcol)`: Establece cada proceso dentro de una malla.
- Destrucción
  - `abort([ictxt,errornum])`: Mata todos los procesos BLACS cuando ha habido algun error.
  - `gridexit(ictxt)`: Libera los recursos del contexto `ictxt`.
  - `exit([ictxt])`: Libera todos los recursos de todos los contextos.
  - `freebuff([ictxt,wait])`: Libera el buffer de las BLACS.
- Informativas
  - `nprow,npcol,myrow,mycol=gridinfo([ictxt])`: Proporciona información de la malla con contexto `ictxt`.
  - `prow,pcol = pcoord(pnum[,ictxt])`: Devuelve las coordenadas de un identificador dentro de la malla `ictxt`.
  - `nprow,npcol,myrow,mycol=gridinfo([ictxt])`: Proporciona información de la malla con contexto `ictxt`.
  - `num = pnum(prow,pcol,[ictxt])`: Proporciona el identificador a partir de las coordenadas de un proceso.
- Envío
  - `gesd2d(a,rdest,cdest,[ictxt,llda])`: Envía los datos de la matriz `a` hacia (`rdesc`,`cdest`).
  - `trsd2d(a,rdest,cdest,[ictxt,llda])`: Envía los datos de la matriz triangular `a` hacia (`rdesc`,`cdest`).

- Recepción
  - `a=gerv2d(a,rsrc,csrc,[ictxt,llda])`: Recibe los datos de la matriz `a` desde (`rsrc`,`csrc`).
  - `a=trrv2d(a,rsrc,csrc,[ictxt,llda,diag,llda])`: Recibe los datos de la matriz triangular `a` desde (`rsrc`,`csrc`).
- Difusión
  - `gebs2d(a,[ictxt,scope,top,llda])`: Inicia la difusión al grupo de destinatarios indicados por `scope`.
  - `trbs2d(a[,ictxt,scope,top,uplo,diag,llda])`: Inicia la difusión de una matriz triangular al grupo de destinatarios indicados por `scope`.
  - `a=gebr2d(a,irsrc,icsrc[,ictxt,scope,top,lda])`: Recibe los datos provenientes de una difusión desde (`irsrc`,`icsrc`).
  - `a=trbr2d(a,irsrc,icsrc[,ictxt,uplo,diag,scope,top,lda])`: Recibe los datos provenientes de una difusión de una matriz triangular desde (`irsrc`, `icsrc`).
- Operaciones combinadas
  - `a=gsum2d(a,rdest,cdest,[ictxt,scope,top,lda])`: Devuelve la suma de los elementos de la matriz.
  - `a,ra,ca=gamx2d(a,rdest,cdest[,ictxt,scope,top,lda,rcflag])`: Devuelve el valor máximo del elemento y su posición en los datos.
  - `a,ra,ca=gamn2d(a,rdest,cdest[,ictxt,scope,top,lda,rcflag])`: Devuelve el valor mínimo del elemento y su posición en los datos.

Las rutinas mostradas se encuentran disponibles en el módulo `PyBLACS` de la distribución PyACTS. De este modo, si se desea enviar una matriz `a` desde un proceso a otro, podemos acceder a las rutinas de envío y recepción como si fueran métodos del módulo (`PyACTS.PyBLACS.gesd2d(...)` y `PyACTS.PyBLACS.gerv2d(...)`. respectivamente). Podemos acceder de una forma más cómoda a las rutinas de PyBLACS si importamos el módulo y le damos un pseudónimo (por ejemplo, `import PyACTS.PyBLACS as PyBLACS`). De este modo, podremos ejecutar cualquiera de los métodos mediante el alias `PyBLACS` creado (`PyBLACS.gesd2d(...)` y `PyBLACS.gerv2d(...)`). Otra opción disponible

sería importar todos los objetos que contiene dicho módulo mediante un comando del tipo `import * from PyACTS.PyBLACS`. Esta acción importa los objetos al espacio de nombres de Python, y podemos ejecutar ambas rutinas sin especificar su módulo (`gesd2d(...)` y `gerv2d(...)`).

En la mayoría de las rutinas se ha conservado el nombre que se corresponde con la librería BLACS, sin embargo los parámetros en algunas de las rutinas pueden no ser obligatorios (los valores que se pueden indicar opcionalmente se muestran dentro de los corchetes, por ejemplo `gebs2d(a,[ictxt,scope,top,llda])`, y en este caso se asumirán valores por defecto. De este modo, si ejecutamos una primera inicialización de una malla de procesos mediante el comando `PyACTS.gridinit()`, obtendremos un identificador de contexto. Este valor generalmente es "`0`" cuando se inicializa la primera malla. Además este valor se establece como la malla por defecto de la librería PyACTS, de modo que `PyACTS.ictxt=0`.

No obstante, si deseamos utilizar más mallas de procesos obtendremos identificadores de contexto diferentes "`2,3,...`" cada uno de ellos correspondiente a una configuración diferente. En algunas rutinas de PyBLACS, el contexto puede ser un valor opcional en muchas de sus rutinas, en el caso de que no se especifique un `ictxt` se utilizará la primera de las configuraciones, es decir el contexto indicado en `PyACTS.ictxt=0`.

6.2. Rutinas de inicialización y liberación

El conjunto de rutinas que se engloban en este grupo permite obtener, establecer, configurar y posteriormente liberar diversos recursos relativos a las mallas de procesos necesarias en un entorno de comunicaciones bajo BLACS.

A continuación mostramos las rutinas de PyBLACS, en el caso de que se desee obtener más información acerca de la funcionalidad y la interfaz de cada una de las funciones, recomendamos acudir al manual del usuario de la librería PyACTS [34]. En dicho manual podremos consultar la funcionalidad de cada rutina, el significado de los parámetros de entrada y salida así como consultar un ejemplo.

- Inicialización:

  - `iam,nprocs=pinfo()`: Proporciona información relativa al número de procesos y la identificación de cada uno de los procesos dentro de un entorno de computación paralela donde intervienen múltiples procesos. Esta rutina se usa para obtener información inicial del sistema antes de inicializar BLACS. En todas las plataformas (excepto PVM), `nprocs` representa el numero de procesos disponibles para su uso donde `nprow` $\times$ `npcol` $\leq$ `nprocs`.

  - `iam,nprocs = setup(nprocs)`: Esta rutina únicamente tiene significado en el caso de estar trabajando bajo un entorno PVM. En otras plataformas, esta funcionalidad la proporciona la rutina `pinfo`. BLACS (y por tanto PyBLACS) asume un sistema estático: el sistema se inicializa con un número dado de procesos, mientras que PVM proporciona un sistema dinámico permitiendo que los procesos se añadan y eliminen al sistema en ejecución.

  - `val = get(what,[ictxt])`: Obtiene valores internos de BLACS. Algunos valores se refieren a un identificador de contexto determinado `ictxt`.

  - `ictxt = gridinit(nprow,npcol[,order])`: Indicaremos cuantos procesos estarán contenidos dentro de la malla que deseamos establecer. Todas las rutinas de BLACS deberán llamar a esta rutina o a su rutina homóloga `gridmap`. Estas rutinas establecen a cada proceso dentro de una malla. Cada malla BLACS está definida en un contexto (que representa su propio universo de paso de mensajes), de este modo no se interfieren distintos contextos o distintas configuraciones de mallas. Esta rutina puede ser utilizada en repetidas ocasiones para definir contextos/mallas adicionales.

    Además, esta rutina de creación de una malla establece variables internas a BLACS que no deberán ser utilizadas antes de la llamada a la funcion `gridinit` y crea una malla de procesos de tamaño `nprow` $\times$ `npcol`, y asigna a cada proceso un orden dentro de esa malla dependiendo del parámetro `order`.

  - `ictxt = gridmap(ictxt,usermap,ldumap,nprow,npcol)`: Sus funcionalidades son muy similares a la rutina `gridinit` descrita anteriormente. Hemos de indicar que ambas rutinas mapean a los procesos en una malla: los procesos no se crean de forma dinámica. En la mayoría de sistemas paralelos, los procesos no se crean dinámicamente y éstos se crean cuando el usuario arranca el ejecutable.

Esta rutina permite al usuario establecer los procesos en una malla de una manera arbitraria. `usermap(i,j)` establece a un determinado proceso a situarse en la coordenada `i,j` de la malla. En los sistemas distribuidos, este número de proceso estará simplemente definido por una maquina con un identificador entre 0 y `nprocs-1`. La rutina `gridmap` no es recomendable para el usuario inexperto, puesto que `gridinit` es mucho mas simple. `gridinit` simplifica las funciones de `gridmap` donde los primeros `nprow` × `npcol` procesos son mapeados dentro de la malla actual en un orden natural por filas.

- Liberación:

  - `abort([ictxt,errornum])`: Cuando ocurre algún error importante, el usuario puede necesitar abortar todos los procesos. Ésta es la razón de la existencia de `abort`. Hemos de destacar que los dos parámetros son de entrada, pero sólo se utilizan para imprimir el mensaje de error. El contexto `ictxt` puede ser cualquiera. Esta rutina mata todos los procesos BLACS, pero sólo aquellos confinados a un particular contexto.
  - `gridexit(ictxt)`: Los contextos consumen recursos, y es posible que el usuario desee liberarlos cuando ya no le sean necesarios mediante esta rutina. Después de la liberación de los recursos, el contexto ya no existe y puede ser reutilizado si se define un nuevo contexto.
  - `exit([ictxt])`: Esta rutina debe ser llamada cuando un proceso ha finalizado todo su uso de BLACS, libera todos los contextos la memoria que PyBLACS ha utilizado.
  - `freebuff([ictxt,wait])`: BLACS tiene al menos un buffer interno utilizado para empaquetar los mensajes. El número de buffers internos depende de la plataforma en la que se ejecuta la librería. En sistemas donde la memoria es escasa, mantener este buffer puede ser caro. La llamada a esta rutina libera el buffer. De todos modos, la próxima llamada a una rutina de comunicaciones que requiera empaquetado provocará que se requiera del buffer.

    El parámetro `wait` determina qué ha de esperar PyBLACS para completar o no las operaciones no-bloqueantes. Si `wait=0`, PyBLACS liberará los buffers que no estén en espera. Si `wait<>0`, PyBLACS liberará todos los buffers internos, incluso aunque los de las operaciones no bloqueantes deban ser completadas previamente.

Debido a que la carga computacional y de comunicaciones es muy pequeña en este tipo de rutinas, la diferencia en cuanto a rendimiento en ejecutarlas en su modo "nativo", es decir, en Fortran o ejecutarlas mediante PyBLACS es muy pequeña, casi despreciable. Por esta razón, no mostraremos en este apartado gráficas comparativas de rendimiento entre BLACS (Fortran) y PyBLACS (Python).

Sin embargo, creemos conveniente mostrar un pequeño ejemplo que ilustre la comodidad en utilizar PyBLACS y su facilidad de configuración. A continuación mostramos un ejemplo en el cual se ejecuta el siguiente *script* en 5 procesos:

```
1 import PyACTS.PyBLACS as PyBLACS
2 iam, nprocs=PyBLACS.pinfo()
3 nprow, npcol=4,1
4 ictxt=PyBLACS.gridinit(nprow, npcol)
5 if ictxt<>-1:
6         print "Soy ",iam,". Estoy en la malla. ictxt=",ictxt
7 else:
8         print "Soy ",iam,". No estoy en la malla. ictxt=",ictxt
9 PyBLACS.gridexit(ictxt)
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 5 mpipython ejemplo.py
Soy  4 .No estoy en la malla. ictxt= -1
Soy  3 .Estoy en la malla. ictxt= 0
Soy  1 .Estoy en la malla. ictxt= 0
Soy  2 .Estoy en la malla. ictxt= 0
Soy  0 .Estoy en la malla. ictxt= 0
```

Como podemos comprobar, estamos ejecutando este código con cinco procesos, y en la llamada a `gridinit(nprow,npcol)` estamos configurando una malla de tamaño 4×1. Se ha de tener en cuenta que `nprow,npcol` son parámetros opcionales y en el caso de que no se hubieran indicado los valores de la configuración de la malla de procesos (de tamaño `nprow` $\times$ `npcol`), el paquete PyBLACS configuraría de forma automática una malla lo más cuadrada posible, intentando utilizar el mayor numero de procesos. De este modo, si el comando de inicialización hubiese sido `ictxt=PyBLACS.gridinit()`, el resultado hubiera sido una malla de tamaño 2×2 puesto que para 5 procesos es la configuración cuadrada más aproximada. Por otro lado, si ejecutáramos, por ejemplo, con 7 procesos,

obtendríamos una configuración de tamaño 3×2, puesto que primero se busca una malla cuadrada (mediante la parte entera de $\sqrt{N_{procs}}$) y después se aumenta el numero de filas de la malla para utilizar el mayor número posible de procesos.

Deseamos destacar la sencillez y facilidad de la inicialización de una malla BLACS, así como la facilidad en el manejo de los parámetros de configuración. Los procesos de inicialización y finalización de la malla de procesos de reducen a una línea cada uno (línea 4 y 9) respectivamente. Además, en apenas 9 líneas hemos mostrado un ejemplo de utilización de las rutinas PyBLACS que nos hubiese ocupado, al menos, cuatro veces más en un lenguaje tradicional como Fortran. Tanto este ejemplo como los próximos que se muestran en este trabajo pueden ejecutarse como si fueran un `script`, o bien podrían ejecutarse paso a paso permitiendo una depuración y ejecución más interactiva.

En el manual de PyBLACS ya citado podremos obtener más ejemplos de utilización que nos permitirán profundizar más en la utilización de estas rutinas.

6.3. Rutinas informativas

Durante la ejecución de un programa en un entorno paralelo en el cual tenemos una malla de procesos configurada, puede ser interesante obtener o consultar algún parámetro específico de la configuración. Para tal finalidad, este conjunto de rutinas nos permiten obtener este tipo de información:

- `nprow,npcol,myrow,mycol=gridinfo([ictxt])`

  Para el contexto `ictxt` indicado devuelve el tamaño de la malla de procesos `nprow` $\times$ `npcol` y la posición del proceso dentro de dicha malla, indicado por las coordenadas `(myrow,mycol)`. El parámetro `ictxt` es opcional, en el caso de no especificar un contexto, se utilizará el contexto almacenado en la propiedad `PyACTS.ictxt`.

- `prow,pcol = pcoord(pnum[,ictxt])`

  Como ya se ha explicado, los procesos están identificados por un valor que en PyACTS se ha llamado `iam` único en una ejecución en paralelo comprendido entre `0` y `numprocs-1`. A partir de este identificador y del identificador de contexto, podemos averiguar las coordenadas de cualquier proceso dentro de la malla de procesos. Estas

coordenadas son útiles y necesarias por ejemplo en el envío o recepción de datos. De este modo, si cada proceso desea obtener su posición dentro de la malla puede ejecutar: `myrow,mycol = pcoord(PyACTS.iam,PyACTS.ictxt)`.

- `num = pnum(prow,pcol,[ictxt]))`

  Esta rutina tiene una funcionalidad inversa a la anterior, es decir, a partir de las coordenadas de un proceso podremos averiguar cual es su identificador `num`.

Del mismo modo que sucede con las rutinas de inicialización y destrucción, la carga computacional y de comunicaciones en este grupo funcional es muy baja por lo que no corresponde realizar un análisis comparativo de rendimiento entre un entorno Fortran y un entorno de programación Python. Sin embargo, sí creemos conveniente mostrar un breve código en el cual se trate de representar la sencillez y facilidad en la utilización de estas rutinas desde su interfaz para Python.

A continuación se muestra un ejemplo en el cual se inicializa una malla de procesos 2×2 a pesar de ejecutar el sistema con 6 procesos. De este modo, los procesos con identificador 4 y 5 deberían quedar fuera de la malla de procesos.

```
1  import PyACTS.PyBLACS as PyBLACS
2  iam, nprocs=PyBLACS.pinfo()
3  nprow, npcol=2,2
4  ictxt=PyBLACS.gridinit(nprow, npcol)
5  nprow, npcol, myrow, mycol=PyBLACS.gridinfo()
6  if ictxt<>-1:
7          print "Soy ",iam,". Estoy en la malla, [",myrow,",",mycol,"]"
8  else:
9          print "Soy ",iam,". No estoy en la malla, ictxt=",ictxt
10 PyBLACS.gridexit(ictxt)
```

En la línea 5 se observa que la llamada `gridinfo` obtiene el tamaño de la malla de procesos configurada *nprow* × *npcol* y también las coordenadas de cada proceso `[myrow,mycol]`. Cuando el identificador de contexto es negativo indica que el proceso no pertenece a ninguna malla de procesos. A continuación se muestra la salida obtenida al ejecutar el ejemplo anterior en seis procesos.

```
[vgaliano@EXAMPLES]$ mpirun -np 6 mpipython ejemplo_inf.py
Soy  5 . No estoy en la malla, ictxt= -1
```

```
Soy  4 . No estoy en la malla, ictxt= -1
Soy  1 . Estoy en la malla, [ 0 , 1 ]
Soy  3 . Estoy en la malla, [ 1 , 1 ]
Soy  2 . Estoy en la malla, [ 1 , 0 ]
Soy  0 . Estoy en la malla, [ 0 , 0 ]
```

6.4. Rutinas de envío y recepción

El intercambio de información nodo a nodo se realiza mediante las rutinas incluidas en este grupo. Estas rutinas permiten el envío de una matriz en formato `Numeric` o `Numpy` de manera unidireccional y no bloqueante.

Se debe distinguir entre dos tipos de rutinas: el primer tipo se corresponde con una rutina para enviar matrices genéricas de tipo denso y vienen identificadas por su prefijo `ge`; no obstante, si se desea enviar una matriz triangular, se utilizarán las rutinas con el prefijo `tr`. Estos detalles ya fueron introducidos en el apartado 4.3.3 referentes a BLACS y guardan mucha similitud para PyBLACS.

El conjunto de rutinas disponibles para este grupo son:

- Envío:
  - `gesd2d(a,rdest,cdest,[ictxt,llda])`: Esta rutina envía la matriz `Numeric` llamada `a` que se le pasa por parámetro, al proceso destino identificado mediante sus coordenadas `(cdest,rdesc)` en la malla de procesos. Esta rutina es localmente bloqueante, es decir, retornará de su llamada incluso si la correspondiente recepción no ha sido realizada.

    Se observan dos parámetros opcionales `ictxt` y `llda`. En caso de no introducir un identificador de contexto diferente, se utilizará el identificador `PyACTS.ictxt`. Por otro lado, `llda` ha sido descrita en el apartado 4.3.5 e indica la distancia entre dos elementos contiguos de la misma fila. Este parámetro es opcional y en caso de no indicarse, se calcula de forma automática y transparente conforme a las dimensiones de la matriz `Numeric` pasada. Por ejemplo, para una matriz `a` de tamaño $m \times n$, el valor de `llda` es m.

- `trsd2d(a,rdest,cdest,[ictxt,llda])`: Rutina muy similar a la vista anteriormente. Los parámetros de esta rutina son similares a la rutina de envío general `gesd2d`, sin embargo esta rutina está destinada al envío de matrices trapezoidales definidas en el apartado 4.3.3.

- Recepción:

  - `a=gerv2d(a,rsrc,csrc,[ictxt,llda])`: Recibe la matriz que se le pasa por parámetro desde el proceso descrito mediante las coordenadas `(csrc,rsrc)` dentro de la malla de procesos. Esta rutina es localmente bloqueante, por ejemplo, no retornará valor si la correspondiente recepción no ha sido realizada. Un detalle importante a tener en cuenta es que a pesar de ser una rutina de recepción, deberemos pasar como parámetro a esta rutina una matriz `a`. Esta matriz será una matriz `Numeric` de ceros (por ejemplo), con las dimensiones adecuadas de la matriz que esperemos recibir. La razón de esta operación se debe a que debemos reservar el espacio de memoria necesario para que la rutina BLACS (en niveles inferiores) copie los datos recibidos en el espacio de memoria reservado. El resultado será una matriz de las dimensiones de la matriz de entrada con sus elementos correspondientes a los elementos recibidos.
  - `a=trrv2d(a,rsrc,csrc,[ictxt,llda,diag,llda])`: Rutina con la misma funcionalidad que `gerv2d` pero destinada a la recepción de matrices trapezoidales.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
1  import PyACTS.PyBLACS as PyBLACS
2  from Numeric import *
3  size=4
4  ictxt=PyBLACS.gridinit()
5  nprow,npcol,myrow,mycol=PyBLACS.gridinfo()
6  if myrow==0 and mycol==0:
7          a=reshape(range(size),[size,1])
8          print "[",myrow,",",mycol,"] envia a=",transpose(a)
9          PyBLACS.gesd2d(a,0,1)
10         PyBLACS.gesd2d(a,1,0)
11         PyBLACS.gesd2d(a,1,1)
12 else:
```

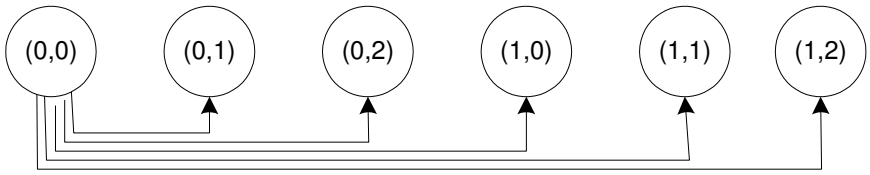
```
13          a=zeros([size,1])
14          a=PyBLACS.gerv2d(a,0,0)
15          print "[",myrow,",",mycol,"] recibe a=",transpose(a)
16 PyBLACS.gridexit(ictxt)
17 PyBLACS.exit()
```

En el ejemplo mostrado, se observa la inicialización de la malla de procesos mediante la rutina `PyBLACS.gridinit()` de la línea 4. A continuación podremos obtener el tamaño de la malla de procesos y la posición de cada proceso mediante la rutina `PyBLACS.gridinfo()`. Estas dos rutinas ya han sido descritas en el apartado 6.2 y 6.3 respectivamente. En este ejemplo, el proceso `(0,0)` será el encargado de enviar un vector de cuatro elementos a todos los procesos mediante comunicaciones unidireccionales, es decir desde un origen a un destino. El resultado de la ejecución de este *script* se muestra a continuación.

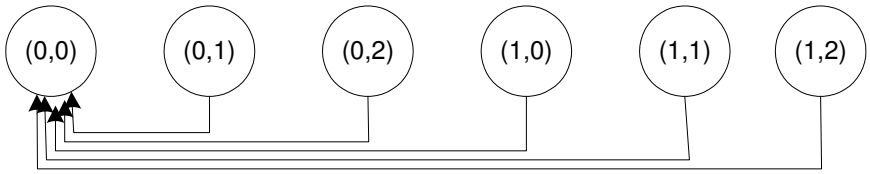
```
[vgaliano@nodo0]$ mpirun -np 4 mpipython exPyblacsgesdrv2d.py
[ 0 , 0 ] envia a= [ [0 1 2 3]]
[ 0 , 1 ] recibe a= [ [0 1 2 3]]
[ 1 , 0 ] recibe a= [ [0 1 2 3]]
[ 1 , 1 ] recibe a= [ [0 1 2 3]]
```

Con el fin de comprobar y comparar el rendimiento de las rutinas descritas anteriormente, se han realizado pruebas en Cluster-umh y en Seaborg. En la figura 6.2 se muestran los tiempos de ejecución necesarios tanto en Fortran como en Python. El algoritmo utilizado para probar el rendimiento de las rutinas de envío y recepción consiste en el envío de una matriz de diferentes tamaños (500, 1000, ...) desde el proceso `(0,0)` hacia el resto de procesos de la malla. Posteriormente, cada nodo de la malla envía de nuevo esa matriz de datos al nodo `(0,0)`.

Como se puede observar en las figuras 6.2(a) y 6.2(b), a medida que aumenta el tamaño de la malla de procesos, aumenta el tiempo necesario para enviar la misma matriz a todos los procesos. Se ha de tener en cuenta que el nodo `(0,0)` envía la matriz `a` cada proceso de forma sucesiva tal y como se describe en la figura 6.1(a). Posteriormente, cada nodo de la malla vuelve a enviar al nodo `(0,0)` la matriz de datos recibida de forma unidireccional. Este segundo paso se ha representado mediante la figura 6.1(b). Tal y como se observa en ambas plataformas, en una malla 2×1, el nodo `(0,0)` envía los datos al nodo `(0,1)` y una vez los ha recibido, éste los envía al nodo `(0,0)`. Sin embargo, cuando aumentamos el



(a) Envío

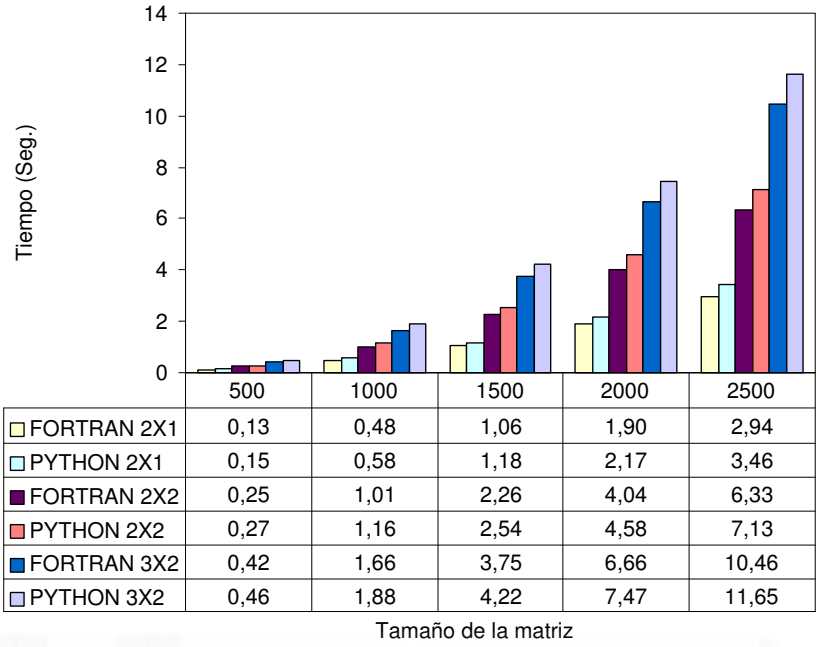


(b) Recepción

Figura 6.1: Algoritmo implementado en las pruebas de rendimiento.

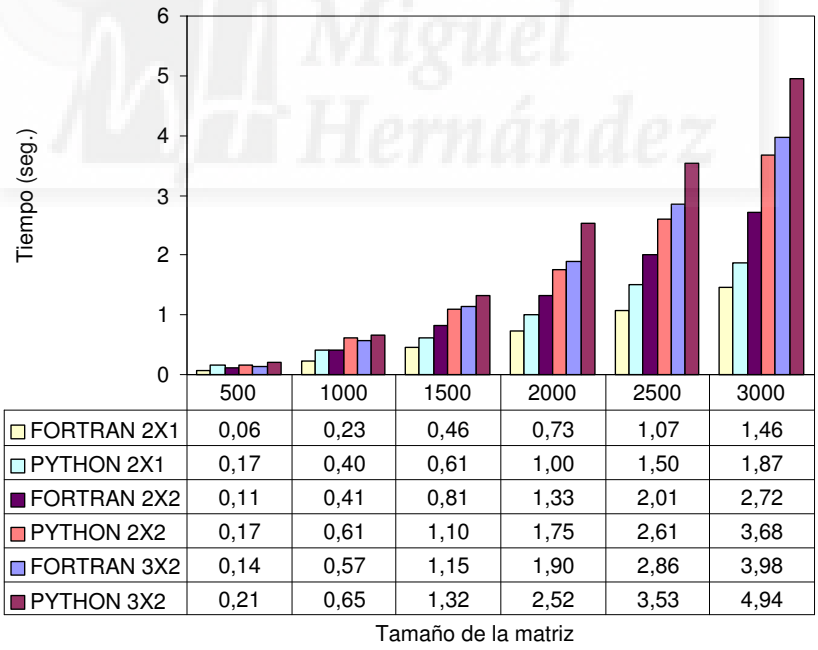
número de procesos, el nodo `(0,0)` deberá enviar de forma dedicada la matriz de datos a cada nodo, y posteriormente recibir dicha matriz de cada nodo siempre con comunicaciones punto a punto. Observamos que los tiempos de ejecución crecen a medida que crece el tamaño del paquete a enviar, pero también crecen con el número de procesos en la malla, puesto que debe ser el nodo `(0,0)` el encargado de centralizar las comunicaciones punto a punto.

Si nos fijamos en la ejecución para dos procesos (primera y segunda fila de las gráficas en las figuras 6.2(a) y 6.2(b), donde la malla de procesos es 2×1), obtenemos el tiempo que tarda una matriz en enviarse y recibirse. Con dos procesos, podríamos obtener el ancho de banda máximo de comunicación entre dos procesos en nuestro sistema. Por esta razón, hemos realizado una prueba más intensiva para dos procesos que se muestra en la figura 6.3. La prueba consiste en enviar un vector de tamaño $N \times 1$ desde el nodo `(0,0)` hacia el nodo `(0,1)` y posteriormente, el nodo `(0,1)` vuelve a enviar el vector al nodo origen `(0,0)`. De este modo, podemos conocer cuál es el ancho de banda obtenido para ese tamaño de vector, o lo que es lo mismo para ese tamaño de mensaje. Se ha de tener en cuenta que el vector generado tanto en Fortran como en Python es de doble precisión (`Numeric.Float` en Python y `DOUBLE PRECISION` en Fortran) por lo que cada elemento del vector ocupa 8 bytes. Si denotamos T_{tot} como el tiempo invertido en enviar y recibir el vector entre dos nodos, y N como el tamaño del vector, podemos obtener el ancho de



	500	1000	1500	2000	2500
FORTRAN 2X1	0,13	0,48	1,06	1,90	2,94
PYTHON 2X1	0,15	0,58	1,18	2,17	3,46
FORTRAN 2X2	0,25	1,01	2,26	4,04	6,33
PYTHON 2X2	0,27	1,16	2,54	4,58	7,13
FORTRAN 3X2	0,42	1,66	3,75	6,66	10,46
PYTHON 3X2	0,46	1,88	4,22	7,47	11,65

(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 2X1	0,06	0,23	0,46	0,73	1,07	1,46
PYTHON 2X1	0,17	0,40	0,61	1,00	1,50	1,87
FORTRAN 2X2	0,11	0,41	0,81	1,33	2,01	2,72
PYTHON 2X2	0,17	0,61	1,10	1,75	2,61	3,68
FORTRAN 3X2	0,14	0,57	1,15	1,90	2,86	3,98
PYTHON 3X2	0,21	0,65	1,32	2,52	3,53	4,94

(b) Seaborg

Figura 6.2: Comparativa de Envío unidireccional (`gesd2d/gerv2d`) con Fortran y Python

banda BW como

$$BW = \frac{2 * 8 * N}{T_{tot} * 10^6} \; Mbytes/seg.$$

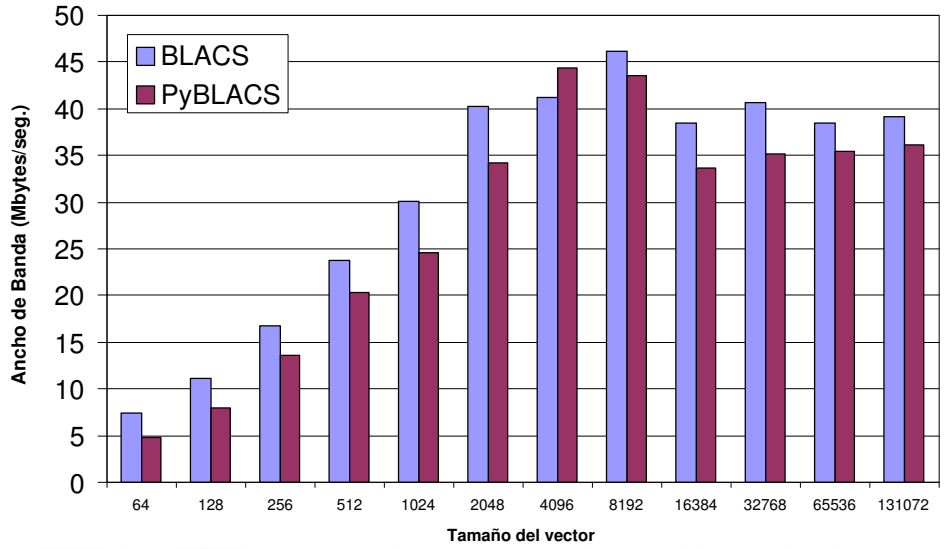
Se ha de tener en cuenta que el vector se envía y se recibe, por lo que el volumen de los bytes enviados por la red será el doble del volumen de bytes del vector. En la figura 6.3(a) observamos que el ancho de banda máximo en Mbytes/seg obtenido en el Cluster-umh es aproximadamente de 46,10 Mbytes/seg. (368 Mbps) con BLACS y 44,43 Mbytes/seg. (355 Mbps) con PyBLACS. Los valores concuerdan con la velocidad obtenida en otras pruebas con librerías a más bajo nivel (MPI). Conviene recordar que en esta plataforma los nodos están conectados mediante una conmutador Gigabit Ethernet.

Por otro lado, en la figura 6.3(b) se observa que el ancho de banda máximo obtenido para Seaborg, es superior al del Cluster-umh, en este caso es de 92,34 Mbytes con BLACS (aproximadamente 738 Mbps) y 85,99 Mbytes (687 Mbps) con PyBLACS. En resumen, el ancho de banda del sistema no sufre una penalización considerable frente a las mejoras de usabilidad en el desarrollo y en la interacción con el usuario.

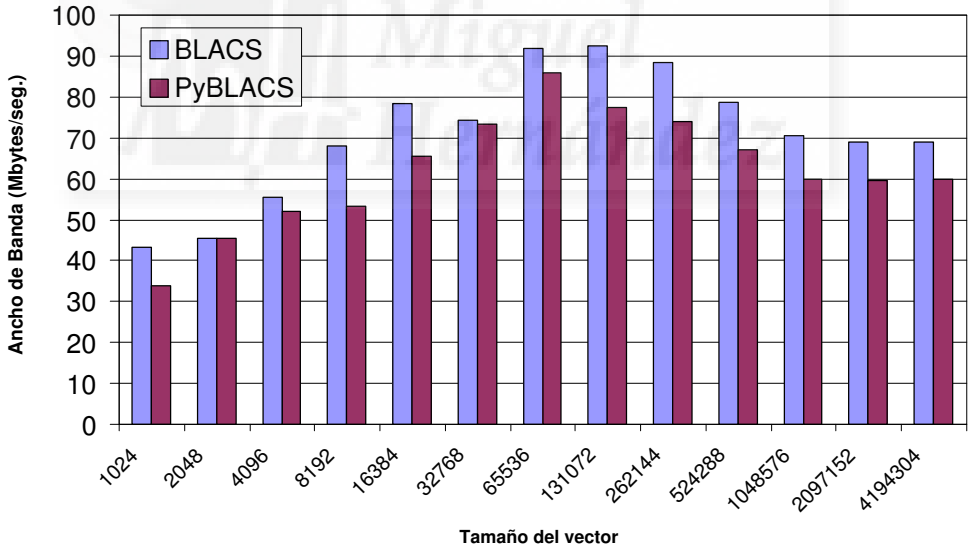
6.5. Rutinas de difusión

En la sección anterior se han visto las rutinas de envío y recepción de datos de carácter unidireccional. Además, el algoritmo mostrado en la figura 6.1 hemos comprobado cómo el tiempo incrementa de forma considerable a medida que se aumenta el número de nodos al que se desea enviar los datos mediante rutinas unidireccionales. Para mejorar estas tareas de difusión, se incluyen en BLACS, y por tanto en PyBLACS rutinas que ofrecen la funcionalidad de difusión y recolección de datos desde y hacia un nodo determinado. El conjunto de rutinas que proporcionan esta facilidad en PyBLACS son las siguientes:

- Difusión:
  - `gebs2d(a,[ictxt,scope,top,lda])`: El proceso origen inicia la difusión a través de un grupo de destinatarios, mientras que todos los demás procesos que pertenecen al grupo de destino deberán llamar a la rutina de recepción de



(a) Cluster-umh



(b) Seaborg

Figura 6.3: Ancho de banda máximo con Fortran y Python para diferentes tamaños de vector

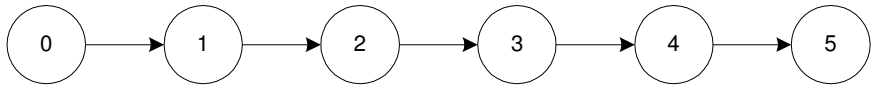
la difusión `gebr2d`. Finalmente, todos los procesos pertenecientes al grupo destinatario recibirán la matriz `a`. La difusión puede ser globalmente bloqueante, es decir, no se garantiza que el proceso vuelva de una difusión hasta que todos los procesos hayan llamado a las rutinas apropiadas (`gebs2d` el proceso origen y `gebr2d` los procesos destino).

- `trbs2d(a[,ictxt,scope,top,uplo,diag,lda])`: Rutina para el envío de difusiones con la misma funcionalidad que `gebs2d` pero enfocada para matrices trapezoidales.

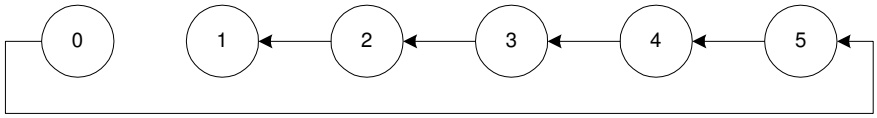
- Recolección:
  - `a=gebr2d(a,irsrc,icsrc[,ictxt,scope,top,lda])`: Implementa la recepción de una difusión de la matriz `a` realizada por un proceso origen `(irsrc, icsrc)`. Los parámetros opcionales `ictxt,scope,top` y `lda` se explican a continuación.
  - `a=trbr2d(a,irsrc,icsrc[,ictxt,uplo,diag,scope,top,lda])`: Tiene la misma funcionalidad que la rutina `gebr2d` pero destinada a matrices trapezoidales.

Se observa en estas rutinas que los parámetros incluidos dentro de los corchetes representan parámetros opcionales, y que en el caso de no indicarse PyBLACS aplicará el parámetro por defecto. Los valores por defecto para cada uno de estos parámetros serían los siguientes:

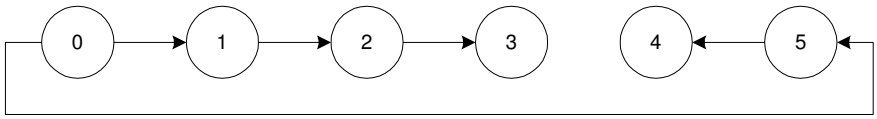
- `ictxt=0`: Representa el indicador de contexto de la malla de procesos, si no se especifica se utilizará `PyACTS.ictxt` obtenido mediante la primera ejecución de `PyACTS.gridinit()` y que suele valer 0.
- `scope`: Indica el alcance de la difusión. Este parámetro puede tener tres valores:
  - `scope='R'`: Afecta a las filas del nodo origen.
  - `scope='C'`: Afecta a las columnas del nodo origen.
  - `scope='A'`: Valor por defecto en PyBLACS. Afecta a todos los nodos de la malla de procesos.
- `top`: Indica la topología de la comunicación utilizada para realizar la difusión. Los valores que puede tomar este parámetro se enumeran a continuación y se representan mediante la figura 6.4:



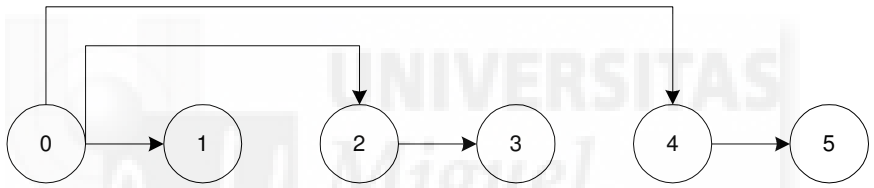
(a) Anillo incremental



(b) Anillo decremental



(c) Anillo segmentado



(d) Multianillo

Figura 6.4: Topologías para la difusión en PyBLACS

- `top=' '`: Valor por defecto en PyBLACS y se corresponde con una topología especial que minimiza el tiempo de operación.
- `scope='I'`: Anillo unidireccional incremental.
- `scope='D'`: Anillo unidireccional decremental.
- `scope='S'`: Anillo segmentado.
- `scope='M'`: Multianillo.

Los parámetros `uplo`, `diag` y `lda` han sido introducidos durante la descripción del tratamiento de matrices que implementa BLACS en el apartado 4.3.3.

En la figura 6.5 se representa la comparación entre los tiempos obtenidos al utilizar el entorno en Fortran y el entorno de programación de Python. El algoritmo utilizado para

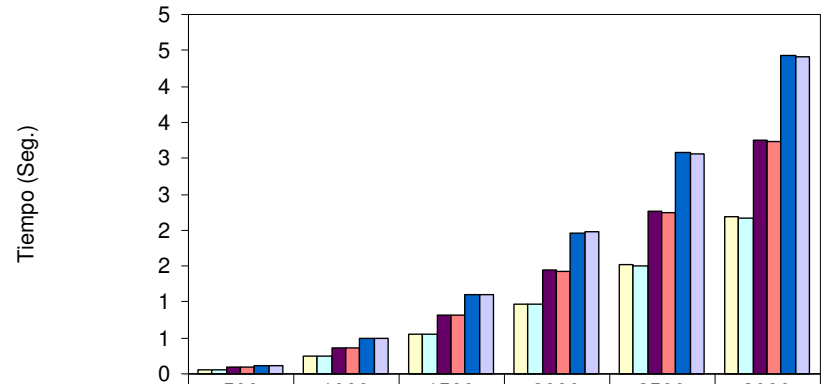
realizar esta prueba ha sido el de realizar la difusión de una matriz cuadrada $N \times N$ del tamaño N indicado en cada columna para diferente número de procesos y configuración de malla. De este modo, será el proceso `(0,0)` el que ejecute la rutina de difusión `gebs2d`, y el resto de nodos de la malla ejecutarán la rutina de recepción de una difusión `gerv2d`. Se puede comprobar que los tiempos entre Fortran y Python obtenidos son extraordinariamente similares en ambos a pesar de una pequeña penalización que prácticamente no supera el 1 %. Por otro lado, es conveniente reseñar que los tiempos de comunicaciones obtenidos son significativamente inferiores a los de la figura 6.2, puesto que en el envío unidireccional la matriz se envía de forma repetitiva desde el proceso origen a cada uno de los otros procesos de la malla. Sin embargo, en este caso las rutinas ejecutadas implementan la difusión por lo que se obtienen tiempos de ejecución inferiores. Además, hemos de recordar que la difusión es localmente bloqueante, es decir, el nodo `(0,0)` no termina la ejecución de la difusión hasta que todos los nodos destinatarios reciben correctamente la matriz enviada.

Por otro lado, se desea destacar que se han utilizado tanto para la implementación en Fortran como para la implementación en Python, los valores de los parámetros por defecto descritos anteriormente.

6.6. Operaciones combinadas

Además de rutinas que implementan tareas de comunicación entre procesos, en PyBLACS (así como en BLACS), se incluyen un conjunto de rutinas que proporcionan una funcionalidad adicional, conocidas como rutinas de operaciones combinadas. A continuación enumeramos cada una de estas rutinas:

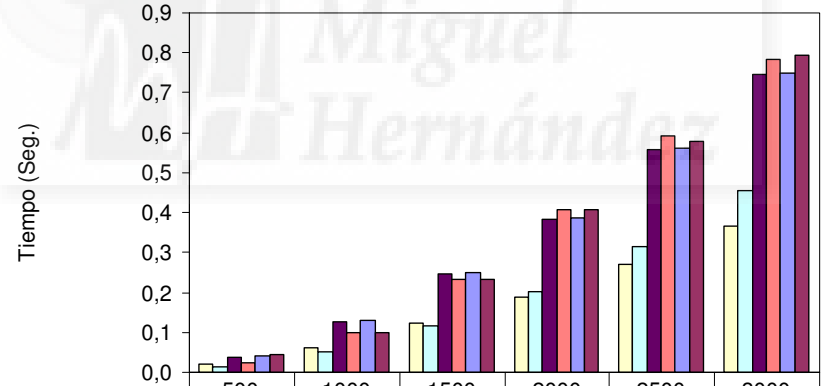
- *Suma*: `a=gsum2d(a,rdest,cdest,[ictxt,scope,top,lda])`
  Mediante la llamada a esta rutina, cada elemento de la matriz `a` es sumado con los correspondientes elementos de las demás matrices de los demás procesos. El resultado se almacena en el proceso identificado por las coordenadas `(rdest,cdest)`. La combinación puede ser globalmente bloqueante, de este modo no se devuelve una respuesta hasta que todos los procesos han llamado a esta rutina.



	500	1000	1500	2000	2500	3000
FORTRAN 2X1	0,06	0,24	0,55	0,97	1,52	2,19
PYTHON 2X1	0,06	0,24	0,55	0,97	1,51	2,17
FORTRAN 2X2	0,09	0,36	0,82	1,44	2,27	3,26
PYTHON 2X2	0,09	0,36	0,81	1,43	2,24	3,23
FORTRAN 3X2	0,12	0,49	1,11	1,96	3,08	4,42
PYTHON 3X2	0,12	0,49	1,11	1,97	3,06	4,42

Tamaño de la matriz

(a) Cluster



	500	1000	1500	2000	2500	3000
FORTRAN 2X1	0,02	0,06	0,12	0,19	0,27	0,37
PYTHON 2X1	0,01	0,05	0,12	0,20	0,32	0,45
FORTRAN 2X2	0,04	0,13	0,25	0,38	0,56	0,75
PYTHON 2X2	0,03	0,10	0,23	0,41	0,59	0,78
FORTRAN 3X2	0,04	0,13	0,25	0,39	0,56	0,75
PYTHON 3X2	0,04	0,10	0,23	0,41	0,58	0,79

Tamaño de la matriz

(b) Seaborg

Figura 6.5: Comparativa de difusión `gebs2d/gebr2d` con Fortran y Python

- *Máximo*: `a,ra,ca=gamx2d(a,rdest,cdest[,ictxt,scope,top,lda,rcflag])`
  Esta rutina obtiene los valores máximos comparando cada elemento de la matriz con los demás elementos de las matrices de cada proceso en la misma posición. El resultado final se almacena en la matriz del proceso con coordenadas `(rdest,cdest)`. El resultado se devuelve en valor absoluto. En el caso de utilizar números complejos se calculará la norma. La ejecución de esta rutina es bloqueante globalmente, por tanto todos los procesos que intervienen deberán ejecutar esta rutina.

- *Mínimo*: `a,ra,ca=gamn2d(a,rdest,cdest[,ictxt,scope,top,lda,rcflag])`
  Esta rutina obtiene los valores mínimos comparando cada elemento de la matriz con los demás elementos de las matrices de cada proceso en la misma posición. El resultado final se almacena en la matriz del proceso con coordenadas `(rdest,cdest)`. El resultado se devuelve en valor absoluto. En el caso de utilizar números complejos se calculará la norma. La ejecución de esta rutina es bloqueante globalmente, por tanto todos los procesos que intervienen deberán ejecutar esta rutina.

Los parámetros `ictxt, scope` y `top` ya han sido introducidos en el apartado 6.5, mientras que el parámetro `lda` fue descrito en el apartado 4.3.3. El parámetro de entrada `rcflag` aparece en las rutinas `gamx2d` y `gamn2d`, y si `rcflag=-1` los valores de la posición del elemento máximo o mínimo `(ra,ca)` no se proporcionarán en la salida.

En el manual de PyACTS [34] se adjunta información más detallada de cada uno de los parámetros de entrada y salida, así como ejemplos de los mismos. Conviene recordar que el número de parámetros y la complejidad de utilización de estas rutinas se ha reducido con respecto a las funciones análogas en BLACS. No obstante, los parámetros mostrados entre corchetes indican su opcionalidad y en el caso de no ser indicados de forma explícita se toma su valor por defecto.

Además, esta mejora en la usabilidad y sencillez de las rutinas no viene acompañada de una perdida de eficiencia significativa en las interfaces a Python creadas. Para comprobar el rendimiento de las rutinas combinadas de PyBLACS, se han realizado una serie de pruebas para diferentes tamaños de matriz y diferentes tamaños de mallas de procesos.

En las figuras 6.6, 6.7 y 6.8 se muestran las comparaciones entre los tiempos obtenidos al utilizar Fortran y al utilizar Python. Como podemos comprobar, no se observa un incremento significativo en los tiempos de ejecución para un mismo tamaño de matriz y para una misma malla de procesos. Además, se comprueba que el comportamiento PyBLACS es muy similar al de PyBLACS independientemente de la plataforma, puesto

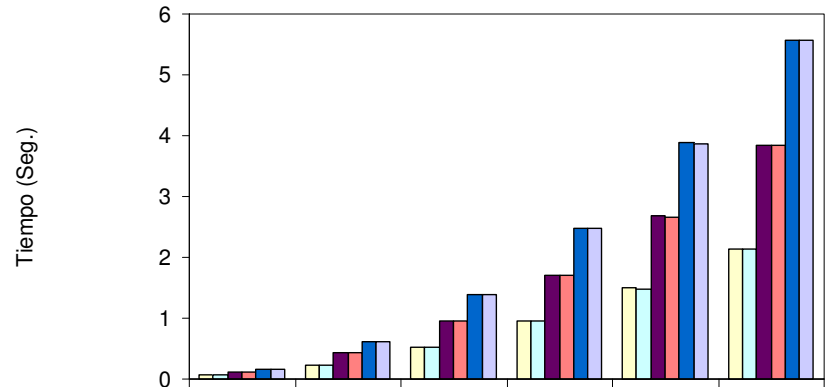
que en la figura 6.6(a) y 6.6(b), PyBLACS mantiene la misma tendencia que BLACS.

6.7. Conclusiones

PyBLACS es el primero de los módulos de PyACTS propuesto, y proporciona un acceso más cómodo y automatizado desde Python a las rutinas incluidas en la librería BLACS. Dentro de PyBLACS se han presentado diferentes grupos funcionales de rutinas, y hemos mostrado ejemplos de utilización para cada una de ellas. Los ejemplos mostrados permiten apreciar la sencillez del lenguaje y de las llamadas a las rutinas de PyBLACS sin perder eficiencia en su ejecución. Para comprobar si se ha producido pérdida de rendimiento, hemos mostrado las pruebas realizadas en dos sistemas con arquitectura diferente: Seaborg y Cluster-umh. Para ambas plataformas, los tiempos obtenidos mediante un entorno de programación tradicional (programa escrito en Fortran) y los obtenidos mediante una ejecución desde un script de Python son muy similares.

Por ejemplo, la figura 6.3 nos permite comparar el ancho de banda obtenido entre dos nodos en Cluster-umh y en Seaborg. En ambas arquitecturas, el ancho de banda varía en función de la longitud del mensaje enviado siguiendo un mismo comportamiento en Fortran y en Python. Únicamente se observa una ligera penalización del ancho de banda que pensamos se compensa con la sencillez y facilidad en la utilización de las rutinas de PyBLACS.

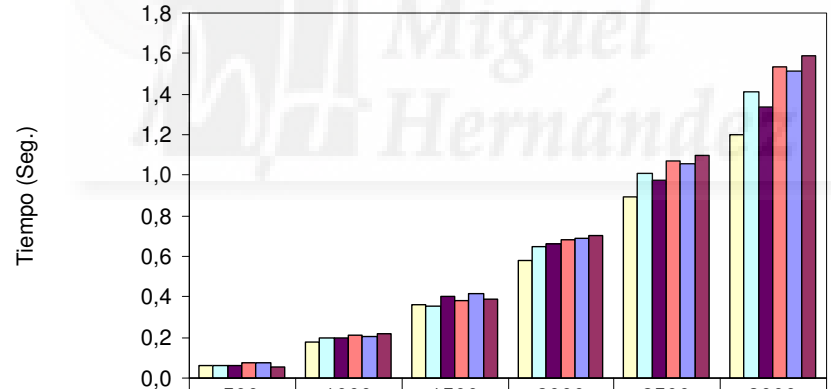
En resumen, mediante PyBLACS se propone una herramienta cómoda y sencilla para Python que permite el intercambio de datos relativos al álgebra lineal entre nodos en un sistema de memoria distribuida.



	500	1000	1500	2000	2500	3000
FORTRAN 2X1	0,06	0,24	0,53	0,94	1,49	2,13
PYTHON 2X1	0,06	0,24	0,53	0,94	1,48	2,14
FORTRAN 2X2	0,11	0,43	0,96	1,70	2,67	3,84
PYTHON 2X2	0,11	0,43	0,96	1,71	2,66	3,84
FORTRAN 3X2	0,16	0,62	1,39	2,47	3,88	5,56
PYTHON 3X2	0,16	0,62	1,39	2,47	3,86	5,57

Tamaño de la matriz

(a) Cluster-umh

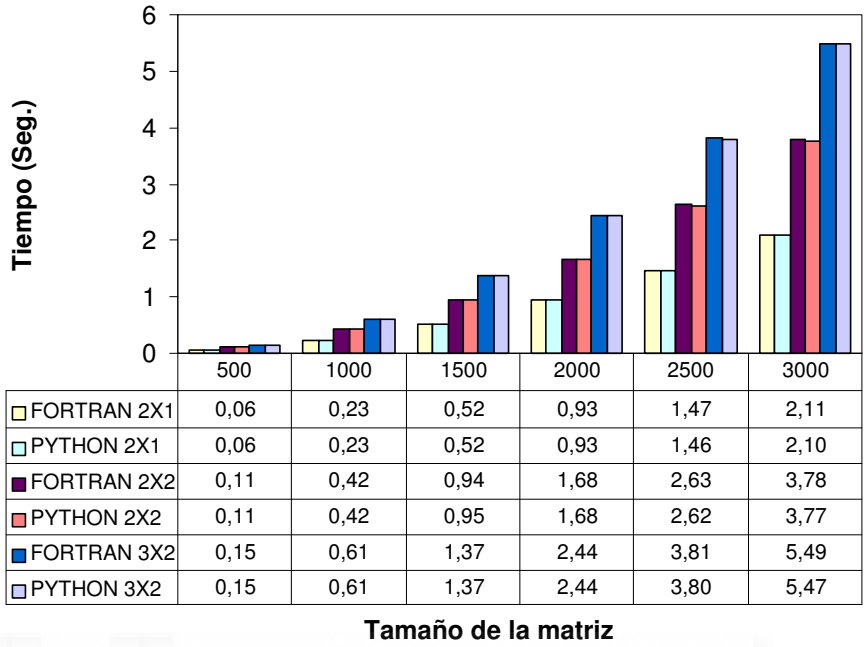


	500	1000	1500	2000	2500	3000
FORTRAN 2X1	0,06	0,18	0,36	0,58	0,90	1,20
PYTHON 2X1	0,06	0,20	0,35	0,65	1,01	1,41
FORTRAN 2X2	0,06	0,19	0,40	0,66	0,97	1,34
PYTHON 2X2	0,07	0,21	0,38	0,68	1,07	1,53
FORTRAN 3X2	0,08	0,20	0,42	0,69	1,06	1,51
PYTHON 3X2	0,05	0,22	0,39	0,70	1,10	1,59

Tamaño de la matriz

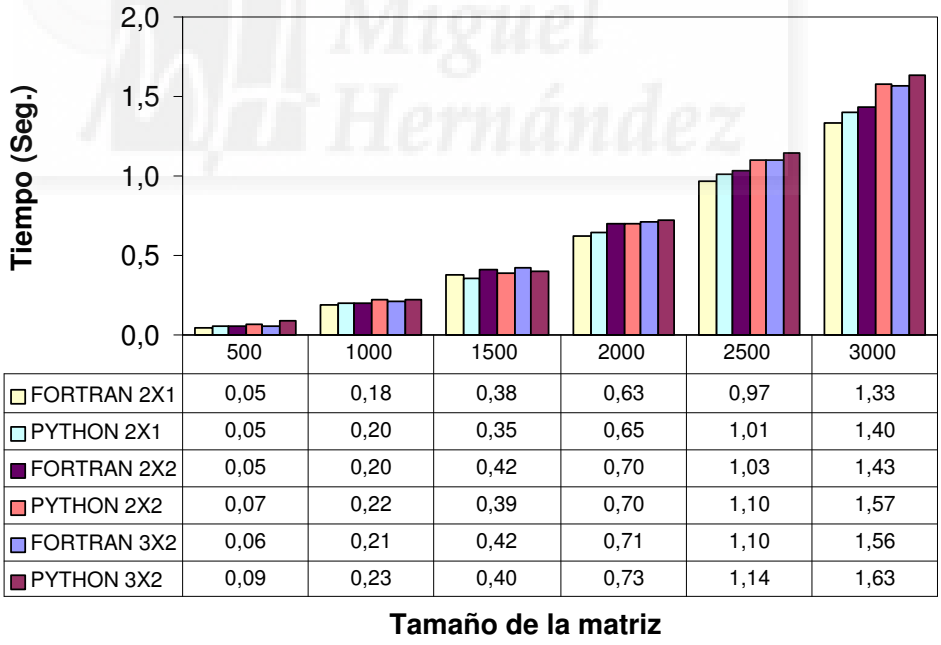
(b) Seaborg

Figura 6.6: Comparativa entre Fortran y Python para la operación combinada `gsum2d`



	500	1000	1500	2000	2500	3000
FORTRAN 2X1	0,06	0,23	0,52	0,93	1,47	2,11
PYTHON 2X1	0,06	0,23	0,52	0,93	1,46	2,10
FORTRAN 2X2	0,11	0,42	0,94	1,68	2,63	3,78
PYTHON 2X2	0,11	0,42	0,95	1,68	2,62	3,77
FORTRAN 3X2	0,15	0,61	1,37	2,44	3,81	5,49
PYTHON 3X2	0,15	0,61	1,37	2,44	3,80	5,47

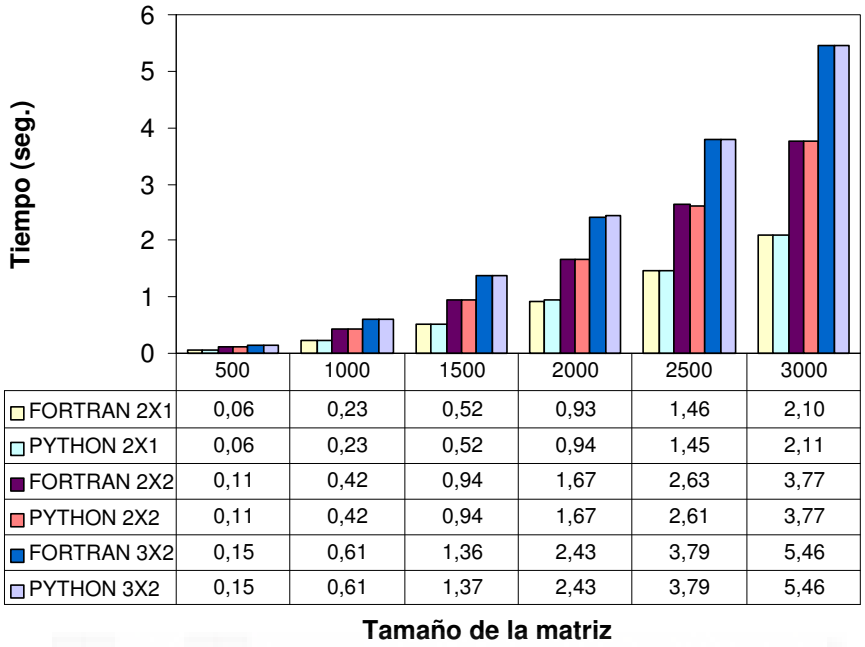
(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 2X1	0,05	0,18	0,38	0,63	0,97	1,33
PYTHON 2X1	0,05	0,20	0,35	0,65	1,01	1,40
FORTRAN 2X2	0,05	0,20	0,42	0,70	1,03	1,43
PYTHON 2X2	0,07	0,22	0,39	0,70	1,10	1,57
FORTRAN 3X2	0,06	0,21	0,42	0,71	1,10	1,56
PYTHON 3X2	0,09	0,23	0,40	0,73	1,14	1,63

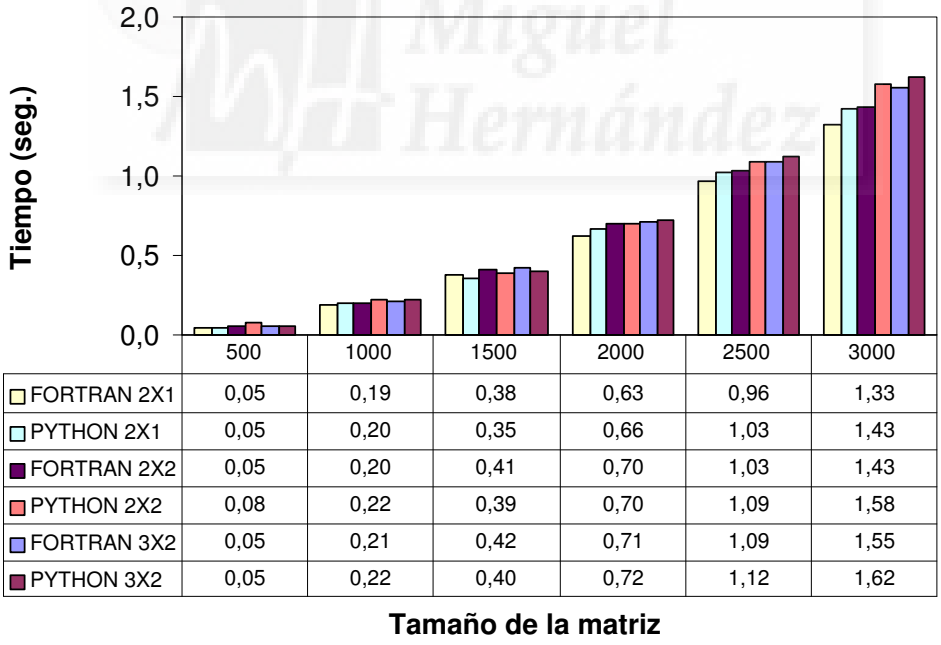
(b) Seaborg

Figura 6.7: Comparativa entre Fortran y Python para la operación combinada `gamx2d`



	500	1000	1500	2000	2500	3000
FORTRAN 2X1	0,06	0,23	0,52	0,93	1,46	2,10
PYTHON 2X1	0,06	0,23	0,52	0,94	1,45	2,11
FORTRAN 2X2	0,11	0,42	0,94	1,67	2,63	3,77
PYTHON 2X2	0,11	0,42	0,94	1,67	2,61	3,77
FORTRAN 3X2	0,15	0,61	1,36	2,43	3,79	5,46
PYTHON 3X2	0,15	0,61	1,37	2,43	3,79	5,46

(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 2X1	0,05	0,19	0,38	0,63	0,96	1,33
PYTHON 2X1	0,05	0,20	0,35	0,66	1,03	1,43
FORTRAN 2X2	0,05	0,20	0,41	0,70	1,03	1,43
PYTHON 2X2	0,08	0,22	0,39	0,70	1,09	1,58
FORTRAN 3X2	0,05	0,21	0,42	0,71	1,09	1,55
PYTHON 3X2	0,05	0,22	0,40	0,72	1,12	1,62

(b) Seaborg

Figura 6.8: Comparativa entre Fortran y Python para la operación combinada `gamn2d`

Capítulo 7

PyPBLAS

En este capítulo se presenta el módulo PyPBLAS [36, 35, 55, 39], así como el conjunto de rutinas englobadas en él. En la sección 7.1 se presenta el módulo principal y se describen sus principales características. Además, en esta primera sección se muestra la totalidad de rutinas de PyPBLAS agrupadas en tres niveles en función de la complejidad de la operación que implementan. De este modo, en las secciones 7.2, 7.3 y 7.4, se describen las rutinas del primer, segundo y tercer nivel de PyPBLAS, respectivamente. Para cada una de estas secciones se describen las principales características de PyPBLAS en dicho nivel, se muestran varios ejemplos de utilización y se ilustra el rendimiento obtenido para cada rutina de PyPBLAS comparándola con el obtenido mediante su rutina análoga de PBLAS. Por último, en la sección 7.5 analizaremos los resultados obtenidos para los diferentes niveles de PyPBLAS y obtendremos conclusiones al respecto.

7.1. Introducción

Dentro de la distribución estándar de ScaLAPACK, se encuentra el conjunto de rutinas denominado PBLAS (*Parallel Basic Linear Algebra Subprograms*, [11]). Este conjunto de rutinas resuelven las principales operaciones algebraicas relacionadas con escalares,

vectores y matrices. Se agrupan en tres niveles dependiendo de la complejidad de las operaciones implementadas:

- `Nivel 1`: Operaciones en las que intervienen únicamente vectores y escalares.
- `Nivel 2`: Operaciones entre vectores y matrices.
- `Nivel 3`: Operaciones entre matrices.

Esta agrupación se ha llevado a cabo en la distribución estándar de PBLAS. En las rutinas que mostramos a continuación hemos intentado respetar al máximo la nomenclatura de cada una de las rutinas así como la nomenclatura en los parámetros utilizados. De este modo, se consigue que cualquier persona con conocimientos básicos en el funcionamiento de PBLAS, le resulte muy fácil poder hacer uso de las rutinas implementadas en PyPBLAS. Además, muchas de las operaciones se automatizan resultando un proceso mucho más sencillo. Del mismo modo, una persona que comience a programar mediante Python y necesite hacer uso de PyPBLAS, encontrará unas rutinas más sencillas y cómodas que le facilitarán el proceso de diseño y aprendizaje. Posteriormente, si este usuario que ya ha adquirido una serie de conocimientos básicos relativos a las rutinas, desea aprender a utilizarlas a más bajo nivel (desde sus interfaces de C o Fortran), le será más sencillo el escribir un programa accediendo directamente a las rutinas de PBLAS sin utilizar las interfaces de Python.

En el presente capítulo agruparemos las rutinas de PyPBLAS por niveles (1,2,3), sin embargo, en el nombre de la rutina no se especifica ninguna relación con el nivel al que pertenece (igual que sucede con la distribución PBLAS). En nuestro caso, todas las rutinas que se corresponden con la colección PBLAS tienen el comienzo identificativo de dicha colección, es decir, las funciones se llamarán `PyPBLAS.pyblasname` desde cualquier script de Python. Se ha de tener en cuenta que las rutinas `PyPBLAS.pyblasname` se encuentran en el módulo `PyPBLAS` del paquete `PyACTS`, por lo que deberemos importar este módulo mediante `import PyACTS.PyPBLAS` o `from PyACTS.PyPBLAS import *`. El término que hemos señalado como `pyblasname` se corresponde con el nombre que cada rutina recibe dentro de la distribución PBLAS, de este modo conseguimos homogeneidad en la nomenclatura de nuestra distribución y facilitamos la migración en la utilización de PBLAS a PyPBLAS o viceversa.

Del mismo modo que se ha comentado en BLACS, los nombres de las rutinas de BLAS y, por tanto de PBLAS, dependen del tipo de dato con el que han de trabajar. De este

Letra	*Tipo*
S	Dato real de precisión simple
D	Dato real de precisión doble
C	Dato complejo de precisión simple
Z	Dato complejo de precisión doble

Tabla 7.1: Tipos de datos en PBLAS

modo, las rutinas de PBLAS tienen el siguiente aspecto `PvXXXX`, donde `P` indica que es la versión paralela de las BLAS y `v` hace referencia al tipo de dato utilizado. Los posibles valores indicativos del tipo de dato se muestran en la tabla 7.1. El valor de `XXXX` no tiene una longitud fija, y representa una abreviatura de la funcionalidad básica de dicha rutina (`scal`, `acopy`, `axpy`, ...). Sin embargo, en PyBLACS el usuario no necesita averiguar el tipo de datos para utilizar la rutina, ya que de forma interna se determina el tipo de dato que contiene la matriz PyACTS y hará uso de la rutina correspondiente de modo interno y transparente al usuario. De este modo, todas las rutinas comienzan por "`pv`" sin necesidad de indicar o averiguar el tipo de dato con el que estamos trabajando, y por tanto, facilitando el trabajo en gran medida a un investigador ajeno al uso de las librerías PBLAS.

Por último, conviene destacar que las rutinas incluidas en PyPBLAS, así como las incluidas en PyScaLAPACK (que se describirá en el capítulo 8), utilizan como datos de entrada y salida variables de tipo matriz PyACTS que ya fue descrita en el apartado 5.3. Mediante este tipo de matrices y mediante las rutinas de conversión adecuadas podremos implementar algoritmos escalables en Python de una manera más sencilla e intuitiva.

A continuación, mostramos el conjunto de rutinas de los 3 niveles definidos en PyPBLAS, indicando mediante notación matemática su funcionalidad:

- ***NIVEL 1***
  - $x \longleftrightarrow y$: `x,y= pvswap(x,y)`
  - $\alpha x \rightarrow x$: `x= pvscal(alpha,x)`
  - $x \rightarrow y$: `y= pvcopy(x)`
  - $\alpha x + y \rightarrow y$: `y= pvaxpy(alpha,x,y)`
  - $x^T y \rightarrow y$: `dot= pvdot(x,y)`

- $x^T y \rightarrow y$: `dot= pvdotu(x,y)`
- $x^H y \rightarrow y$: `dot= pvdotc(x,y)`
- $\|x\|_2 \rightarrow y$: `nrm2= pvnrm2(x)`
- $\|re(x)\|_1 + \|img(x)\|_1 \rightarrow asum$: `asum= pvasum(x)`
- $1^{st} k \ni |re(x_k)| + |img(x_k)| \rightarrow indx,\ max|re(x_i)| + |img(x_i)| \rightarrow amax$:
  `amax,indx= pvamax(x)`

▪ ***NIVEL 2***

- $\alpha op(A)x + \beta y \rightarrow y$: `y= pvgemv(alpha,a,x,beta,y[,trans])`
- $\alpha Ax + \beta y \rightarrow y$: `y= pvhemv(alpha,a,x,beta,y[,uplo])`
- $\alpha Ax + \beta y \rightarrow y$: `y= pvsymv(alpha,a,x,beta,y)`
- $Ax \rightarrow x; A^T x \rightarrow x; A^H x \rightarrow x$: `x= pvtrmv(a,x,[uplo,trans,diag])`
- $A^{-1}x \rightarrow x; A^{-T}x \rightarrow x; A^{-H}x \rightarrow x$:
  `x= pvtrsv(a,x,[uplo,trans,diag])`
- $\alpha xy^T + A \rightarrow A$: `a= pvger(alpha,x,y,a)`
- $\alpha xy^T + A \rightarrow A$: `a= pvgeru(alpha,x,y,a)`
- $\alpha xy^H + A \rightarrow A$: `a= pvgerc(alpha,x,y,a)`
- $\alpha xx^H + A \rightarrow A$: `a= pvher(alpha,x,a,[uplo])`
- $\alpha xy^H + y(\alpha x)^H + A \rightarrow A$: `a= pvher2(alpha,x,y,a,[uplo])`
- $\alpha xx^T + A \rightarrow A$: `a= pvsyr(alpha,x,a,[uplo='U'])`
- $\alpha xy^T + y(\alpha x)^T + A \rightarrow A$: `a= pvsyr2(alpha,x,y,a,[uplo])`

▪ ***NIVEL 3***

- $\alpha op(A) op(B) + \beta C \rightarrow C$:
  `c= pvgemm(alpha,a,b,beta,c,[transa,transb])`
- $\alpha AB + \beta C \rightarrow C; \alpha BA + \beta C \rightarrow C$:
  `c=pvsymm(alpha,a,b,beta,c[,side,uplo])`
- $\alpha AB + \beta C \rightarrow C; \alpha BA + \beta C \rightarrow C$:
  `c=pvhemm(alpha,a,b,beta,c[,side,uplo])`
- $\alpha AA^T + \beta C \rightarrow C; \alpha A^T A + \beta C \rightarrow C$:
  `c=pvsyrk(alpha,a,beta,c[,uplo,trans])`

- $\alpha AA^H + \beta C \rightarrow C; \alpha A^H A + \beta C \rightarrow C$:
  `c=pvherk(alpha,a,beta,c[,uplo,trans])`
- $\alpha AB^T + \alpha BA^T + \beta C \rightarrow C$:
  `c=pvsyr2k(alpha,a,b,beta,c[,uplo,trans])`
- $\alpha AB^H + \alpha BA^H + \beta C \rightarrow C$:
  `c=pvher2k(alpha,a,b,beta,c[,uplo,trans])`
- $\beta C + \alpha A^T \rightarrow C$: `c=pvtran(alpha,a,beta,c)`
- $\beta C + \alpha A^T \rightarrow C$: `c=pvtranu(alpha,a,beta,c)`
- $\beta C + \alpha A^H \rightarrow C$: `c=pvtranc(alpha,a,beta,c)`
- $\alpha op(A)B \rightarrow B; \alpha Bop(A) \rightarrow B$:
  `b=pvtrmm(alpha,a,b[,side,uplo,transa,diag])`
- $\alpha op(A^{-1})B \rightarrow B; \alpha Bop(A^{-1}) \rightarrow B$:
  `b=pvtrsm(alpha,a,b[,side,uplo,transa,diag])`

Si comparamos las rutinas de PyBLACS con las rutinas de PBLAS [11], se comprueba que el número de parámetros se reduce considerablemente en su versión para Python. Por ejemplo, la rutina `PyPBLAS.pvgemm` tiene cinco parámetros de entrada obligatorios que únicamente hacen referencia a las matrices y escalares que intervienen en la operación. En el otro lado, la rutina `PDGEMM` de PBLAS requiere de 19 parámetros de entrada:

```
SUBROUTINE PDGEMM( TRANSA, TRANSB, M, N, K, ALPHA, A, IA, JA,
$ DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC )
```

Para conseguir esta reducción en el número de parámetros de entrada, que en muchos casos significa reducir la complejidad, hemos clasificado algunos de estos parámetros como opcionales dejando un valor por defecto (por ejemplo, `trans='N',uplo='U', ...`). Estos parámetros opcionales se describen en el siguiente párrafo con mayor detalle. En otros casos, hemos conseguido reducir el número de parámetros agrupando los parámetros de entrada como matrices PyACTS. De este modo, cada matriz PyACTS tiene dos propiedades (`PyACTS.data` y `PyACTS.desc`) que contienen los datos y el descriptor de cada matriz global y que se utilizarán como datos de entrada en las llamadas a las rutinas de PBLAS en niveles inferiores de forma interna y oculta al usuario de PyPBLAS.

Los parámetros que se han asumido como opcionales, pueden indicarse de forma explícita si se desea un valor diferente al que se obtiene por defecto. Por ejemplo, si

deseamos realizar la operación $\alpha A^T B + \beta C \rightarrow C$, debemos indicarlo en la llamada a la rutina: `PyBLAS.pvgemm(alpha,a,b,beta,c,transa='T')`. En este caso, no es necesario especificar `transb` puesto que por defecto no se desea realizar la traspuesta a la matriz `b`. Creemos conveniente recordar que los escalares, los vectores y las matrices deben de ser previamente convertidos a matrices o escalares PyACTS con las rutinas descritas en el apartado 5.5.

El significado de cada uno de los parámetros opcionales se detalla a continuación:

- `trans`: Indica si se realiza una operación o acción sobre la matriz. En la notación matemática se ha descrito como "*op*( )" y sus posibles valores son :
  - `trans='N'`: Valor por defecto. No se realiza operación sobre la matriz.
  - `trans='T'`: Se realiza la traspuesta.
  - `trans='C'`: Se realiza la traspuesta conjugada en el caso de matrices con sus elementos de tipo complejo.
- `uplo`: En matrices triangulares, indicamos si la matriz es triangular superior (`trans='U'`, valor por defecto) o triangular inferior (`trans='L'`), es decir, indicamos en que parte de la diagonal se encuentran los datos con los que operar.
- `diag`: Se utiliza para especificar si la matriz `a` es triangular unitaria (`diag='U'`), o bien no se asume que ésta lo sea (`diag='N'`, valor por defecto).

7.2. Rutinas PyPBLAS de nivel 1

Las rutinas de nivel 1 de PBLAS agrupan las funciones computacionalmente más sencillas, y en las que intervienen únicamente vectores y escalares. El resultado de cada rutina suele ser un vector o un escalar y depende de la funcionalidad de cada rutina. Las funcionalidades de las rutinas agrupadas en el nivel 1 se han mostrado en el apartado anterior y si se desea más información de los parámetros de entrada y salida, así como ejemplos de todas y cada una de las rutinas, recomendamos la lectura del manual de usuario de PyACTS [34].

Se expone a continuación, un ejemplo de utilización de una de ellas como muestra de la sencillez en el manejo de dichas rutinas. En este caso, se realiza una llamada a la rutina `PyPBLAS.pvaxpy` que implementa una multiplicación de un escalar por un vector y una adición del resultado a otro vector ($\alpha x + y \rightarrow y$).

```
1  from PyACTS import *
2  import PyACTS.PyPBLAS as PyPBLAS
3  #Inicializamos la malla con los parámetros por defecto
4  gridinit()
5  #Indicamos el tipo de matriz PyACTS
6  ACTS_lib=1 # 1=Scalapack
7  #Generamos los datos de forma aleatoria
8  x=Rand2PyACTS(n,1,ACTS_lib)
9  y=Rand2PyACTS(n,1,ACTS_lib)
10 alpha=Scal2PyACTS(2,ACTS_lib)
11 #Llamamos a la función
12 y=PyPBLAS.pvaxpy(alpha,x,y)
13 #Liberamos la malla de procesos
14 gridexit()
```

Como se observa en el ejemplo, primero se importan los módulos correspondientes de PyACTS. En la primera línea, mediante `from PyACTS import *` se importan todos los objetos y funciones definidos en el módulo principal (es decir, aquellas rutinas definidas en el archivo `__init__.py`). De este modo, podremos llamar directamente a las rutinas `gridinit`, `gridexit`, `Rand2PyACTS` y `Scal2PyACTS`. Sin embargo, si en lugar de importar todos los símbolos que contiene dicho módulo, se importa únicamente el módulo, deberemos referenciar a cada rutina mediante el prefijo "`PyACTS.`" (por ejemplo, `PyACTS.gridinit`, `PyACTS.gridexit`, `PyACTS.Rand2PyACTS`, ...). Además, recordamos que la funcionalidad de estas rutinas de inicialización, liberación y conversión de datos ya fueron introducidas en el apartado 5.5 de este trabajo.

Como ya se ha comentado, las matrices o vectores de entrada y salida de las rutinas de PyPBLAS han de estar en el formato de una matriz PyACTS por lo que de forma previa necesitaremos convertir o generar los vectores a dicho formato. Además, se ha de reseñar que el escalar también se ha de convertir a formato PyACTS. La razón para definir este tipo de necesidades viene marcada por la exigencia de verificación de los datos, es decir, antes de llamar a una rutina en niveles inferiores (en este caso PBLAS), se ha de comprobar si el dato se encuentra correctamente distribuido. De comprobar la coherencia

de la distribución de los datos se encarga de forma transparente al usuario la distribución PyACTS en todas las llamadas a cualquiera de sus rutinas. En el caso que uno de los parámetros de entrada fuera una matriz de tipo `Numeric`, y no de tipo PyACTS, antes de la ejecución de la rutina de PBLAS, se devuelve un mensaje de error informando de tipo incorrecto en los parámetros.

En la línea 12 se realiza la llamada a la rutina de nivel 1 `pvaxpy`. Como se observa en el ejemplo, esta rutina tiene un parámetro de salida (`y=pvaxpy(alpha,x,y)`) que se denomina igual que uno de los parámetros de entrada. Esta denominación no es casual puesto que el resultado de dicha operación ($\alpha x+y \rightarrow y$) se almacenará en la variable de Python `y`. Esta aclaración puede no sorprender al lector ya que el resultado que devuelve dicha función se sobreescribe en `y`. Sin embargo, si escribiéramos el código "`z=pvaxpy(alpha,x,y)`" parece claro pensar que el resultado se almacenará en una matriz PyACTS denominada `z`. Esto es cierto, no obstante mediante esta asignación podemos pensar que el valor de `y` no ha sido modificado y es en este punto donde estaríamos equivocados. De forma general, en la distribución PyACTS el contenido de la memoria de los parámetros de entrada se reescribe para proporcionar la salida. De este modo, en este ejemplo el contenido de `z` e `y` sería el mismo puesto que ambas variables apuntan a la misma dirección de memoria. Además de esto, si ya sabemos que el contenido de la variable `y` se va a sobreescribir podemos realizarnos la siguiente pregunta: "Si no se realiza una asignación de memoria dinámica al resultado, ¿qué sentido tiene utilizar por tanto variables de salida?", y la respuesta a dicha pregunta es que se han diseñado valores del script de salida para facilitar la interpretación y legibilidad de los códigos que utilicen la distribución PyACTS, ya que al utilizar parámetros de entrada y salida (a la derecha e izquierda del signo "=" respectivamente) se identifican éstos de una forma clara y sencilla.

Una solución alternativa a la característica descrita anteriormente puede ser que la rutina de PyPBLAS de forma interna genere una nueva matriz PyACTS y copie el contenido de la matriz de entrada en la que supuestamente se sobreescribirían los datos. De este modo preservamos el contenido original de todas las variables de entrada. Lamentablemente, esta solución se considera inadecuada ya que requiere mucho más espacio de memoria al duplicar una matriz que puede ser de un elevado tamaño. Además de esto, se consume un tiempo adicional considerable en crear y copiar el contenido de los datos desde el parámetro de entrada a esta nueva variable de salida.

Por tanto, de forma general para las rutinas incluidas en la distribución PyACTS, cuando el parámetro de entrada y de salida tengan la misma nomenclatura (por ejem-

plo, `c=pvherk(alpha,a,beta,c)`, o `b=pvtrmm(alpha,a,b)`) indicará que el contenido de dicha variable se sobrescribirá.

Para comprobar el correcto funcionamiento de cada una de las funciones agrupadas en el nivel 1 de PyPBLAS, se han realizado una serie de pruebas que se muestran a continuación. En esta serie de gráficas se exponen, para cada una de las rutinas, los tiempos de ejecución obtenidos para diferentes tamaños de vectores y para diferentes configuraciones de mallas de procesos. Las operaciones de nivel 1 de PyPBLAS no son costosas computacionalmente y no tienen una alta carga en las comunicaciones por lo que se obtienen tiempos de ejecución considerablemente menores con respecto al resto de rutinas de PyPBLAS. Es quizás, en estas interfaces para PyPBLAS de nivel 1 donde más se pueda apreciar la penalización por las interfaces debido únicamente al reducido tiempo de ejecución que lo hace sensible a pequeños incrementos.

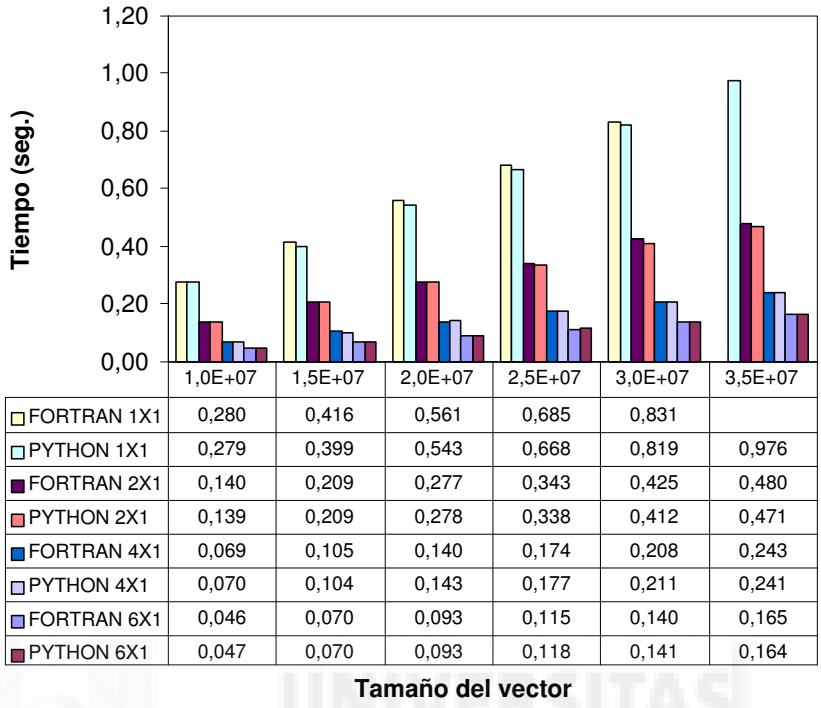
Por ejemplo, la figura 7.1 muestra los tiempos de ejecución obtenidos para realizar el intercambio de datos entre dos variables mediante la rutina de nivel 1 `pvswap`. Se ha de tener en cuenta que este intercambio de datos se produce a nivel de memoria, es decir, en el nivel más bajo y no se corresponde con un intercambio del puntero de cada una de las variables. Para todas las rutinas se han realizado pruebas en dos sistemas bien diferenciados: Cluster-umh y Seaborg. A pesar de ser sistemas con arquitecturas y estructuras de comunicaciones diferentes, el comportamiento en ambas es similar. En este primer ejemplo, se aprecia cómo la penalización sufrida por utilizar las interfaces desde Python no es demasiado elevada, en todo caso en el Cluster-umh nunca supera el 3 %. Además se ha de tener en cuenta que los tiempos de ejecución mostrados para el nivel 1 son muy pequeños debido a la sencillez de los cálculos a realizar, o dicho de otro modo, la carga computacional de los algoritmos de nivel 1 de PyPBLAS es muy baja.

El comportamiento observado para la rutina `pvswap` se observa también para el resto de las rutinas de nivel 1. En la figura 7.2 se muestran los tiempos obtenidos para la ejecución de la rutina `pvscal`, donde el comportamiento entre la interfaz para Python y la interfaz tradicional para Fortran es muy similar. Un detalle importante a tener en cuenta en las figuras de los tres niveles es el tipo de dato utilizado. El tipo de datos de una matriz PyACTS viene determinado por el tipo de datos que la matriz `Numeric` de la propiedad `PyACTS.data`. Si la propiedad `PyACTS.data` es `Float32`, se corresponde con precisión simple y viene identificado con el caracter "`s`"; por otro lado, si la matriz se define como `Float` se corresponde con precisión doble y está identificada con el carácter "`d`"). Si disponemos de una matriz `Numeric` y deseamos consultar su tipo de dato pode-

mos hacerlo mediante la propiedad `typecode` (por ejemplo, `print a.data.typecode()` imprime el caracter correspondiente a su tipo de datos). De forma general (incluido en el tipo complejo), utilizaremos elementos de matrices o vectores de doble precisión en todas las pruebas que se muestran para los tres niveles de PyPBLAS. De este modo, si ejecutamos la rutina de PyPBLAS `pvswap` y el tipo de dato definido es de doble precisión, desde el método `PyPBLAS.pvswap` se averigua el tipo de dato utilizado y llamaremos a la rutina `pyscalapack.pdswap`. La rutina `pyscalapack.pdswap` es la interfaz a Python desde la librería de objetos `pyscalapack.so` y se corresponde con la rutina `PDSWAP` de PBLAS.

Del mismo modo que para la rutina `pvswap`, se muestran las comparativas en los tiempos de ejecución obtenidos para las rutinas `pvcopy` (figura 7.3), `pvaxpy` (figura 7.4), `pvdot` (figura 7.5), `pvdotc` (figura 7.6), `pvdotu` (figura 7.7), `pvnrm2` (figura 7.8), `pvasum` (figura 7.9), y `pvamax` (figura 7.10).

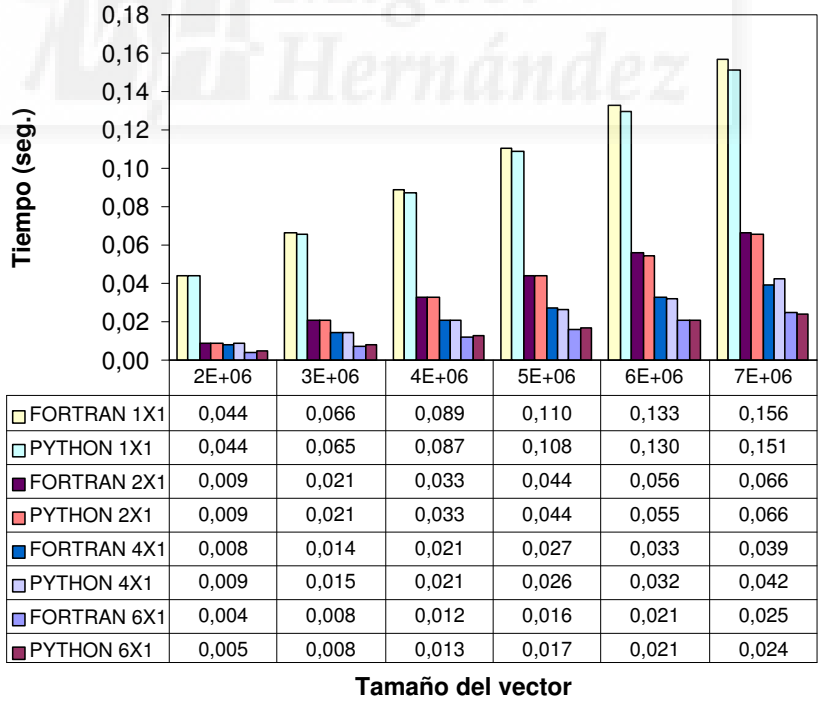
Además de obtener tiempos y comportamiento similares entre las interfaces de Python y Fortran, deseamos destacar una peculiaridad que se refleja de forma clara en las figuras 7.1(a), 7.6(b) y 7.7(b). En ellas se observa que para las configuraciones de la malla de procesos 1×1 en su ejecución con la interfaz de Fortran, no se refleja ningún dato, es decir, dicha celda está en blanco. La razón de esta ausencia del dato se debe a que la ejecución proporcionaba un fallo de memoria, puesto que era necesario reservar más memoria para la ejecución y al aumentar la memoria necesaria del vector, el compilador informaba que no había más memoria disponible. Sin embargo, cuando se intentaba la ejecución en Python con ese tamaño de vector o vectores, se ejecutaba el proceso sin problemas de memoria. En resumen, para determinados tamaños de vectores encontramos problemas en la gestión de memoria con Fortran, sin embargo si definimos las matrices mediante Python esos tamaños se pueden ejecutar en el sistema. La razón de esta ventaja de Python frente a Fortran radica en que Python maneja y almacena en memoria las matrices y vectores de forma transparente al usuario, mientras que la programación implementada en Fortran nos obliga a contabilizar los bytes que disponemos en memoria y el tamaño de cada vector. Por otro lado, consideramos que seguramente sea posible optimizar el compilador para ampliar los recursos de memoria disponible, pero hemos de recordar que tratamos de facilitar el acceso a las herramientas de altas prestaciones para usuarios no avanzados en conocimientos de programación. Como vemos en las figuras, hemos conseguido mejorar la gestión de los recursos de memoria y ocultarla al programador, consiguiendo una herramienta más cómoda, sencilla y que maneja mejor los recursos de memoria.



	1,0E+07	1,5E+07	2,0E+07	2,5E+07	3,0E+07	3,5E+07
FORTRAN 1X1	0,280	0,416	0,561	0,685	0,831	
PYTHON 1X1	0,279	0,399	0,543	0,668	0,819	0,976
FORTRAN 2X1	0,140	0,209	0,277	0,343	0,425	0,480
PYTHON 2X1	0,139	0,209	0,278	0,338	0,412	0,471
FORTRAN 4X1	0,069	0,105	0,140	0,174	0,208	0,243
PYTHON 4X1	0,070	0,104	0,143	0,177	0,211	0,241
FORTRAN 6X1	0,046	0,070	0,093	0,115	0,140	0,165
PYTHON 6X1	0,047	0,070	0,093	0,118	0,141	0,164

Tamaño del vector

(a) Cluster-umh

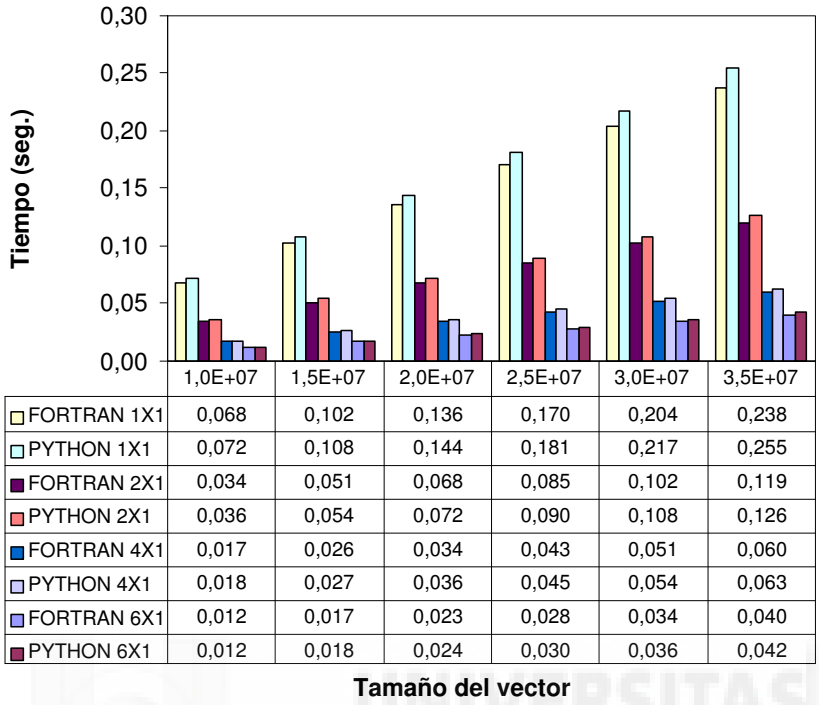


	2E+06	3E+06	4E+06	5E+06	6E+06	7E+06
FORTRAN 1X1	0,044	0,066	0,089	0,110	0,133	0,156
PYTHON 1X1	0,044	0,065	0,087	0,108	0,130	0,151
FORTRAN 2X1	0,009	0,021	0,033	0,044	0,056	0,066
PYTHON 2X1	0,009	0,021	0,033	0,044	0,055	0,066
FORTRAN 4X1	0,008	0,014	0,021	0,027	0,033	0,039
PYTHON 4X1	0,009	0,015	0,021	0,026	0,032	0,042
FORTRAN 6X1	0,004	0,008	0,012	0,016	0,021	0,025
PYTHON 6X1	0,005	0,008	0,013	0,017	0,021	0,024

Tamaño del vector

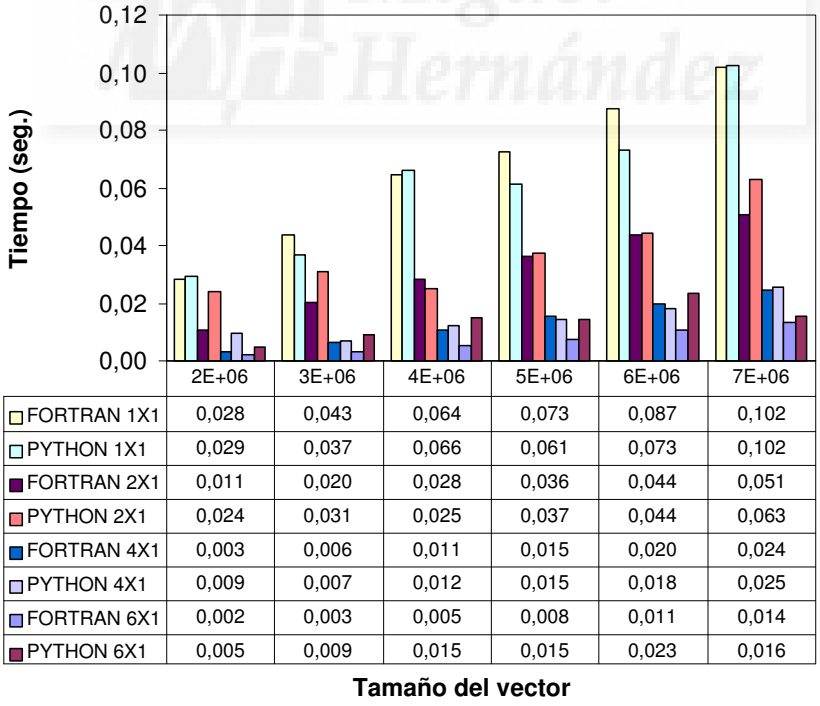
(b) Seaborg

Figura 7.1: Comparativa de la rutina `pvswap` en lenguaje Fortran vs Python



	1,0E+07	1,5E+07	2,0E+07	2,5E+07	3,0E+07	3,5E+07
FORTRAN 1X1	0,068	0,102	0,136	0,170	0,204	0,238
PYTHON 1X1	0,072	0,108	0,144	0,181	0,217	0,255
FORTRAN 2X1	0,034	0,051	0,068	0,085	0,102	0,119
PYTHON 2X1	0,036	0,054	0,072	0,090	0,108	0,126
FORTRAN 4X1	0,017	0,026	0,034	0,043	0,051	0,060
PYTHON 4X1	0,018	0,027	0,036	0,045	0,054	0,063
FORTRAN 6X1	0,012	0,017	0,023	0,028	0,034	0,040
PYTHON 6X1	0,012	0,018	0,024	0,030	0,036	0,042

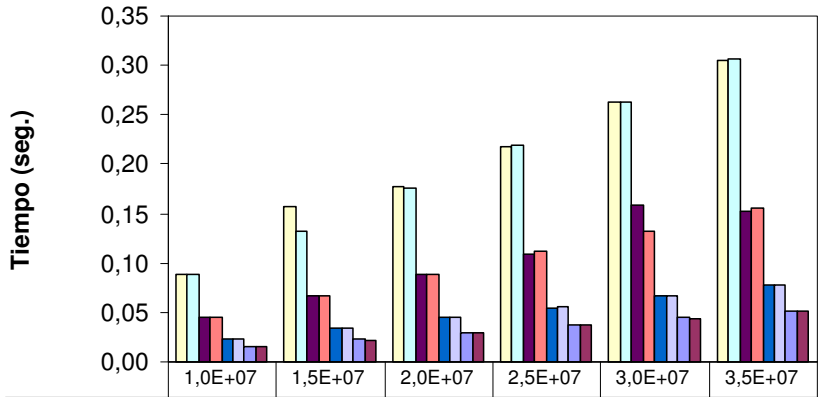
(a) Cluster-umh



	2E+06	3E+06	4E+06	5E+06	6E+06	7E+06
FORTRAN 1X1	0,028	0,043	0,064	0,073	0,087	0,102
PYTHON 1X1	0,029	0,037	0,066	0,061	0,073	0,102
FORTRAN 2X1	0,011	0,020	0,028	0,036	0,044	0,051
PYTHON 2X1	0,024	0,031	0,025	0,037	0,044	0,063
FORTRAN 4X1	0,003	0,006	0,011	0,015	0,020	0,024
PYTHON 4X1	0,009	0,007	0,012	0,015	0,018	0,025
FORTRAN 6X1	0,002	0,003	0,005	0,008	0,011	0,014
PYTHON 6X1	0,005	0,009	0,015	0,015	0,023	0,016

(b) Seaborg

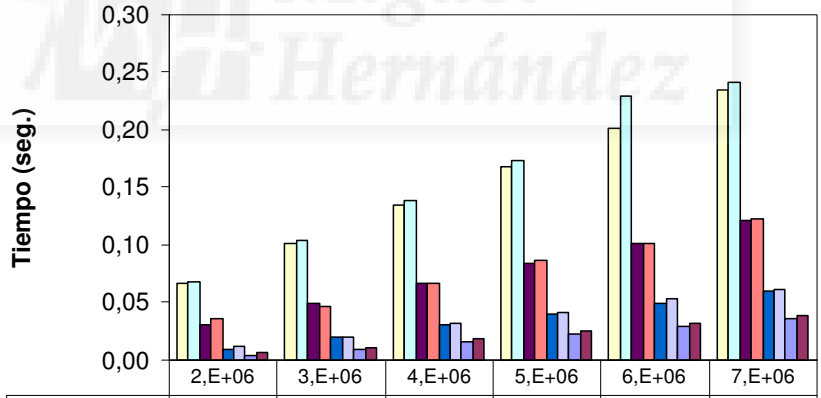
Figura 7.2: Comparativa de la rutina `pvscal` en lenguaje Fortran vs Python



	1,0E+07	1,5E+07	2,0E+07	2,5E+07	3,0E+07	3,5E+07
FORTRAN 1X1	0,088	0,157	0,178	0,218	0,263	0,305
PYTHON 1X1	0,088	0,132	0,176	0,220	0,263	0,307
FORTRAN 2X1	0,045	0,067	0,088	0,109	0,158	0,153
PYTHON 2X1	0,045	0,067	0,089	0,111	0,133	0,155
FORTRAN 4X1	0,023	0,033	0,044	0,055	0,067	0,077
PYTHON 4X1	0,023	0,034	0,045	0,056	0,067	0,078
FORTRAN 6X1	0,015	0,023	0,030	0,037	0,044	0,052
PYTHON 6X1	0,015	0,022	0,030	0,037	0,044	0,052

Tamaño del vector

(a) Cluster-umh

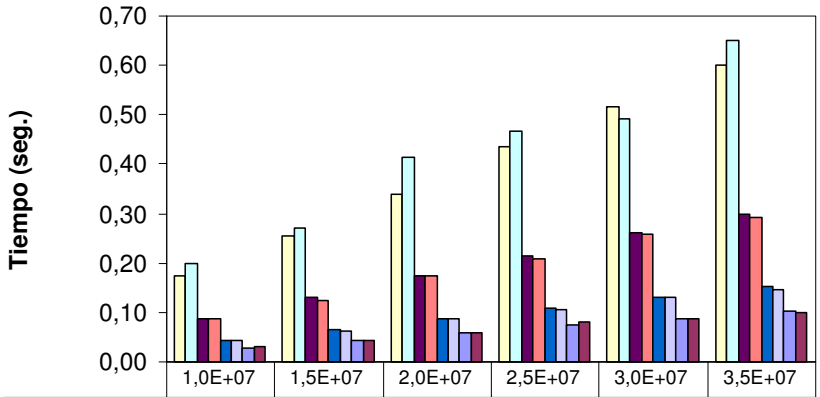


	2,E+06	3,E+06	4,E+06	5,E+06	6,E+06	7,E+06
FORTRAN 1X1	0,067	0,101	0,134	0,167	0,201	0,234
PYTHON 1X1	0,068	0,105	0,139	0,173	0,229	0,242
FORTRAN 2X1	0,030	0,049	0,067	0,084	0,101	0,121
PYTHON 2X1	0,037	0,047	0,067	0,087	0,101	0,123
FORTRAN 4X1	0,010	0,020	0,030	0,040	0,049	0,059
PYTHON 4X1	0,012	0,020	0,031	0,041	0,054	0,061
FORTRAN 6X1	0,005	0,009	0,016	0,023	0,030	0,037
PYTHON 6X1	0,006	0,011	0,018	0,025	0,032	0,038

Tamaño del vector

(b) Seaborg

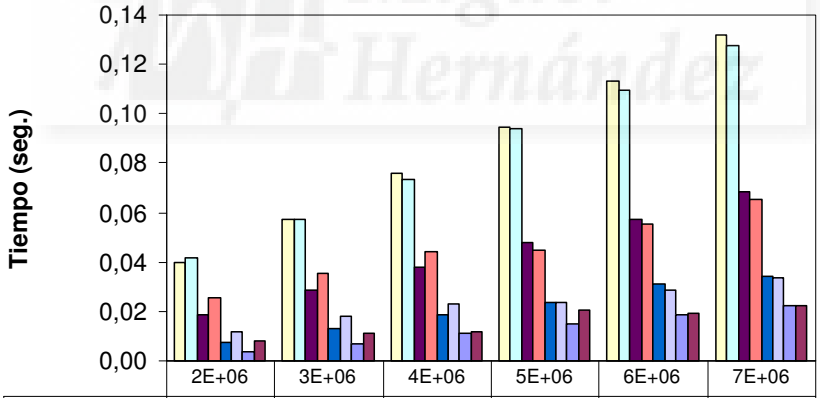
Figura 7.3: Comparativa de la rutina `pvcopy` en lenguaje Fortran vs Python



	1,0E+07	1,5E+07	2,0E+07	2,5E+07	3,0E+07	3,5E+07
FORTRAN 1X1	0,175	0,256	0,340	0,435	0,517	0,599
PYTHON 1X1	0,198	0,269	0,413	0,468	0,493	0,649
FORTRAN 2X1	0,087	0,131	0,175	0,216	0,260	0,297
PYTHON 2X1	0,086	0,125	0,174	0,210	0,258	0,294
FORTRAN 4X1	0,044	0,066	0,088	0,110	0,131	0,151
PYTHON 4X1	0,043	0,063	0,087	0,107	0,129	0,147
FORTRAN 6X1	0,029	0,044	0,058	0,074	0,087	0,102
PYTHON 6X1	0,031	0,044	0,058	0,081	0,086	0,100

Tamaño de la matriz

(a) Cluster-umh

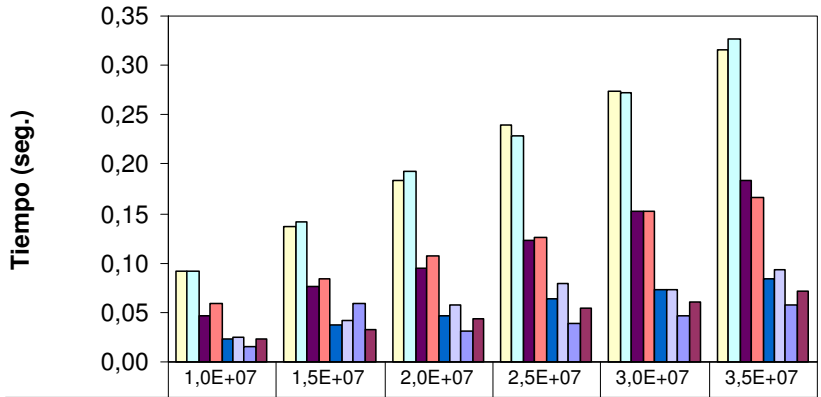


	2E+06	3E+06	4E+06	5E+06	6E+06	7E+06
FORTRAN 1X1	0,040	0,057	0,076	0,095	0,113	0,132
PYTHON 1X1	0,042	0,058	0,073	0,094	0,110	0,128
FORTRAN 2X1	0,019	0,029	0,038	0,048	0,057	0,068
PYTHON 2X1	0,026	0,035	0,044	0,045	0,055	0,065
FORTRAN 4X1	0,007	0,013	0,018	0,023	0,031	0,034
PYTHON 4X1	0,012	0,018	0,023	0,024	0,028	0,033
FORTRAN 6X1	0,004	0,007	0,011	0,015	0,019	0,022
PYTHON 6X1	0,008	0,011	0,012	0,021	0,019	0,022

Tamaño del vector

(b) Seaborg

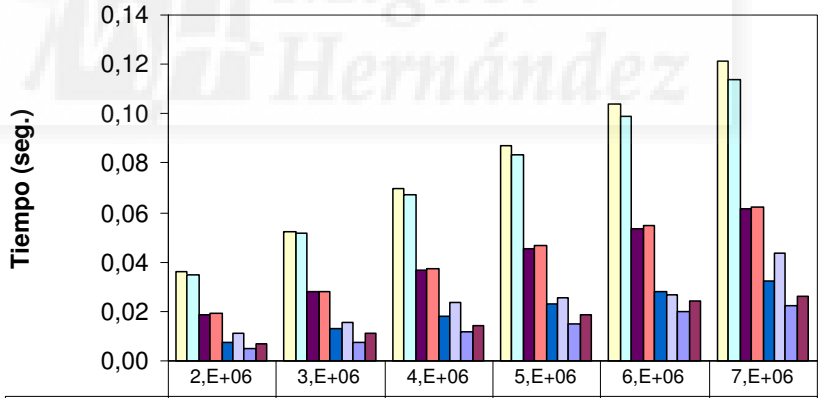
Figura 7.4: Comparativa de la rutina `pvaxpy` en lenguaje Fortran vs Python



	1,0E+07	1,5E+07	2,0E+07	2,5E+07	3,0E+07	3,5E+07
FORTRAN 1X1	0,091	0,137	0,184	0,239	0,274	0,316
PYTHON 1X1	0,091	0,142	0,194	0,229	0,271	0,326
FORTRAN 2X1	0,047	0,077	0,096	0,123	0,153	0,184
PYTHON 2X1	0,059	0,084	0,108	0,126	0,153	0,167
FORTRAN 4X1	0,024	0,038	0,047	0,063	0,074	0,084
PYTHON 4X1	0,026	0,042	0,057	0,079	0,074	0,093
FORTRAN 6X1	0,016	0,060	0,032	0,039	0,047	0,057
PYTHON 6X1	0,024	0,033	0,044	0,054	0,061	0,072

Tamaño del vector

(a) Cluster-umh

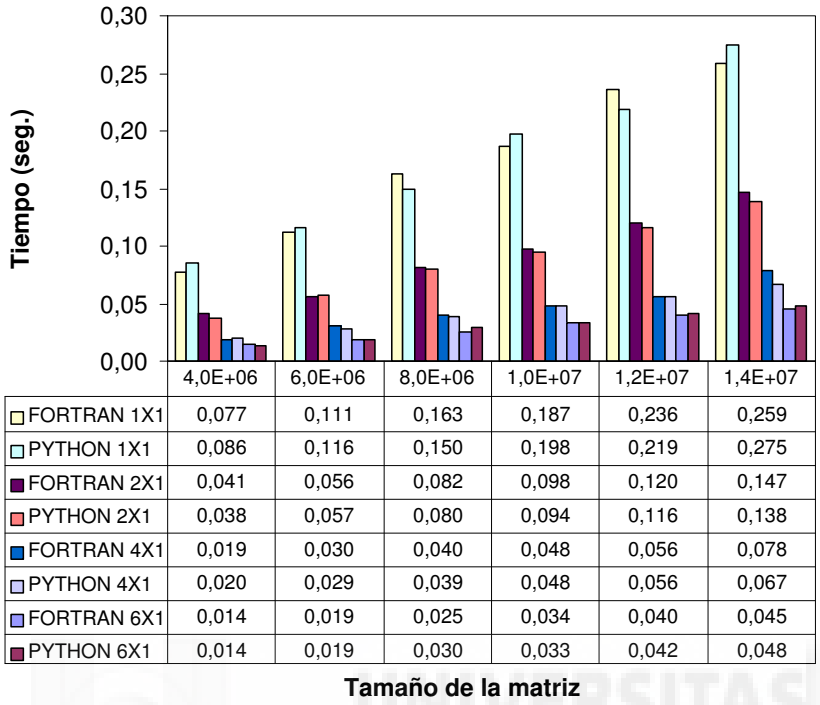


	2,E+06	3,E+06	4,E+06	5,E+06	6,E+06	7,E+06
FORTRAN 1X1	0,036	0,052	0,070	0,087	0,104	0,121
PYTHON 1X1	0,035	0,052	0,067	0,083	0,099	0,114
FORTRAN 2X1	0,018	0,028	0,037	0,045	0,053	0,062
PYTHON 2X1	0,019	0,028	0,038	0,047	0,054	0,062
FORTRAN 4X1	0,007	0,013	0,018	0,023	0,028	0,032
PYTHON 4X1	0,011	0,015	0,024	0,026	0,027	0,044
FORTRAN 6X1	0,005	0,008	0,012	0,015	0,020	0,023
PYTHON 6X1	0,007	0,011	0,014	0,019	0,024	0,026

Tamaño del vector

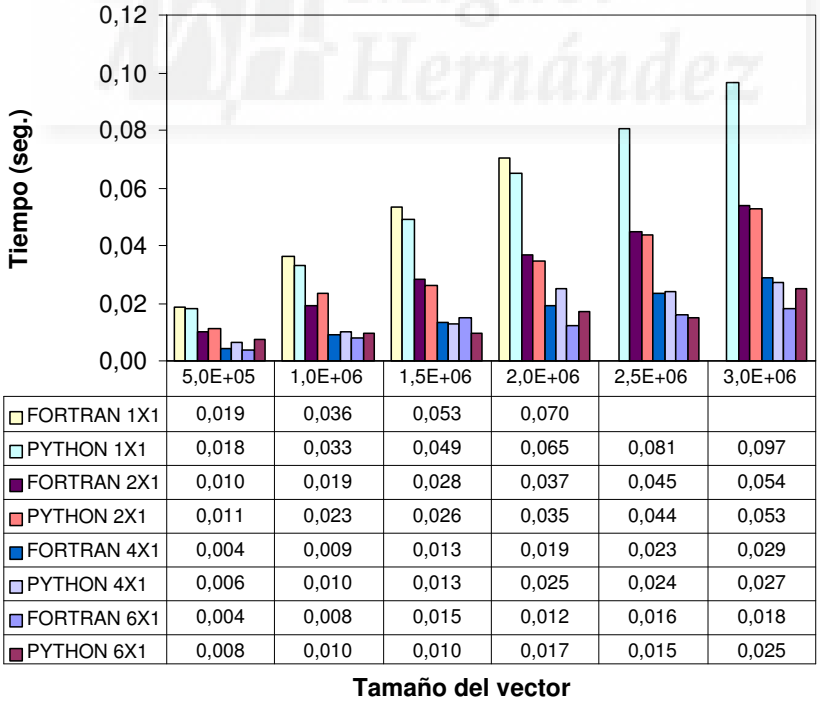
(b) Seaborg

Figura 7.5: Comparativa de la rutina `pvdot` en lenguaje Fortran vs Python



	4,0E+06	6,0E+06	8,0E+06	1,0E+07	1,2E+07	1,4E+07
FORTRAN 1X1	0,077	0,111	0,163	0,187	0,236	0,259
PYTHON 1X1	0,086	0,116	0,150	0,198	0,219	0,275
FORTRAN 2X1	0,041	0,056	0,082	0,098	0,120	0,147
PYTHON 2X1	0,038	0,057	0,080	0,094	0,116	0,138
FORTRAN 4X1	0,019	0,030	0,040	0,048	0,056	0,078
PYTHON 4X1	0,020	0,029	0,039	0,048	0,056	0,067
FORTRAN 6X1	0,014	0,019	0,025	0,034	0,040	0,045
PYTHON 6X1	0,014	0,019	0,030	0,033	0,042	0,048

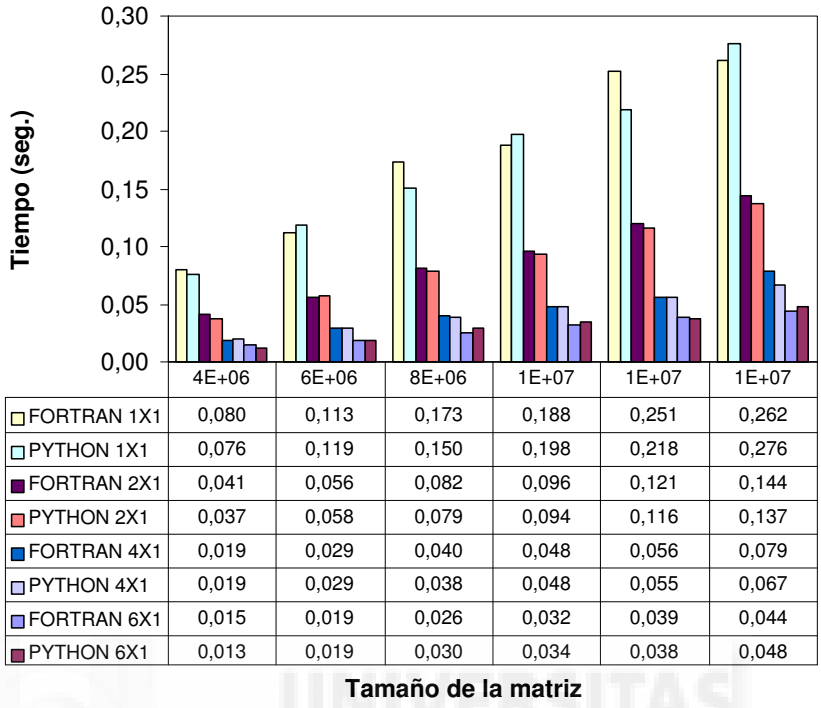
(a) Cluster-umh



	5,0E+05	1,0E+06	1,5E+06	2,0E+06	2,5E+06	3,0E+06
FORTRAN 1X1	0,019	0,036	0,053	0,070		
PYTHON 1X1	0,018	0,033	0,049	0,065	0,081	0,097
FORTRAN 2X1	0,010	0,019	0,028	0,037	0,045	0,054
PYTHON 2X1	0,011	0,023	0,026	0,035	0,044	0,053
FORTRAN 4X1	0,004	0,009	0,013	0,019	0,023	0,029
PYTHON 4X1	0,006	0,010	0,013	0,025	0,024	0,027
FORTRAN 6X1	0,004	0,008	0,015	0,012	0,016	0,018
PYTHON 6X1	0,008	0,010	0,010	0,017	0,015	0,025

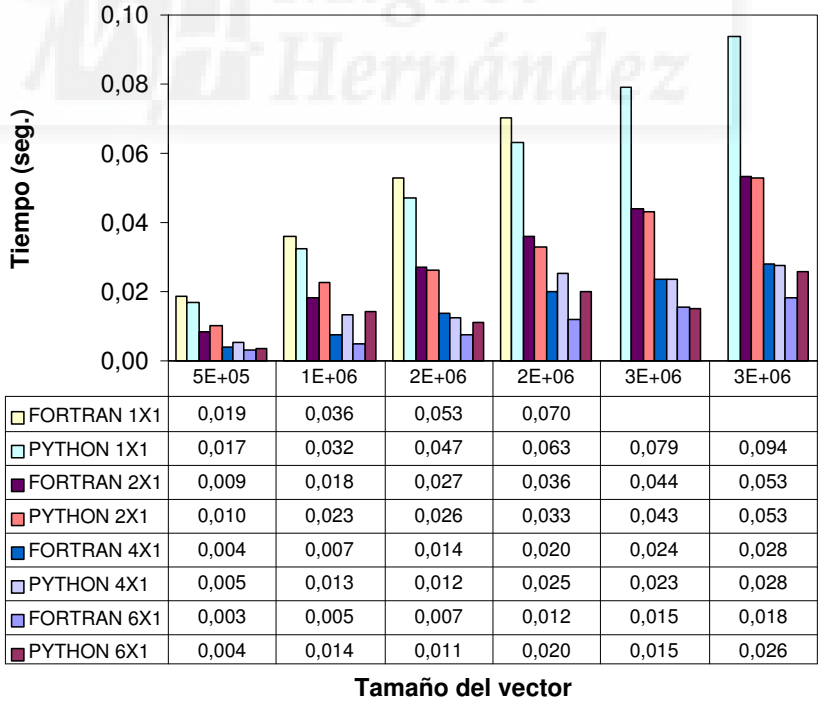
(b) Seaborg

Figura 7.6: Comparativa de la rutina `pvdotc` en lenguaje Fortran vs Python



	4E+06	6E+06	8E+06	1E+07	1E+07	1E+07
FORTRAN 1X1	0,080	0,113	0,173	0,188	0,251	0,262
PYTHON 1X1	0,076	0,119	0,150	0,198	0,218	0,276
FORTRAN 2X1	0,041	0,056	0,082	0,096	0,121	0,144
PYTHON 2X1	0,037	0,058	0,079	0,094	0,116	0,137
FORTRAN 4X1	0,019	0,029	0,040	0,048	0,056	0,079
PYTHON 4X1	0,019	0,029	0,038	0,048	0,055	0,067
FORTRAN 6X1	0,015	0,019	0,026	0,032	0,039	0,044
PYTHON 6X1	0,013	0,019	0,030	0,034	0,038	0,048

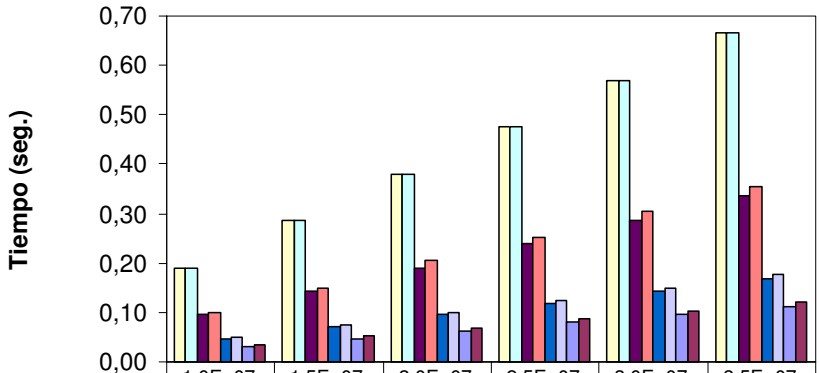
(a) Cluster-umh



	5E+05	1E+06	2E+06	2E+06	3E+06	3E+06
FORTRAN 1X1	0,019	0,036	0,053	0,070		
PYTHON 1X1	0,017	0,032	0,047	0,063	0,079	0,094
FORTRAN 2X1	0,009	0,018	0,027	0,036	0,044	0,053
PYTHON 2X1	0,010	0,023	0,026	0,033	0,043	0,053
FORTRAN 4X1	0,004	0,007	0,014	0,020	0,024	0,028
PYTHON 4X1	0,005	0,013	0,012	0,025	0,023	0,028
FORTRAN 6X1	0,003	0,005	0,007	0,012	0,015	0,018
PYTHON 6X1	0,004	0,014	0,011	0,020	0,015	0,026

(b) Seaborg

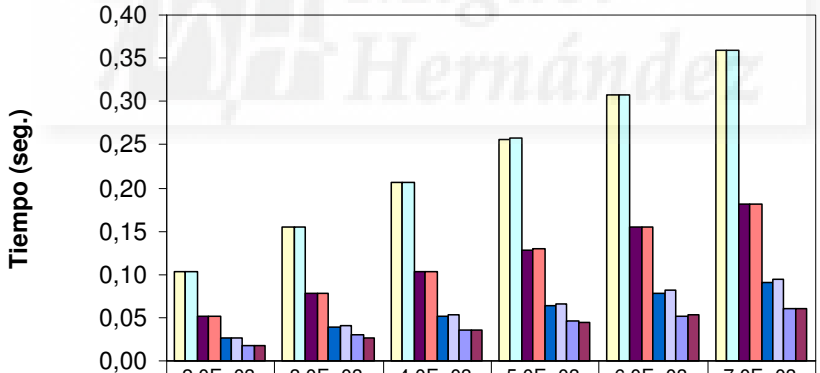
Figura 7.7: Comparativa de la rutina `pvdotu` en lenguaje Fortran vs Python



	1,0E+07	1,5E+07	2,0E+07	2,5E+07	3,0E+07	3,5E+07
FORTRAN 1X1	0,190	0,285	0,380	0,475	0,570	0,665
PYTHON 1X1	0,190	0,285	0,380	0,475	0,570	0,665
FORTRAN 2X1	0,095	0,144	0,191	0,239	0,287	0,335
PYTHON 2X1	0,101	0,150	0,204	0,252	0,304	0,353
FORTRAN 4X1	0,048	0,071	0,095	0,119	0,144	0,167
PYTHON 4X1	0,049	0,075	0,099	0,126	0,150	0,178
FORTRAN 6X1	0,032	0,048	0,063	0,079	0,095	0,111
PYTHON 6X1	0,033	0,052	0,068	0,088	0,102	0,121

Tamaño de la matriz

(a) Cluster-umh

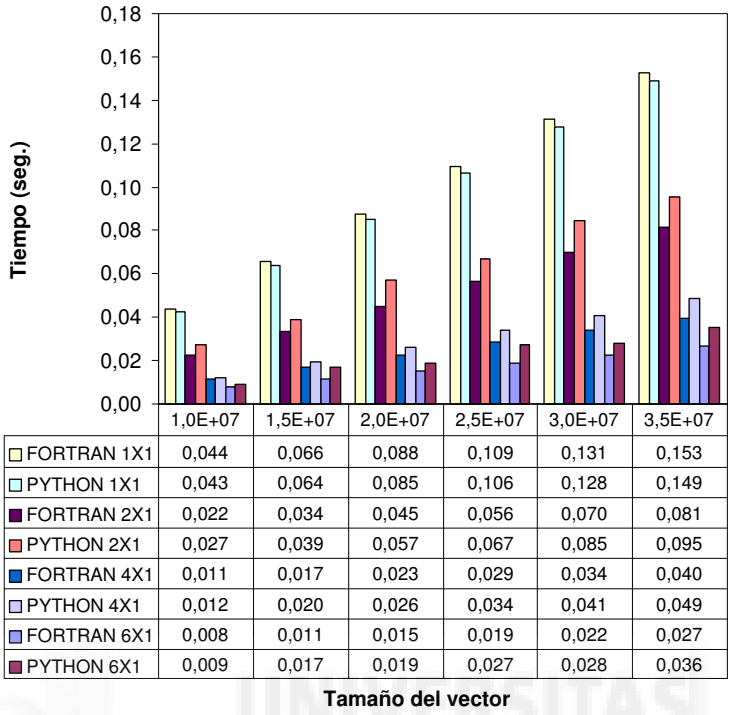


	2,0E+03	3,0E+03	4,0E+03	5,0E+03	6,0E+03	7,0E+03
FORTRAN 1X1	0,103	0,154	0,205	0,257	0,308	0,359
PYTHON 1X1	0,103	0,154	0,206	0,257	0,308	0,359
FORTRAN 2X1	0,052	0,078	0,103	0,129	0,155	0,181
PYTHON 2X1	0,052	0,078	0,104	0,130	0,155	0,181
FORTRAN 4X1	0,026	0,039	0,052	0,065	0,077	0,091
PYTHON 4X1	0,027	0,041	0,053	0,066	0,082	0,094
FORTRAN 6X1	0,018	0,031	0,035	0,047	0,052	0,060
PYTHON 6X1	0,018	0,026	0,035	0,044	0,053	0,061

Tamaño de la matriz

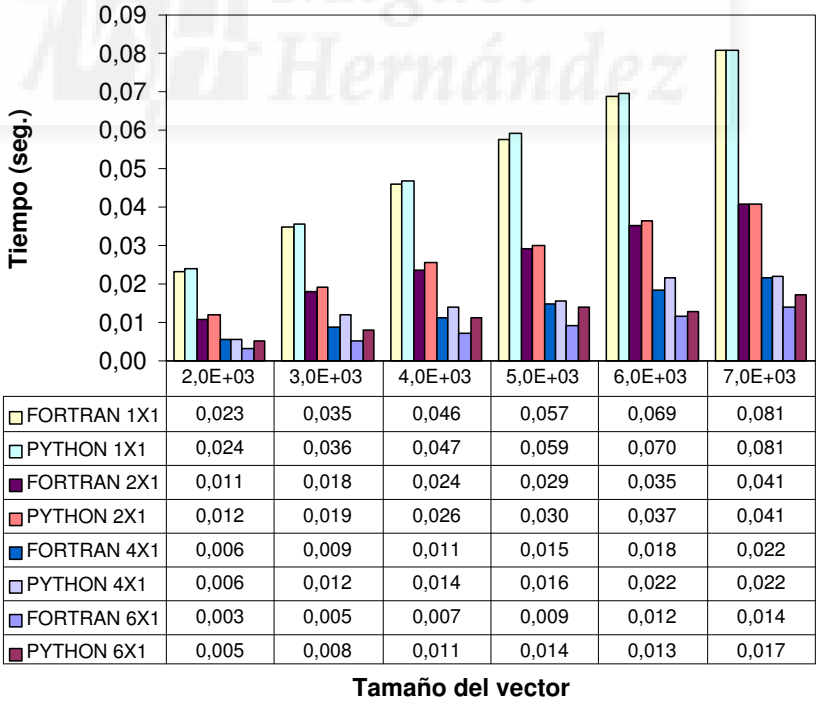
(b) Seaborg

Figura 7.8: Comparativa de la rutina `pvnrm2` en lenguaje Fortran vs Python



	1,0E+07	1,5E+07	2,0E+07	2,5E+07	3,0E+07	3,5E+07
FORTRAN 1X1	0,044	0,066	0,088	0,109	0,131	0,153
PYTHON 1X1	0,043	0,064	0,085	0,106	0,128	0,149
FORTRAN 2X1	0,022	0,034	0,045	0,056	0,070	0,081
PYTHON 2X1	0,027	0,039	0,057	0,067	0,085	0,095
FORTRAN 4X1	0,011	0,017	0,023	0,029	0,034	0,040
PYTHON 4X1	0,012	0,020	0,026	0,034	0,041	0,049
FORTRAN 6X1	0,008	0,011	0,015	0,019	0,022	0,027
PYTHON 6X1	0,009	0,017	0,019	0,027	0,028	0,036

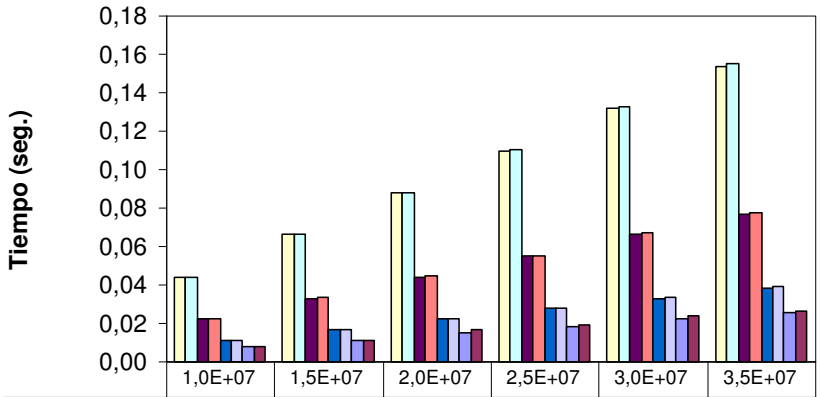
(a) Cluster-umh



	2,0E+03	3,0E+03	4,0E+03	5,0E+03	6,0E+03	7,0E+03
FORTRAN 1X1	0,023	0,035	0,046	0,057	0,069	0,081
PYTHON 1X1	0,024	0,036	0,047	0,059	0,070	0,081
FORTRAN 2X1	0,011	0,018	0,024	0,029	0,035	0,041
PYTHON 2X1	0,012	0,019	0,026	0,030	0,037	0,041
FORTRAN 4X1	0,006	0,009	0,011	0,015	0,018	0,022
PYTHON 4X1	0,006	0,012	0,014	0,016	0,022	0,022
FORTRAN 6X1	0,003	0,005	0,007	0,009	0,012	0,014
PYTHON 6X1	0,005	0,008	0,011	0,014	0,013	0,017

(b) Seaborg

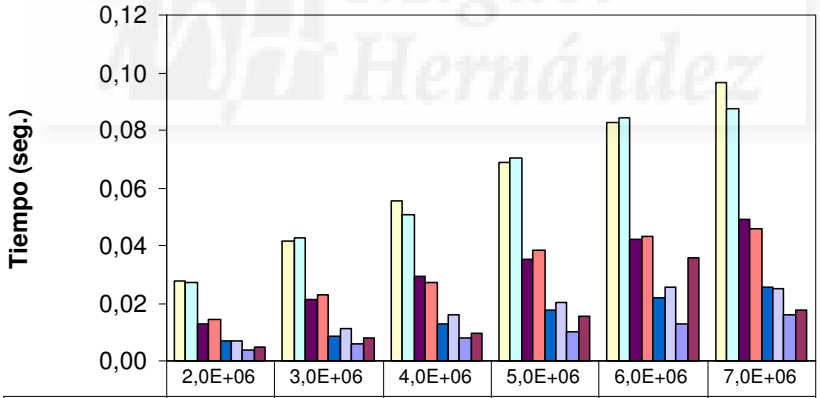
Figura 7.9: Comparativa de la rutina `pvasum` en lenguaje Fortran vs Python



	1,0E+07	1,5E+07	2,0E+07	2,5E+07	3,0E+07	3,5E+07
FORTRAN 1X1	0,044	0,066	0,088	0,110	0,132	0,154
PYTHON 1X1	0,044	0,066	0,088	0,110	0,133	0,155
FORTRAN 2X1	0,022	0,033	0,044	0,055	0,066	0,077
PYTHON 2X1	0,022	0,034	0,045	0,056	0,067	0,078
FORTRAN 4X1	0,011	0,017	0,022	0,028	0,033	0,039
PYTHON 4X1	0,012	0,017	0,023	0,028	0,033	0,039
FORTRAN 6X1	0,008	0,011	0,015	0,019	0,022	0,026
PYTHON 6X1	0,008	0,012	0,017	0,019	0,024	0,026

Tamaño de la matriz

(a) Cluster-umh



	2,0E+06	3,0E+06	4,0E+06	5,0E+06	6,0E+06	7,0E+06
FORTRAN 1X1	0,028	0,042	0,055	0,069	0,082	0,096
PYTHON 1X1	0,027	0,043	0,050	0,070	0,084	0,087
FORTRAN 2X1	0,013	0,021	0,029	0,035	0,042	0,049
PYTHON 2X1	0,014	0,023	0,027	0,038	0,043	0,046
FORTRAN 4X1	0,007	0,008	0,013	0,018	0,022	0,025
PYTHON 4X1	0,007	0,011	0,016	0,020	0,026	0,025
FORTRAN 6X1	0,004	0,006	0,008	0,010	0,013	0,016
PYTHON 6X1	0,005	0,008	0,010	0,015	0,036	0,017

Tamaño del vector

(b) Seaborg

Figura 7.10: Comparativa de la rutina `pvamax` en lenguaje Fortran vs Python

7.3. Rutinas PyPBLAS de nivel 2

Las rutinas de nivel 2 de PyPBLAS engloban funciones computacionalmente más complejas que en el nivel 1. En este caso, las rutinas de nivel 2 implementan funciones entre escalares, vectores y matrices. Sin embargo, no se implementan operaciones entre matrices (como la multiplicación o la suma) que se reservan al nivel 3 de PyPBLAS. En el apartado 7.1 se expusieron las rutinas englobadas dentro del nivel 2 de PyPBLAS. Creemos conveniente recordar que si se desea más información acerca de cualquiera de estas rutinas, podremos acudir al manual de usuario [34].

Se expone a continuación, un ejemplo de utilización de una de ellas como muestra de la sencillez en el manejo de dichas rutinas. En este caso, se realiza una llamada a la rutina `PyPBLAS.pvgemv` que implementa una multiplicación de un escalar por una matriz y por un vector y una adición del resultado a otro vector multiplicado por otro escalar ($\alpha\ op(A)x + \beta y \rightarrow y$).

```
1  from PyACTS import *
2  import PyACTS.PyPBLAS as PyPBLAS
3  #Inicializamos la malla con los parámetros por defecto
4  gridinit()
5  #Indicamos el tipo de Matriz PyACTS
6  ACTS_lib=1 # 1=Scalapack
7  #Generamos los datos de forma aleatoria
8  a=Rand2PyACTS(n,n,ACTS_lib)
9  x=Rand2PyACTS(n,1,ACTS_lib)
10 y=Rand2PyACTS(n,1,ACTS_lib)
11 alpha=Scal2PyACTS(2,ACTS_lib)
12 beta=Scal2PyACTS(3,ACTS_lib)
13 #Llamamos a la función
14 y= PyPBLAS.pvgemv(alpha,a,x,beta,y)
15 #Liberamos la malla de procesos
16 gridexit()
```

En el ejemplo de la rutina de PyPBLAS de nivel 1, ya se comentaron las rutinas de inicialización (línea 4) y liberación (línea 16) incluidas en la distribución PyACTS. Además, en este caso se observa en la línea 8 la sencillez para generar una matriz PyACTS distribuida cuyos elementos son datos aleatorios generados mediante la rutina auxiliar

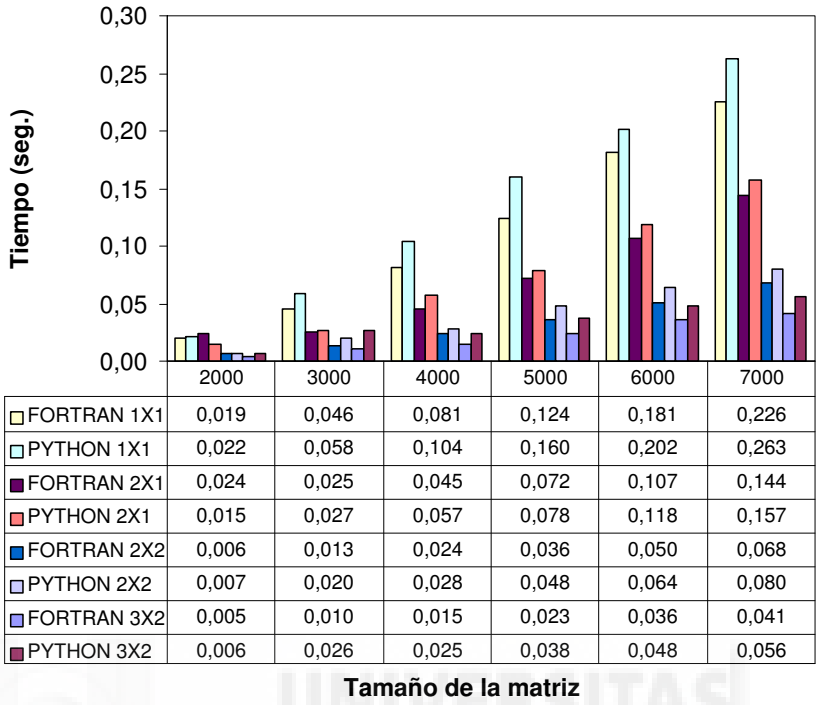
`PyACTS.Rand2PyACTS`.

Del mismo modo que se ha explicado con PyPBLAS de nivel 1, el resultado se almacena en la matriz PyACTS `y` de tamaño distribuido `n` $\times$ `1`, es decir, podemos considerar a `y` como un vector almacenado en una variable de tipo matriz PyACTS. Se ha de tener en cuenta que el contenido de estos datos se modifica de forma interna en la rutina, por tanto, el contenido de `y` se modifica durante la ejecución de la rutina en niveles inferiores y no en la asignación de la línea 14.

La forma de utilizar y de llamar a las rutinas incluidas en el nivel 2 de PyPBLAS es muy similar para todas, y por ello se ha deseado mostrar el ejemplo anterior. Sin embargo, la complejidad y los requisitos computacionales varían entre ellas en función de la operación que implementen.

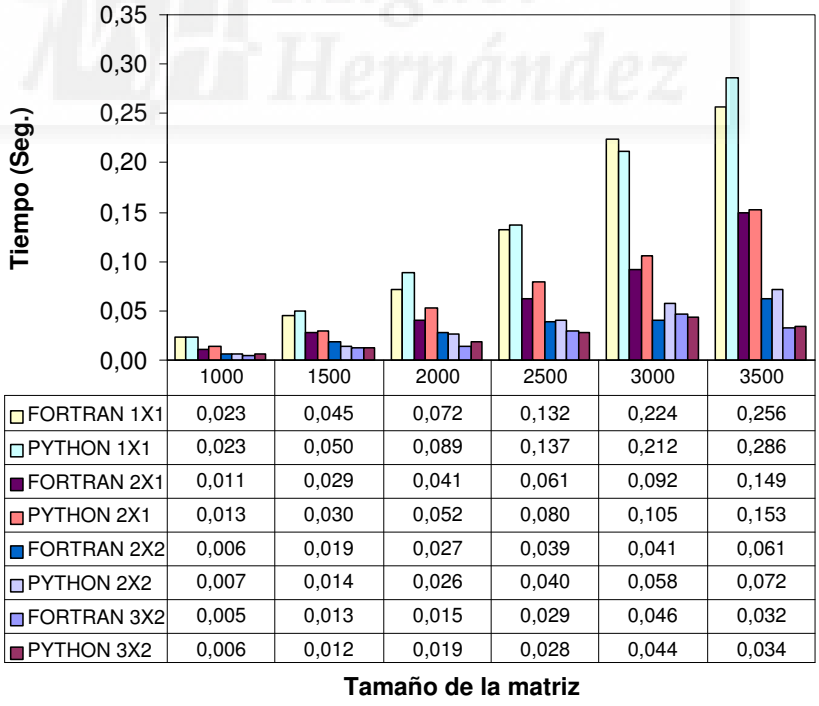
Para comprobar el correcto funcionamiento de las rutinas incluidas en el nivel 2 de PyPBLAS se han desarrollado una serie de pruebas similares a las de la sección anterior, pero en este caso realizando las llamadas a las rutinas de nivel 2. Por ejemplo, para la rutina `pvgemv` ($\alpha Ax + \beta y \rightarrow y$) se han realizado las pruebas representadas en la figura 7.11, en la que se aprecian tiempos de ejecución similares para un mismo tamaño de los vectores x e y y de la matriz A. Por otro lado, debido a que la carga computacional de los algoritmos implementados no es demasiado elevada, los tiempos de ejecución total suelen ser relativamente pequeños y esto hace que pequeñas variaciones o incrementos en el tiempo invertido en el interfaz, puedan apreciarse en algunas de las figuras comparativas mostradas a continuación.

Del mismo modo, se muestran las comparativas en los tiempos de ejecución obtenidos para todas las rutinas de nivel 2 de PyPBLAS: `pvhemv` (figura 7.12), `pvsymv` (figura 7.13), `pvtrmv` (figura 7.14), `pvtrsv` (figura 7.14), `pvger` (figura 7.16), `pvgeru` (figura 7.17), `pvgerc` (figura 7.18), `pvher` (figura 7.19), `pvher2` (figura 7.20), `pvsyr` (figura 7.21) y `pvsyr2` (figura 7.22).



	2000	3000	4000	5000	6000	7000
FORTRAN 1X1	0,019	0,046	0,081	0,124	0,181	0,226
PYTHON 1X1	0,022	0,058	0,104	0,160	0,202	0,263
FORTRAN 2X1	0,024	0,025	0,045	0,072	0,107	0,144
PYTHON 2X1	0,015	0,027	0,057	0,078	0,118	0,157
FORTRAN 2X2	0,006	0,013	0,024	0,036	0,050	0,068
PYTHON 2X2	0,007	0,020	0,028	0,048	0,064	0,080
FORTRAN 3X2	0,005	0,010	0,015	0,023	0,036	0,041
PYTHON 3X2	0,006	0,026	0,025	0,038	0,048	0,056

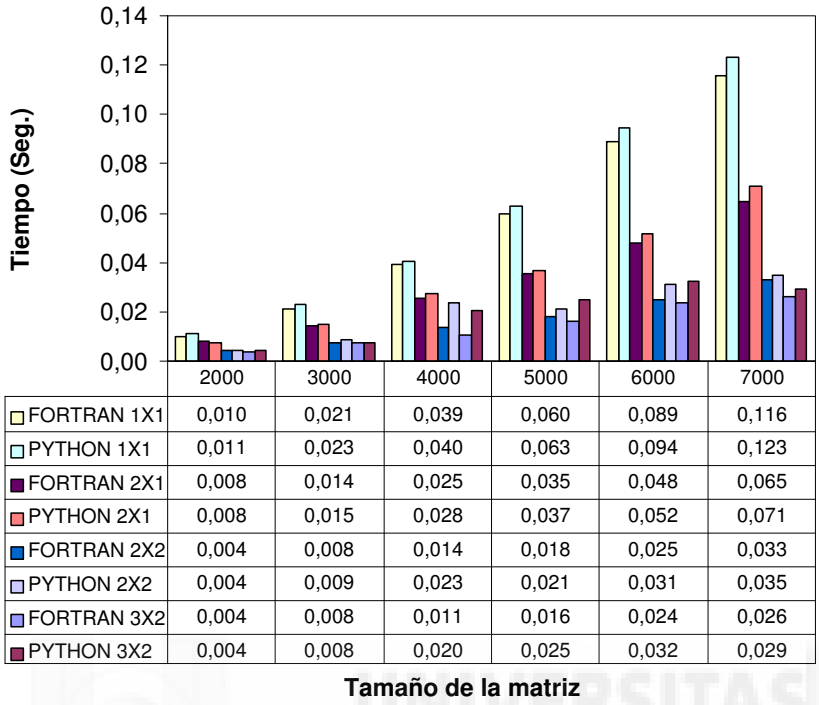
(a) Cluster-umh



	1000	1500	2000	2500	3000	3500
FORTRAN 1X1	0,023	0,045	0,072	0,132	0,224	0,256
PYTHON 1X1	0,023	0,050	0,089	0,137	0,212	0,286
FORTRAN 2X1	0,011	0,029	0,041	0,061	0,092	0,149
PYTHON 2X1	0,013	0,030	0,052	0,080	0,105	0,153
FORTRAN 2X2	0,006	0,019	0,027	0,039	0,041	0,061
PYTHON 2X2	0,007	0,014	0,026	0,040	0,058	0,072
FORTRAN 3X2	0,005	0,013	0,015	0,029	0,046	0,032
PYTHON 3X2	0,006	0,012	0,019	0,028	0,044	0,034

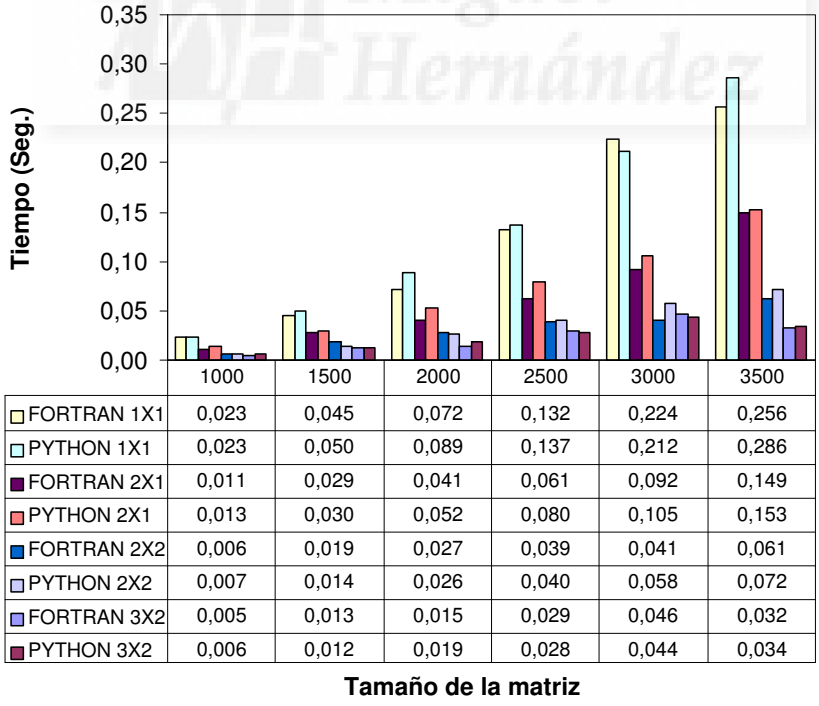
(b) Seaborg

Figura 7.11: Comparativa de la rutina `pvgemv` en lenguaje Fortran vs Python



	2000	3000	4000	5000	6000	7000
FORTRAN 1X1	0,010	0,021	0,039	0,060	0,089	0,116
PYTHON 1X1	0,011	0,023	0,040	0,063	0,094	0,123
FORTRAN 2X1	0,008	0,014	0,025	0,035	0,048	0,065
PYTHON 2X1	0,008	0,015	0,028	0,037	0,052	0,071
FORTRAN 2X2	0,004	0,008	0,014	0,018	0,025	0,033
PYTHON 2X2	0,004	0,009	0,023	0,021	0,031	0,035
FORTRAN 3X2	0,004	0,008	0,011	0,016	0,024	0,026
PYTHON 3X2	0,004	0,008	0,020	0,025	0,032	0,029

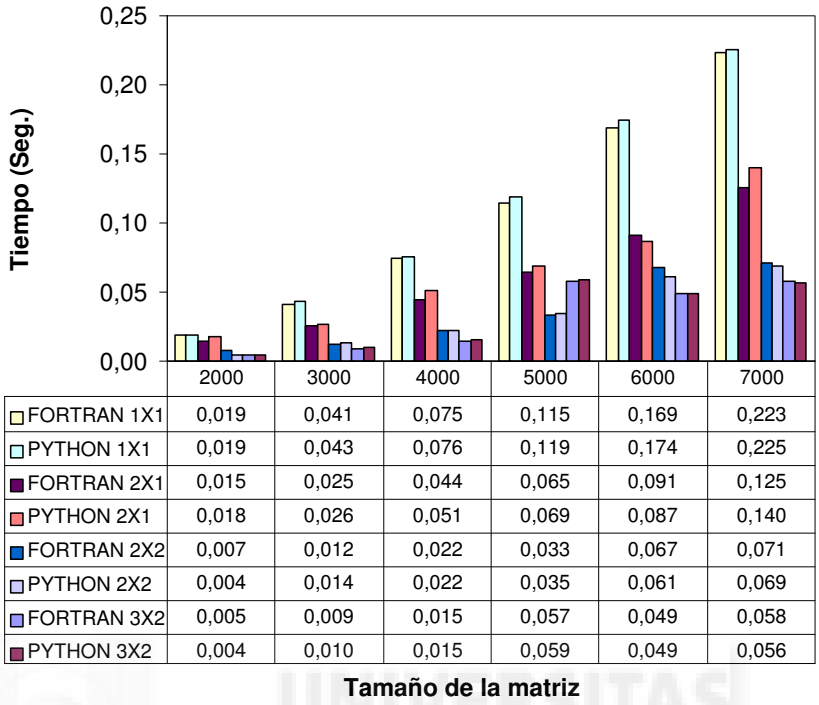
(a) Cluster-umh



	1000	1500	2000	2500	3000	3500
FORTRAN 1X1	0,023	0,045	0,072	0,132	0,224	0,256
PYTHON 1X1	0,023	0,050	0,089	0,137	0,212	0,286
FORTRAN 2X1	0,011	0,029	0,041	0,061	0,092	0,149
PYTHON 2X1	0,013	0,030	0,052	0,080	0,105	0,153
FORTRAN 2X2	0,006	0,019	0,027	0,039	0,041	0,061
PYTHON 2X2	0,007	0,014	0,026	0,040	0,058	0,072
FORTRAN 3X2	0,005	0,013	0,015	0,029	0,046	0,032
PYTHON 3X2	0,006	0,012	0,019	0,028	0,044	0,034

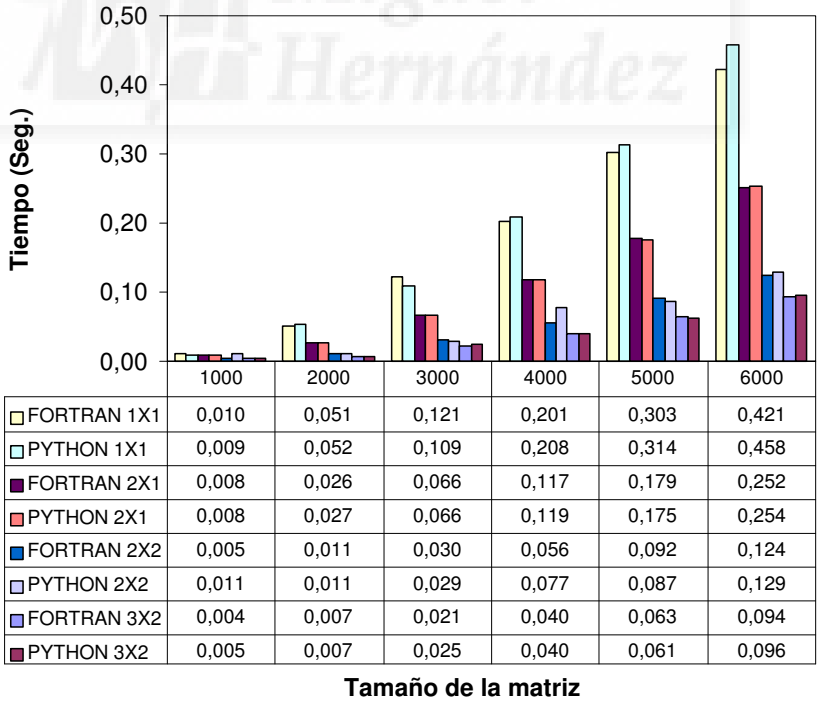
(b) Seaborg

Figura 7.12: Comparativa de la rutina `pvhemv` en lenguaje Fortran vs Python



	2000	3000	4000	5000	6000	7000
FORTRAN 1X1	0,019	0,041	0,075	0,115	0,169	0,223
PYTHON 1X1	0,019	0,043	0,076	0,119	0,174	0,225
FORTRAN 2X1	0,015	0,025	0,044	0,065	0,091	0,125
PYTHON 2X1	0,018	0,026	0,051	0,069	0,087	0,140
FORTRAN 2X2	0,007	0,012	0,022	0,033	0,067	0,071
PYTHON 2X2	0,004	0,014	0,022	0,035	0,061	0,069
FORTRAN 3X2	0,005	0,009	0,015	0,057	0,049	0,058
PYTHON 3X2	0,004	0,010	0,015	0,059	0,049	0,056

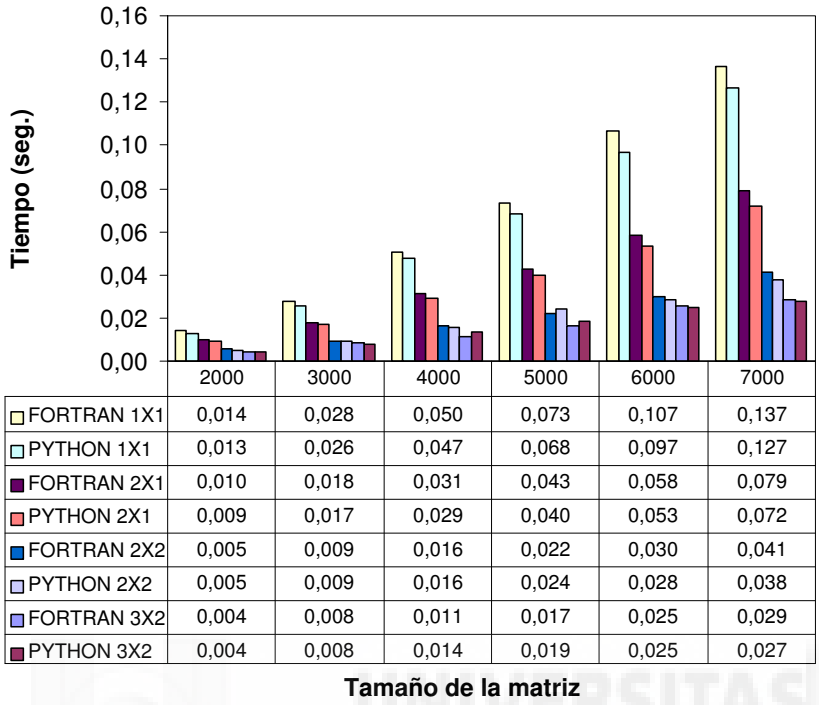
(a) Cluster-umh



	1000	2000	3000	4000	5000	6000
FORTRAN 1X1	0,010	0,051	0,121	0,201	0,303	0,421
PYTHON 1X1	0,009	0,052	0,109	0,208	0,314	0,458
FORTRAN 2X1	0,008	0,026	0,066	0,117	0,179	0,252
PYTHON 2X1	0,008	0,027	0,066	0,119	0,175	0,254
FORTRAN 2X2	0,005	0,011	0,030	0,056	0,092	0,124
PYTHON 2X2	0,011	0,011	0,029	0,077	0,087	0,129
FORTRAN 3X2	0,004	0,007	0,021	0,040	0,063	0,094
PYTHON 3X2	0,005	0,007	0,025	0,040	0,061	0,096

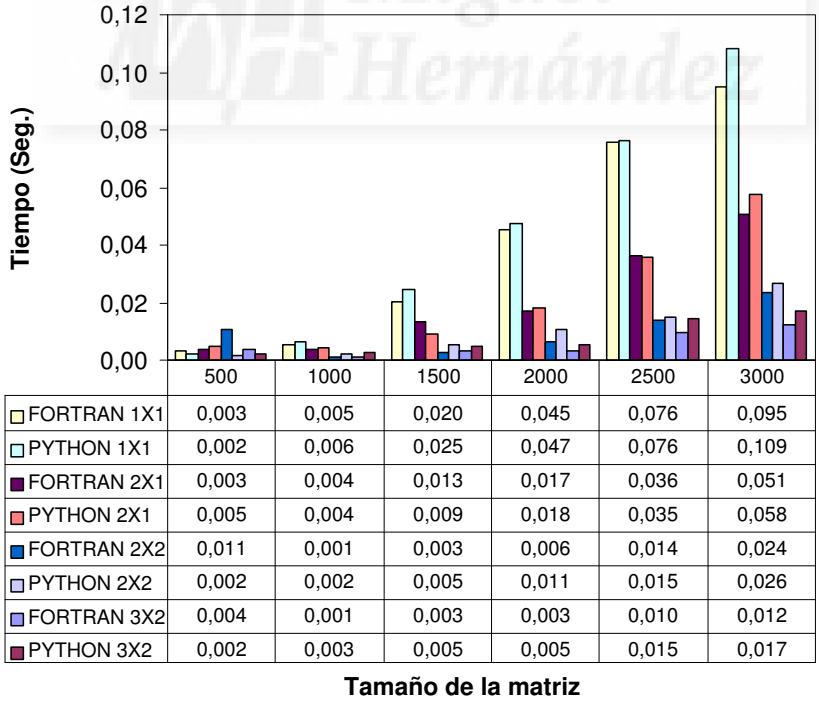
(b) Seaborg

Figura 7.13: Comparativa de la rutina `pvsymv` en lenguaje Fortran vs Python



	2000	3000	4000	5000	6000	7000
FORTRAN 1X1	0,014	0,028	0,050	0,073	0,107	0,137
PYTHON 1X1	0,013	0,026	0,047	0,068	0,097	0,127
FORTRAN 2X1	0,010	0,018	0,031	0,043	0,058	0,079
PYTHON 2X1	0,009	0,017	0,029	0,040	0,053	0,072
FORTRAN 2X2	0,005	0,009	0,016	0,022	0,030	0,041
PYTHON 2X2	0,005	0,009	0,016	0,024	0,028	0,038
FORTRAN 3X2	0,004	0,008	0,011	0,017	0,025	0,029
PYTHON 3X2	0,004	0,008	0,014	0,019	0,025	0,027

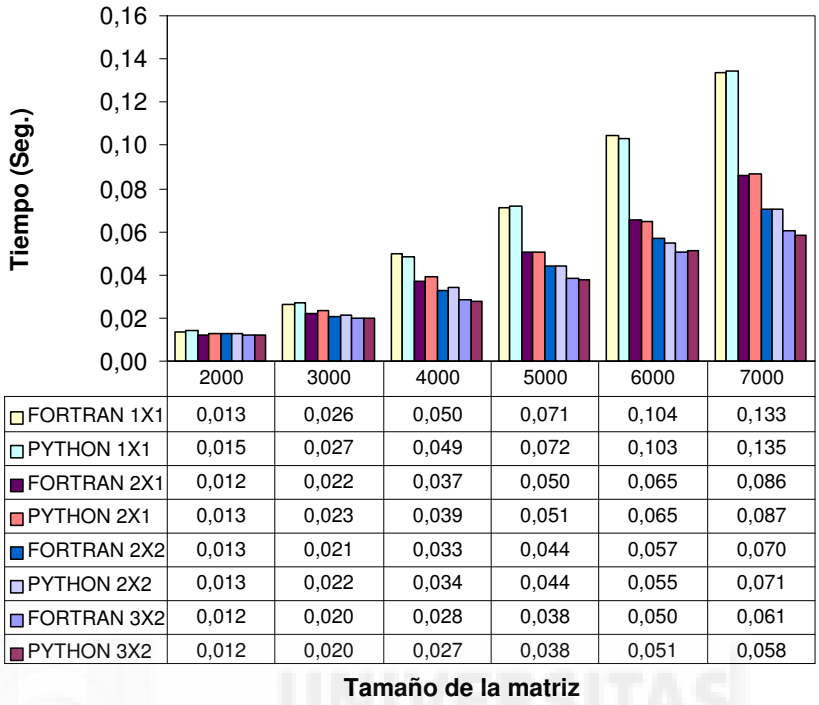
(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,003	0,005	0,020	0,045	0,076	0,095
PYTHON 1X1	0,002	0,006	0,025	0,047	0,076	0,109
FORTRAN 2X1	0,003	0,004	0,013	0,017	0,036	0,051
PYTHON 2X1	0,005	0,004	0,009	0,018	0,035	0,058
FORTRAN 2X2	0,011	0,001	0,003	0,006	0,014	0,024
PYTHON 2X2	0,002	0,002	0,005	0,011	0,015	0,026
FORTRAN 3X2	0,004	0,001	0,003	0,003	0,010	0,012
PYTHON 3X2	0,002	0,003	0,005	0,005	0,015	0,017

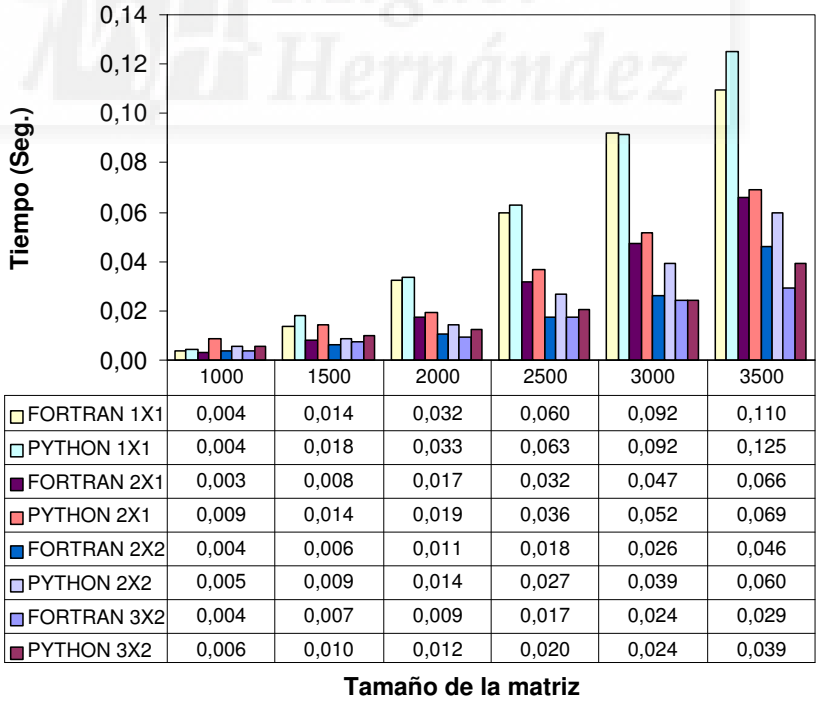
(b) Seaborg

Figura 7.14: Comparativa de la rutina `pvtrmv` en lenguaje Fortran vs Python



	2000	3000	4000	5000	6000	7000
FORTRAN 1X1	0,013	0,026	0,050	0,071	0,104	0,133
PYTHON 1X1	0,015	0,027	0,049	0,072	0,103	0,135
FORTRAN 2X1	0,012	0,022	0,037	0,050	0,065	0,086
PYTHON 2X1	0,013	0,023	0,039	0,051	0,065	0,087
FORTRAN 2X2	0,013	0,021	0,033	0,044	0,057	0,070
PYTHON 2X2	0,013	0,022	0,034	0,044	0,055	0,071
FORTRAN 3X2	0,012	0,020	0,028	0,038	0,050	0,061
PYTHON 3X2	0,012	0,020	0,027	0,038	0,051	0,058

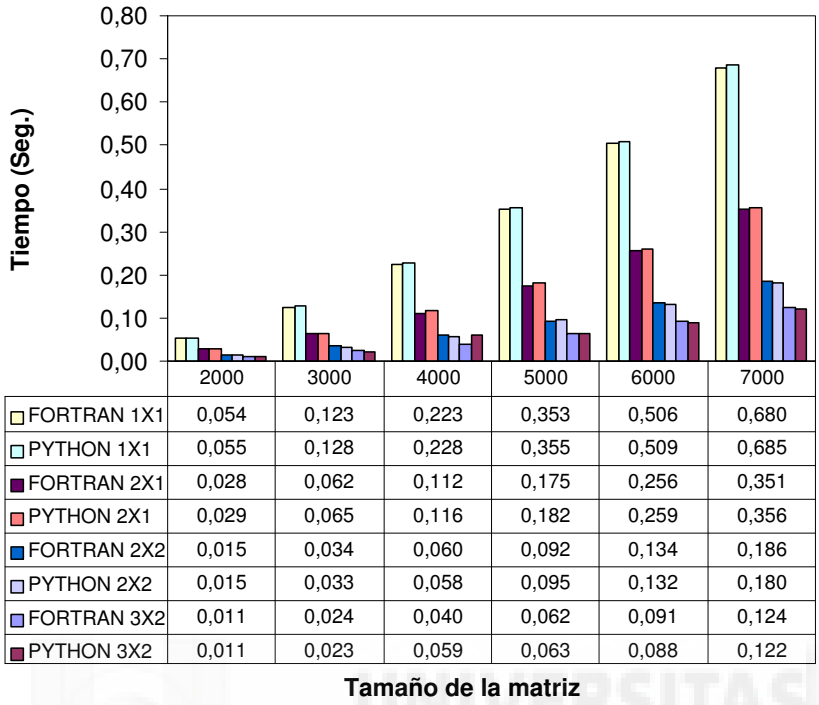
(a) Cluster-umh



	1000	1500	2000	2500	3000	3500
FORTRAN 1X1	0,004	0,014	0,032	0,060	0,092	0,110
PYTHON 1X1	0,004	0,018	0,033	0,063	0,092	0,125
FORTRAN 2X1	0,003	0,008	0,017	0,032	0,047	0,066
PYTHON 2X1	0,009	0,014	0,019	0,036	0,052	0,069
FORTRAN 2X2	0,004	0,006	0,011	0,018	0,026	0,046
PYTHON 2X2	0,005	0,009	0,014	0,027	0,039	0,060
FORTRAN 3X2	0,004	0,007	0,009	0,017	0,024	0,029
PYTHON 3X2	0,006	0,010	0,012	0,020	0,024	0,039

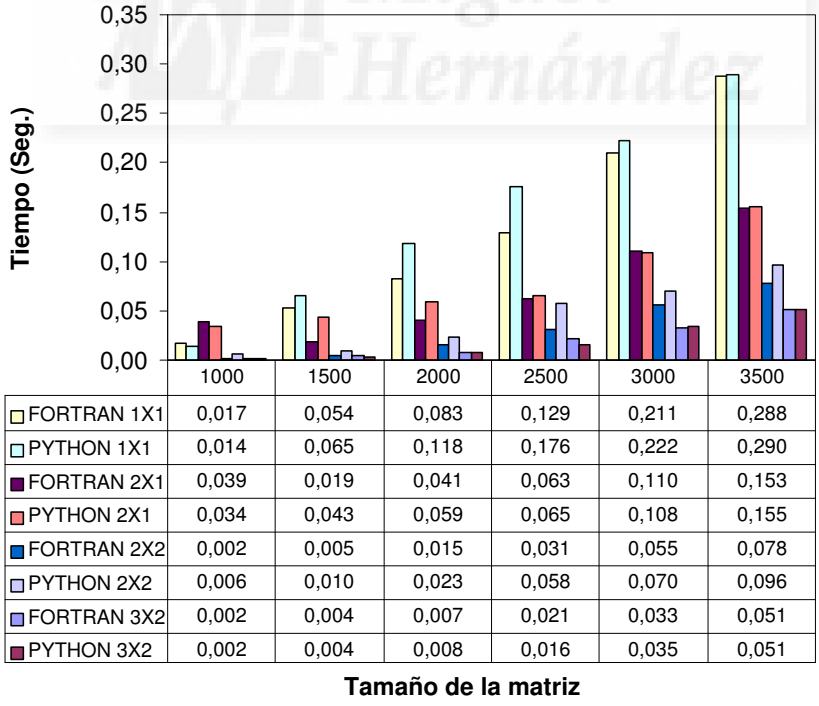
(b) Seaborg

Figura 7.15: Comparativa de la rutina `pvtrsv` en lenguaje Fortran vs Python



	2000	3000	4000	5000	6000	7000
FORTRAN 1X1	0,054	0,123	0,223	0,353	0,506	0,680
PYTHON 1X1	0,055	0,128	0,228	0,355	0,509	0,685
FORTRAN 2X1	0,028	0,062	0,112	0,175	0,256	0,351
PYTHON 2X1	0,029	0,065	0,116	0,182	0,259	0,356
FORTRAN 2X2	0,015	0,034	0,060	0,092	0,134	0,186
PYTHON 2X2	0,015	0,033	0,058	0,095	0,132	0,180
FORTRAN 3X2	0,011	0,024	0,040	0,062	0,091	0,124
PYTHON 3X2	0,011	0,023	0,059	0,063	0,088	0,122

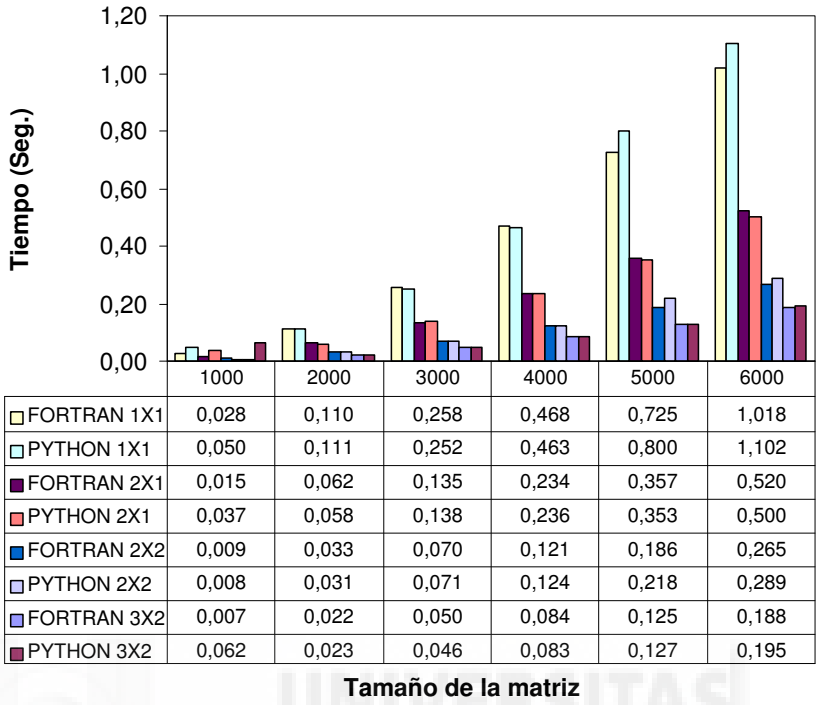
(a) Cluster-umh



	1000	1500	2000	2500	3000	3500
FORTRAN 1X1	0,017	0,054	0,083	0,129	0,211	0,288
PYTHON 1X1	0,014	0,065	0,118	0,176	0,222	0,290
FORTRAN 2X1	0,039	0,019	0,041	0,063	0,110	0,153
PYTHON 2X1	0,034	0,043	0,059	0,065	0,108	0,155
FORTRAN 2X2	0,002	0,005	0,015	0,031	0,055	0,078
PYTHON 2X2	0,006	0,010	0,023	0,058	0,070	0,096
FORTRAN 3X2	0,002	0,004	0,007	0,021	0,033	0,051
PYTHON 3X2	0,002	0,004	0,008	0,016	0,035	0,051

(b) Seaborg

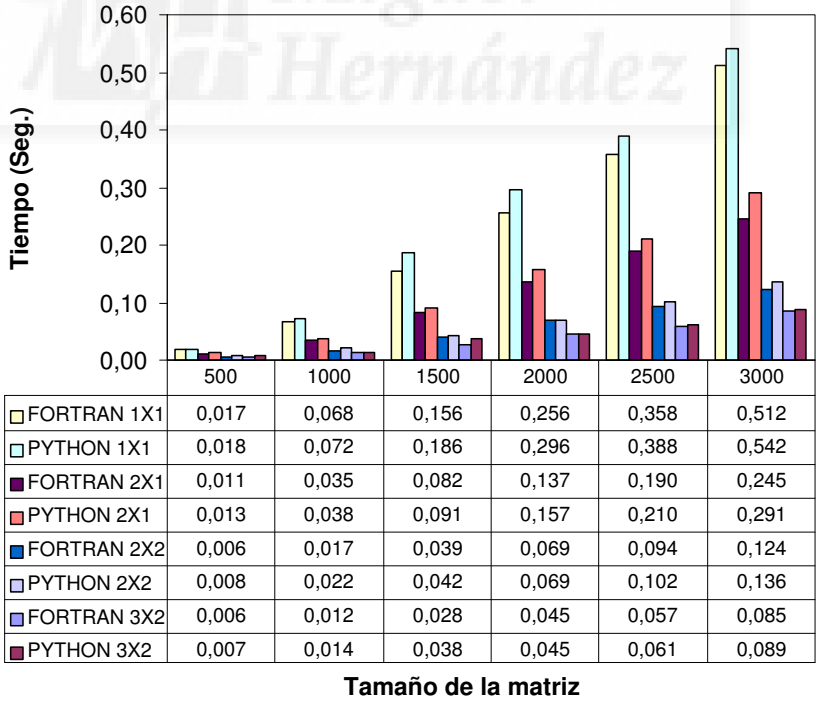
Figura 7.16: Comparativa de la rutina `pvger` en lenguaje Fortran vs Python



	1000	2000	3000	4000	5000	6000
FORTRAN 1X1	0,028	0,110	0,258	0,468	0,725	1,018
PYTHON 1X1	0,050	0,111	0,252	0,463	0,800	1,102
FORTRAN 2X1	0,015	0,062	0,135	0,234	0,357	0,520
PYTHON 2X1	0,037	0,058	0,138	0,236	0,353	0,500
FORTRAN 2X2	0,009	0,033	0,070	0,121	0,186	0,265
PYTHON 2X2	0,008	0,031	0,071	0,124	0,218	0,289
FORTRAN 3X2	0,007	0,022	0,050	0,084	0,125	0,188
PYTHON 3X2	0,062	0,023	0,046	0,083	0,127	0,195

Tamaño de la matriz

(a) Cluster-umh

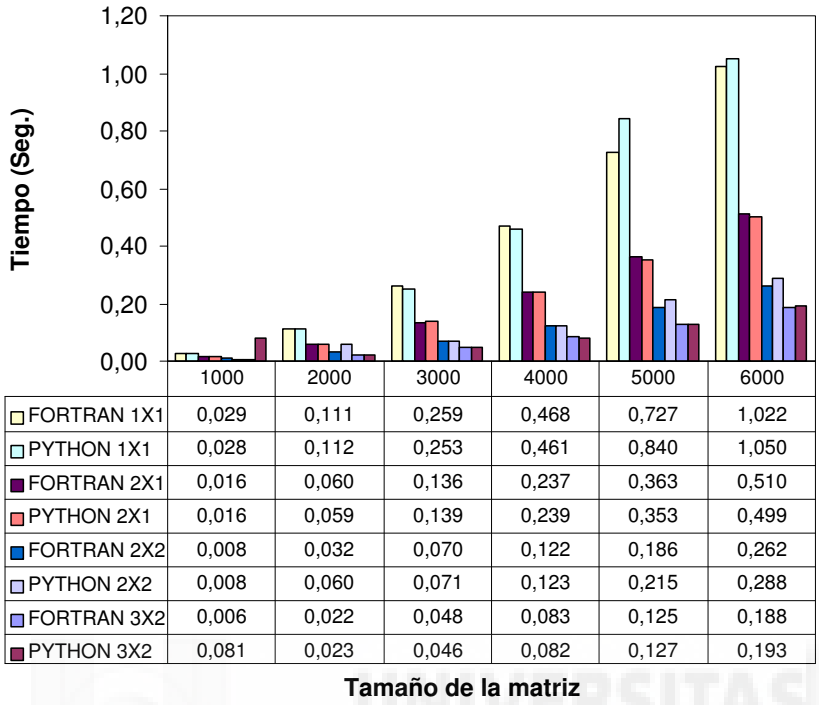


	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,017	0,068	0,156	0,256	0,358	0,512
PYTHON 1X1	0,018	0,072	0,186	0,296	0,388	0,542
FORTRAN 2X1	0,011	0,035	0,082	0,137	0,190	0,245
PYTHON 2X1	0,013	0,038	0,091	0,157	0,210	0,291
FORTRAN 2X2	0,006	0,017	0,039	0,069	0,094	0,124
PYTHON 2X2	0,008	0,022	0,042	0,069	0,102	0,136
FORTRAN 3X2	0,006	0,012	0,028	0,045	0,057	0,085
PYTHON 3X2	0,007	0,014	0,038	0,045	0,061	0,089

Tamaño de la matriz

(b) Seaborg

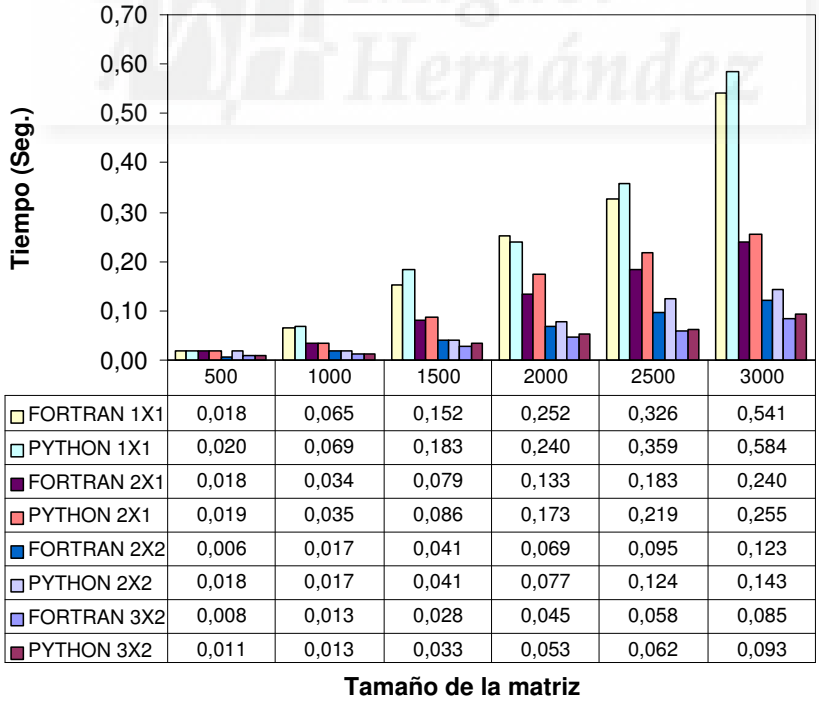
Figura 7.17: Comparativa de la rutina `pvgeru` en lenguaje Fortran vs Python



	1000	2000	3000	4000	5000	6000
FORTRAN 1X1	0,029	0,111	0,259	0,468	0,727	1,022
PYTHON 1X1	0,028	0,112	0,253	0,461	0,840	1,050
FORTRAN 2X1	0,016	0,060	0,136	0,237	0,363	0,510
PYTHON 2X1	0,016	0,059	0,139	0,239	0,353	0,499
FORTRAN 2X2	0,008	0,032	0,070	0,122	0,186	0,262
PYTHON 2X2	0,008	0,060	0,071	0,123	0,215	0,288
FORTRAN 3X2	0,006	0,022	0,048	0,083	0,125	0,188
PYTHON 3X2	0,081	0,023	0,046	0,082	0,127	0,193

Tamaño de la matriz

(a) Cluster-umh

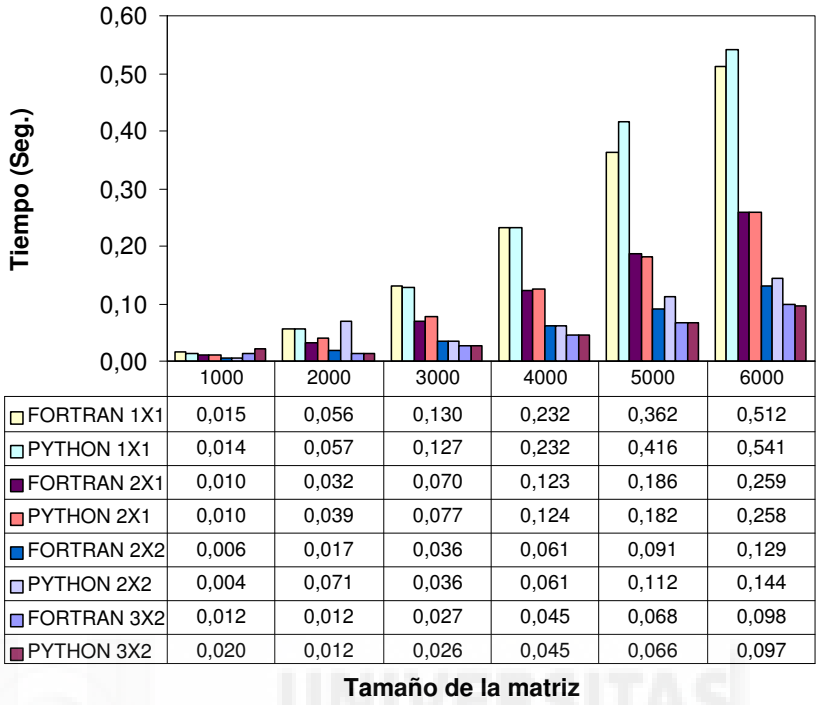


	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,018	0,065	0,152	0,252	0,326	0,541
PYTHON 1X1	0,020	0,069	0,183	0,240	0,359	0,584
FORTRAN 2X1	0,018	0,034	0,079	0,133	0,183	0,240
PYTHON 2X1	0,019	0,035	0,086	0,173	0,219	0,255
FORTRAN 2X2	0,006	0,017	0,041	0,069	0,095	0,123
PYTHON 2X2	0,018	0,017	0,041	0,077	0,124	0,143
FORTRAN 3X2	0,008	0,013	0,028	0,045	0,058	0,085
PYTHON 3X2	0,011	0,013	0,033	0,053	0,062	0,093

Tamaño de la matriz

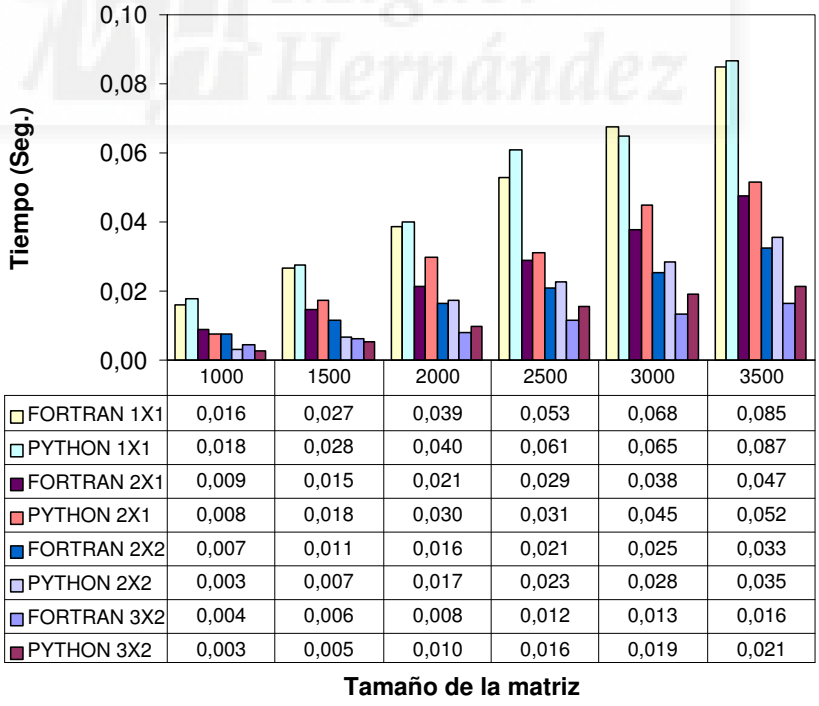
(b) Seaborg

Figura 7.18: Comparativa de la rutina `pvgerc` en lenguaje Fortran vs Python



	1000	2000	3000	4000	5000	6000
FORTRAN 1X1	0,015	0,056	0,130	0,232	0,362	0,512
PYTHON 1X1	0,014	0,057	0,127	0,232	0,416	0,541
FORTRAN 2X1	0,010	0,032	0,070	0,123	0,186	0,259
PYTHON 2X1	0,010	0,039	0,077	0,124	0,182	0,258
FORTRAN 2X2	0,006	0,017	0,036	0,061	0,091	0,129
PYTHON 2X2	0,004	0,071	0,036	0,061	0,112	0,144
FORTRAN 3X2	0,012	0,012	0,027	0,045	0,068	0,098
PYTHON 3X2	0,020	0,012	0,026	0,045	0,066	0,097

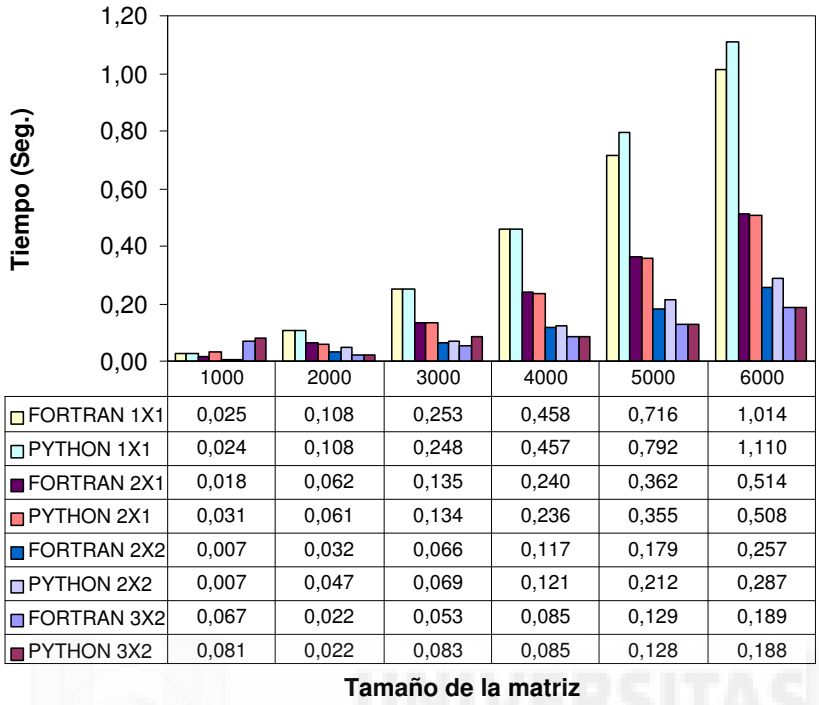
(a) Cluster-umh



	1000	1500	2000	2500	3000	3500
FORTRAN 1X1	0,016	0,027	0,039	0,053	0,068	0,085
PYTHON 1X1	0,018	0,028	0,040	0,061	0,065	0,087
FORTRAN 2X1	0,009	0,015	0,021	0,029	0,038	0,047
PYTHON 2X1	0,008	0,018	0,030	0,031	0,045	0,052
FORTRAN 2X2	0,007	0,011	0,016	0,021	0,025	0,033
PYTHON 2X2	0,003	0,007	0,017	0,023	0,028	0,035
FORTRAN 3X2	0,004	0,006	0,008	0,012	0,013	0,016
PYTHON 3X2	0,003	0,005	0,010	0,016	0,019	0,021

(b) Seaborg

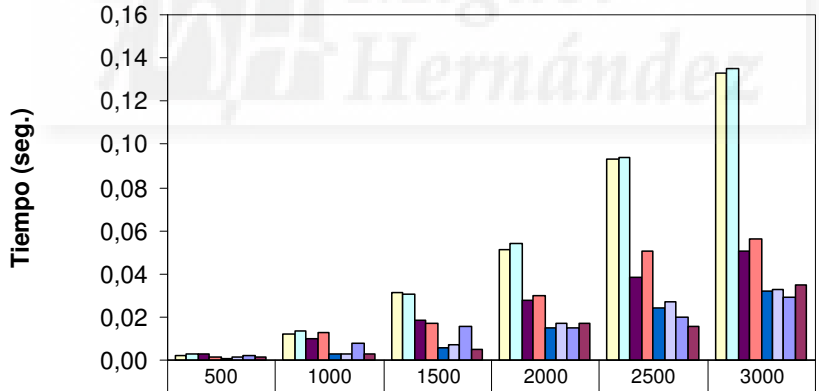
Figura 7.19: Comparativa de la rutina `pvher` en lenguaje Fortran vs Python



	1000	2000	3000	4000	5000	6000
FORTRAN 1X1	0,025	0,108	0,253	0,458	0,716	1,014
PYTHON 1X1	0,024	0,108	0,248	0,457	0,792	1,110
FORTRAN 2X1	0,018	0,062	0,135	0,240	0,362	0,514
PYTHON 2X1	0,031	0,061	0,134	0,236	0,355	0,508
FORTRAN 2X2	0,007	0,032	0,066	0,117	0,179	0,257
PYTHON 2X2	0,007	0,047	0,069	0,121	0,212	0,287
FORTRAN 3X2	0,067	0,022	0,053	0,085	0,129	0,189
PYTHON 3X2	0,081	0,022	0,083	0,085	0,128	0,188

Tamaño de la matriz

(a) Cluster-umh

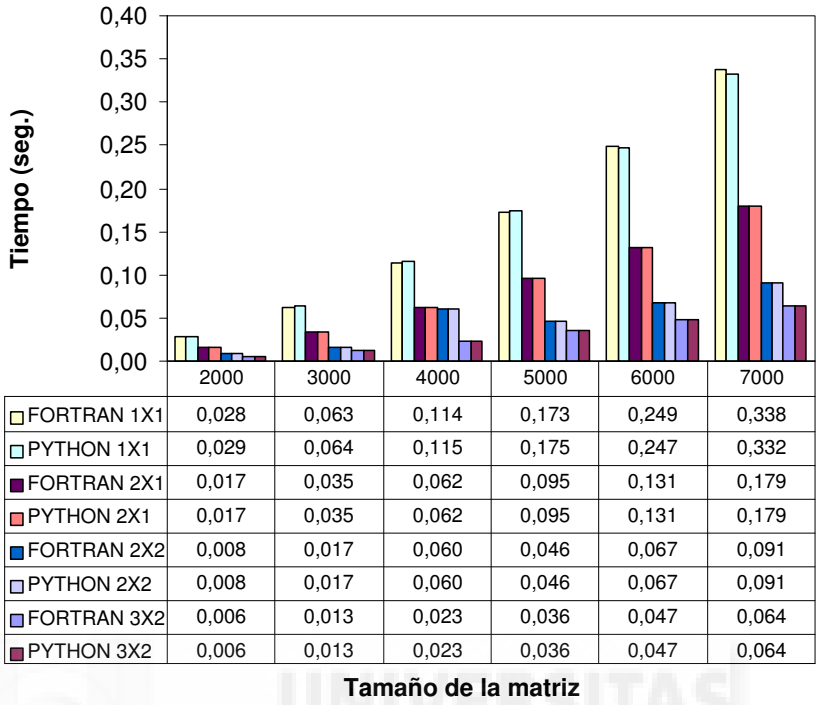


	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,002	0,012	0,031	0,051	0,093	0,133
PYTHON 1X1	0,003	0,014	0,031	0,054	0,094	0,135
FORTRAN 2X1	0,003	0,010	0,018	0,028	0,039	0,051
PYTHON 2X1	0,001	0,013	0,017	0,030	0,051	0,056
FORTRAN 2X2	0,001	0,003	0,006	0,015	0,024	0,032
PYTHON 2X2	0,001	0,003	0,007	0,017	0,027	0,033
FORTRAN 3X2	0,002	0,008	0,016	0,015	0,020	0,029
PYTHON 3X2	0,002	0,003	0,005	0,017	0,016	0,035

Tamaño de la matriz

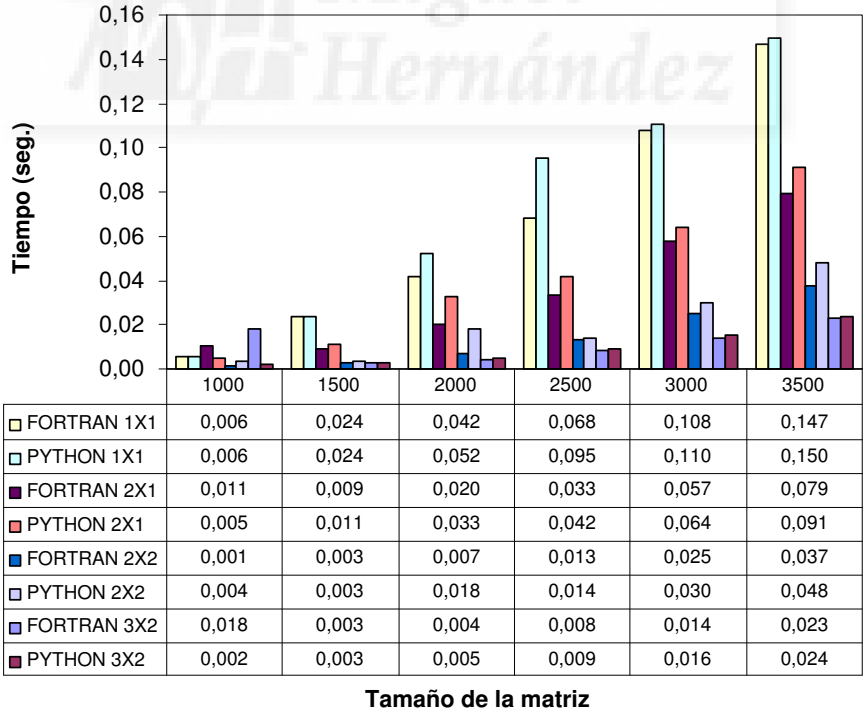
(b) Seaborg

Figura 7.20: Comparativa de la rutina `pvher2` en lenguaje Fortran vs Python



	2000	3000	4000	5000	6000	7000
FORTRAN 1X1	0,028	0,063	0,114	0,173	0,249	0,338
PYTHON 1X1	0,029	0,064	0,115	0,175	0,247	0,332
FORTRAN 2X1	0,017	0,035	0,062	0,095	0,131	0,179
PYTHON 2X1	0,017	0,035	0,062	0,095	0,131	0,179
FORTRAN 2X2	0,008	0,017	0,060	0,046	0,067	0,091
PYTHON 2X2	0,008	0,017	0,060	0,046	0,067	0,091
FORTRAN 3X2	0,006	0,013	0,023	0,036	0,047	0,064
PYTHON 3X2	0,006	0,013	0,023	0,036	0,047	0,064

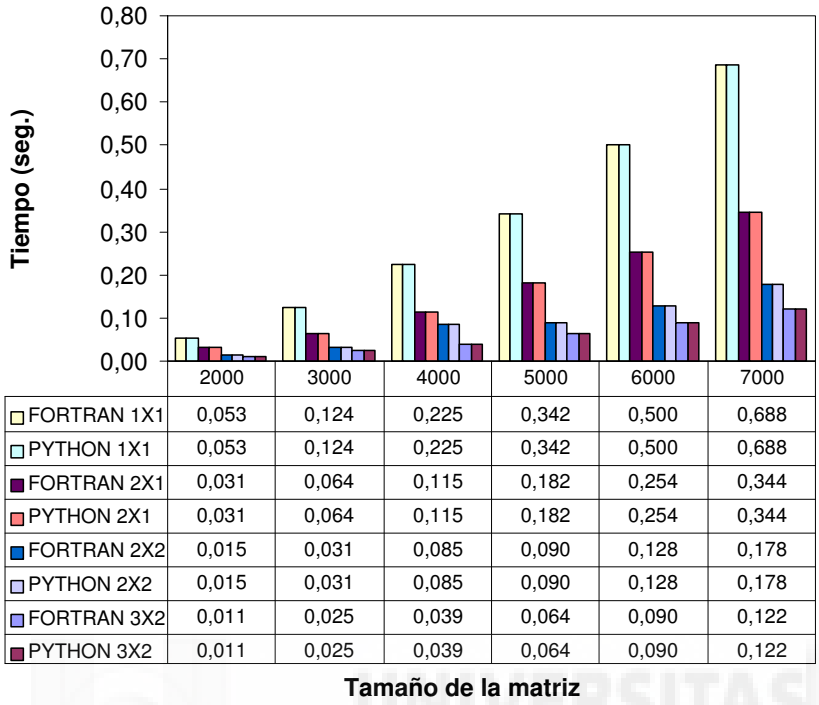
(a) Cluster-umh



	1000	1500	2000	2500	3000	3500
FORTRAN 1X1	0,006	0,024	0,042	0,068	0,108	0,147
PYTHON 1X1	0,006	0,024	0,052	0,095	0,110	0,150
FORTRAN 2X1	0,011	0,009	0,020	0,033	0,057	0,079
PYTHON 2X1	0,005	0,011	0,033	0,042	0,064	0,091
FORTRAN 2X2	0,001	0,003	0,007	0,013	0,025	0,037
PYTHON 2X2	0,004	0,003	0,018	0,014	0,030	0,048
FORTRAN 3X2	0,018	0,003	0,004	0,008	0,014	0,023
PYTHON 3X2	0,002	0,003	0,005	0,009	0,016	0,024

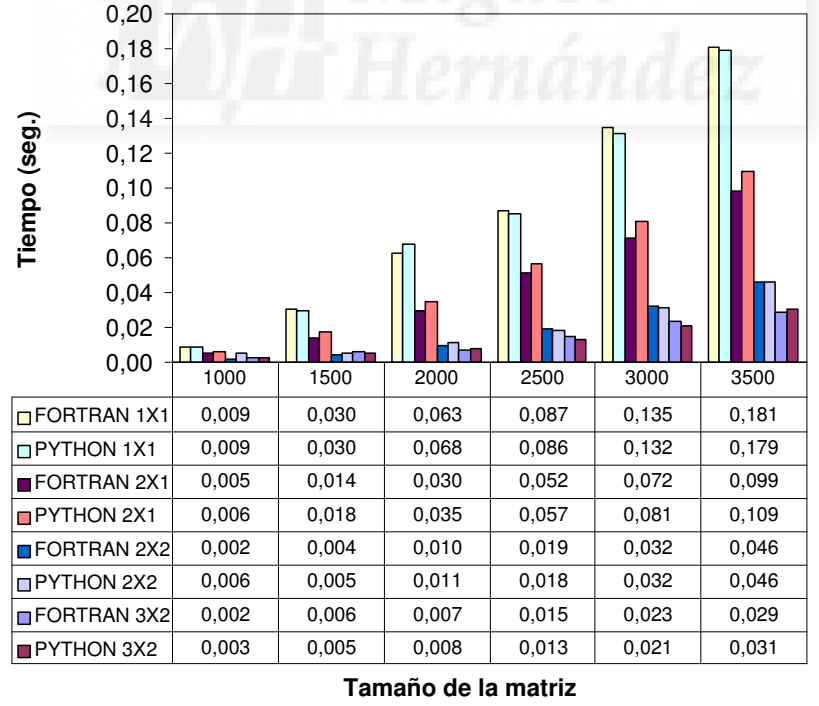
(b) Seaborg

Figura 7.21: Comparativa de la rutina `pvsyr` en lenguaje Fortran vs Python



	2000	3000	4000	5000	6000	7000
FORTRAN 1X1	0,053	0,124	0,225	0,342	0,500	0,688
PYTHON 1X1	0,053	0,124	0,225	0,342	0,500	0,688
FORTRAN 2X1	0,031	0,064	0,115	0,182	0,254	0,344
PYTHON 2X1	0,031	0,064	0,115	0,182	0,254	0,344
FORTRAN 2X2	0,015	0,031	0,085	0,090	0,128	0,178
PYTHON 2X2	0,015	0,031	0,085	0,090	0,128	0,178
FORTRAN 3X2	0,011	0,025	0,039	0,064	0,090	0,122
PYTHON 3X2	0,011	0,025	0,039	0,064	0,090	0,122

(a) Cluster-umh



	1000	1500	2000	2500	3000	3500
FORTRAN 1X1	0,009	0,030	0,063	0,087	0,135	0,181
PYTHON 1X1	0,009	0,030	0,068	0,086	0,132	0,179
FORTRAN 2X1	0,005	0,014	0,030	0,052	0,072	0,099
PYTHON 2X1	0,006	0,018	0,035	0,057	0,081	0,109
FORTRAN 2X2	0,002	0,004	0,010	0,019	0,032	0,046
PYTHON 2X2	0,006	0,005	0,011	0,018	0,032	0,046
FORTRAN 3X2	0,002	0,006	0,007	0,015	0,023	0,029
PYTHON 3X2	0,003	0,005	0,008	0,013	0,021	0,031

(b) Seaborg

Figura 7.22: Comparativa de la rutina `pvsyr2` en lenguaje Fortran vs Python

7.4. Rutinas PyPBLAS de nivel 3

Las rutinas de nivel 3 de PyPBLAS son más complejas computacionalmente por lo que sus tiempos de ejecución son considerablemente superiores. En la introducción de este capitulo, se mostraron el conjunto de rutinas agrupadas en este nivel indicando de una forma sencilla su funcionalidad. En este grupo, las rutinas incluidas realizan operaciones entre matrices y escalares, y en este caso no aparecen vectores como sucede en el nivel 2.

La legibilidad y usabilidad de este conjunto de rutinas es muy similar a las de niveles anteriores y también utilizan como variables de entrada y salida las matrices PyACTS definidas en el apartado 5.3.

A continuación, mostraremos un ejemplo de utilización de una rutina de nivel 3. En este caso hemos optado por mostrar un ejemplo de una rutina que opera con complejos. La interacción entre Python y los números complejos es muy sencilla e intuitiva y su adaptación a las rutinas pertenecientes a PBLAS no implica una penalización en el rendimiento de dichas rutinas. El ejemplo en cuestión muestra el uso de la rutina `pvhemm` cuya operación implementada es $\alpha AB + \beta C \rightarrow C$, *o* $\alpha BA + \beta C \rightarrow C$, según se determine con el parámetro opcional `side`.

```
1  from PyACTS import *
2  import PyACTS.PyPBLAS as PyPBLAS
3  from Numeric import *
4  #Dimension of Arrays
5  m,n=6,6
6  #Initiliaze the Grid
7  PyACTS.gridinit()
8  if PyACTS.iread==1:
9          print "Example of using PyPBLAS 3: PvHEMM"
10         print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
11         print "Block's size:",PyACTS.mb,"*",PyACTS.nb
12         a=ones([m,m])+1.j*reshape(range(m*m),[m,m])
13         print "a=",a
14         b=ones([m,m])-1.j*reshape(range(m*n),[m,n])
15         print "b=",b
16         c=1.j*ones([m,n])
17         print "c=",c
```

```
18          alpha,beta=2.,3.
19          print "alpha=",alpha,";","beta=",beta
20  else:
21          alpha,a,b,beta,c=None,None,None,None,None
22  #We convert Numeric Array to PyACTS.Scalapack Array
23  ACTS_lib=1 # 1=Scalapack
24  alpha=Scal2PyACTS(alpha,ACTS_lib)
25  beta=Scal2PyACTS(beta,ACTS_lib)
26  a=Num2PyACTS(a,ACTS_lib)
27  b=Num2PyACTS(b,ACTS_lib)
28  c=Num2PyACTS(c,ACTS_lib)
29  #We call PBLAS routine
30  c= PyPBLAS.phemm(alpha,a,b,beta,c)
31  c_num=PyACTS2Num(c)
32  if PyACTS.iread==1:
33          print "PvHEMM=",c_num
34  PyACTS.gridexit()
```

Se observa en este ejemplo cómo se crean los datos en un sólo proceso (aquel con `PyACTS.iread=1`) y la rutina auxiliar `Num2PyACTS` se encarga de convertir la variable `Numeric` en el proceso origen en una matriz distribuida en función de los parámetros configurados en el entorno PyACTS, como pueden ser el tamaño de la malla de procesos o el tamaño de bloque de datos. Por otro lado, se aprecia en la línea 31 cómo se realiza el proceso inverso, es decir, la conversión de una matriz PyACTS a una variable de tipo `Numeric` para que sea el nodo origen el encargado de imprimirla en la salida.

A continuación se muestra la salida a dicho código. Deseamos destacar la facilidad en la representación de los datos de tipo complejo así como la transparencia al usuario final del tipo de datos que estemos utilizando. De este modo, si un usuario que quiere realizar la misma funcionalidad que `pvhemm` pero los datos que utiliza son de tipo real con doble precisión, de forma interna la rutina `PyPBLAS.pvhemm` comprobará el tipo de dato de los parámetros y en el caso de ser todos ellos de tipo real de doble precisión realizará una llamada a la rutina `PDGEMM` que implementa la misma funcionalidad pero con ese tipo de datos. Del mismo modo, si un usuario realiza una llamada a la rutina `PyPBLAS.pvgemm` y alguna de las matrices contiene datos de tipo complejo, de forma interna y transparente se ejecuta la rutina `PZHEMM` destinada a datos de tipo complejo de doble precisión. Esta es una de las distintas automatizaciones y simplificaciones que se llevan a cabo en PyACTS

para ocultar y simplificar la programación al usuario final.

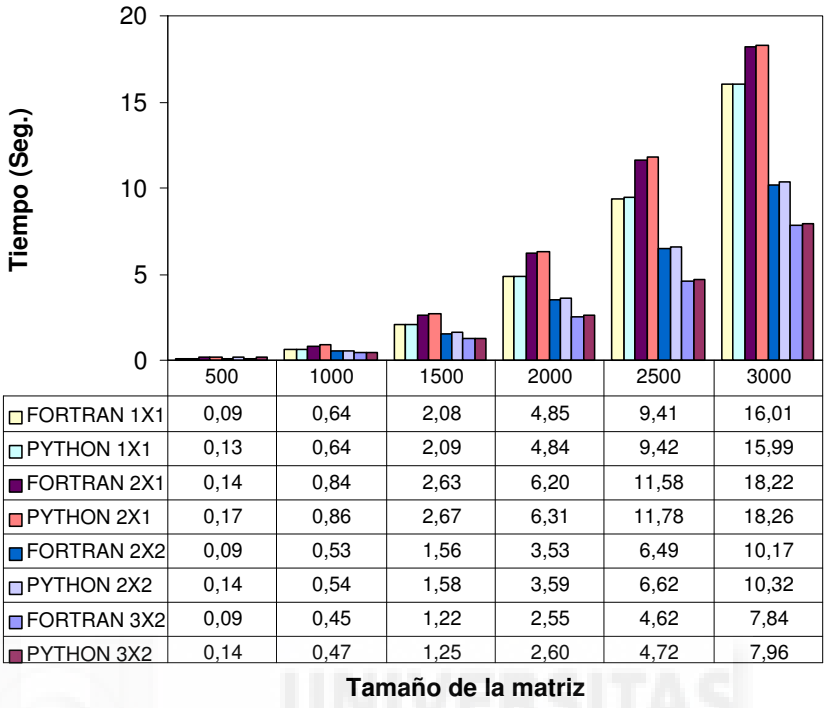
```
Example of using PyPBLAS 3: PvHEMM
N= 6 ;nprow x npcol: 3 x 2
Block's size: 64 * 64
a= [[ 1. +0.j  1. +1.j  1. +2.j  1. +3.j  1. +4.j  1. +5.j]
 [ 1. +6.j  1. +7.j  1. +8.j  1. +9.j  1.+10.j  1.+11.j]
 [ 1.+12.j  1.+13.j  1.+14.j  1.+15.j  1.+16.j  1.+17.j]
 [ 1.+18.j  1.+19.j  1.+20.j  1.+21.j  1.+22.j  1.+23.j]
 [ 1.+24.j  1.+25.j  1.+26.j  1.+27.j  1.+28.j  1.+29.j]
 [ 1.+30.j  1.+31.j  1.+32.j  1.+33.j  1.+34.j  1.+35.j]]
b= [[ 1. +0.j  1. -1.j  1. -2.j  1. -3.j  1. -4.j  1. -5.j]
 [ 1. -6.j  1. -7.j  1. -8.j  1. -9.j  1.-10.j  1.-11.j]
 [ 1.-12.j  1.-13.j  1.-14.j  1.-15.j  1.-16.j  1.-17.j]
 [ 1.-18.j  1.-19.j  1.-20.j  1.-21.j  1.-22.j  1.-23.j]
 [ 1.-24.j  1.-25.j  1.-26.j  1.-27.j  1.-28.j  1.-29.j]
 [ 1.-30.j  1.-31.j  1.-32.j  1.-33.j  1.-34.j  1.-35.j]]
c= [[ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]
 [ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]
 [ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]
 [ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]
 [ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]
 [ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]]
alpha= 2.0 ; beta= 3.0
PvHEMM= [[  672.-147.j  1668.-103.j  2244.-101.j  1980.-141.j   456.-223.j
       -2748.-347.j]
 [  702.-159.j  1742.-115.j  2320.-113.j  2016.-153.j   410.-235.j
       -2918.-359.j]
 [  732.-171.j  1816.-127.j  2396.-125.j  2052.-165.j   364.-247.j
       -3088.-371.j]
 [  762.-183.j  1890.-139.j  2472.-137.j  2088.-177.j   318.-259.j
       -3258.-383.j]
 [  792.-195.j  1964.-151.j  2548.-149.j  2124.-189.j   272.-271.j
       -3428.-395.j]
 [  822.-207.j  2038.-163.j  2624.-161.j  2160.-201.j   226.-283.j
```

```
-3598.-407.j]]
```

Para comprobar el correcto funcionamiento de las rutinas incluidas en el nivel 3 de PyPBLAS se han desarrollado una serie de pruebas similares a las realizadas para los niveles inferiores de PyPBLAS. Por ejemplo, para la rutina `pvgemm` se han realizado las pruebas que se muestran en la figura 7.23. En esta figura se muestran los tiempos de ejecución para diferentes tamaños de matrices cuadradas $N \times N$ y diferente configuración de la malla de procesos. Para cada rutina se han realizado las pruebas en dos plataformas diferentes, en el Cluster-umh y en Seaborg, mostrándose los tiempos obtenidos en las figuras 7.23(a) y 7.23(b) respectivamente. En ambas figuras, se aprecian tiempos de ejecución muy similares entre los lenguajes Python y Fortran para diferentes tamaños de matrices y diferentes configuraciones en la malla de procesos. Del mismo modo, se muestran las comparativas en los tiempos de ejecución obtenidos para todas las rutinas de nivel 3 de PyPBLAS: `pvsymm` (figura 7.25), `pvhemm` (figura 7.26), `pvsyrk` (figura 7.27), `pvherk` (figura 7.28), `pvsyr2k` (figura 7.29), `pvtran` (figura 7.30), `pvtranc` (figura 7.31), `pvtranu` (figura 7.32), `pvtrmm` (figura 7.33) y `pvtrsm` (figura 7.34).

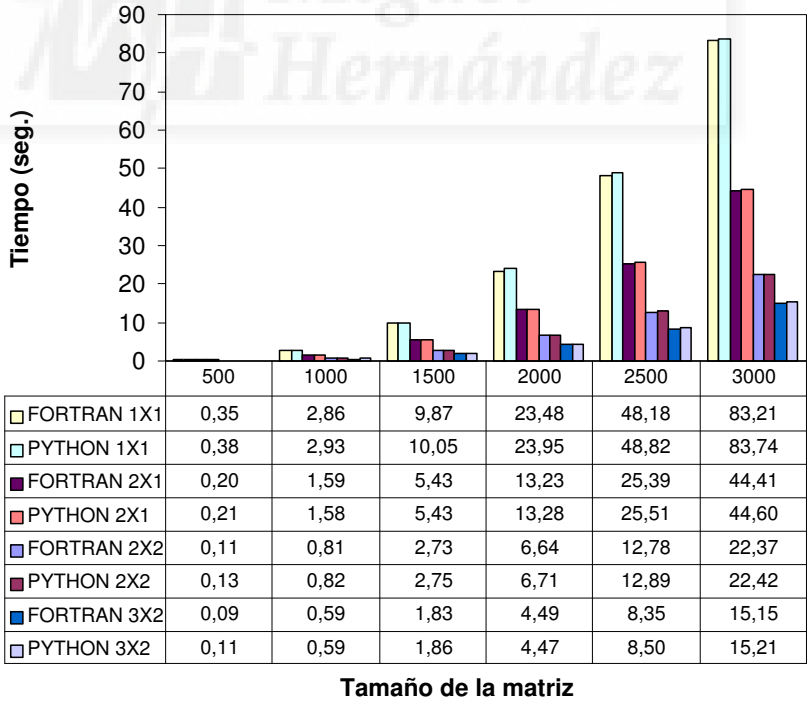
Por otro lado, en la figura 7.24 se muestra el rendimiento obtenido en el Cluster-umh y en Seaborg expresado en MFlops. En ambas plataformas, el comportamiento del rendimiento es similar utilizando el módulo PyPBLAS y la librería PBLAS desde Fortran. De hecho, en Seaborg se observa de forma clara cómo se incrementa el rendimiento del sistema a medida que añadimos procesos en la ejecución y se obtienen eficiencias similares para todos los tamaños de las matrices cuadradas $N \times N$ probadas.

Cabe mencionar que en las rutinas que emplean datos de tipo complejo como `pvherk` (figura 7.28(b)), `pvtranc` (figura 7.31(b)), `pvtranu` (figura 7.32(b)) o en `pvtrmm` (figura 7.33(b)) se observan determinadas celdas en blanco en la ejecución de Fortran que no aparecen en la ejecución en Python análoga. Esto significa que mediante Python y PyPBLAS hemos podido ejecutar algunas rutinas con tamaños de matrices elevados que en Fortran no había sido posible. Esto se debe a la gestión más eficiente de los recursos de memoria que realiza Python mediante las matrices `Numpy`. La gestión de la memoria en Fortran utilizada en las pruebas realizadas se corresponde con la realizada en los ejemplos de la distribución oficial de PBLAS. En estos ejemplos, se define un vector de una memoria máxima y cada una de las matrices se sitúan en dicho vector utilizando índices de posición. En definitiva, la gestión de los recursos de memoria en Fortran se realiza desde un nivel muy bajo que para un usuario no iniciado o con conocimientos medios, es muy complejo realizar una correcta y óptima asignación de memoria.



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,09	0,64	2,08	4,85	9,41	16,01
PYTHON 1X1	0,13	0,64	2,09	4,84	9,42	15,99
FORTRAN 2X1	0,14	0,84	2,63	6,20	11,58	18,22
PYTHON 2X1	0,17	0,86	2,67	6,31	11,78	18,26
FORTRAN 2X2	0,09	0,53	1,56	3,53	6,49	10,17
PYTHON 2X2	0,14	0,54	1,58	3,59	6,62	10,32
FORTRAN 3X2	0,09	0,45	1,22	2,55	4,62	7,84
PYTHON 3X2	0,14	0,47	1,25	2,60	4,72	7,96

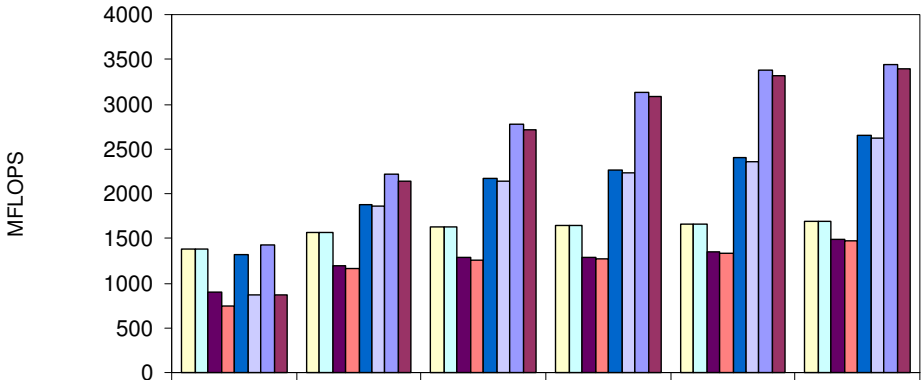
(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,35	2,86	9,87	23,48	48,18	83,21
PYTHON 1X1	0,38	2,93	10,05	23,95	48,82	83,74
FORTRAN 2X1	0,20	1,59	5,43	13,23	25,39	44,41
PYTHON 2X1	0,21	1,58	5,43	13,28	25,51	44,60
FORTRAN 2X2	0,11	0,81	2,73	6,64	12,78	22,37
PYTHON 2X2	0,13	0,82	2,75	6,71	12,89	22,42
FORTRAN 3X2	0,09	0,59	1,83	4,49	8,35	15,15
PYTHON 3X2	0,11	0,59	1,86	4,47	8,50	15,21

(b) Seaborg

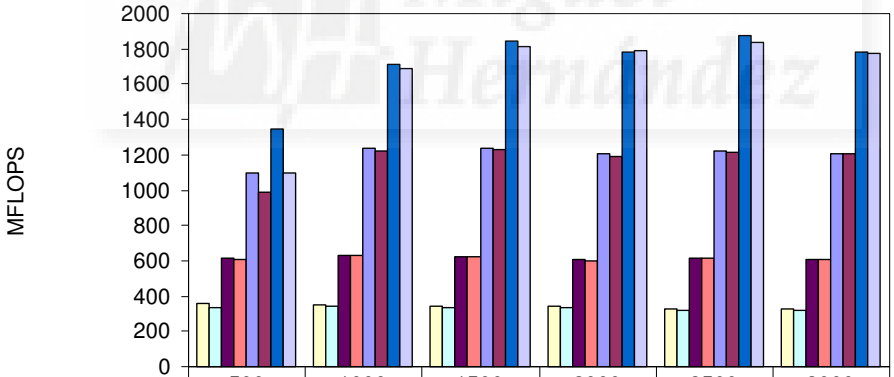
Figura 7.23: Comparativa de la rutina `pvgemm` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	1372,6	1563,9	1623,7	1650,5	1662,3	1687,8
PYTHON 1X1	1372,6	1563,9	1623,7	1650,5	1662,3	1687,8
FORTRAN 2X1	906,9	1187,2	1282,8	1291,5	1350,4	1482,8
PYTHON 2X1	743,2	1167,5	1263,5	1269,4	1327,3	1479,4
FORTRAN 2X2	1323,4	1883,7	2172,0	2267,7	2410,5	2657,1
PYTHON 2X2	870,3	1867,7	2140,7	2231,0	2361,8	2618,5
FORTRAN 3X2	1419,4	2216,1	2777,5	3137,0	3383,5	3444,3
PYTHON 3X2	873,0	2146,2	2709,3	3080,8	3314,8	3394,9

Tamaño de la matriz

(a) Cluster-umh

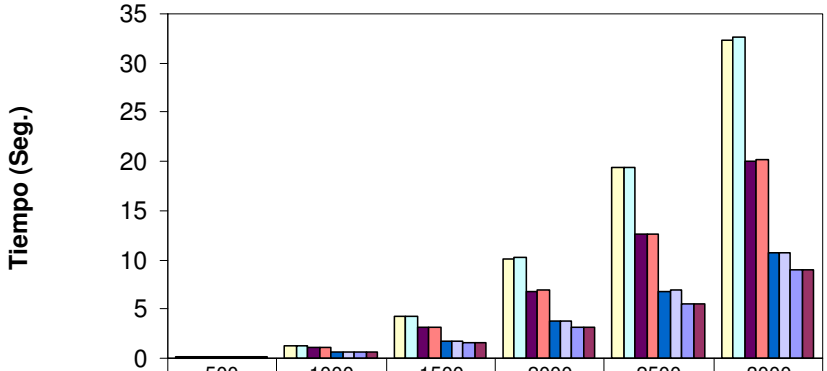


	500	1000	1500	2000	2500	3000
FORTRAN 1X1	354,4	350,8	342,2	341,1	324,6	324,7
PYTHON 1X1	334,3	342,0	336,4	334,4	320,3	322,7
FORTRAN 2X1	613,6	629,8	622,1	605,1	615,9	608,4
PYTHON 2X1	606,9	632,3	622,6	603,0	613,0	605,8
FORTRAN 2X2	1099,9	1236,2	1237,4	1206,7	1223,5	1207,7
PYTHON 2X2	988,8	1224,1	1229,5	1193,5	1213,0	1205,1
FORTRAN 3X2	1347,6	1711,5	1843,1	1785,3	1872,6	1783,8
PYTHON 3X2	1095,3	1691,9	1814,5	1790,7	1839,8	1776,4

Tamaño de la matriz

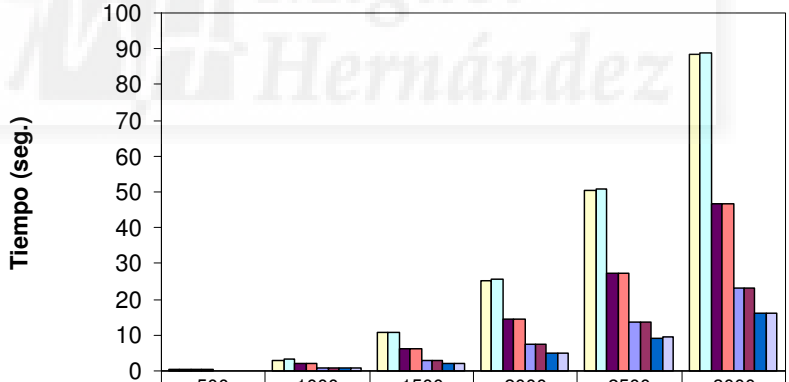
(b) Seaborg

Figura 7.24: Comparativa de MFlops de la rutina `pvgemm` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,17	1,31	4,22	10,17	19,35	32,40
PYTHON 1X1	0,20	1,33	4,26	10,20	19,45	32,65
FORTRAN 2X1	0,20	1,06	3,09	6,82	12,57	20,06
PYTHON 2X1	0,20	1,08	3,12	6,91	12,66	20,18
FORTRAN 2X2	0,11	0,61	1,72	3,72	6,85	10,67
PYTHON 2X2	0,19	0,61	1,73	3,77	6,87	10,72
FORTRAN 3X2	0,11	0,59	1,52	3,12	5,52	8,94
PYTHON 3X2	0,16	0,59	1,54	3,13	5,59	9,00

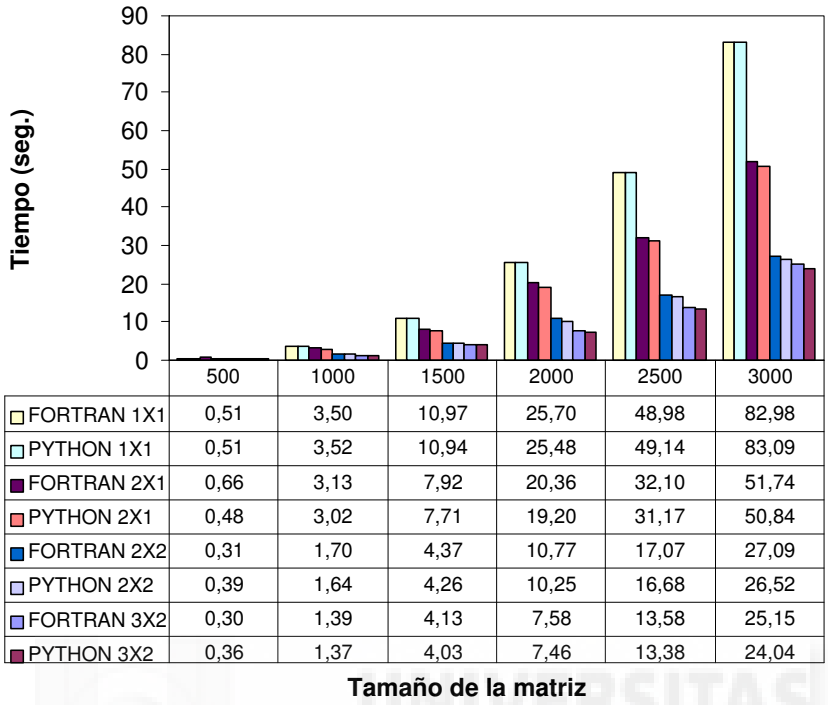
(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,39	3,07	10,56	25,26	50,21	88,46
PYTHON 1X1	0,41	3,20	10,83	25,67	50,96	88,67
FORTRAN 2X1	0,27	1,88	6,04	14,50	27,31	46,75
PYTHON 2X1	0,26	1,88	6,05	14,50	27,30	46,65
FORTRAN 2X2	0,14	0,94	3,03	7,31	13,65	23,05
PYTHON 2X2	0,15	0,94	3,02	7,24	13,58	23,13
FORTRAN 3X2	0,12	0,69	2,10	4,83	9,18	16,28
PYTHON 3X2	0,15	0,69	2,17	4,93	9,41	16,32

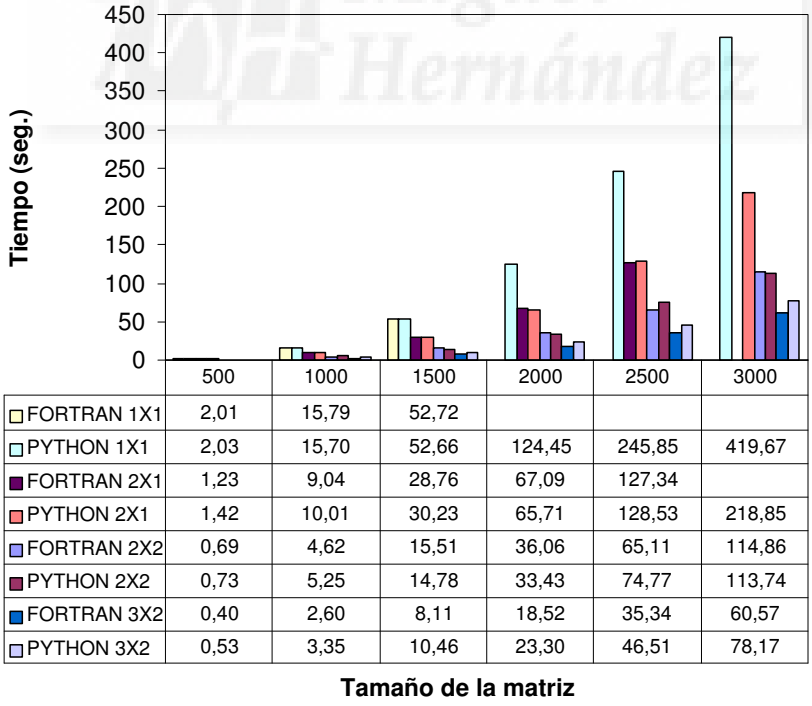
(b) Seaborg

Figura 7.25: Comparativa de la rutina `pvsymm` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,51	3,50	10,97	25,70	48,98	82,98
PYTHON 1X1	0,51	3,52	10,94	25,48	49,14	83,09
FORTRAN 2X1	0,66	3,13	7,92	20,36	32,10	51,74
PYTHON 2X1	0,48	3,02	7,71	19,20	31,17	50,84
FORTRAN 2X2	0,31	1,70	4,37	10,77	17,07	27,09
PYTHON 2X2	0,39	1,64	4,26	10,25	16,68	26,52
FORTRAN 3X2	0,30	1,39	4,13	7,58	13,58	25,15
PYTHON 3X2	0,36	1,37	4,03	7,46	13,38	24,04

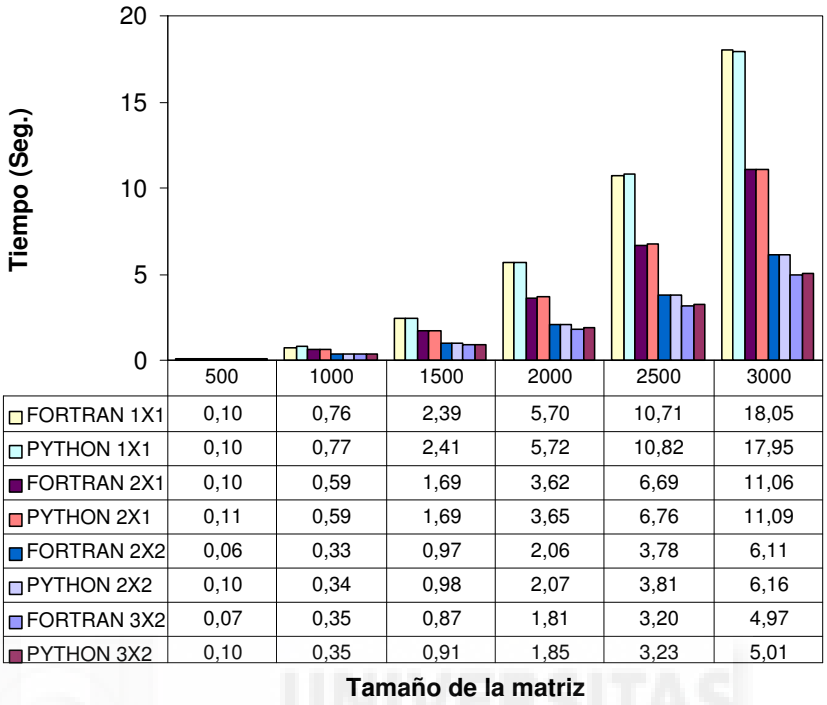
(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	2,01	15,79	52,72			
PYTHON 1X1	2,03	15,70	52,66	124,45	245,85	419,67
FORTRAN 2X1	1,23	9,04	28,76	67,09	127,34	
PYTHON 2X1	1,42	10,01	30,23	65,71	128,53	218,85
FORTRAN 2X2	0,69	4,62	15,51	36,06	65,11	114,86
PYTHON 2X2	0,73	5,25	14,78	33,43	74,77	113,74
FORTRAN 3X2	0,40	2,60	8,11	18,52	35,34	60,57
PYTHON 3X2	0,53	3,35	10,46	23,30	46,51	78,17

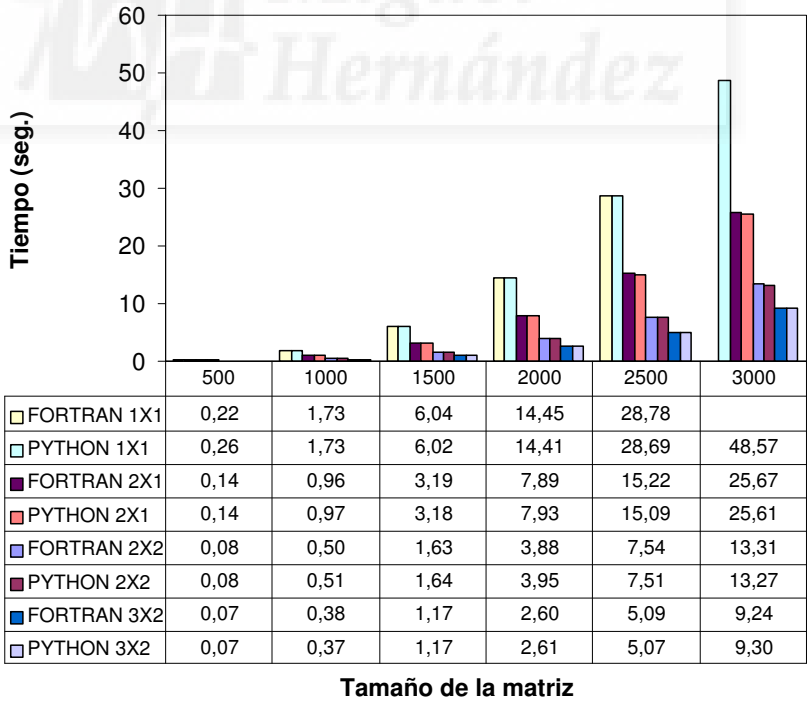
(b) Seaborg

Figura 7.26: Comparativa de la rutina `pvhemm` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,10	0,76	2,39	5,70	10,71	18,05
PYTHON 1X1	0,10	0,77	2,41	5,72	10,82	17,95
FORTRAN 2X1	0,10	0,59	1,69	3,62	6,69	11,06
PYTHON 2X1	0,11	0,59	1,69	3,65	6,76	11,09
FORTRAN 2X2	0,06	0,33	0,97	2,06	3,78	6,11
PYTHON 2X2	0,10	0,34	0,98	2,07	3,81	6,16
FORTRAN 3X2	0,07	0,35	0,87	1,81	3,20	4,97
PYTHON 3X2	0,10	0,35	0,91	1,85	3,23	5,01

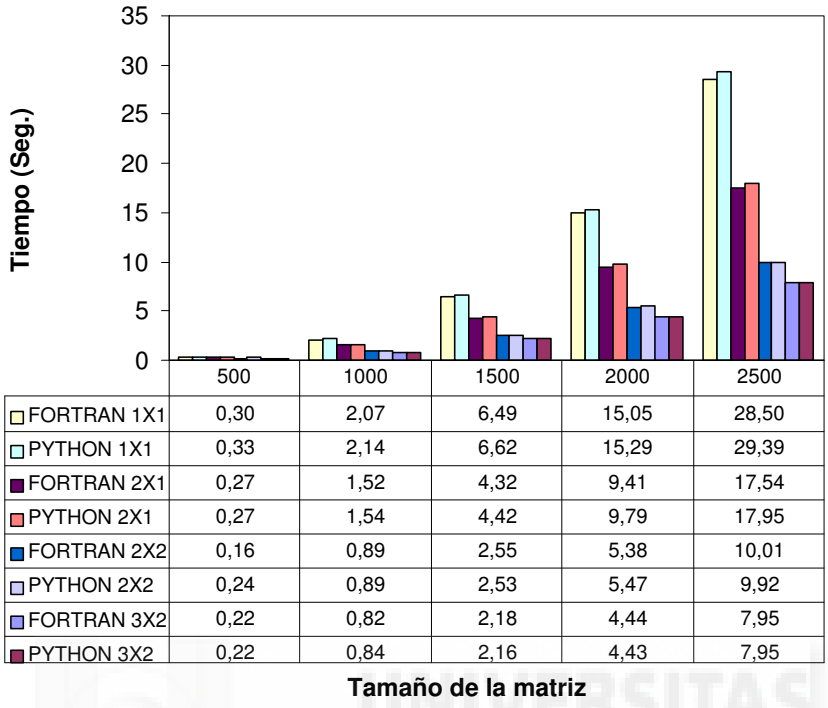
(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,22	1,73	6,04	14,45	28,78	
PYTHON 1X1	0,26	1,73	6,02	14,41	28,69	48,57
FORTRAN 2X1	0,14	0,96	3,19	7,89	15,22	25,67
PYTHON 2X1	0,14	0,97	3,18	7,93	15,09	25,61
FORTRAN 2X2	0,08	0,50	1,63	3,88	7,54	13,31
PYTHON 2X2	0,08	0,51	1,64	3,95	7,51	13,27
FORTRAN 3X2	0,07	0,38	1,17	2,60	5,09	9,24
PYTHON 3X2	0,07	0,37	1,17	2,61	5,07	9,30

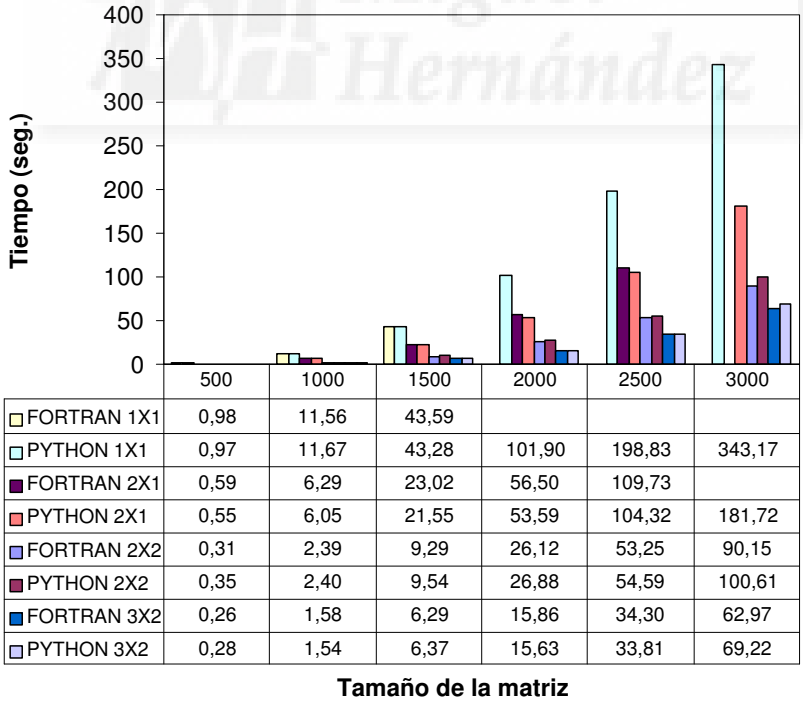
(b) Seaborg

Figura 7.27: Comparativa de la rutina `pvsyrk` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500
FORTRAN 1X1	0,30	2,07	6,49	15,05	28,50
PYTHON 1X1	0,33	2,14	6,62	15,29	29,39
FORTRAN 2X1	0,27	1,52	4,32	9,41	17,54
PYTHON 2X1	0,27	1,54	4,42	9,79	17,95
FORTRAN 2X2	0,16	0,89	2,55	5,38	10,01
PYTHON 2X2	0,24	0,89	2,53	5,47	9,92
FORTRAN 3X2	0,22	0,82	2,18	4,44	7,95
PYTHON 3X2	0,22	0,84	2,16	4,43	7,95

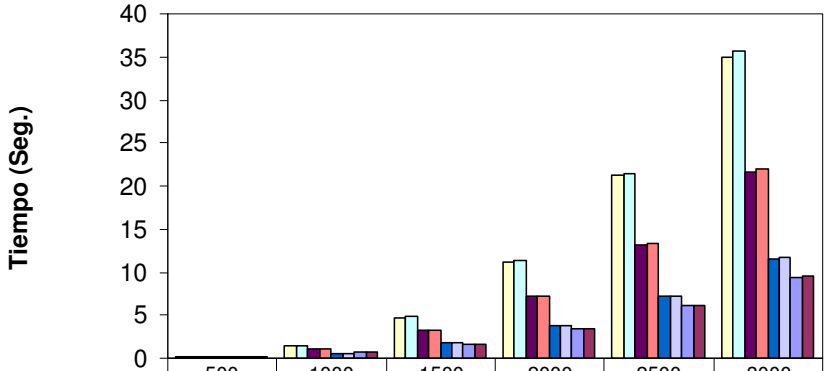
(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,98	11,56	43,59			
PYTHON 1X1	0,97	11,67	43,28	101,90	198,83	343,17
FORTRAN 2X1	0,59	6,29	23,02	56,50	109,73	
PYTHON 2X1	0,55	6,05	21,55	53,59	104,32	181,72
FORTRAN 2X2	0,31	2,39	9,29	26,12	53,25	90,15
PYTHON 2X2	0,35	2,40	9,54	26,88	54,59	100,61
FORTRAN 3X2	0,26	1,58	6,29	15,86	34,30	62,97
PYTHON 3X2	0,28	1,54	6,37	15,63	33,81	69,22

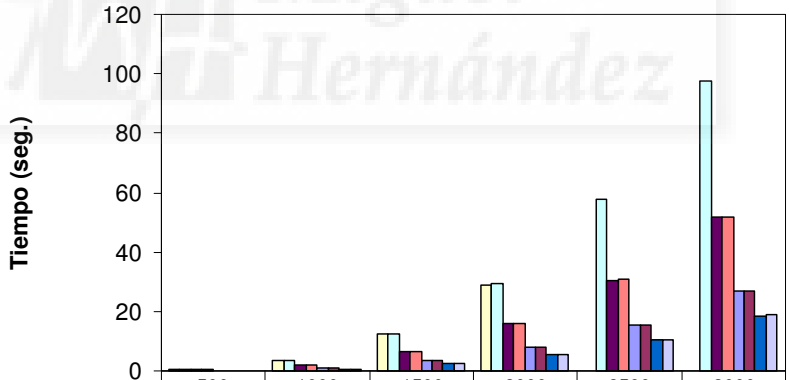
(b) Seaborg

Figura 7.28: Comparativa de la rutina `pvherk` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,20	1,48	4,71	11,17	21,35	35,04
PYTHON 1X1	0,20	1,51	4,78	11,35	21,46	35,68
FORTRAN 2X1	0,21	1,17	3,30	7,14	13,19	21,70
PYTHON 2X1	0,21	1,17	3,33	7,26	13,36	22,03
FORTRAN 2X2	0,12	0,60	1,79	3,84	7,13	11,59
PYTHON 2X2	0,15	0,60	1,80	3,85	7,25	11,76
FORTRAN 3X2	0,17	0,64	1,63	3,42	6,06	9,45
PYTHON 3X2	0,16	0,67	1,69	3,48	6,14	9,58

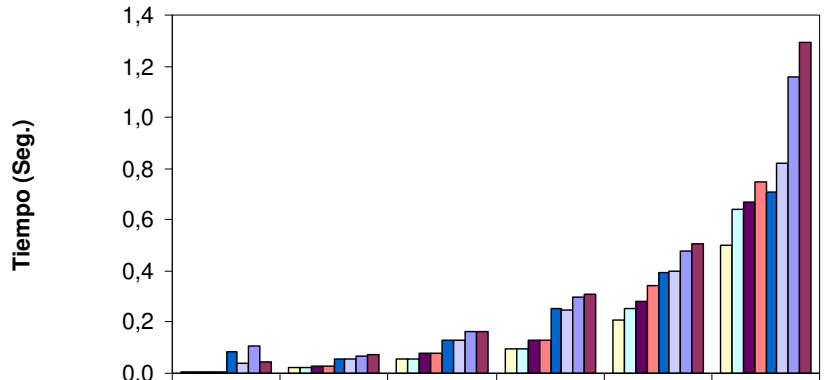
(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,43	3,62	12,31	29,06		
PYTHON 1X1	0,45	3,63	12,29	29,15	57,74	97,58
FORTRAN 2X1	0,29	2,03	6,63	15,89	30,61	51,56
PYTHON 2X1	0,29	2,04	6,64	15,97	30,68	51,73
FORTRAN 2X2	0,17	1,01	3,33	8,09	15,28	26,75
PYTHON 2X2	0,17	1,01	3,35	8,16	15,28	26,81
FORTRAN 3X2	0,13	0,74	2,39	5,40	10,53	18,61
PYTHON 3X2	0,13	0,73	2,39	5,41	10,52	18,70

(b) Seaborg

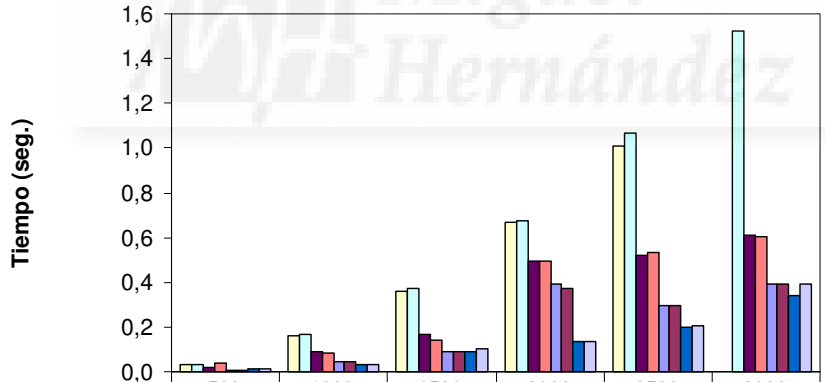
Figura 7.29: Comparativa de la rutina `pvsyr2k` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,01	0,02	0,06	0,10	0,21	0,50
PYTHON 1X1	0,01	0,02	0,06	0,10	0,26	0,64
FORTRAN 2X1	0,01	0,03	0,08	0,13	0,28	0,67
PYTHON 2X1	0,01	0,03	0,08	0,13	0,34	0,75
FORTRAN 2X2	0,09	0,05	0,13	0,25	0,39	0,71
PYTHON 2X2	0,04	0,06	0,13	0,25	0,40	0,82
FORTRAN 3X2	0,11	0,07	0,16	0,30	0,48	1,16
PYTHON 3X2	0,05	0,07	0,16	0,31	0,51	1,29

Tamaño de la matriz

(a) Cluster-umh

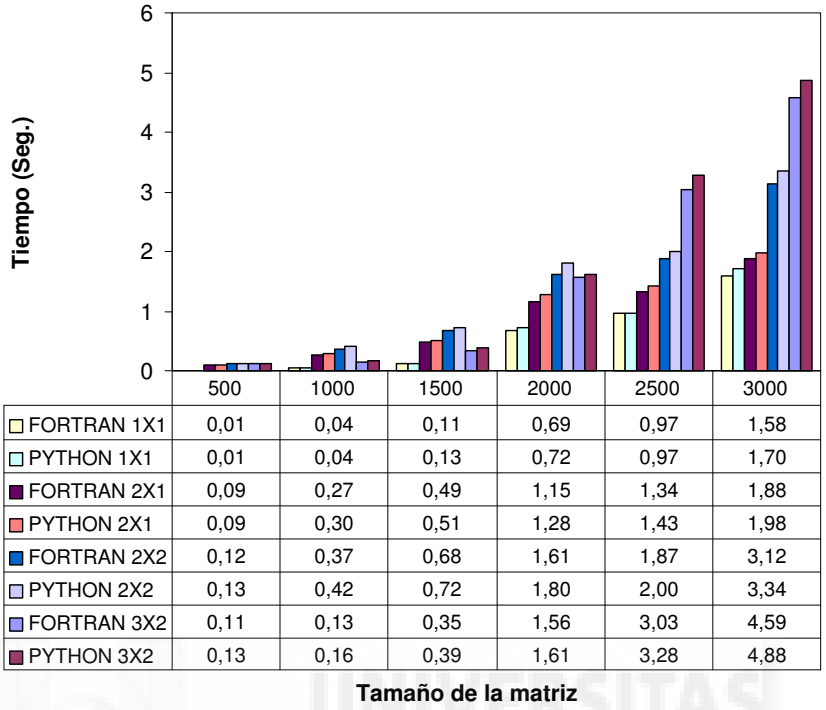


	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,04	0,16	0,36	0,67	1,01	
PYTHON 1X1	0,03	0,17	0,37	0,67	1,07	1,52
FORTRAN 2X1	0,02	0,09	0,17	0,50	0,52	0,61
PYTHON 2X1	0,04	0,08	0,14	0,49	0,53	0,61
FORTRAN 2X2	0,01	0,05	0,09	0,39	0,29	0,39
PYTHON 2X2	0,00	0,05	0,09	0,37	0,30	0,39
FORTRAN 3X2	0,01	0,03	0,09	0,14	0,20	0,34
PYTHON 3X2	0,01	0,03	0,10	0,14	0,21	0,39

Tamaño de la matriz

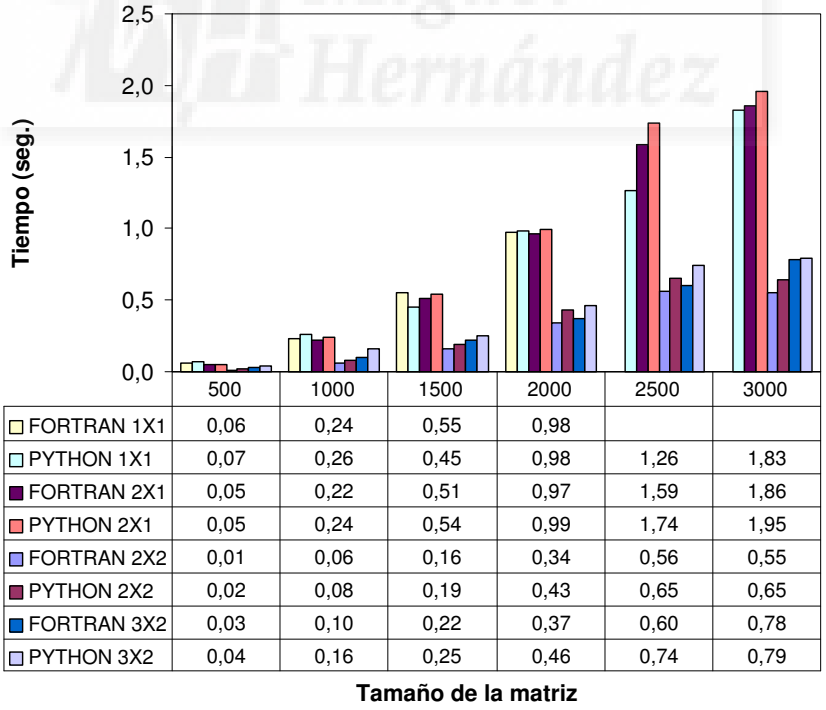
(b) Seaborg

Figura 7.30: Comparativa de la rutina `pvtran` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,01	0,04	0,11	0,69	0,97	1,58
PYTHON 1X1	0,01	0,04	0,13	0,72	0,97	1,70
FORTRAN 2X1	0,09	0,27	0,49	1,15	1,34	1,88
PYTHON 2X1	0,09	0,30	0,51	1,28	1,43	1,98
FORTRAN 2X2	0,12	0,37	0,68	1,61	1,87	3,12
PYTHON 2X2	0,13	0,42	0,72	1,80	2,00	3,34
FORTRAN 3X2	0,11	0,13	0,35	1,56	3,03	4,59
PYTHON 3X2	0,13	0,16	0,39	1,61	3,28	4,88

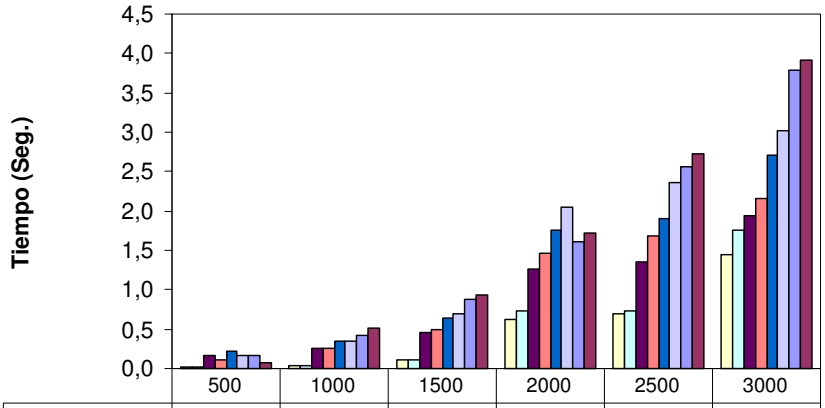
(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,06	0,24	0,55	0,98		
PYTHON 1X1	0,07	0,26	0,45	0,98	1,26	1,83
FORTRAN 2X1	0,05	0,22	0,51	0,97	1,59	1,86
PYTHON 2X1	0,05	0,24	0,54	0,99	1,74	1,95
FORTRAN 2X2	0,01	0,06	0,16	0,34	0,56	0,55
PYTHON 2X2	0,02	0,08	0,19	0,43	0,65	0,65
FORTRAN 3X2	0,03	0,10	0,22	0,37	0,60	0,78
PYTHON 3X2	0,04	0,16	0,25	0,46	0,74	0,79

(b) Seaborg

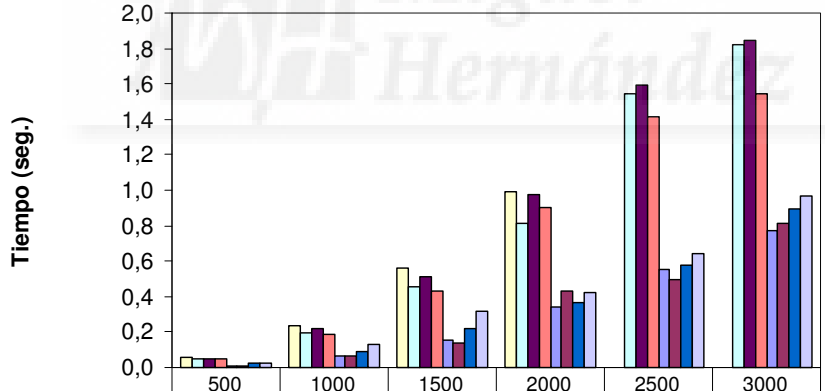
Figura 7.31: Comparativa de la rutina `pvtranc` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,01	0,04	0,11	0,62	0,69	1,45
PYTHON 1X1	0,01	0,04	0,11	0,73	0,73	1,76
FORTRAN 2X1	0,16	0,25	0,46	1,26	1,36	1,94
PYTHON 2X1	0,11	0,25	0,49	1,47	1,68	2,15
FORTRAN 2X2	0,22	0,35	0,64	1,76	1,90	2,71
PYTHON 2X2	0,16	0,36	0,69	2,06	2,35	3,01
FORTRAN 3X2	0,16	0,42	0,88	1,61	2,56	3,78
PYTHON 3X2	0,08	0,52	0,94	1,72	2,72	3,92

Tamaño de la matriz

(a) Cluster-umh

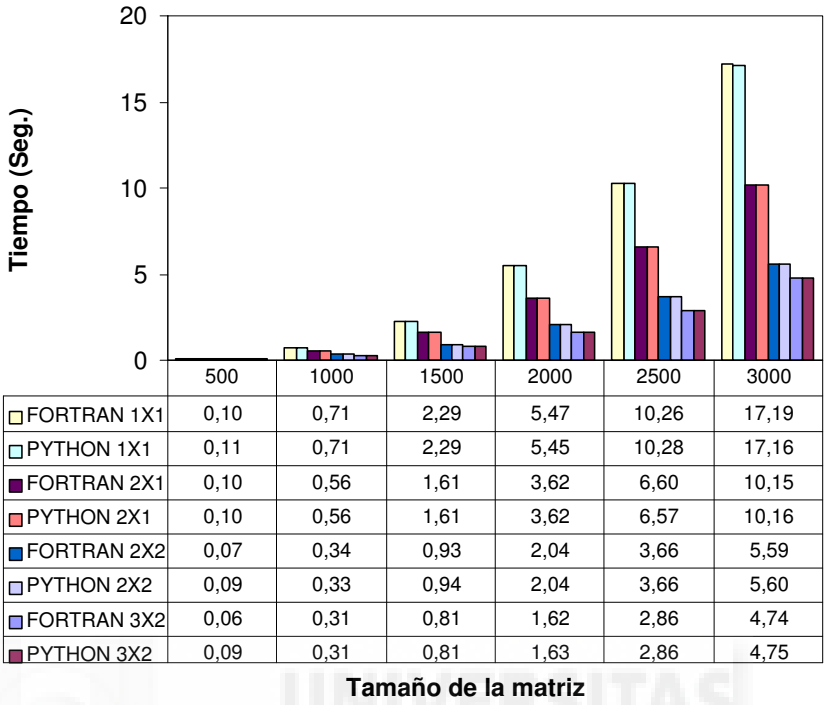


	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,06	0,24	0,56	0,99		
PYTHON 1X1	0,05	0,20	0,46	0,81	1,54	1,82
FORTRAN 2X1	0,04	0,22	0,51	0,97	1,59	1,84
PYTHON 2X1	0,05	0,19	0,43	0,90	1,41	1,55
FORTRAN 2X2	0,01	0,07	0,16	0,34	0,55	0,77
PYTHON 2X2	0,01	0,07	0,14	0,43	0,49	0,81
FORTRAN 3X2	0,02	0,09	0,22	0,36	0,58	0,89
PYTHON 3X2	0,03	0,13	0,32	0,43	0,64	0,97

Tamaño de la matriz

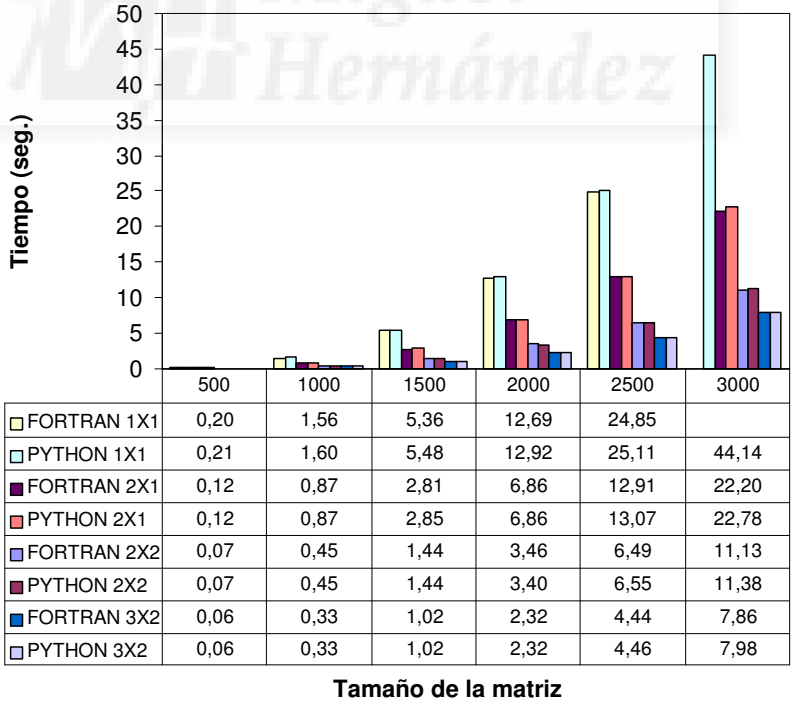
(b) Seaborg

Figura 7.32: Comparativa de la rutina `pvtranu` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,10	0,71	2,29	5,47	10,26	17,19
PYTHON 1X1	0,11	0,71	2,29	5,45	10,28	17,16
FORTRAN 2X1	0,10	0,56	1,61	3,62	6,60	10,15
PYTHON 2X1	0,10	0,56	1,61	3,62	6,57	10,16
FORTRAN 2X2	0,07	0,34	0,93	2,04	3,66	5,59
PYTHON 2X2	0,09	0,33	0,94	2,04	3,66	5,60
FORTRAN 3X2	0,06	0,31	0,81	1,62	2,86	4,74
PYTHON 3X2	0,09	0,31	0,81	1,63	2,86	4,75

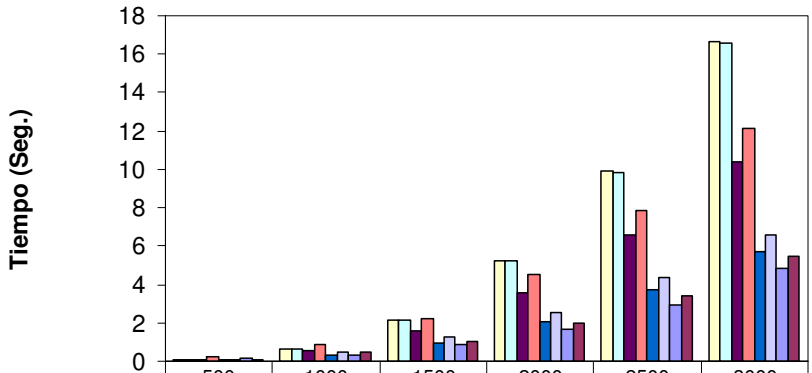
(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,20	1,56	5,36	12,69	24,85	
PYTHON 1X1	0,21	1,60	5,48	12,92	25,11	44,14
FORTRAN 2X1	0,12	0,87	2,81	6,86	12,91	22,20
PYTHON 2X1	0,12	0,87	2,85	6,86	13,07	22,78
FORTRAN 2X2	0,07	0,45	1,44	3,46	6,49	11,13
PYTHON 2X2	0,07	0,45	1,44	3,40	6,55	11,38
FORTRAN 3X2	0,06	0,33	1,02	2,32	4,44	7,86
PYTHON 3X2	0,06	0,33	1,02	2,32	4,46	7,98

(b) Seaborg

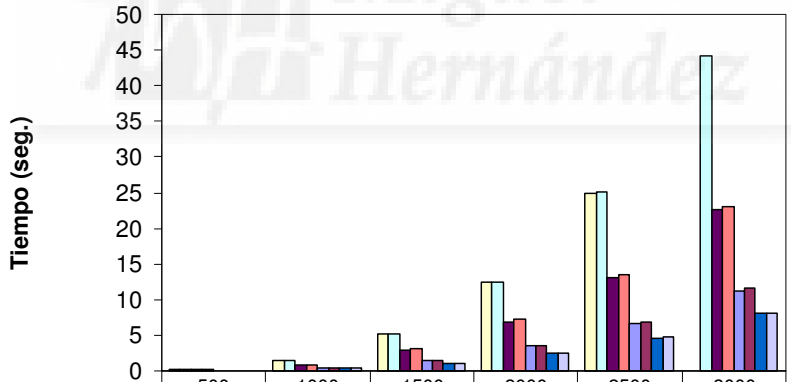
Figura 7.33: Comparativa de la rutina `pvtrmm` en lenguaje Fortran vs Python



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,08	0,65	2,14	5,21	9,88	16,69
PYTHON 1X1	0,08	0,65	2,15	5,20	9,86	16,58
FORTRAN 2X1	0,10	0,56	1,62	3,60	6,57	10,35
PYTHON 2X1	0,25	0,88	2,19	4,51	7,87	12,14
FORTRAN 2X2	0,06	0,33	0,94	2,04	3,69	5,70
PYTHON 2X2	0,09	0,50	1,24	2,51	4,34	6,61
FORTRAN 3X2	0,12	0,33	0,85	1,69	2,97	4,86
PYTHON 3X2	0,09	0,44	1,04	2,00	3,43	5,47

Tamaño de la matriz

(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	0,19	1,51	5,19	12,50	24,90	
PYTHON 1X1	0,19	1,53	5,24	12,52	25,10	44,17
FORTRAN 2X1	0,12	0,87	2,84	6,88	13,13	22,58
PYTHON 2X1	0,18	0,89	3,04	7,17	13,40	23,11
FORTRAN 2X2	0,08	0,46	1,47	3,51	6,60	11,31
PYTHON 2X2	0,09	0,51	1,50	3,49	6,76	11,62
FORTRAN 3X2	0,09	0,35	1,07	2,42	4,60	8,05
PYTHON 3X2	0,08	0,38	1,12	2,51	4,74	8,14

Tamaño de la matriz

(b) Seaborg

Figura 7.34: Comparativa de la rutina `pvtrsm` en lenguaje Fortran vs Python

7.5. Conclusiones

En este capítulo hemos introducido el módulo PyPBLAS y se han presentado las rutinas que incluye agrupadas en los tres niveles. Para cada uno de los tres niveles, se ha mostrado un ejemplo de utilización de una rutina. En todos los ejemplos mostrados, las rutinas utilizadas para la inicialización y liberación del entorno paralelo y las rutinas auxiliares mostradas para conversión de datos han sido las presentadas en el capítulo 5. De este modo, comprobamos la integración e interrelación entre los diferentes módulos que componen PyACTS.

El número de rutinas incluidas en PyPBLAS es elevado, además se ha de tener en cuenta que las rutinas utilizadas en niveles inferiores (PBLAS) son cuatro veces más, puesto que para cada funcionalidad se proporcionan cuatro rutinas que varían su nombre a partir del tipo de dato (S, D, C y Z). De este modo, se ha conseguido una mayor simplicidad en la utilización de PyPBLAS puesto que el tipo de dato va implícito en la propiedad `data` de las matrices PyACTS que son realmente matrices Numpy.

Según las pruebas mostradas desde la figura 7.1 hasta la figura 7.34, los tiempos de ejecución obtenidos con PyPBLAS son muy similares a los obtenidos utilizando las librerías mediante su interfaz a Fortran. Además, en la figura 7.24 hemos mostrado el rendimiento en MFlops obtenido para Cluster-umh y para Seaborg.

Por otro lado, fundamentalmente en las rutinas de nivel 3, hemos comprobado la mejora en la gestión de la memoria que realiza Python frente a una aplicación desarrollada en Fortran. La sencillez en la creación dinámica de las matrices de datos viene acompañada de una gestión eficiente de memoria que permite que se puedan ejecutar las operaciones en Python para algunos tamaños de matrices mientras que en Fortran se necesitan más recursos de memoria.

En resumen, podemos concluir que mediante el módulo PyPBLAS conseguimos una herramienta que amplia el espectro de usuarios de librerías computacionales de altas prestaciones al facilitar su utilización desde un lenguaje interpretado. Las automatizaciones utilizadas y los objetos definidos permiten simplificar muchos de los parámetros necesarios en cada rutina sin perder eficiencia en las llamadas a las mismas. Se consigue por tanto una herramienta escalable, con la misma robustez que la librería PBLAS pero con una mejor interacción y dinamismo. Además proporcionamos un amplio conjunto de ejemplos dentro del manual de referencia de PyACTS, que permiten a los usuarios comprobar y aprender

el funcionamiento de cada una de las rutinas. Consideramos por tanto que PyPBLAS es una herramienta adecuada para la implementación de aplicaciones con altos requisitos computacionales, que permite una sintaxis y comprensión sencilla sin detrimento alguno de su rendimiento.

Capítulo 8

PyScaLAPACK

En este capítulo nos centraremos en el conjunto de rutinas de PyScaLAPACK [39, 52] disponibles en el paquete PyACTS. En la primera sección realizaremos una clasificación de los grupos funcionales basándonos en la clasificación realizada en el apartado 4.3.5. En la sección 8.2 se presentan las rutinas clasificadas como rutinas driver que permiten resolver un problema de manera completa, tal como la resolución de un sistema de ecuaciones lineales (apartado 8.2.1), la obtención de una solución por mínimos cuadrados (apartado 8.2.2), o el cálculo de los valores propios y la descomposición en valores singulares de una matriz (apartado 8.2.3). En la sección 8.3 presentamos las rutinas computacionales que realizan una tarea específica sin llegar a resolver un problema de manera completa. La concatenación de rutinas computacionales conforman una rutina driver y por tanto, las rutinas computaciones se subdividen atendiendo al problema que tratan: sistemas de ecuaciones (apartado 8.3.1), mínimos cuadrados (apartado 8.3.2), valores propios (apartado 8.3.3) y descomposición en valores singulares (apartado 8.3.4). Una vez presentado PyACTS y todos los módulos que lo componen, en la sección 8.4 se describen las pruebas realizadas y se justifica la elección planteada en el apartado 5.8 de los parámetros y configuraciones que realiza por defecto PyACTS. Por último, en la sección 8.5 se exponen las conclusiones obtenidas a partir del conjunto de rutinas mostradas y de las pruebas efectuadas en distintas plataformas.

8.1. Introducción

En el apartado 4.3.5 se realizó una introducción a la librería ScaLAPACK describiendo sus principales funcionalidades, su estructura y los pasos necesarios para poder utilizar sus rutinas. También en dicho apartado se realizó un recorrido por la totalidad de las rutinas proporcionadas en ScaLAPACK atendiendo a su clasificación. Es por ello que en este capítulo presuponemos que se conocen los conceptos tratados en este apartado, así como los tratados en el apartado de BLACS (4.3.3) y de PBLAS (4.3.4).

Recordando la estructura funcional de ScaLAPACK vista en el apartado 4.3.5, las rutinas se pueden agrupar en tres grandes bloques funcionales y sus correspondientes subbloques que se enumeran a continuación:

- Rutinas driver
  - Ecuaciones lineales.
  - Problemas de mínimos cuadrados.
  - Obtención de valores propios y descomposición en valores singulares.
  - Obtención de valores propios en matrices simétricas definidas.
- Rutinas computacionales
  - Ecuaciones lineales.
  - Problemas de mínimos cuadrados y factorizaciones ortogonales.
    - Factorización QR.
    - Factorización LQ.
    - Factorización QR con pivotación por columnas.
    - Factorización ortogonal completa.
  - Factorizaciones ortogonales generalizadas.
    - Factorización QR generalizada.
    - Factorización RQ generalizada.
  - Problemas de valores propios en matrices simétricas.
  - Problemas de valores propios en matrices no simétricas.

  - Descomposición en valores singulares.
  - Problemas de valores propios en matrices simétricas definidas.

- Rutinas auxiliares

En los dos siguientes apartados se describen con más detalle las rutinas driver y las rutinas computacionales incluidas en PyScaLAPACK. Sin embargo, en la distribución de PyScaLAPACK no se han incluido las rutinas auxiliares de ScaLAPACK puesto que éstas son útiles en el entorno de programación Fortran y carecen de utilidad en el entorno Python. Para cada uno de los siguientes apartados, se vuelven a clasificar en grupos, en función del problema que tratan de abordar las rutinas. Para cada uno de estos subgrupos se muestra un ejemplo de utilización y pruebas de ejecución que comparan la eficiencia obtenida entre el entorno propuesto y el tradicional. Sin embargo, creemos conveniente comentar que no se realizará una descripción detallada de la funcionalidad de cada rutina de PyScaLAPACK, ya que en el apartado 4.3.5 se realizó una descripción de las rutinas de ScaLAPACK. De este modo, si en los siguientes apartados se desea obtener más información de una rutina de PyScaLAPACK, podremos consultar la funcionalidad de la rutina análoga de ScaLAPACK descrita en el apartado 4.3.5 o bien en la referencia [17].

8.2. Rutinas driver

Este tipo de rutinas de PyScaLAPACK (y por tanto, de ScaLAPACK) resuelven un problema completo, es decir, implementan una funcionalidad para resolver un problema determinado en todas sus etapas. La resolución de un sistema de ecuación o la descomposición en valores singulares son dos ejemplos de problemas completos a resolver por estas rutinas.

A su vez, las rutinas driver se encuentran clasificadas atendiendo al tipo de problema que resuelven. En los siguientes subapartados se describen las rutinas de PyScaLAPACK para cada funcionalidad.

8.2.1. Ecuaciones lineales

Para la resoluciones de sistemas lineales, en PyScaLAPACK se incluyen dos tipos de rutinas driver agrupadas en función de la complejidad de su interfaz. Se ha de tener en cuenta que la mayor complejidad se debe a que se establecen nuevos parámetros en la rutina que permiten configurar determinadas opciones que en el caso de las rutinas sencillas se asumen por defecto. Sin embargo, en el caso de las rutinas avanzadas, dichos parámetros son configurables y ésto exige un conocimiento elevado en el manejo de estas rutinas. En este punto, consideramos de mayor importancia y utilización las rutinas sencillas puesto que aquellos usuarios que utilicen PyScaLAPACK seguramente harán uso de las rutinas sencillas, ya que desean obtener el resultado de una manera cómoda y fácil mediante una solución escalable en un lenguaje de alto nivel.

Se ha de tener en consideración que los parámetros de entrada y salida de cada rutina pueden ser matrices o escalares de tipo `Numeric` o PyACTS. Mediante la consulta del manual de usuario de PyACTS, podremos averiguar más información acerca de cada rutina o incluso en la propia interfaz de Python a través de la instrucción `help` (por ejemplo, "`>>help(PyScaLAPACK.pvgesv)`").

En general, las rutinas que describimos a continuación resuelven el sistema lineal de ecuaciones $Ax = B$. Debido a las peculiaridades de las matrices A y B se presentan diferentes rutinas que hacen uso de diferentes métodos numéricos para su resolución y que se detallan en los siguientes puntos. Debemos recordar que no profundizaremos en las características de las matrices ni en la funcionalidades de las rutinas puesto que ya fueron introducidas en el apartado 4.3.5. A continuación se describen las rutinas de tipo driver de PyScaLAPACK que implementan la resolución de sistemas lineales clasificadas en función del tipo de la matriz de coeficientes y de la complejidad de la interfaz.

- Matriz general (pivotación parcial).
  - Sencilla: `b,info=pvgesv(a,b)`

    Implementa la resolución de un sistema lineal de ecuaciones a partir de la matriz PyACTS `a` de coeficientes, de dimensiones $n \times n$, y la matriz PyACTS `b` de términos independientes y dimensiones $n \times nrhs$.

    Consideramos interesante comparar y analizar si se ha obtenido una mejora en cuanto a la comprensibilidad y facilidad a la hora de desarrollar una aplicación en Fortran o bien en un lenguaje de alto nivel como el propuesto. Para

poder realizar dicha comparación, debemos tener como referencia los ejemplos distribuidos en el sitio oficial de ScaLAPACK [12]. No adjuntamos en este trabajo ningún ejemplo de utilización de las rutinas de ScaLAPACK en Fortran y recomendamos acudir a las referencias si se desea comparar ambos ejemplos de una manera más directa. En resumen, un programa que utilice una rutina de ScaLAPACK como por ejemplo `PDGESV` puede ocupar más de 100 líneas puesto que de forma previa a la ejecución de la rutina se debían realizar unos pasos básicos ya descritos en el apartado 4.3.5.

A continuación se muestra un ejemplo de utilización de `pvgesv` que resuelve un sistema de ecuaciones mediante un código más cómodo, sencillo y reducido que en el entorno tradicional Fortran.

```
1  from PyACTS import *
2  import PyACTS.PyScaLAPACK as PySLK
3  from Numeric import *
4  n,nrhs=8,2
5  #Inicializa la malla
6  PyACTS.gridinit()
7  if PyACTS.iread==1:
8      #Definimos la matriz de coeficientes
9      a=n*identity(n,Float)+ ones([n,n],Float)
10     #y de terminos independientes
11                 b=ones((n,nrhs))
12 else:
13     a,b=None,None
14 ACTS_lib=1 # 1=Scalapack
15 a=Num2PyACTS(a,ACTS_lib)
16 b=Num2PyACTS(b,ACTS_lib)
17 #Llamamos a la rutina de PyScaLAPACK
18 b,info= PySLK.pvgesv(a,b)
19 b_num=PyACTS2Num(b)
20 if PyACTS.iread==1:
21         print "a*x=b-->x'=",transpose(b_num)
22         print "Info:",info
23 PyACTS.gridexit()
```

Del mismo modo que en el módulo PyBLACS y PyPBLAS, necesitamos inicializar la malla de procesos mediante el comando `PyACTS.gridinit()`. Se

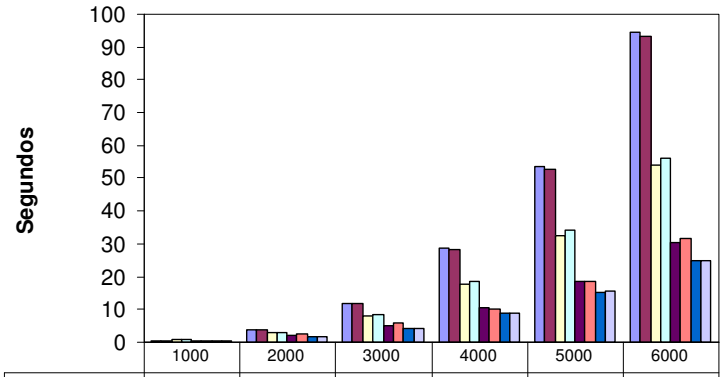
observa a partir de la línea 7, que el proceso origen de los datos (aquel con PyACTS.iread=1) crea las matrices de datos de tipo Numeric, mientras que el resto de nodos definen una variable con contenido vacío (en Python, este valor se denomina `None`). Esta definición en el resto de nodos es necesaria puesto que de no ser así la llamada de la línea 15 daría error en los nodos que no sean origen de datos por desconocer el valor de la variable `a`. De forma interna, la rutina `Num2PyACTS` comprueba cual es el nodo origen de los datos, y los distribuye al resto de nodos pertenecientes a la malla de procesos conforme a una distribución cíclica 2D.

En la línea 18 se realiza la llamada a la rutina `pvgesv` que resuelve el sistema de ecuaciones devolviendo el resultado en la matriz `b`. Se ha de tener en cuenta que esta variable es de tipo PyACTS, y si queremos imprimirla en pantalla en el orden adecuado tendremos que recoger al nodo origen el contenido distribuido de la variable entre los procesos pertenecientes a la malla de procesos y almacenarlo en una variable de tipo Numeric. Esta conversión de matriz PyACTS distribuida a matriz `Numeric` se realiza en la línea 19.

Podemos considerar este ejemplo como trivial puesto que los datos generados son muy simples y el tamaño del sistema es pequeño. Sin embargo, este sencillo ejemplo muestra la funcionalidad del entorno PyACTS ya que para resolver sistemas lineales de mayor tamaño los cambios necesarios para su resolución serían nulos.

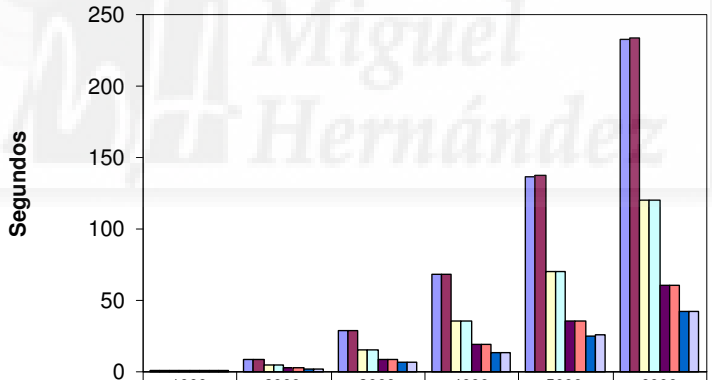
Del mismo modo que en los capítulos referidos a PyBLACS y PyPBLAS (capítulos 6 y 7, respectivamente), se muestra en la figura 8.1 la comparativa de los tiempos obtenidos para la resolución de un sistema lineal en entornos de programación Fortran y Python. Para ambos lenguajes, se ha realizado los cálculos variando el tamaño del sistema de ecuaciones (siendo la matriz A del tamaño $N \times N$ indicado en cada columna y B un vector $N \times 1$) y la configuración de la malla de procesos. Además, para garantizar que el sistema tiene solución hemos utilizado (tanto en Python como en Fortran) un sistema determinado mediante la definición de la matriz A y el vector B utilizada en la línea 8 del ejemplo de `pvgesv` visto anteriormente .

Si comparamos los tiempos entre Python y Fortran se observan tiempos de ejecución muy similares y en algunos casos incluso inferiores, aunque se sobreentiende que ésto se debe a pequeñas variaciones externas del algoritmo a realizar. Las diferencias de los tiempos de ejecución entre ambos entornos



	1000	2000	3000	4000	5000	6000
FORTRAN 1X1	0,57	3,84	12,00	28,58	53,63	94,52
PYTHON 1X1	0,49	3,77	11,76	28,31	52,81	93,34
FORTRAN 2X1	0,65	2,94	8,13	17,92	32,44	54,18
PYTHON 2X1	0,70	2,85	8,46	18,47	34,20	56,01
FORTRAN 2X2	0,52	1,93	5,03	10,53	18,56	30,33
PYTHON 2X2	0,53	2,33	6,05	10,27	18,58	31,61
FORTRAN 3X2	0,59	1,86	4,34	8,67	15,24	24,82
PYTHON 3X2	0,58	1,87	4,33	8,73	15,56	24,77

(a) Cluster-umh



	1000	2000	3000	4000	5000	6000
FORTRAN 1X1	1,06	8,35	28,63	68,21	136,85	232,99
PYTHON 1X1	1,06	8,38	28,90	68,63	137,24	233,88
FORTRAN 2X1	0,79	5,07	15,41	35,89	70,32	120,38
PYTHON 2X1	0,80	5,03	15,51	35,75	70,21	120,23
FORTRAN 2X2	0,69	2,82	8,45	18,89	35,88	60,42
PYTHON 2X2	0,55	2,87	8,40	18,84	35,94	60,45
FORTRAN 3X2	0,73	2,31	6,34	13,65	25,33	42,28
PYTHON 3X2	0,57	2,30	6,39	13,66	25,79	42,17

(b) Seaborg

Figura 8.1: Comparativa de `pvgesv` en lenguaje Fortran vs Python

de programación son mínimas, y el tiempo invertido en validar y convertir los parámetros para poder ser pasados a la rutina correspondiente (en este caso `PDGESV`) es despreciable con respecto a los tiempos de ejecución totales. En resumen, la facilidad de la herramienta y la legibilidad de las aplicaciones desarrolladas nos permite utilizar estas interfaces sin perder eficiencia en las mismas.

Se obtiene, por tanto, que el comportamiento para las rutinas de resolución de ecuaciones lineales de ScaLAPACK es similar al obtenido para PyBLACS y PyPBLAS, es decir, se obtiene una penalización que no suele superar el 5 %.

- Avanzada: `a,af,equed,r,c,b,x,rcond,ferr,berr,info=`
  `=pvgesvx(a,b[,fact='N', trans='N',equed='N',rcond=0])`

  Obtiene la resolución de un sistema lineal de ecuaciones y otras funciones adicionales (estimar el número de condición, obtener errores de redondeo y equilibrar el sistema). Debido a la mayor complejidad de esta rutina, dispone de un mayor número de parámetros de entrada que establecen los pasos a realizar en la rutina. Debido al elevado número de parámetros de entrada y de salida de las rutinas avanzadas, preferimos no extendernos y recomendamos la lectura del manual de PyScaLAPACK [34] y del manual de ScaLAPACK [14] para ampliar la información de las mismas.

- Matriz general en bandas (sencilla, pivotación parcial) :
  `b,info=pvgbsv(a,b,[bwl=n-1,bwu=n-1])`

  Implementa la resolución de un sistema lineal de ecuaciones a partir de la matriz PyACTS `a` de coeficientes en bandas y el vector PyACTS `b` de términos independientes. Se ha de tener en cuenta que en este caso, la matriz `a` es una matriz en bandas y la distribución de los datos de la matriz se debe realizar mediante las rutinas proporcionadas para convertir las matrices `Numpy` a matrices PyACTS en bandas (por ejemplo, `a=Num2PyACTS(a,ACTS_lib=501,bwu=2,bwl=2)`).

La disposición de los datos en las matrices en bandas es diferente a la distribución cíclica 2D vista para las matrices densas de tipo general en ScaLAPACK. A continuación, mostramos un ejemplo distribución de datos en memoria para una matriz A una matriz no simétrica de tamaño 7×7 en bandas donde `bwl=bwu=2`. La representación de la matriz A se puede expresar mediante

$$A = \left\{ \begin{array}{ccccccc} a_{11} & a_{11} & a_{13} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} \end{array} \right\}.$$

Entonces la distribución de los datos en memoria para matrices en banda con pivotación parcial, suponiendo que se ejecuta en tres procesos, se debe realizar como se presenta en la figura 8.2. Además, se observa el espacio que se debe reservar entre columnas identificado mediante "F" que debe tener un ancho (`bwl`+`bwl`). El número total de filas en esta distribución es $2 * (\texttt{bwl} + \texttt{bwl}) + 1$.

Procesos						
0			1			2
F	F	F	F	F	F	F
F	F	F	F	F	F	F
F	F	F	F	F	F	F
F	F	F	F	F	F	F
*	*	a_{13}	a_{24}	a_{35}	a_{46}	a_{57}
*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}
a_{11}	a_{22}	a_{33}	a_{44}	a_{45}	a_{66}	a_{77}
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	*
a_{31}	a_{42}	a_{53}	a_{64}	a_{75}	*	*

Figura 8.2: Distribución en memoria de una matriz A en bandas no simétrica con pivotación parcial

Las matrices en banda con este tipo de distribución vienen identificadas mediante el tipo de descriptor 501, por esta razón utilizaremos `ACTS_lib=501` en la llamada a `Num2PyACTS`.

- Matriz general en bandas (sencilla, sin pivotación) :

  `b,info=pvdbsv(a,b,[bwl=n-1,bwu=n-1])`

  Implementa la resolución de un sistema lineal de ecuaciones a partir de la matriz

PyACTS `a` de coeficientes en bandas y el vector PyACTS `b` de términos independientes. En este caso, la distribución de datos de la matriz `a` se debe realizar de acuerdo a la representación de los datos mostrados en la figura 8.3. Este tipo de distribución es la utilizada por ScaLAPACK (y por tanto, por PyScaLAPACK) en matrices en bandas y en rutinas en las que no se realiza pivotación y por tanto no se necesita el espacio de memoria adicional reservado en la pivotación parcial.

Procesos

0			1			2
$*$	$*$	a_{13}	a_{24}	a_{35}	a_{46}	a_{57}
$*$	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}
a_{11}	a_{22}	a_{33}	a_{44}	a_{45}	a_{66}	a_{77}
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	$*$
a_{31}	a_{42}	a_{53}	a_{64}	a_{75}	$*$	$*$

Figura 8.3: Distribución en memoria de una matriz A en bandas no simétrica sin pivotación

En resumen, se observa que la distribución en memoria de los datos es diferente con respecto a la pivotación parcial. Sin embargo, ambas matrices `a` tienen un mismo tipo de descriptor identificado mediante el valor de su primer elemento (`a.desc[0]=501`). Para indicar el tipo de matriz y la pivotación utilizada, hemos definido el parámetro opcional `piv`. Por tanto, si `ACTS_lib=501` indica que es una distribución de una matriz en bandas y por tanto debemos atender al valor de `piv` que guarda mucha relación con la tabla 4.7 del apartado 4.3.5:

- `piv='gb'`: Valor por defecto. Matriz general en bandas con pivotación parcial.
- `piv='db'`: General en bandas sin pivotación.
- `piv='dt'`: General en bandas tridiagonal.
- `piv='pb'`: En bandas simétrica o hermítica definida positiva.
- `piv='pt'`: Tridiagonal simétrica o hermítica definida positiva.

A continuación, mostramos un ejemplo de utilización de la rutina `pvdbsv` en la cual se convierte una matriz que realmente no es en bandas, a una distribución de bandas suponiendo que los elementos fuera de las diagonales principales son cero.

Esta conversión se realiza en la línea 12 y en ella se indica el ancho de las bandas superior e inferior así como el tipo de pivotación utilizada.

```
1  from PyACTS import *
2  import PyACTS.PyScaLAPACK as PySLK
3  from Numeric import *
4  n,nrhs=8,1
5  PyACTS.gridinit(npcol=PyACTS.nprocs,nprow=1)
6  if PyACTS.iread==1:
7          a=8*identity(n,Float)+ones([n,n],Float)
8          b=ones((n,nrhs))
9  else:
10         a,b=None,None
11 pivot,bw,ACTS_lib='db',2,501
12 a=Num2PyACTS(a,ACTS_lib,bwu=bw,bwl=bw,piv=pivot)
13 b=Num2PyACTS(b,502)
14 b,info= PySLK.pvdbsv(a,b,bwl=bw,bwu=bw)
15 b_num=PyACTS2Num(b)
16 if PyACTS.iread==1:
17         print "a*x=b --> x'=",transpose(b.data)
18         print "Info:",info
19 PyACTS.gridexit()
```

Por otro lado, se observa en la línea 13 que la distribución de los términos independientes se realiza mediante un indicador de librería diferente al habitual (`ACTS_lib=502`). Este indicador corresponde a la distribución de matrices denominada *right-hand side vector* utilizada para distribuir los datos de los vectores de términos independientes para sistemas en bandas.

En resumen, se observa en este ejemplo que las rutinas de inicialización y conversión utilizadas en el caso de utilizar matrices en bandas son las mismas que para matrices generales que utilizan una distribución cíclica 2D. Esto contribuye a una mayor simplificación de las librerías y una mejor compresión y fácil utilización por parte del usuario de PyACTS.

- Matriz general tridiagonal (sencilla, sin pivotación) :

  `b,info=pvdtsv(a,b)`

  Implementa la resolución de un sistema lineal de ecuaciones a partir de la matriz

PyACTS `a` de coeficientes tridiagonal y el vector PyACTS `b` de términos independientes. Esta rutina está implementada para matrices tridiagonales y como se puede observar en el manual de usuario de ScaLAPACK para las rutinas `PvDTyyy`, la matriz A se suele convertir a un trio de vectores `d,dl,du` que representan la diagonal principal, la primera diagonal inferior y la primera diagonal superior respectivamente. El resto de elementos son nulos, ya que estamos trabajando con una matriz tridiagonal. Buscando como objetivo la sencillez, mediante el comando `a=Num2PyACTS(a,501,piv='dt')` estaremos convirtiendo nuestra matriz `Numeric` al formato definido en ScaLAPACK mediante estos tres vectores. Para ocultar al usuario dicha conversión, se almacenan los tres vectores como un diccionario dentro del campo de datos. A continuación mostramos un ejemplo de este tipo de conversión en PyScaLAPACK. Si suponemos que se ha inicializado la malla en un único proceso y se han importado los módulos correspondientes, ejecutaremos:

```
1 a=8*identity(n,Float)+ones([n,n],Float)
2 print "a=",a
3 a=Num2PyACTS(a,501,piv='dt')
4 print "dl=",a.data[0]
5 print "d=",a.data[1]
6 print "du=",a.data[2]
```

En este ejemplo, se define la matriz `a` y se distribuye conforme a la distribución por bandas (`ACTS_lib=501`) para matrices tridiagonales (`piv='dt'`). El resultado de la ejecución de este ejemplo es:

```
a=[[ 9.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.]]
dl= [ 1.  1.  1.  1.  1.  1.]
d= [ 9.  9.  9.  9.  9.  9.  9.]
du= [ 1.  1.  1.  1.  1.  1.]
```

De este modo, mediante la conversión al formato utilizado en ScaLAPACK para

matrices tridiagonales, y de forma prácticamente transparente al usuario, podremos ejecutar la rutina `pvdtsv` de una forma similar al resto de rutinas de PyScaLAPACK:

```
1 ...
2 a=Num2PyACTS(a,ACTS_lib,piv=pivot)
3 b=Num2PyACTS(b,502)
4 print "dl=",a.data[0]
5 print "d=",a.data[1]
6 print "du=",a.data[2]
7 ...
8 b,info= PySLK.pvdtsv(a,b)
9 ...
```

- Matriz simétrica o hermítica definida positiva.

  - Sencilla: `b,info=pvposv(a,b,[uplo='U'])`

    Implementa la resolución de un sistema lineal de ecuaciones a partir de la matriz PyACTS `a` simétrica (o hermítica en el caso complejo), y la matriz PyACTS `b` de términos independientes y dimensiones $n \times nrhs$. Este tipo de rutinas utilizan la distribución cíclica 2D por lo que `ACTS_lib=1`. Por otro lado, como se trata de matrices simétricas o hermíticas debemos indicar mediante el parámetro `uplo` en qué parte de la diagonal deseamos trabajar para la lectura de los datos. Por defecto, PyScaLAPACK configura la matriz parte superior a la diagonal principal (`uplo='U'`) de la matriz simétrica (o hermítica en el caso complejo). Si se desea utilizar la parte inferior a la diagonal principal, se debe indicar mediante `uplo='L'`.

  - Avanzada: `a,af,equed,sr,sc,b,x,rcond,ferr,berr,info=`
    `pvposvx(a,b,[fact='N',uplo='U',equed='N'])`

    Esta rutina permite obtener mayor información de la factorización realizada (ver apartado 4.3.5), los errores de redondeo y el número de condición. Debido a que se trata de una rutina avanzada y para evitar extendernos en la longitud de este apartado, nos remitimos al manual de usuario de PyScaLAPACK [34] para obtener más información de los parámetros opcionales y de salida de esta rutina.

- Matriz en bandas simétrica o hermítica definida positiva (sencilla).
  `b,info=pvpbsv(a,b[,uplo,bw])`

En este caso, la matriz `a` es una matriz en bandas simétrica y definida positiva por lo que deberemos utilizar la conversión de matrices `Numeric` a matrices PyACTS en bandas implementada en PyScaLAPACK. A continuación se muestran las líneas más destacadas de un ejemplo que hace uso de esta rutina.

```
1 ...
2 a=Num2PyACTS(a,501,bwu=1,bwl=1,piv='pb')
3 b=Num2PyACTS(b,502)
4 ...
5 b,info= PySLK.pvpbsv(a,b,uplo='U',bw=1)
6 b_num=PyACTS2Num(b)
7 ...
8 PyACTS.gridexit()
```

Observamos en la línea 2, la indicación mediante `ACTS_lib=501` que se trata de una matriz en bandas y mediante el parámetro `piv='pb'` que estamos ante una matriz simétrica o hermítica definida positiva.

- Matriz tridiagonal simétrica o hermítica definida positiva (sencilla).

  `b,info=pvptsv(a,b)`

  La matriz `a` es una matriz tridiagonal simétrica y definida positiva por lo que deberemos utilizar la conversión de matrices `Numeric` a matrices PyACTS en bandas implementada en PyScaLAPACK. En el caso de las matrices tridiagonales vistas anteriormente `pvdtsv`, se convertía la matriz `Numeric` a las tres diagonales principales. En este caso, como la matriz es simétrica o hermítica (`dl=du`), únicamente obtendremos dos vectores: `d,dl`. El vector `dl` se corresponde con la diagonal superior de la matriz `Numeric`. En este caso, se debe realizar una conversión del tipo `a=Num2PyACTS(a,501,piv='pt')`, obteniendo una matriz PyACTS `a`, cuya propiedad `data` contiene realmente los vectores `d` y `dl`.

8.2.2. Problemas de mínimos cuadrados

El segundo grupo funcional definido en PyScaLAPACK resuelve un sistema mediante mínimos cuadrados. Estas rutinas también harán uso de matrices PyACTS y por tanto deberemos convertir o leer los datos a dicho formato. Sin embargo, a diferencia del

apartado anterior, el número de rutinas incluidas en este grupo funcional es mucho más reducido. De hecho sólo tenemos una rutina para matrices generales y en su interfaz más sencillo. A continuación explicamos brevemente la interfaz de dicha rutina:

- `b,info=pvgels(a,b,[trans='N'])`:
  Se corresponde con la rutina `PvGELS` de ScaLAPACK vista en el apartado 4.3.5, donde `a` y `b` son matrices PyACTS y el resultado es la misma matriz PyACTS `b` pero que contiene el vector solución x.

La forma de realizar una llamada a la rutina `pvgels` es muy similar a la mostrada en el ejemplo de `pvgesv` mostrado en las rutinas de resolución de ecuaciones lineales (apartado 8.2.1). La diferencia fundamental con respecto a ese código sería que en la línea 18, realizaríamos la llamada a la rutina `pvgels` en lugar de la rutina `pvgesv` mediante una instrucción del tipo `b,info= PySLK.pvgels(a,b)`.

Sin embargo, en la rutina `pvgels` ocurre una diferencia notable con respecto a la rutina `pvgesv`. A continuación, mostramos la interfaz definida para Fortran de la rutina `PDGELS` a la cual hemos desarrollado una interfaz a Python llamada `pvgels`.

```
1   SUBROUTINE PDGELS( TRANS, M, N, NRHS, A, IA, JA, DESCA, B, IB, JB,
2  $                   DESCB, WORK, LWORK, INFO )
```

Como se observa en esta interfaz y comparándola con la interfaz definida para `pvgels`, muchos de los parámetros que deberemos pasar a `PDGELS` podemos obtenerlos de los parámetros opcionales y de las matrices PyACTS:

- `TRANS←[trans='N']`
- `A←a.data`
- `IA←[ia=1]`
- `JA←[ja=1]`
- `DESCA←a.desc`
- `B←b.data`
- `IB←[ib=1]`
- `JB←[jb=1]`
- `DESCB←b.desc`

Los parámetros `ia, ja, ib` y `jb` no se han mostrado en la interfaz de `pvgels`, sin embargo por defecto estos parámetros siempre valen uno y puede modificarse su valor

indicándolo en la llamada a la rutina (por ejemplo, `pvgels(a,b,ia=10,ja=10)`). Esta pareja de parámetros se define para cada matriz utilizada en todas las rutinas, tanto en ScaLAPACK como en BLACS y PBLAS [14].

Sin embargo, además de las variables opcionales descritas, se observa en la interfaz de `PDGELS` la aparición de dos parámetros nuevos `LWORK` y `WORK`. Éstos se corresponden con la longitud del espacio de memoria necesario y con la dirección del espacio de memoria. De este modo, para muchas de las rutinas de PyScaLAPACK que presentamos en este capítulo necesitaremos calcular y reservar el espacio de memoria de trabajo necesario para cada rutina, puesto que éste varía en función de la rutina a utilizar.

En la mayor parte de las rutinas de ScaLAPACK, el tamaño del espacio de memoria de trabajo necesario se obtiene realizando una llamada previa a dicha rutina con el parámetro `LWORK=-1`. La propia rutina nos informa del tamaño necesario mediante la primera posición del vector `WORK`. En esta primera llamada únicamente es necesario reservar un elemento de coma flotante con doble precisión para obtener el tamaño de memoria de trabajo. En este punto, a partir de `LWORK=WORK(0)` reservamos la memoria indicada en `LWORK` y llamamos de nuevo a la rutina `PDGELS` con el tamaño de memoria `LWORK` adecuado, y con espacio suficiente en la memoria de trabajo.

Estas operaciones las debería realizar un programador de Fortran que desee hacer uso de algunas de las rutinas de ScaLAPACK. Sin embargo, estos detalles se ocultan al usuario de PyScaLAPACK para simplificarle la llamada a `pvgels`, y de este modo únicamente sea necesario pasar las matrices PyACTS `a` y `b` para resolver el sistema mediante mínimos cuadrados. Por tanto, en el módulo `PyScaLAPACK.py`, donde se definen las interfaces de cada rutina, en la definición de la rutina de `pvgels` realizaremos la reserva del espacio de memoria de trabajo.

A continuación, adjuntamos el código Python en el que se realiza la definición de la rutina `pvgels`. La estructura de este código ha sido desarrollada en el apartado 5.6 y comentaremos únicamente las acciones realizadas para la gestión de la memoria de trabajo.

```
1
2 def  pvgels(ACTS_a,ACTS_b,trans='N',ia=1,ja=1,ib=1,jb=1):
3 try:
4   ACTS_vars_in=[ACTS_a,ACTS_b]
5   ACTS_inc_in=[ia,ja,ib,jb]
6   ACTS_Error=PySLK_tools.checkerrors(ACTS_vars_in,ACTS_inc_in)
```

```
7   if ACTS_Error<>"":
8    raise PySLKError(ACTS_Error)
9
10   PySLKtype= PySLK_tools.getcommontype(
11              [ACTS_a.data.typecode(), ACTS_b.data.typecode()])
12
13  if PySLKtype=='s':
14   PvScaLAPACK=pyscalapack.psgels
15  elif PySLKtype=='d':
16   PvScaLAPACK=pyscalapack.pdgels
17  elif PySLKtype=='c':
18   PvScaLAPACK=pyscalapack.pcgels
19  elif PySLKtype=='z':
20   PvScaLAPACK=pyscalapack.pcgels
21
22  lwork=-1
23  work=Numeric.zeros((10,1),ACTS_a.data.typecode())
24  ACTS_a.data,ACTS_b.data,work,info=PvScaLAPACK(trans, m, n, nrhs,
25          ACTS_a.data, ia, ja, ACTS_a.desc, ACTS_b.data, ib, jb,
26          ACTS_b.desc, work,lwork,info)
27
28  if PySLKtype=='s' or PySLKtype=='d':
29   lwork=work[0,0]
30  elif PySLKtype=='c' or PySLKtype=='z':
31   lwork=work[0,0].real
32
33  work=Numeric.zeros((int(lwork),1),ACTS_a.data.typecode())
34  ACTS_a.data,ACTS_b.data,work,info=PvScaLAPACK(trans, m, n, nrhs,
35          ACTS_a.data, ia, ja, ACTS_a.desc, ACTS_b.data, ib, jb,
36          ACTS_b.desc, work,lwork,info)
37
38 except PySLKError,Error:
39                              ACTS_b.data=[]
40
41 return ACTS_b,info
```

En la línea 24 se observa la primera llamada a la rutina de ScaLAPACK donde previamente se ha asignado `lwork=-1`. La longitud adecuada del espacio de memoria se obtiene

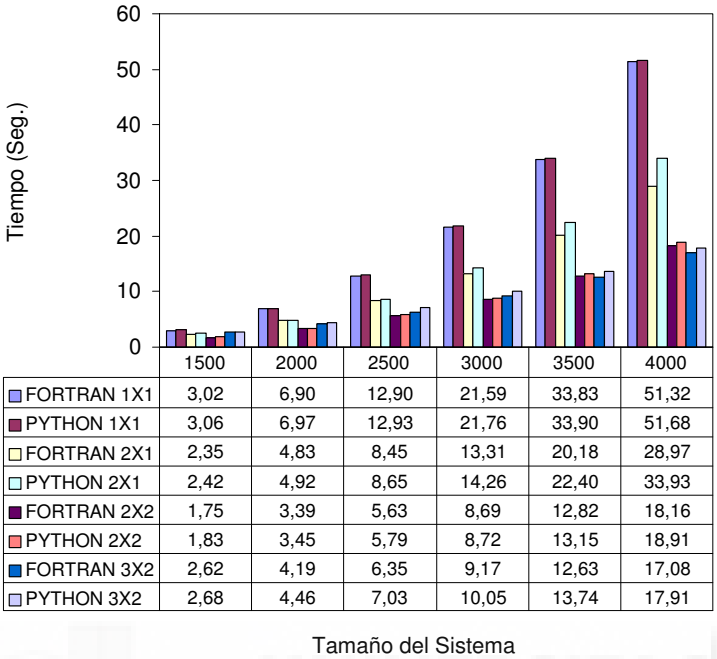
en el vector `work` (líneas 28 a 31) y se genera un nuevo vector `Numpy` con el tamaño indicado en la primera llamada. En este ejemplo, se realiza una sola reserva de memoria de trabajo de la rutina. Sin embargo, en otras rutinas es necesario dos espacios de memoria de trabajo además de espacios de memoria para implementar la pivotación u otros algoritmos intermedios. Todas estas acciones previas a la propia ejecución de la rutina hacen más compleja la utilización de ScaLAPACK. No obstante, mediante PyScaLAPACK se ocultan estos detalles al usuario simplificando en gran medida la complejidad de la aplicación.

En la figura 8.4 mostramos los tiempos de ejecución empleados para resolver el sistema con las mismas matrices A y B utilizadas para `pvgesv`. En este caso, se observa el mismo comportamiento, es decir, los tiempos de ejecución en Fortran y Python son prácticamente similares en ambas plataformas (Cluster-umh y Seaborg). Además, a pesar de resolver el mismo sistema de ecuaciones se observa que los tiempos de ejecución `pvgels` son superiores a `pvgesv` en ambos lenguajes. Esto nos demuestra que el tiempo de cómputo se consume fundamentalmente en el algoritmo implementado en las librerías, independientemente del lenguaje utilizado por el programador en niveles superiores. Sin embargo, esta afirmación se cumple siempre que los datos de las matrices PyACTS se encuentren dispuestos en memoria en el orden adecuado (por ejemplo, continuidad en memoria) y no se necesite convertir tipos de datos o realizar modificaciones en los mismos que incrementen los tiempos totales de ejecución.

8.2.3. Problemas de valores singulares y valores propios

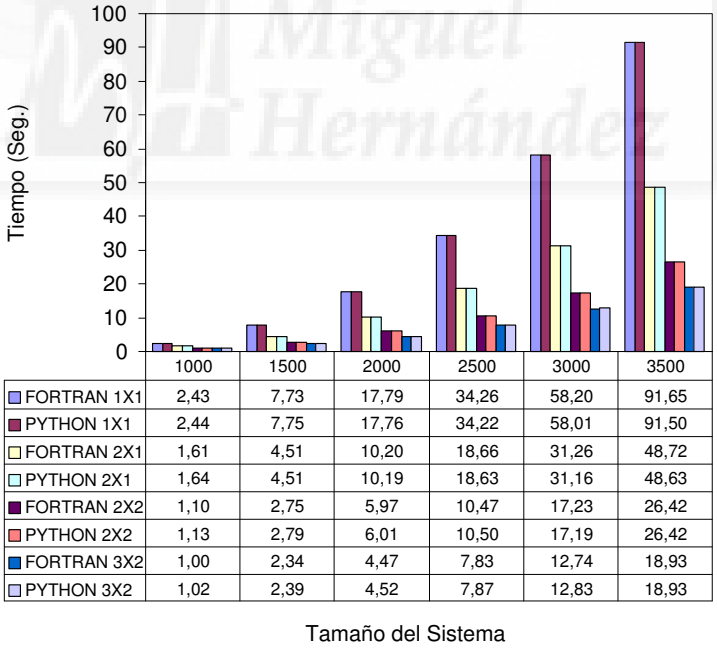
El tercer grupo funcional incluido en el módulo PyScaLAPACK obtiene los valores y vectores propios y la descomposición en valores singulares de una matriz introducida como parámetro. El número de rutinas en este grupo es reducido, y guarda estrecha relación con las rutinas mostradas en la tabla 4.10 del apartado 4.3.5. A continuación se detallan estas rutinas indicando su funcionalidad básica, no obstante, creemos conveniente recordar que para obtener más información de cada uno de los parámetros de entrada y salida, así como ejemplos de utilización de cada rutina, es recomendable la lectura del manual de usuario de PyACTS [34].

- Matriz simétrica.



	1500	2000	2500	3000	3500	4000
FORTRAN 1X1	3,02	6,90	12,90	21,59	33,83	51,32
PYTHON 1X1	3,06	6,97	12,93	21,76	33,90	51,68
FORTRAN 2X1	2,35	4,83	8,45	13,31	20,18	28,97
PYTHON 2X1	2,42	4,92	8,65	14,26	22,40	33,93
FORTRAN 2X2	1,75	3,39	5,63	8,69	12,82	18,16
PYTHON 2X2	1,83	3,45	5,79	8,72	13,15	18,91
FORTRAN 3X2	2,62	4,19	6,35	9,17	12,63	17,08
PYTHON 3X2	2,68	4,46	7,03	10,05	13,74	17,91

(a) Cluster-umh



	1000	1500	2000	2500	3000	3500
FORTRAN 1X1	2,43	7,73	17,79	34,26	58,20	91,65
PYTHON 1X1	2,44	7,75	17,76	34,22	58,01	91,50
FORTRAN 2X1	1,61	4,51	10,20	18,66	31,26	48,72
PYTHON 2X1	1,64	4,51	10,19	18,63	31,16	48,63
FORTRAN 2X2	1,10	2,75	5,97	10,47	17,23	26,42
PYTHON 2X2	1,13	2,79	6,01	10,50	17,19	26,42
FORTRAN 3X2	1,00	2,34	4,47	7,83	12,74	18,93
PYTHON 3X2	1,02	2,39	4,52	7,87	12,83	18,93

(b) Seaborg

Figura 8.4: Comparativa de `pvgels` en lenguaje Fortran vs Python

- Sencilla `w,z,info= pvsyev(a[,jobz='N',uplo='U'])`

  En esta rutina se obtienen todos los valores propios `w` y opcionalmente los vectores propios `z` de una matriz A real simétrica. Esta rutina tiene dos parámetros opcionales, mediante el primero de ellos (`jobz`) indicaremos si se desean calcular los vectores propios de la matriz. Por defecto `jobz='N'`, por lo que no se calculan los vectores propios y `z` es una matriz PyACTS de ceros; sin embargo, indicando `jobz='V'` se obtienen los vectores propios en `z`. El parámetro opcional `uplo` ya ha sido explicado para las rutinas simétricas o las hermíticas. El parámetro `info` de salida, como en el resto de rutinas vistas, informa del resultado de la ejecución.

- Avanzada: `m,nz,w,z,ifail,iclustr,gap,info=pvsyevx(a`
  `[,jobz='N',range='A',uplo='U',orfac=0,rcond=0,`
  `vl=None,vu=None,il=None,iu=None, abstol=0])`

  Obtiene los valores propios y, opcionalmente, los vectores propios de una matriz simétrica (caso real) o hermítica (caso complejo) A. Un detalle importante que deseamos destacar es el siguiente: según se observa en la tabla 4.11, en ScaLAPACK existen un par de rutinas para el caso real (`PSSYEVX` y `PDSYEVX`) y otro par para el caso complejo (`PCHEEVX` y `PZHEEVX`). La funcionalidad de las cuatro rutinas es la misma pero el nombre varía por el tipo de dato utilizado, y por que, en el caso real, tendremos matrices simétricas (de ahí el "`SY`") y en el caso complejo, matrices hermíticas ("`HE`"). Sin embargo, en PyScaLAPACK no se necesita indicar el tipo de datos en el nombre de la rutina, por tanto, hemos tomado como solución que si el usuario de PySCALAPACK llama a la rutina `PyScaLAPACK.pvsyevx(a)` siendo `a` una matriz PyACTS de elementos complejos (por ejemplo de doble precisión), de forma interna, en lugar de llamar a `PDSYEVX` llamaremos a la rutina `PZHEEVX`.

  Por otro lado, también se proporciona en PyScaLAPACK la rutina `pvheevx` cuyos parámetros de entrada y salida son similares a `pvsyevx`. Del mismo modo, si llamamos a la rutina `pvheevx` mediante datos de tipo real, de forma interna se ejecutarán las rutinas de ScaLAPACK para el caso real.

  Este comportamiento de PyScaLAPACK, se repite para todas aquellas rutinas cuya funcionalidad entre el caso real y el complejo sea la misma pero que difieren en el nombre de la rutina en algo más que en la letra que indica el tipo de dato. Por ejemplo, también sucede con las matrices ortogonales en el caso real (`'OR'` en ScaLAPACK) y unitarias (`'UN'`) para complejos.

- Matriz general.

  - Sencilla: `s,u,vt,info= pvgesvd(a[,jobu='N',jobvt='N'])`:
    Obtiene la descomposición en valores singulares, variable `s` de tipo `Numeric`, de una matriz `a` general y calcula de forma opcional los vectores propios tanto a la izquierda `u` como a la derecha `v` almacenados en matrices PyACTS, si así se configura mediante los parámetros opcionales `jobu` y `jobvt`, respectivamente. Cabe destacar que el parámetro `s` es un vector de tipo `Numeric` cuyos datos son los mismos para todos los procesos que han intervenido en el cálculo, por tanto cada proceso almacena todos los valores singulares en dicho vector, es decir, los valores singulares no están distribuidos sino que todos los procesos poseen dichos valores singulares.

La utilización de las rutinas en este grupo funcional es muy similar a las vistas en grupos anteriores. A continuación, se expone un ejemplo de utilización de la descomposición en valores singulares mediante la rutina `pvgesvd` de una matriz muy sencilla generada previamente. Se observa en este ejemplo la sencillez en la llamada a la rutina de PyScaLAPACK ocultando al usuario conceptos de distribución de datos, descriptores, reserva de espacios de memoria de trabajo y otros detalles que son necesarios conocer cuando un programador implementa una aplicación desde lenguajes más tradicionales como Fortran.

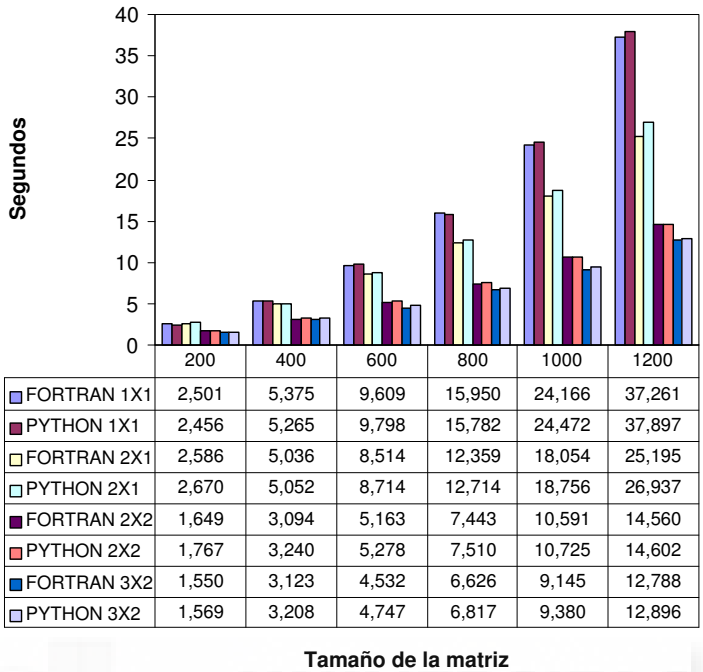
```
1  from PyACTS import *
2  import PyACTS.PyScaLAPACK as PySLK
3  from Numeric import *
4  n=8
5  #Inicializa la malla
6  PyACTS.gridinit()
7  if PyACTS.iread==1:
8      print "N=",n,";"
9      print PyACTS.nprow,"x",PyACTS.npcol
10     print "NBxMB:",PyACTS.mb,"*",PyACTS.nb
11     a=8*identity(n,Float)
12     print "a=",a
13 else:
14     a=None
15 ACTS_lib=1 # 1=Scalapack
16 a=Num2PyACTS(a,ACTS_lib)
```

```
17 #Llamamos a la rutina de PyScaLAPACK
18 s,u,vt,info= PySLK.pvgesvd(a,jobu='V',jobvt='V')
19 u_num=PyACTS2Num(u)
20 vt_num=PyACTS2Num(vt)
21 if PyACTS.iread==1:
22     print "s'=",transpose(s)
23     print "u=",u_num
24     print "vt=",vt_num
25 PyACTS.gridexit()
```

Este ejemplo tiene muchas similitudes con los ejemplos mostrados para otras rutinas de PyScaLAPACK, así como otras rutinas del módulo PyBLACS y PyPBLAS. Como particularidad de esta rutina, se puede observar cómo en la línea 18 se realiza la llamada a la rutina de PyScaLAPACK. En esta llamada se indica el valor de dos parámetros opcionales (`jobu` y `jobvt`) que indican si se ha de calcular las matrices U y V^T respectivamente. Además se observa que las matrices de salida no tienen relación con las matrices de entrada, es decir, el parámetro de entrada es la matriz `a` mientras que las variables de salida serán los valores singulares y las matrices de vectores singulares de `a`. De este modo, estos datos (y especialmente estos espacios de memoria) se generan de forma interna a la rutina, y los datos de la matriz `a` no son modificados por la rutina `pvgesvd`.

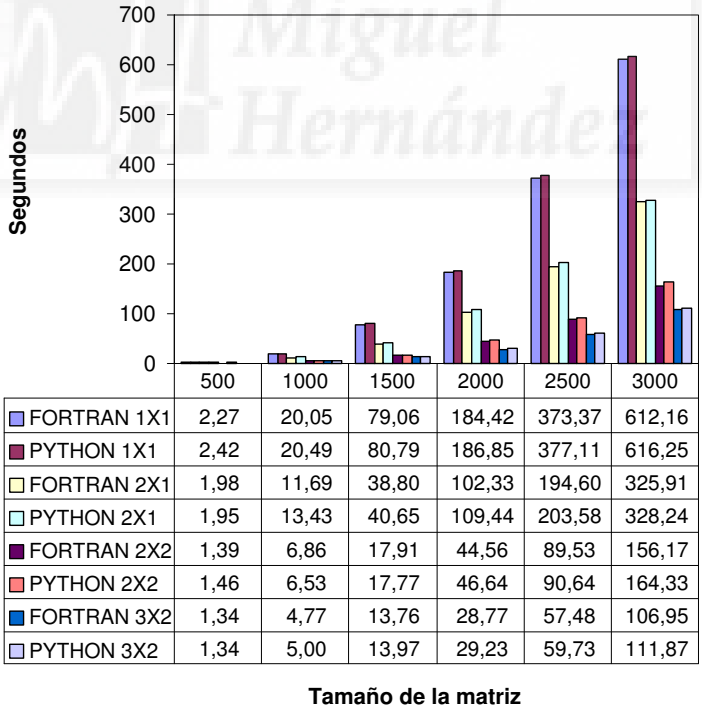
Una diferencia considerable en la utilización de esta rutina con respecto a la interfaz clásica en Fortran es que el usuario no se debe preocupar de calcular los tamaños de las matrices Σ, U y V^T y por tanto, no ha de reservar espacio de memoria para las mismas así como para la matriz de trabajo temporal (denominada `WORK` en la interfaz de Fortran).

En la figura 8.5 se muestra una comparativa de los tiempos de ejecución para la descomposición en valores singulares mediante la rutina `pvgesvd` en Python y `PDGESVD` en Fortran. Los resultados se muestran para dos plataformas diferentes; por un lado en el Cluster-umh (figura 8.5(a)) se observan tiempos de ejecución muy parecidos entre ambos entornos de programación. Del mismo modo, los resultados obtenidos en Seaborg (figura 8.5(b)) nos demuestran que la penalización sufrida por utilizar lenguajes de alto nivel como PyScaLAPACK es muy pequeña independientemente de la arquitectura paralela que estemos utilizando.



	200	400	600	800	1000	1200
FORTRAN 1X1	2,501	5,375	9,609	15,950	24,166	37,261
PYTHON 1X1	2,456	5,265	9,798	15,782	24,472	37,897
FORTRAN 2X1	2,586	5,036	8,514	12,359	18,054	25,195
PYTHON 2X1	2,670	5,052	8,714	12,714	18,756	26,937
FORTRAN 2X2	1,649	3,094	5,163	7,443	10,591	14,560
PYTHON 2X2	1,767	3,240	5,278	7,510	10,725	14,602
FORTRAN 3X2	1,550	3,123	4,532	6,626	9,145	12,788
PYTHON 3X2	1,569	3,208	4,747	6,817	9,380	12,896

(a) Cluster-umh



	500	1000	1500	2000	2500	3000
FORTRAN 1X1	2,27	20,05	79,06	184,42	373,37	612,16
PYTHON 1X1	2,42	20,49	80,79	186,85	377,11	616,25
FORTRAN 2X1	1,98	11,69	38,80	102,33	194,60	325,91
PYTHON 2X1	1,95	13,43	40,65	109,44	203,58	328,24
FORTRAN 2X2	1,39	6,86	17,91	44,56	89,53	156,17
PYTHON 2X2	1,46	6,53	17,77	46,64	90,64	164,33
FORTRAN 3X2	1,34	4,77	13,76	28,77	57,48	106,95
PYTHON 3X2	1,34	5,00	13,97	29,23	59,73	111,87

(b) Seaborg

Figura 8.5: Comparativa de `pvgesvd` en lenguaje Fortran vs Python

8.2.4. Problemas de valores propios en matrices simétricas definidas

En este cuarto grupo funcional únicamente se proporciona en PyScaLAPACK la rutina que se muestra a continuación y que además se puede clasificar como avanzada.

- Avanzada: `m,nz,w,z,ifail,iclustr,gap,info=pvsygvx(a,b,ibtype,`
  `[jobz='N',range='A',uplo='U',orfac=0,rcond=0,`
  `vl=None,vu=None,il=None,iu=None, abstol=0])`

  Dependiendo del tipo de dato que contengan las matrices PyACTS, esta rutina realizará la llamada a la correspondiente rutina de ScaLAPACK (`PSSYGVX`, `PCHEGVX`, `PDSYGVX` o `PZHEGVX`, dependiendo si el tipo de dato es S, C, D o Z). Como se trata de una rutina avanzada y no deseamos extendernos explicando sus parámetros, nos remitimos al manual de referencia de PyACTS.

En la utilización de estas rutinas se ha tratado de minimizar el número de parámetros de entrada, definiendo muchos de ellos como opcionales. A continuación, mostramos la parte del código en la que se realizaría la llamada a esta rutina. El código expresado con puntos suspensivos indica que las operaciones son las mismas que en el ejemplo del apartado anterior.

```
1  ...
2  if PyACTS.iread==1:
3       a=8*identity(n,Float)
4       b=0.5 * identity(n,Float)
5       else:
6       a,b=None,None
7  ACTS_lib=1
8  a=Num2PyACTS(a,ACTS_lib)
9  b=Num2PyACTS(b,ACTS_lib)
10 m,nz,w,z,ifail,iclustr,gap,info= PySLK.pvsygvx(a,b,1)
11 ...
12 PyACTS.gridexit()
```

8.3. Rutinas computacionales

Las rutinas rutinas driver que hemos visto hasta el momento, realmente son una secuencia de llamadas a rutinas computacionales. Serán las rutinas computacionales, las enfocadas para realizar una tarea especifica y la concatenación de estás tareas obtiene la resolución de un problema completo. En PyScaLAPACK también se han incluido las rutinas computacionales definidas en ScaLAPACK, de este modo, si un usuario simplemente desea obtener una factorización de una matriz u otras tareas que se han detallado en el apartado 4.3.5, podrá hacer uso de las rutinas que se exponen a continuación.

Debemos destacar, que en la distribución de ScaLAPACK no se adjuntan rutinas computacionales avanzadas (acabadas en "`X`"), por lo que no se detallan tampoco rutinas avanzadas para PyScaLAPACK.

8.3.1. Ecuaciones lineales

En la tabla 4.12, se ha presentado el conjunto de rutinas computacionales (`PxyyTRF`, `PxyyTRS`, `PxyyCON`, `PxyyRFS`, `PxyyTRI`, y `PxyyEQU`) relacionadas con la resolución de sistemas de ecuaciones lineales y la funcionalidad de cada una de las rutinas. Los nombres de estas rutinas varían en función del tipo de dato de la matriz `x`, y del esquema de almacenamiento utilizado para la matriz `yy`. En PyScaLAPACK, el tipo de dato de los elementos de la matriz no es necesario especificarlo, por lo que en todas ellas `x`="`v`". Sin embargo, el esquema de almacenamiento que se realiza sí se debe indicar e incluso afecta a las rutinas auxiliares tal como ya se ha introducido en el apartado 8.2.1 mediante las figuras 8.2 y 8.3.

Del mismo modo que se han introducido las rutinas driver para ecuaciones lineales, clasificadas según el esquema de almacenamiento, a continuación mostramos las rutinas computacionales atendiendo a la misma clasificación.

- Matriz general (pivotación parcial).

```
af,info= pvgetrf(a)
b,info=pvgetrs(af,b)
```

```
rcond,info=pvgecon(af,[,anorm=1])
x,ferr,berr,info= pvgerfs(a,af,b,x[,trans='N'])
a,info= pvgetri(af)
r,c,rowcnd,colcnd,amax,info=pvgeequ(a)
```

En este tipo de rutinas, `a` y `b` son matrices PyACTS distribuidas de forma general, es decir, con distribución cíclica 2D entre los procesos de la malla. Del mismo modo que en las rutinas driver, `a` contiene la matriz de coeficientes y `b` la matriz de términos independientes.

La ejecución de la rutina `pvgetrf` realiza la factorización de la matriz `a`, devuelta como `af`, sin embargo, recordamos que la matriz `a` y la matriz `af` son la misma puesto que apuntan al mismo espacio de memoria en el que residen los datos. A continuación, dicha matriz factorizada se utilizará en `pvgetrs` para resolver el sistema.

La rutina `pvgecon` obtiene el número de condición `rcond` a partir de la matriz factorizada `af` devuelta por `pvgetrf`. El parámetro opcional `anorm` indica si la norma utilizada es la norma 1 (`anorm='1'`) o bien se utiliza una norma infinita donde `anorm='I'`.

Por otro lado, la rutina `pvgerfs` obtiene la solución `x` al sistema y también limita el error producido devuelto mediante las variables PyACTS `ferr` y `berr`. Se observa que en esta rutina debemos utilizar como parámetro de entrada, la matriz PyACTS `a` y la forma factorizada `af`. El parámetro opcional de entrada `trans` indica el aspecto del sistema de ecuaciones, de este modo en su valor por defecto `trans='N'` el sistema es $Ax = B$, si `trans='T'` el sistema es $A^T x = B$ y si `trans='C'` el sistema es $A^H x = B$.

Para obtener la inversa de una matriz PyACTS `a`, utilizaremos la rutina `pvgetri` donde la matriz `af` ha sido obtenida mediante ejecución previa de la rutina `pvgetrf`.

Por ultimo, si deseamos obtener los factores de escalados en filas `r` y columnas `c` para equilibrar la matriz `a`, haremos uso la rutina `pvgeequ`. Mediante los parámetros `rowcnd` y `colcnd` se nos informa de la mayor variación entre los factores de escalados para filas y para columnas, respectivamente, y mediante `amax` se devuelve el elemento de mayor valor absoluto de la matriz.

La distribución de PyACTS incluye ejemplos para todas las rutinas mostradas

en este trabajo incluyendo las rutinas driver y las computacionales. Para evitar extendernos en este trabajo, presentamos a continuación un ejemplo en el que se utilizan algunas de las rutinas descritas.

```
1  from PyACTS import *
2  import PyACTS.PyScaLAPACK as PySLK
3  from Numeric import *
4  n,nrhs=8,1
5  PyACTS.gridinit(nb=2)
6  if PyACTS.iread==1:
7      print "Ejemplo de Utilizacion ScaLAPACK: PvGERFS"
8      print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
9      print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
10     a=8*identity(n,Float)+ones([n,n],Float)
11     print "a=",a
12     af=8*identity(n,Float)+ones([n,n],Float)
13     b=ones((n,nrhs))
14     print "b'=",transpose(b)
15     x=ones((n,nrhs))
16     print "x'=",transpose(x)
17 else:
18     a,af,b,x=None,None,None,None
19 #We convert Numeric Array to PyACTS.Scalapack Array
20 ACTS_lib=1 # 1=Scalapack
21 a=Num2PyACTS(a,ACTS_lib)
22 af=Num2PyACTS(af,ACTS_lib)
23 b=Num2PyACTS(b,ACTS_lib)
24 x=Num2PyACTS(x,ACTS_lib)
25 #We call PBLAS routine
26 af,info= PySLK.pvgetrf(af)
27 x,ferr,berr,info= PySLK.pvgerfs(a,af,b,x)
28 x=PyACTS2Num(x)
29 if PyACTS.iread==1:
30     print "Solucion:"
31     print "x:"transpose(x)
32     print "ferr:",transpose(ferr)
33     print "berr:",transpose(berr)
34     print "Info:",info
```

35 PyACTS.gridexit()

Como se observa en este ejemplo, primero se realiza la factorización mediante la rutina `pvgetrf` y el resultado se almacena en la variable `af`. Se observa en las líneas 22 y 23, que el nodo con `PyACTS.iread=1` define la matriz `a` y la matriz `af` con los mismos valores, o sea, son la misma matriz. Esto se debe a que la rutina `pvgetrf` almacena la factorización `af` en la dirección de memoria de la matriz de entrada, por lo que si utilizamos `a`, sus datos se sobreescriben. En la línea 27, a partir de la matriz factorizada `af` y de las matrices del sistema de ecuaciones se obtiene la solución y los errores máximos producidos.

El resultado de la ejecución del código mostrado en dos nodos (2×1) con un tamaño de bloque reducido `nb=mb=2` se ilustra a continuación.

```
vgaliano$ mpirun -np 2 mpipython exPyScapvgerfs.py
Ejemplo de Utilizacion ScaLAPACK: PvGERFS
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
b'= [ [1 1 1 1 1 1 1 1]]
x'= [ [1 1 1 1 1 1 1 1]]
Solucion:
x:
[[0.0625  0.0625  0.0625  0.0625  0.0625  0.0625  0.0625  0.0625]]
ferr:
[[9.92261828e-15   0.00000000e+00   0.00000000e+00   0.00000000e+00
  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]]
berr:
[[9.43689571e-16   0.00000000e+00   0.00000000e+00   0.00000000e+00
  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]]
```

```
Info: 0
```

Del mismo modo que se ha realizado con las rutinas driver, hemos realizado una serie de pruebas para comprobar el funcionamiento de todas las rutinas computacionales para ecuaciones lineales. Para evitar extendernos en la longitud de esta sección, mostraremos las pruebas realizadas para una función de cada grupo de rutinas computacionales. En la figura 8.6 se representan los tiempos necesarios para factorizar una matriz A cuadrada del tamaño indicado en cada columna y utilizando la configuración de la malla de procesos indicada en cada fila para los dos entornos de programación Fortran y Python que deseamos comparar. La rutina que se ha utilizado ha sido `pvgetrf` que implementa la factorización $A = PLU$ de la matriz PyACTS `a` y se han realizado estas pruebas en dos plataformas diferentes, en el Cluster-umh (figura 8.6(a)) y en Seaborg (figura 8.6(b)).

- Matriz general en bandas (pivotación parcial).

```
af,info= pvgbtrf(a)
b,info=pvgbtrs(af,b)
```

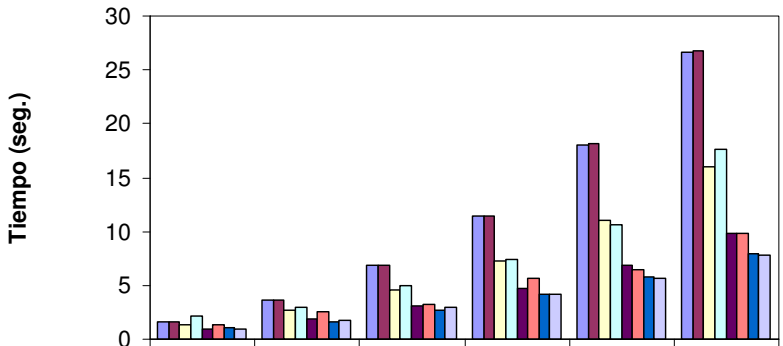
- Matriz general en bandas (sin pivotación).

```
af,info= pvdbtrf(a)
b,info=pvdbtrs(af,b)
```

Si se desea obtener la factorización LU de una matriz en bandas sin realizar pivotación y posteriormente resolver el sistema de ecuaciones deberemos usar estas rutinas incluidas en ScaLAPACK, donde la matriz PyACTS `a` se encuentra distribuida en bandas (`ACTS_a=Num2PyACTS(a, 501, bwu, bwl, piv='db')`, y el vector PyACTS `b` está distribuido de la forma *right-hand side vector*. Para el resto de rutinas de este apartado se debe seguir el mismo esquema de utilización del parámetro `piv` descrito en el apartado 8.2.1.

- Matriz general tridiagonal (sin pivotación).

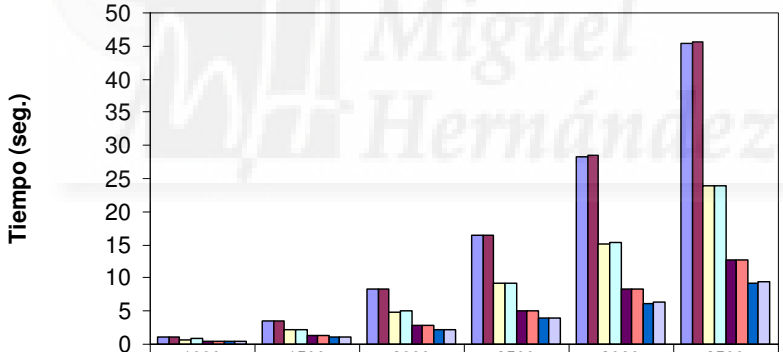
```
af,info= pvdttrf(a)
b,info=pvdttrs(af,b)
```



	1500	2000	2500	3000	3500	4000
FORTRAN 1X1	1,60	3,65	6,82	11,43	18,06	26,57
PYTHON 1X1	1,61	3,67	6,86	11,46	18,19	26,71
FORTRAN 2X1	1,29	2,66	4,64	7,27	11,01	15,97
PYTHON 2X1	2,20	2,99	4,99	7,34	10,69	17,60
FORTRAN 2X2	0,95	1,85	3,07	4,68	6,89	9,78
PYTHON 2X2	1,37	2,51	3,24	5,61	6,51	9,84
FORTRAN 3X2	1,02	1,63	2,70	4,23	5,80	7,96
PYTHON 3X2	0,95	1,69	2,93	4,12	5,64	7,84

Tamaño de la matriz

(a) Cluster-umh



	1000	1500	2000	2500	3000	3500
FORTRAN 1X1	1,03	3,51	8,25	16,49	28,35	45,43
PYTHON 1X1	1,03	3,52	8,27	16,54	28,49	45,53
FORTRAN 2X1	0,76	2,15	4,86	9,10	15,23	23,87
PYTHON 2X1	0,77	2,18	5,00	9,15	15,35	24,01
FORTRAN 2X2	0,53	1,32	2,79	5,05	8,29	12,76
PYTHON 2X2	0,53	1,33	2,81	5,04	8,29	12,76
FORTRAN 3X2	0,52	1,15	2,21	3,84	6,22	9,28
PYTHON 3X2	0,53	1,16	2,22	3,85	6,30	9,36

Tamaño del Sistema

(b) Seaborg

Figura 8.6: Comparativa de `pvgetrf` en lenguaje Fortran vs Python

- Matriz simétrica o hermítica definida positiva.

```
af,info= pvpotrf(a[,uplo='U'])
b,info=pvpotrs(af,b[,uplo='U'])
rcond,info=pvocon(af,[,anorm=1,uplo='U'])
x,ferr,berr,info= pvporfs(a,af,b,x[,trans='N',uplo='U'])
a,info= pvpotri(af[,uplo='U'])
r,c,rowcnd,colcnd,amax,info=pvpoequ(a)
```

  Tanto las rutinas como sus parámetros de entrada y salida son similares a los vistos para las matrices generales. La única particularidad es que al ser matrices simétricas, debemos indicar mediante el parámetro `uplo` si utilizar la parte triangular superior (`uplo='U'`) o inferior (`uplo='L'`).

- Matriz en bandas simétrica o hermítica definida positiva.

```
af,info= pvpbtrf(a)
b,info=pvpbtrs(af,b)
```

- Matriz tridiagonal simétrica o hermítica definida positiva.

```
af,info= pvpttrf(a)
b,info=pvpttrs(af,b)
```

8.3.2. Problemas de mínimos cuadrados y factorizaciones ortogonales

En estas rutinas computacionales, se proporcionan diferentes factorizaciones que ya fueron descritas en el apartado de ScaLAPACK 4.3.5. En la distribución de PyScaLAPACK, también se realiza una distinción del esquema de almacenamiento de las matrices de datos, sin embargo, no es necesario indicar el tipo de datos que contienen. A continuación se muestran las rutinas, clasificadas a partir de la factorización que implementan:

- Factorización QR general.

```
aqr,tau,info= pvgeqpf(a)
```

	side='L'	side='R'
trans='N'	QC	CQ
trans='T'	Q^TC	CQ^T

Tabla 8.1: Funcionalidad de pvormqr

```
aqr,tau,info= pvgeqrf(a)
q,info= pvorgqr(aqr,tau)
c,info= pvormqr(aqr,c,tau,k[,side='L',trans='N'])
```

Las rutinas pvgeqpf y pvgeqrf obtienen la factorización QR con y sin pivotación parcial, respectivamente. Además, se obtiene tau que contiene los factores escalares de los reflectores elementales. A partir de la factorización, podemos obtener q con sus n columnas ortonormales [14]. A partir de la matriz PyACTS c, mediante la rutina pvormqr podemos operar con la matriz q según se indique mediante los parámetros side y trans según la tabla 8.1. El parámetro k indica el número de reflectores elementales tau que define el producto de la matriz Q, donde $M \geq$ k ≥ 0 si side='L' o $N \geq$ k ≥ 0 si side='R'. Se debe destacar que antes de la ejecución de las rutinas pvorgqr y pvormqr, debemos realizar previamente la factorización QR con la rutina correspondiente.

Para ilustrar la utilización de estas rutinas computacionales, adjuntamos un ejemplo de utilización de pvormqr y por tanto, también de pvgeqpf ya que su ejecución es un paso previo y necesario. No mostramos la ejecución de este ejemplo para no extendernos, y por esta razón tampoco aparecen impresiones en pantalla (comando print) en el código.

```
1  from  PyACTS import *
2  import PyACTS.PyScaLAPACK as PySLK
3  from Numeric import *
4  n=8
5  #Initiliaze the Grid
6  PyACTS.gridinit(nb=2)
7  if PyACTS.iread==1:
8          a=8*identity(n,Float)+ones([n,n],Float)
9          c=8*identity(n,Float)
10 else:
```

```
11              a , c=None , None
12  ACTS_lib=1 # 1=Scalapack
13  a=Num2PyACTS( a , ACTS_lib )
14  c=Num2PyACTS( c , ACTS_lib )
15  a , tau , info= PySLK. pvgeqrf( a )
16  c , info= PySLK. pvormqr( a , c , tau , k=1, side='L' , trans='N' )
17  c=PyACTS2Num( c )
18  PyACTS. gridexit ()
```

Se aprecia en la línea 15 la llamada a la rutina de factorización con pivotación para, a continuación, realizar una llamada a `pvormqr` donde se realiza la operación reflejada en la tabla 8.1, siendo `c` una matriz identidad.

En la figura 8.7 se representan los tiempos de ejecución obtenidos para realizar la factorización mediante `pvgeqpf`. Del mismo modo que en las rutinas computacionales evaluadas hasta el momento, los tiempos son similares en entornos de programación Python y Fortran. Para la ejecución en el Cluster-umh, los tiempos en Fortran son ligeramente superiores a los tiempos obtenidos en Python, seguramente debido a una mejor gestión de memoria que evite las paginaciones. Además, se observa que este tipo de factorización es computacionalmente más costosa que la factorización implementada por `pvgetrf`.

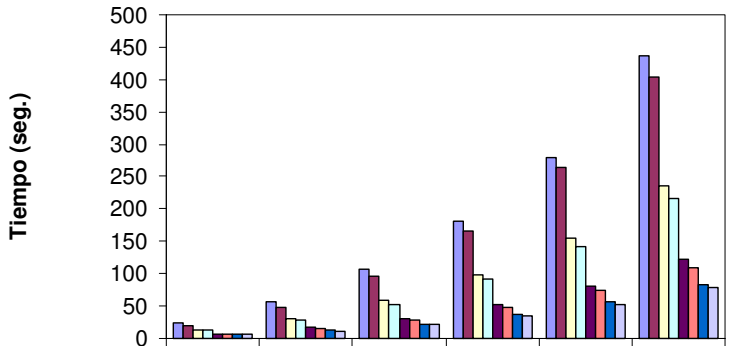
- Factorización LQ general.

```
alq,tau,info= pvgelqf(a)
q,info= pvorglq(alq,tau)
c,info= pvormlq(alq,c,tau,k[,side='L',trans='N'])
```

Las rutinas proporcionadas para la factorización LQ tienen una interfaz muy similar a la vista para las rutinas de la factorización QR. Sin embargo, para realizar la factorización LQ únicamente se proporciona una rutina que hace uso de pivotación, a diferencia de la factorización QR en la que se proporcionan dos rutinas. La funcionalidad y utilización de las rutinas para este tipo de factorización y para las factorizaciones expuestas en el listado siguiente, es totalmente análoga a la vista para QR.

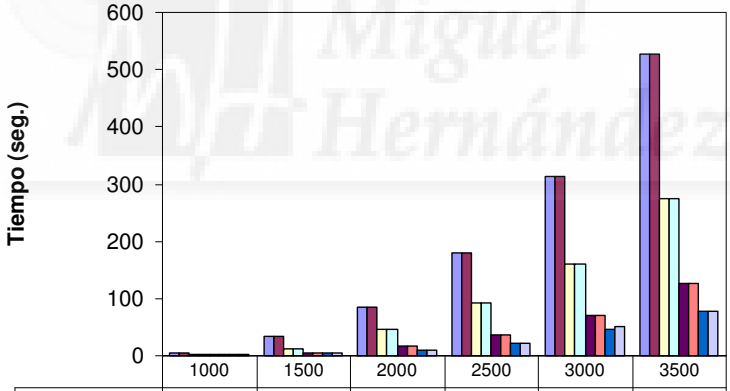
- Factorización QL general.

```
aql,tau,info= pvgeqlf(a)
```



	1500	2000	2500	3000	3500	4000
FORTRAN 1X1	23,06	56,08	106,32	180,26	280,21	436,16
PYTHON 1X1	20,57	48,74	95,08	166,37	264,90	402,92
FORTRAN 2X1	13,08	30,65	58,21	97,83	155,13	235,75
PYTHON 2X1	12,19	27,90	53,30	91,04	141,20	216,25
FORTRAN 2X2	7,27	16,56	30,90	51,38	80,71	121,78
PYTHON 2X2	6,91	15,19	28,24	47,23	73,92	109,77
FORTRAN 3X2	5,73	12,01	22,29	37,68	57,11	83,40
PYTHON 3X2	5,56	11,39	21,27	35,14	52,84	78,05

(a) Cluster-umh



	1000	1500	2000	2500	3000	3500
FORTRAN 1X1	5,65	33,78	84,73	180,79	313,10	526,55
PYTHON 1X1	5,53	33,74	85,43	180,53	313,98	526,47
FORTRAN 2X1	3,46	12,79	47,34	91,65	159,39	273,55
PYTHON 2X1	3,50	12,80	46,83	91,54	160,21	273,92
FORTRAN 2X2	1,94	5,18	17,03	36,55	70,03	126,48
PYTHON 2X2	1,98	5,18	17,67	37,28	70,10	127,12
FORTRAN 3X2	1,63	4,63	9,25	20,90	46,18	76,60
PYTHON 3X2	1,66	4,61	9,31	20,95	49,91	77,19

(b) Seaborg

Figura 8.7: Comparativa de `pvgeqpf` en lenguaje Fortran vs Python

```
q,info= pvorgql(aql,tau)
c,info= pvormql(aql,c,tau,k[,side='L',trans='N'])
```

- Factorización RQ general.

```
arq,tau,info= pvgerqf(a)
q,info= pvorgrq(arq,tau)
c,info= pvormrq(arq,c,tau,k[,side='L',trans='N'])
```

- Factorización RZ trapezoidal.

```
az,tau,info= pvtzrzf(a)
c,info= pvormrz(az,c,tau,k[,side='L',trans='N'])
```

8.3.3. Problemas de valores propios

En este grupo funcional de rutinas de PyScaLAPACK, agruparemos las rutinas computacionales que permiten la obtención de valores y vectores propios tanto para matrices simétricas como no simétricas. Estas rutinas proporcionan una interfaz desde Python, más cómoda y sencilla a las rutinas incluidas en ScaLAPACK mostradas en las tablas 4.14 y 4.15 del apartado 4.3.5.

Las rutinas computacionales para la obtención de valores y vectores propios se enumeran a continuación.

- Matrices simétricas.

```
aq,tau,info= pvsytrd(a[,uplo='U']
c,info=pvormtr(aq,c,tau[,side='L',uplo='U',trans='N']
```

  La rutina `pvsytrd` reduce una matriz PyACTS simétrica a su forma tridiagonal T ($A = QTQ^T$ o $A = QTQ^H$) devuelta como la matriz PyACTS `aq`, donde las características de la transformación han sido descritas en el apartado de ScaLAPACK. Si `uplo='U'`, en la entrada, la rutina referencia la parte triangular superior de

la matriz `a`, y en la salida la diagonal y la diagonal superior se sobreescriben con los elementos de la matriz tridiagonal T. En el caso que `uplo='L'`, a la entrada la rutina referencia la parte triangular inferior de la matriz `a` y a la salida la diagonal y la diagonal inferior se sobreescriben con los elementos de la matriz tridiagonal T. El parámetro `tau` es un vector `Numeric` que contiene los factores escalares de los reflectores elementales vinculado a la matriz `at`.

Por otro lado, mediante la rutina `pvormtr` podemos operar con la matriz `q` utilizando los parámetros `side` y `trans` de forma similar a como se ha visto para la rutina `pvormqr` en la tabla 8.1. El parámetro `k` indica el número de reflectores elementales `tau` que define el producto de la matriz Q.

A continuación mostramos un ejemplo muy sencillo de la utilización de este tipo de rutinas.

```
1  from  PyACTS import *
2  import PyACTS.PyScaLAPACK as PySLK
3  from Numeric import *
4  n=8
5  PyACTS.gridinit(nb=2)
6  if PyACTS.iread==1:
7          a=8*identity(n,Float)+ones([n,n],Float)
8          c=identity(n,Float)
9  else:
10         a,c=None,None
11 ACTS_lib=1 # 1=Scalapack
12 a=Num2PyACTS(a,ACTS_lib)
13 c=Num2PyACTS(c,ACTS_lib)
14 a,tau,info= PySLK.pvsytrd(a,uplo='U')
15 c,info= PySLK.pvormtr(a,c,tau,side='L',uplo='U',trans='N')
16 c=PyACTS2Num(c)
17 PyACTS.gridexit()
```

Del mismo modo que en ejemplos anteriores, preferimos no mostrar la ejecución del código para no extendernos en la longitud de este apartado. Recordamos que en la distribución PyACTS, se proporcionan ejemplos de utilización para todas las rutinas, tanto de PyScaLAPACK como PyBLACS o PyPBLAS. Deseamos destacar en este ejemplo que la ejecución de la rutina `pvsytrd` (línea 14) es necesaria para obtener la forma tridiagonal `a`, que será utilizada en la llamada a la rutina `pvormtr`.

En este ejemplo, el tipo de dato utilizado en las matrices `a` y `c` es coma flotante de doble precisión (indicado por `Float` en la línea 7 y 8). De este modo, de forma interna, PyScaLAPACK llamará a la rutina `PDSYTRD` y `PDHETRD`, respectivamente. No obstante, si se hubiera definido un tipo de dato complejo (por ejemplo, mediante `c=(1+1.j)*identity(n,Float)`), la rutina de ScaLAPACK utilizada hubiera sido `PDHETRD`.

- Matrices no simétricas.

```
ah,tau,info= pvgehrd(a[ilo=1,ihi=-1]
c,info=pvormhr(ah,c,tau[ilo=1,ihi=-1,side='L',trans='N']
```

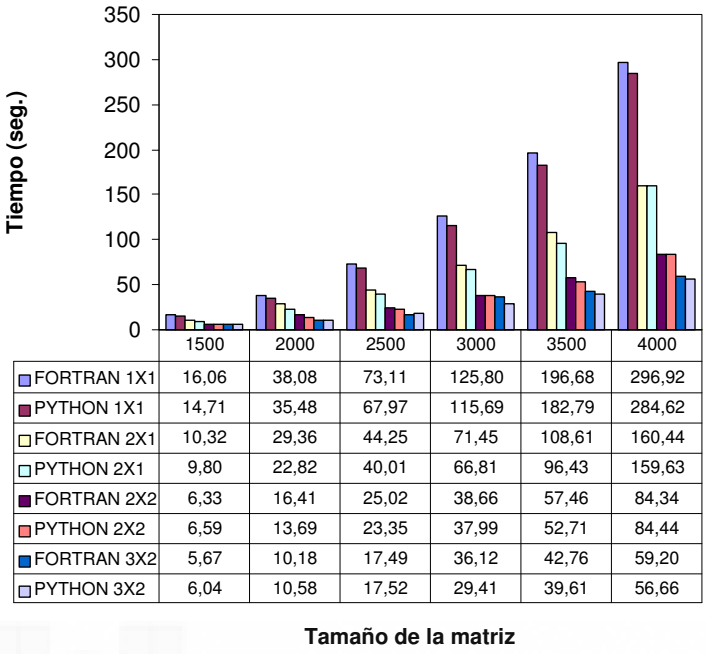
La rutina `pvgehrd` reduce una matriz PyACTS `a` a su forma superior de Hessenberg H mediante una transformación ortogonal $A = QHQ^T$. En la parte superior y en la primera subdiagonal de la matriz PyACTS `ah` de salida, los elementos se sobreescriben con la parte superior de la matriz de Hessenberg. Por otro lado, `tau` proporciona los factores escalares de los reflectores elementales en la factorización. La matriz Q se representa como producto de los `ihi` a `ilo` elementos reflectores $Q = H_{ilo}H_{ilo+1}\dots H_{ihi-1}$, donde cada $H_i = I - \tau vv'$ [14].

Mediante la rutina `pvormhr` podemos operar con la matriz H utilizando los parámetros `side` y `trans` de forma similar a como se ha visto para la rutina `pvormqr` en la tabla 8.1. La utilización de ambas rutinas es muy similar a la mostrada en el ejemplo de `pvormtr`.

- Matrices simétricas definidas positivas.

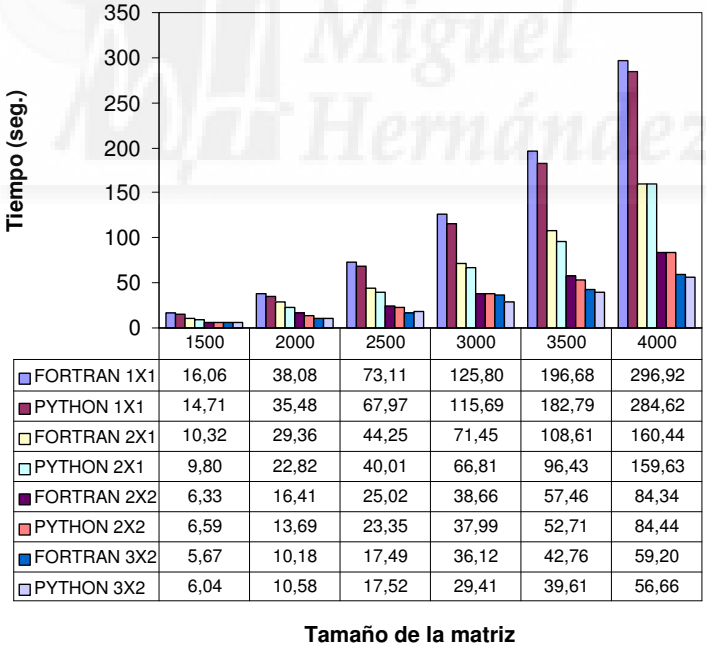
```
a,scale,info= PySLK.pvsygst(a,b,ibtype=1,uplo='U')
```

Este grupo de rutinas trata de obtener la solución un cualquiera de los sistemas planteados en la tabla 4.17 del apartado 4.3.5. Dependiendo del tipo de problema (indicado mediante `ibtype`) y a partir de las matrices PyACTS `a` y `b`, se obtiene la factorización y obtención de los vectores propios realizando la conversión adecuada. El parámetro de salida `scale` es un número en coma flotante que indica los valores propios que debieran ser escalados para compensar el escalado en esta rutina. Actualmente, siempre se devuelve `scale=1` pero se ha reservado su uso en la interfaz, para una futura implementación.



	1500	2000	2500	3000	3500	4000
FORTRAN 1X1	16,06	38,08	73,11	125,80	196,68	296,92
PYTHON 1X1	14,71	35,48	67,97	115,69	182,79	284,62
FORTRAN 2X1	10,32	29,36	44,25	71,45	108,61	160,44
PYTHON 2X1	9,80	22,82	40,01	66,81	96,43	159,63
FORTRAN 2X2	6,33	16,41	25,02	38,66	57,46	84,34
PYTHON 2X2	6,59	13,69	23,35	37,99	52,71	84,44
FORTRAN 3X2	5,67	10,18	17,49	36,12	42,76	59,20
PYTHON 3X2	6,04	10,58	17,52	29,41	39,61	56,66

(a) Cluster-umh



	1500	2000	2500	3000	3500	4000
FORTRAN 1X1	16,06	38,08	73,11	125,80	196,68	296,92
PYTHON 1X1	14,71	35,48	67,97	115,69	182,79	284,62
FORTRAN 2X1	10,32	29,36	44,25	71,45	108,61	160,44
PYTHON 2X1	9,80	22,82	40,01	66,81	96,43	159,63
FORTRAN 2X2	6,33	16,41	25,02	38,66	57,46	84,34
PYTHON 2X2	6,59	13,69	23,35	37,99	52,71	84,44
FORTRAN 3X2	5,67	10,18	17,49	36,12	42,76	59,20
PYTHON 3X2	6,04	10,58	17,52	29,41	39,61	56,66

(b) Seaborg

Figura 8.8: Comparativa de `pvgehrd` en lenguaje Fortran vs Python

8.3.4. Descomposición en valores singulares

El cuarto y último grupo funcional en las rutinas computacionales de PyScaLAPACK permite la obtención de los valores y vectores singulares de una matriz mediante la ejecución de las mismas paso a paso. A continuación mostramos las rutinas agrupadas en este tipo de rutinas computacionales.

```
a,d,e,tauq,taup,info= PySLK.pvgebrd(a)
c,info= pvormbr(a,c,tau[,vect='Q',side='L',trans='N'])
```

La rutina `pvgebrd` reduce una matriz PyACTS de tamaño $M \times N$ a su forma bidiagonal superior o inferior mediante una transformación ortogonal del tipo $Q'AP = B$. En esta rutina, si $M \geq N$, se sobreescriben los datos de la diagonal y subdiagonal superior de A con los datos de bidiagonales superiores de B, los elementos por debajo de la diagonal de `a` junto con `tauq`, representan la matriz ortogonal Q como producto de reflectores elementales y los elementos por encima de la primera superdiagonal junto con el vector `taup` representan la matriz ortogonal P como producto de reflectores elementales.

Por otro lado, si $M < N$, se sobreescriben los datos de la diagonal y subdiagonal inferior de A con los datos de bidiagonales inferiores de B, los elementos por encima de la diagonal de `a` junto con `taup`, representan la matriz ortogonal P como producto de reflectores elementales y los elementos por debajo de la primera subdiagonal inferior junto con el vector `tauq` representan la matriz ortogonal Q como producto de reflectores elementales. Los vectores `d` y `e` devueltos por la rutina `pvgebrd` proporcionan la matriz bidiagonal B siendo `d` los elementos de la diagonal principal, y `e` los elementos de la subdiagonal.

Por último, la rutina `pvormbr` sobreescribe la matriz PyACTS `c` en función de los parámetros de entrada según se indica en la tabla 8.2.

A continuación, mostramos las líneas mas relevantes de un ejemplo de utilización de las rutinas `pvgebrd` y `pvormbr`, el resto de líneas se han sustituido por puntos suspensivos y serían similares al resto de ejemplos mostrados durante este apartado. Se observa en este ejemplo la obtención de los vectores `d` y `e` así como de los reflectores elementales en la línea 2. En la línea 3 se realiza la llamada a `pvormbr` indicando cuál de los vectores `vect='Q'` o `vect='P'` es el utilizado y por tanto qué vector de elementos reflectores debemos utilizar

vect='Q'		
	side='L'	side='R'
trans='N'	QC	CQ
trans='T'	Q^TC	CQ^T
vect='P'		
	side='L'	side='R'
trans='N'	PC	CP
trans='T'	P^TC	CP^T

Tabla 8.2: Funcionalidad de `pvormbr`

(en este caso `tauq`).

```
1 ...
2 a,d,e,tauq,taup,info= PySLK.pvgebrd(a)
3 c,info= PySLK.pvormbr(a,c,tauq,vect='Q',side='L',trans='N')
4 ...
```

8.4. Justificación de los parámetros por defecto

En el apartado 5.8 del capítulo de PyACTS se introdujeron los conceptos principales de la distribución. Uno de los objetivos de dicha herramienta es el de facilitar y automatizar determinados parámetros del entorno paralelo que el usuario final puede desear no modificar o ni siquiera conocer su funcionamiento. Para ocultar determinados aspectos del paralelismo a los usuarios de PyACTS, se tomaron ciertas decisiones que afectaban al tamaño de bloque más adecuado en una ejecución y a la configuración de la malla de procesos.

Tanto el tamaño de bloque como la configuración de la malla de procesos tienen una elevada importancia en la distribución o en la ejecución de cualquiera de las rutinas de PyBLACS, PyPBLAS o PyScaLAPACK. Una vez introducida la distribución PyACTS y presentado cada uno de sus módulos, presentamos en este apartado las pruebas realizadas que nos permiten comprobar el comportamiento de PyACTS en función de estos

parámetros y que serán descritas en el siguiente apartado. Las pruebas mostradas han sido realizadas en Jacquard, sin embargo, se han realizado pruebas en todas las plataformas mencionadas en el apartado 2.6 obteniendo resultados similares que no mostramos en este apartado para evitar extendernos en la longitud del mismo.

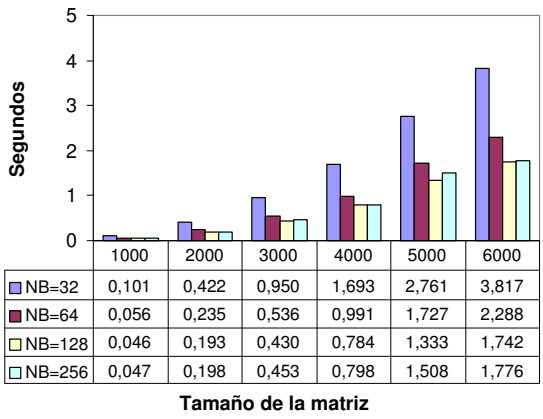
8.4.1. Influencia del tamaño del bloque de datos

Tal y como se describe en el apartado 4.3.5, la distribución de matrices de tipo denso en PBLAS y ScaLAPACK es de tipo cíclica 2D. Además de ser ésta la configuración utilizada para distribuir los datos entre los diferentes procesos, también es la configuración que esperan las rutinas de PBLAS y ScaLAPACK para realizar sus cálculos. Es conveniente recordar que a una rutina de este tipo, por cada dato hemos de proporcionar su posición en memoria así como un descriptor que contiene información relativa a esta distribución cíclica 2D.

Por tanto, los factores que intervienen para determinar una distribución cíclica 2D son:

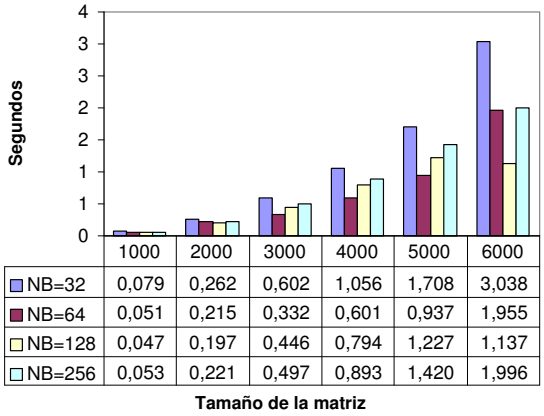
- Tamaño de la matriz $M \times N$
- Tamaño del bloque de distribución $MB \times NB$
- Configuración de la malla de procesos `nprow` $\times$ `npcol`

En las pruebas que mostramos a continuación, hemos establecido $MB = NB$ ya que los tiempos obtenidos para valores diferentes entre $MBy NB$ han sido peores y no los mostramos en este apartado para evitar extendernos. En la figura 8.9 se muestran los tiempos necesarios para distribuir una matriz cuadrada de diferentes tamaños entre los procesos pertenecientes a una malla de procesos (2×1, 2×2, 4×4 y 6×6) utilizando un tamaño de bloque determinado. Por ejemplo, en la figura 8.9(a) se muestran los tiempos necesarios para distribuir matrices cuadradas entre dos procesos (por tanto, con una malla 2×1 y utilizando diferentes tamaños de bloque (32, 64, 128 y 256). Desde un punto de vista lógico, cabe pensar que cuanto mayor sea el tamaño del bloque, menor será el tiempo necesario para distribuir los datos puesto que se aumenta la eficiencia de ese mensaje. Esta característica se puede apreciar en la figura, ya que para un tamaño de bloque NB



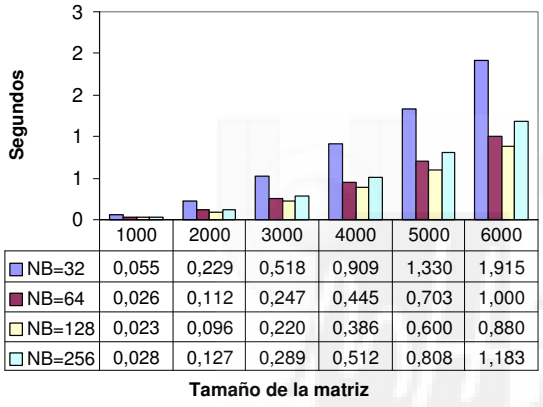
	1000	2000	3000	4000	5000	6000
NB=32	0,101	0,422	0,950	1,693	2,761	3,817
NB=64	0,056	0,235	0,536	0,991	1,727	2,288
NB=128	0,046	0,193	0,430	0,784	1,333	1,742
NB=256	0,047	0,198	0,453	0,798	1,508	1,776

(a) 2×1



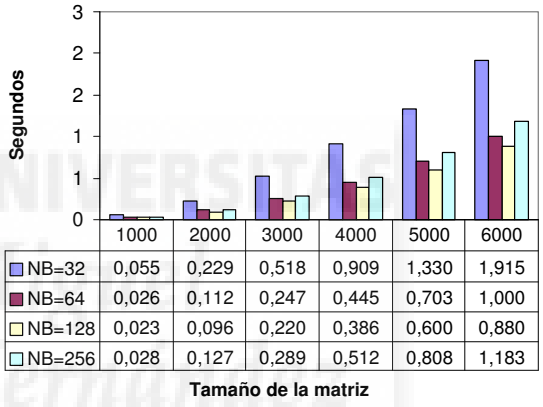
	1000	2000	3000	4000	5000	6000
NB=32	0,079	0,262	0,602	1,056	1,708	3,038
NB=64	0,051	0,215	0,332	0,601	0,937	1,955
NB=128	0,047	0,197	0,446	0,794	1,227	1,137
NB=256	0,053	0,221	0,497	0,893	1,420	1,996

(b) 2×2



	1000	2000	3000	4000	5000	6000
NB=32	0,055	0,229	0,518	0,909	1,330	1,915
NB=64	0,026	0,112	0,247	0,445	0,703	1,000
NB=128	0,023	0,096	0,220	0,386	0,600	0,880
NB=256	0,028	0,127	0,289	0,512	0,808	1,183

(c) 4×4



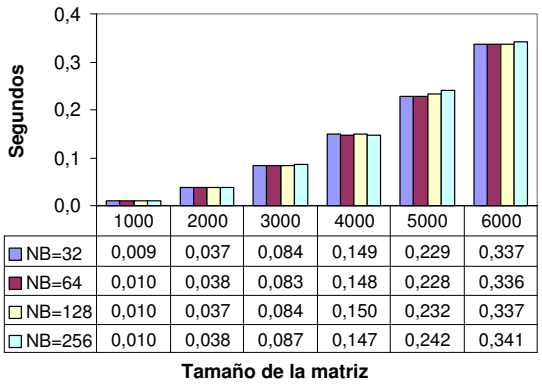
	1000	2000	3000	4000	5000	6000
NB=32	0,055	0,229	0,518	0,909	1,330	1,915
NB=64	0,026	0,112	0,247	0,445	0,703	1,000
NB=128	0,023	0,096	0,220	0,386	0,600	0,880
NB=256	0,028	0,127	0,289	0,512	0,808	1,183

(d) 6×6

Figura 8.9: Distribución de matrices de datos para diferentes tamaños de Bloque

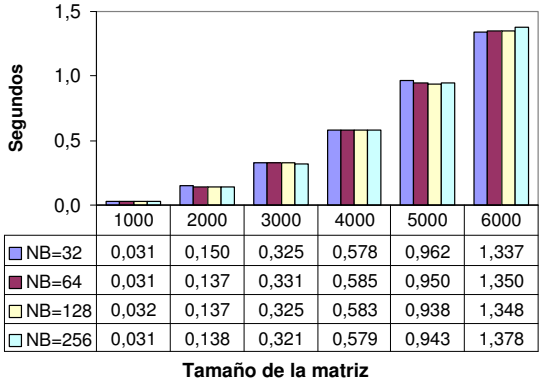
de 32 elementos, los tiempos suelen ser considerablemente superiores que con tamaños mayores. Sin embargo, para un tamaño de 64 los tiempos de distribución son similares a los obtenidos con tamaños de 128. Por tanto, podemos ver que un tamaño de bloque 64×64 ya se puede considerar aceptable.

Un detalle que llama la atención es el hecho que para NB=64 y NB=128 obtenemos valores similares en la mayoría de las configuraciones, aunque se obtiene ligeramente un mejor resultado para NB=128. Sin embargo, con NB=256 se consiguen peores resultados que con tamaños de bloque inferiores a pesar que aumentamos en la eficiencia de cada mensaje. En definitiva, podemos concluir mediante estas pruebas realizadas, que para



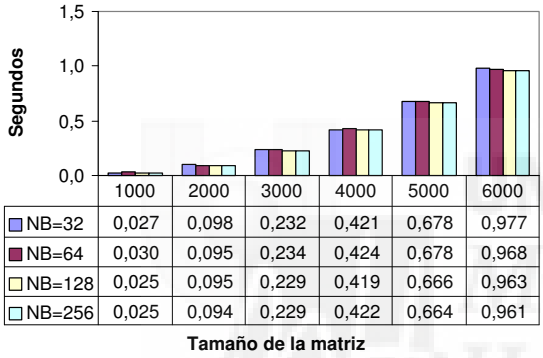
	1000	2000	3000	4000	5000	6000
NB=32	0,009	0,037	0,084	0,149	0,229	0,337
NB=64	0,010	0,038	0,083	0,148	0,228	0,336
NB=128	0,010	0,037	0,084	0,150	0,232	0,337
NB=256	0,010	0,038	0,087	0,147	0,242	0,341

(a) 1×1



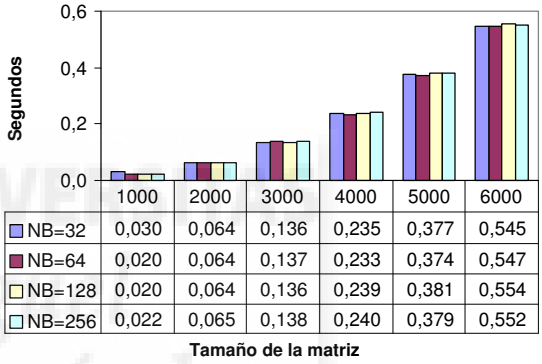
	1000	2000	3000	4000	5000	6000
NB=32	0,031	0,150	0,325	0,578	0,962	1,337
NB=64	0,031	0,137	0,331	0,585	0,950	1,350
NB=128	0,032	0,137	0,325	0,583	0,938	1,348
NB=256	0,031	0,138	0,321	0,579	0,943	1,378

(b) 2×1



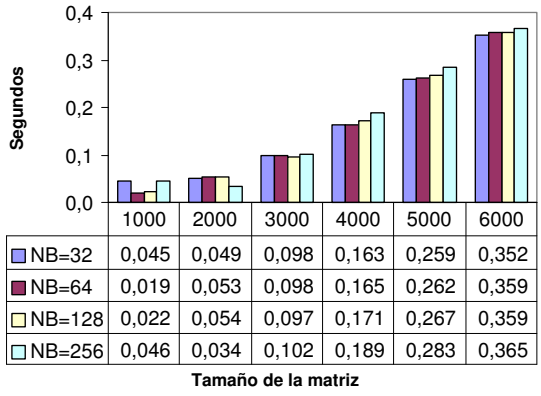
	1000	2000	3000	4000	5000	6000
NB=32	0,027	0,098	0,232	0,421	0,678	0,977
NB=64	0,030	0,095	0,234	0,424	0,678	0,968
NB=128	0,025	0,095	0,229	0,419	0,666	0,963
NB=256	0,025	0,094	0,229	0,422	0,664	0,961

(c) 2×2



	1000	2000	3000	4000	5000	6000
NB=32	0,030	0,064	0,136	0,235	0,377	0,545
NB=64	0,020	0,064	0,137	0,233	0,374	0,547
NB=128	0,020	0,064	0,136	0,239	0,381	0,554
NB=256	0,022	0,065	0,138	0,240	0,379	0,552

(d) 4×4



	1000	2000	3000	4000	5000	6000
NB=32	0,045	0,049	0,098	0,163	0,259	0,352
NB=64	0,019	0,053	0,098	0,165	0,262	0,359
NB=128	0,022	0,054	0,097	0,171	0,267	0,359
NB=256	0,046	0,034	0,102	0,189	0,283	0,365

(e) 6×6

Figura 8.10: Ejecución de pvgemm para diferentes tamaños de bloque y mallas de procesos

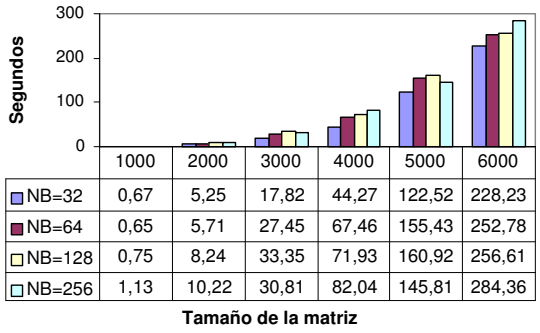
realizar la distribución de los datos en diferentes configuraciones de la malla de procesos y tamaño de la matriz, se considera apropiado un tamaño de bloque NB=64 o NB=128.

No obstante, podemos plantearnos si los tamaños de bloque antes indicados también son adecuados para realizar los cálculos en las rutinas incluidas en PyPBLAS y PyScaLAPACK. Para tratar de aclarar esta cuestión mostramos las pruebas realizadas en Jacquard ejecutando una rutina de PyPBLAS (`pvgemm`, figura 8.10) y dos rutinas de PyScaLAPACK (`pvgesv` y `pvgesvd`, figuras 8.11 y 8.12 respectivamente). En el presente trabajo mostramos los resultados obtenidos como muestra de las múltiples pruebas realizadas para un amplio conjunto de las rutinas incluidas en cada uno de los módulos PyBLACS, PyPBLAS y PyScaLAPACK y preferimos no mostrar todas las pruebas con un comportamiento similar a las presentadas en este trabajo.

En primer término, se observa que para la rutina PyPBLAS de nivel 3 `pvgemm` ($\alpha op(A)op(B) + \beta C \rightarrow C$), los tiempos obtenidos utilizando diferentes tamaño de bloques son muy similares en la mayoría de las configuraciones de malla y tamaños de matrices, por lo que parece no influir mucho en las rutinas incluidas en PBLAS. Podemos concluir que para las rutinas de las PyPBLAS el tamaño de bloque parece no ser un factor decisivo obteniéndose tiempos similares para diferentes valores del tamaño de bloque.

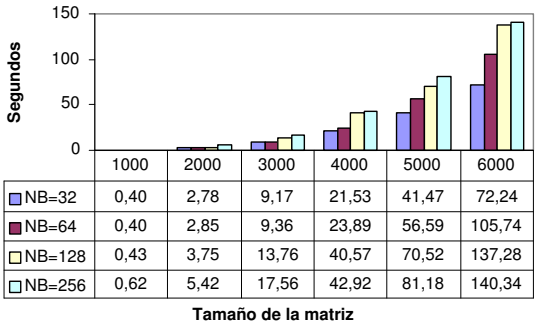
Sin embargo, si realizamos las mismas pruebas pero con rutinas incluidas en PyScaLAPACK los resultados son significativamente diferentes. En la figura 8.11 se representan los tiempos de ejecución para resolver un sistema del tipo $Ax = B$ con la rutina `pvgesv` mediante diferentes tamaños de bloque (32, 64, 128 y 256), con diferentes configuraciones de malla de procesos y para diferentes tamaños del sistema a resolver. Se observa de una manera clara, cómo a medida que incrementamos el tamaño de bloque, los tiempos de ejecución se incrementan considerablemente. Por ejemplo, para una configuración de malla 4×4 y un tamaño del sistema de 6000 valores, se consiguen tiempos con $NB = 32$ de 9.8 segundos aproximadamente, mientras que con $NB = 128$ los tiempos obtenidos representan más del doble que el anterior (22,8 segundos) .Por otro lado, se puede observar que para un numero relativamente elevado de procesos, por ejemplo una malla 6×6, la diferencia de tiempos obtenidos entre $NB = 32$ y $NB = 64$ es pequeña mientras que con pocos procesos (malla 2×1 o 2×2) la diferencia entre ambos tiempos es perceptible.

En resumen podemos concluir que la rutina `pvgesv` de PyScaLAPACK se comporta mejor con valores pequeños del tamaño de bloque, siendo valores adecuados $NB = 32$ y $NB = 64$.



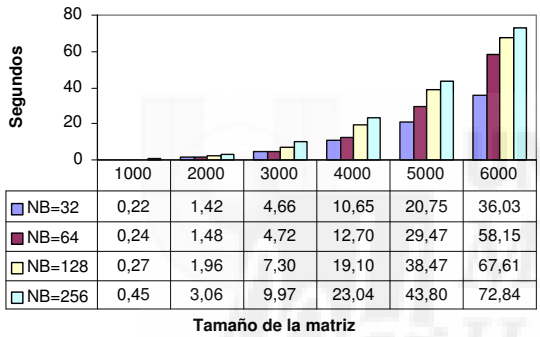
	1000	2000	3000	4000	5000	6000
NB=32	0,67	5,25	17,82	44,27	122,52	228,23
NB=64	0,65	5,71	27,45	67,46	155,43	252,78
NB=128	0,75	8,24	33,35	71,93	160,92	256,61
NB=256	1,13	10,22	30,81	82,04	145,81	284,36

(a) 1×1



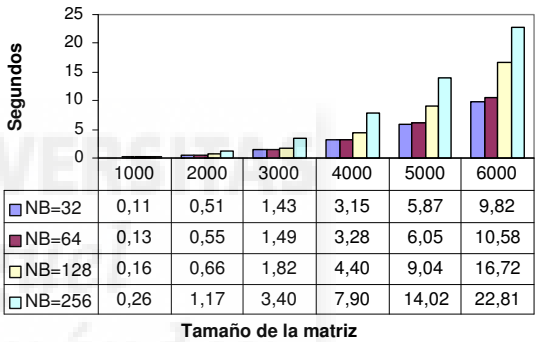
	1000	2000	3000	4000	5000	6000
NB=32	0,40	2,78	9,17	21,53	41,47	72,24
NB=64	0,40	2,85	9,36	23,89	56,59	105,74
NB=128	0,43	3,75	13,76	40,57	70,52	137,28
NB=256	0,62	5,42	17,56	42,92	81,18	140,34

(b) 2×1



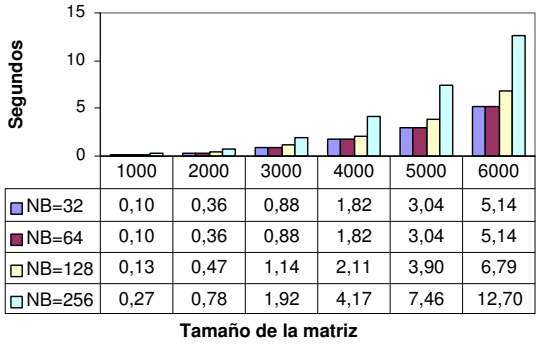
	1000	2000	3000	4000	5000	6000
NB=32	0,22	1,42	4,66	10,65	20,75	36,03
NB=64	0,24	1,48	4,72	12,70	29,47	58,15
NB=128	0,27	1,96	7,30	19,10	38,47	67,61
NB=256	0,45	3,06	9,97	23,04	43,80	72,84

(c) 2×2



	1000	2000	3000	4000	5000	6000
NB=32	0,11	0,51	1,43	3,15	5,87	9,82
NB=64	0,13	0,55	1,49	3,28	6,05	10,58
NB=128	0,16	0,66	1,82	4,40	9,04	16,72
NB=256	0,26	1,17	3,40	7,90	14,02	22,81

(d) 4×4



	1000	2000	3000	4000	5000	6000
NB=32	0,10	0,36	0,88	1,82	3,04	5,14
NB=64	0,10	0,36	0,88	1,82	3,04	5,14
NB=128	0,13	0,47	1,14	2,11	3,90	6,79
NB=256	0,27	0,78	1,92	4,17	7,46	12,70

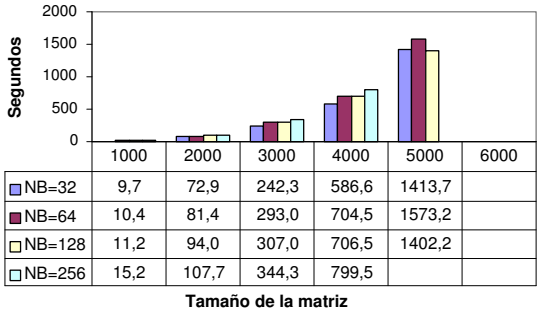
(e) 6×6

Figura 8.11: Ejecución de pvgesv para diferentes tamaños de bloque y mallas de procesos

En la figura 8.12 se muestran los tiempos obtenidos al realizar este tipo de pruebas con la rutina `pvgesvd`, cuya funcionalidad es la de obtener la descomposición en valores singulares de una matriz. En esta figura se observan comportamientos similares a los de la rutina `pvgesv`. Por ejemplo, para configuraciones de malla 2×1 o 4×4 se observa claramente el incremento los tiempos de ejecución para un mismo tamaño de la matriz a resolver cuando aumentamos el tamaño de bloque utilizado. Sin embargo observamos en esta evolución como el incremento del tiempo de ejecución de $NB = 32$ a $NB = 64$ es menor que en otros tamaños. Por tanto, de esta rutina podemos sacar las mismas conclusiones que para la otra rutina de PyScaLAPACK: el comportamiento es mejor a medida que sea menor el tamaño de bloque, por lo que consideramos tamaños de bloques adecuados para la ejecución de rutinas de PyScaLAPACK $NB = 32$ y $NB = 64$.

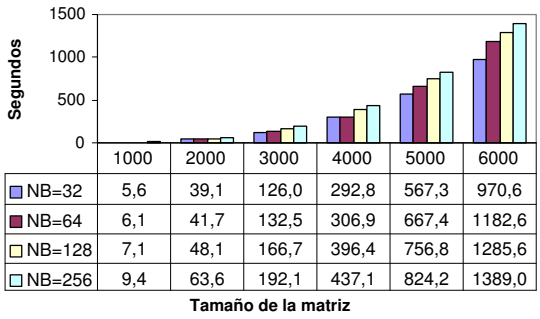
Por último, se desea destacar que el tamaño de bloque recomendado por los creadores de ScaLAPACK se corresponde con un valor de $MB = NB = 64$ [13]. De hecho, este valor ha sido considerado como adecuado tanto en la distribución de los datos, y en la resolución de los diferentes cálculos planteados. En resumen, a partir de las pruebas mostradas y de las recomendaciones proporcionadas anteriormente, concluimos que $MB = NB = 64$ es un tamaño adecuado para utilizar en la distribución cíclica 2D implementada en PyACTS, y por tanto éste será su valor por defecto en el caso de que el usuario no desee indicar uno de forma explícita mediante una ejecución como la realizada en la línea 4 del siguiente ejemplo.

```
1  from PyACTS import *
2  import PyACTS.PyPBLAS as PyPBLAS
3  ACTS_lib=1                          # ScaLAPACK ID
4  PyACTS.gridinit(mb=32,nb=32)        # Grid initialization
5  alpha=Scal2PyACTS(2,ACTS_lib)       # Convert scalar to PyACTS scalar
6  a=Txt2PyACTS("data_a.txt",ACTS_lib) # Read Text file and
7  b=Txt2PyACTS("data_b.txt",ACTS_lib) # store in PyACTS Array
8  result=PyPBLAS.pvaxpy(alpha,a,b)    # Call level 1 PBLAS routine
9  PyACTS2Text("data_result.txt",res)  # Write results to Text
10 PyACTS.gridexit()
```



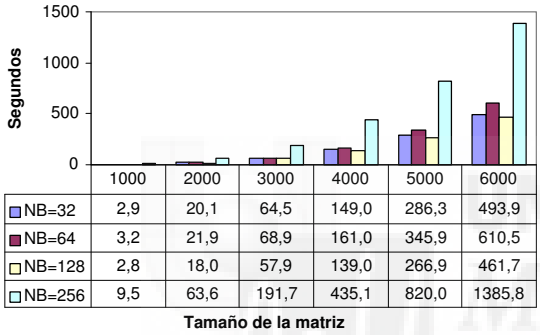
	1000	2000	3000	4000	5000	6000
NB=32	9,7	72,9	242,3	586,6	1413,7	
NB=64	10,4	81,4	293,0	704,5	1573,2	
NB=128	11,2	94,0	307,0	706,5	1402,2	
NB=256	15,2	107,7	344,3	799,5		

(a) 1×1



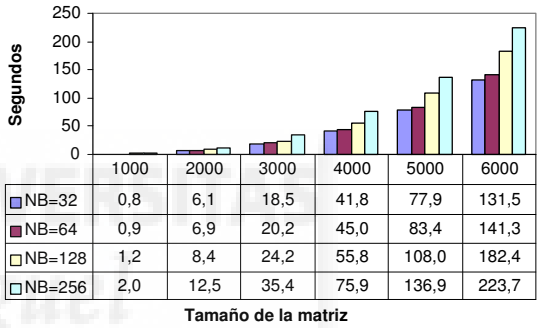
	1000	2000	3000	4000	5000	6000
NB=32	5,6	39,1	126,0	292,8	567,3	970,6
NB=64	6,1	41,7	132,5	306,9	667,4	1182,6
NB=128	7,1	48,1	166,7	396,4	756,8	1285,6
NB=256	9,4	63,6	192,1	437,1	824,2	1389,0

(b) 2×1



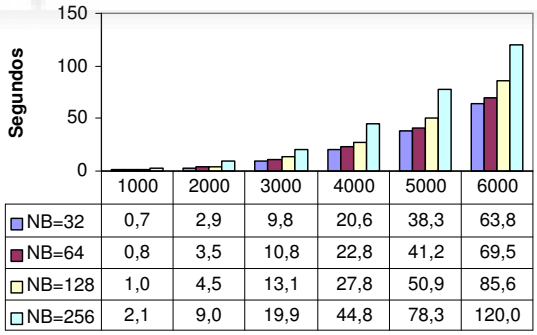
	1000	2000	3000	4000	5000	6000
NB=32	2,9	20,1	64,5	149,0	286,3	493,9
NB=64	3,2	21,9	68,9	161,0	345,9	610,5
NB=128	2,8	18,0	57,9	139,0	266,9	461,7
NB=256	9,5	63,6	191,7	435,1	820,0	1385,8

(c) 2×2



	1000	2000	3000	4000	5000	6000
NB=32	0,8	6,1	18,5	41,8	77,9	131,5
NB=64	0,9	6,9	20,2	45,0	83,4	141,3
NB=128	1,2	8,4	24,2	55,8	108,0	182,4
NB=256	2,0	12,5	35,4	75,9	136,9	223,7

(d) 4×4



	1000	2000	3000	4000	5000	6000
NB=32	0,7	2,9	9,8	20,6	38,3	63,8
NB=64	0,8	3,5	10,8	22,8	41,2	69,5
NB=128	1,0	4,5	13,1	27,8	50,9	85,6
NB=256	2,1	9,0	19,9	44,8	78,3	120,0

(e) 6×6

Figura 8.12: Ejecución de pvgesvd para diferentes tamaños de bloque y mallas de procesos

8.4.2. Influencia de la configuración de la malla de procesos

En el apartado anterior, se ha determinado cuál es el tamaño de bloque adecuado para realizar la distribución de datos y la ejecución de las rutinas de PyPBLAS y PyScaLAPACK. Además, al comienzo de esta sección se plantearon otras cuestiones relativas a la configuración idónea de la malla de procesos para realizar una eficiente distribución o cálculo mediante estas rutinas. En definitiva, suponiendo que deseamos ejecutar un determinado código en un numero de procesos `nprocs` cuya ejecución requiere la distribución de datos, la consiguiente ejecución de rutinas en paralelo y la recolección de este resultado en un único fichero, ¿cual sería la configuración más adecuada para la matriz de procesos?.

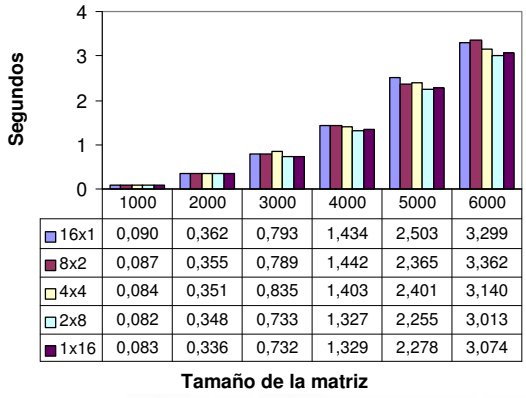
Ante esta cuestión, veremos en esta sección cuál ha sido la solución adoptada en la distribución PyACTS argumentada fundamentalmente por las pruebas realizadas.

Para determinar cuál es la configuración más adecuada de la malla de procesos, procederemos de un modo similar al apartado anterior. En este caso, utilizaremos 16 nodos para realizar las pruebas con un mismo tamaño de bloque ($MB = NB = 64$) e incrementaremos el tamaño del sistema a resolver. Por tanto, para 16 nodos tenemos las siguientes configuraciones posibles: 16×1, 8×2, 4×4, 2×8, 1×16.

En la figura 8.13 se representan los tiempos de ejecución obtenidos para realizar diferentes tareas: distribución de una matriz de datos (figura 8.13(a)), ejecución de la rutina de nivel 3 PyPBLAS `pvgemm` (figura 8.13(b)), ejecución de las rutinas de PyScaLAPACK `pvgesv` y `pvgesvd` (figuras 8.13(c) y 8.13(d), respectivamente). En estas ejecuciones se han realizado pruebas para diferentes tamaños del problema y aplicando a la malla de procesos las configuraciones anteriormente enumeradas.

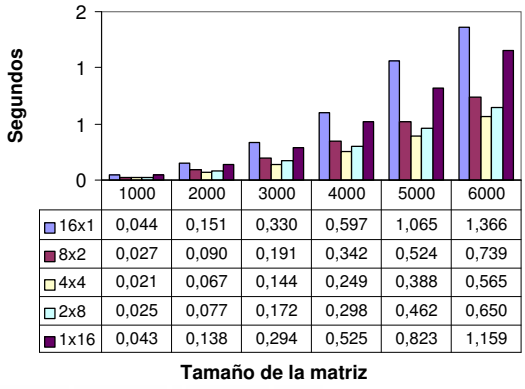
Con respecto al comportamiento de la configuración de la malla de procesos en la distribución de una matriz de datos, se puede observar en la figura 8.13(a) que los tiempos necesarios para su distribución son relativamente similares para todas ellas. Se aprecia una pequeña diferencia a medida que aumenta el numero de columnas en la malla de procesos, en el caso extremo 1×16, la matriz se distribuye por columnas entre los procesos. A pesar de esta pequeña diferencia, no la consideramos decisiva para concluir que tiene un comportamiento mejor cuanto mayor sea el numero de columnas posible.

Por otro lado, si observamos el comportamiento de la variación del aspecto de la malla de procesos en la ejecución de alguna de las rutinas de PyPBLAS y PyScaLAPACK podemos obtener resultados de interesante análisis. Como se puede apreciar en



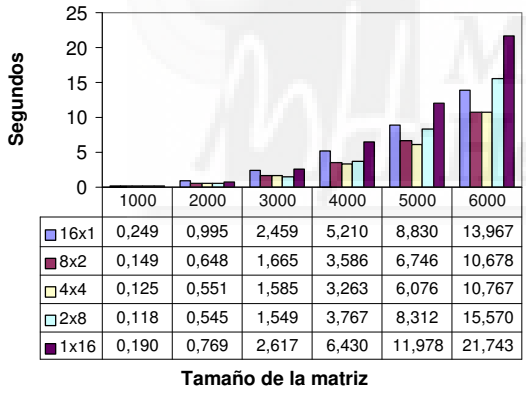
	1000	2000	3000	4000	5000	6000
16x1	0,090	0,362	0,793	1,434	2,503	3,299
8x2	0,087	0,355	0,789	1,442	2,365	3,362
4x4	0,084	0,351	0,835	1,403	2,401	3,140
2x8	0,082	0,348	0,733	1,327	2,255	3,013
1x16	0,083	0,336	0,732	1,329	2,278	3,074

(a) Distribución



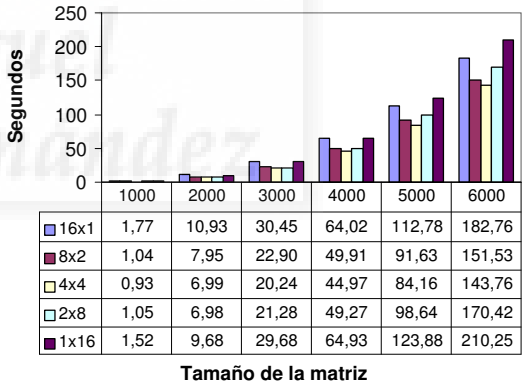
	1000	2000	3000	4000	5000	6000
16x1	0,044	0,151	0,330	0,597	1,065	1,366
8x2	0,027	0,090	0,191	0,342	0,524	0,739
4x4	0,021	0,067	0,144	0,249	0,388	0,565
2x8	0,025	0,077	0,172	0,298	0,462	0,650
1x16	0,043	0,138	0,294	0,525	0,823	1,159

(b) pvgemm



	1000	2000	3000	4000	5000	6000
16x1	0,249	0,995	2,459	5,210	8,830	13,967
8x2	0,149	0,648	1,665	3,586	6,746	10,678
4x4	0,125	0,551	1,585	3,263	6,076	10,767
2x8	0,118	0,545	1,549	3,767	8,312	15,570
1x16	0,190	0,769	2,617	6,430	11,978	21,743

(c) pvgesv



	1000	2000	3000	4000	5000	6000
16x1	1,77	10,93	30,45	64,02	112,78	182,76
8x2	1,04	7,95	22,90	49,91	91,63	151,53
4x4	0,93	6,99	20,24	44,97	84,16	143,76
2x8	1,05	6,98	21,28	49,27	98,64	170,42
1x16	1,52	9,68	29,68	64,93	123,88	210,25

(d) pvgesvd

Figura 8.13: Comparativa de configuraciones de malla para un mismo número de procesos

la figura 8.13(b), los mejores tiempos de ejecución para todos los tamaños de la matriz se consiguen siempre con una malla cuadrada de procesos, en este caso 4×4. Del mismo modo, se observa el mismo comportamiento para las rutinas de PyScaLAPACK (figuras 8.13(c) y 8.13(d)). En ellas se aprecia también de una forma clara, como los tiempos de ejecución necesarios son menores cuando se ejecutan con una malla de procesos cuadrada.

Las pruebas realizadas nos permiten concluir, que en el caso de buscar una configuración de la malla de procesos por defecto óptima, ésta deba ser una malla de procesos lo más cuadrada posible. De este modo se procede en la librería PyACTS, puesto que a partir de un numero de procesos `nprocs` se calcula la parte entera de la raíz cuadrada y a continuación si es posible se le va incrementando el número de filas. Por ejemplo, en una ejecución con `nprocs` = `13`, la configuración por defecto que se obtendría al ejecutar `gridinit()` sería de 4×3.

Además de los resultados obtenidos, los creadores de ScaLAPACK recomiendan utilizar mallas de procesos tan cuadradas como sea posible a partir de un determinado numero de procesos [13], por lo que esta recomendación confirma los resultados que hemos obtenido.

En resumen, a partir de las pruebas realizadas podemos obtener las siguientes conclusiones:

- El tamaño de bloque que se configurará por defecto será cuadrado con 64 elementos en filas y columnas ($MB = NB = 64$).
- A partir del número de procesos, se configurará una malla de procesos tan cuadrada como sea posible, y posteriormente se incrementarán las filas hasta que `nprow x npcol` $\leq$ `nprocs`

8.5. Conclusiones

El módulo PyScaLAPACK es el último módulo presentado que forma parte de PyACTS y ofrece una interfaz más sencilla y cómoda al usuario a las rutinas incluidas en ScaLA-

PACK. Las rutinas incluidas en PyScaLAPACK se dividen en dos grandes grupos: rutinas driver y rutinas computacionales. Durante este capítulo hemos presentado las rutinas de cada uno de los grupos. En las rutinas driver, se resuelve un problema completo que puede ser un sistema de ecuaciones, una resolución de mínimos cuadrados o la obtención de los valores singulares. Por otro lado, las rutinas computacionales realizan una tarea especifica (por ejemplo, una factorización) y la concatenación de estás tareas obtiene la resolución de un problema completo. De este modo, una rutina driver de ScaLAPACK de forma interna realiza una concatenación de llamadas a rutinas computacionales de ScaLAPACK.

Las llamadas a las rutinas de ScaLAPACK son considerablemente más complejas que las llamadas a las rutinas de otras librerías utilizadas como PBLAS o BLACS. De hecho, en ScaLAPACK se utilizan vectores y matrices temporales utilizados como espacios de trabajo o vectores de pivotación. Las dimensiones de estos espacios de memoria auxiliares, varían conforme a las dimensiones del problema y a la complejidad del mismo, por tanto, el usuario debe conocer el funcionamiento de cada rutina de ScaLAPACK y de cada uno de los parámetros para reservar el espacio de memoria necesario. Sin embargo, durante este capítulo hemos comprobado cómo en PyScaLAPACK se reducen considerablemente los parámetros necesarios y se calculan los espacios de memoria de trabajo de forma interna y ocultándolo al usuario. Ocultando la gestión de los espacios de memoria de trabajo, conseguimos simplificar considerablemente muchas de las rutinas de PyScaLAPACK y por tanto facilitar la programación a usuarios no iniciados en librerías numéricas en plataformas de memoria distribuida.

Durante los apartados 8.2 y 8.3 se han presentado las rutinas driver y computacionales de PyScaLAPACK. Cada uno de estos grupos se ha subdividido atendiendo al tipo de problema que se trata de resolver del mismo modo que se clasifica en ScaLAPACK. Además, dentro de cada grupo funcional se diferencia entre diferentes rutinas en función del tipo de matriz y de la distribución de los datos en memoria. Para facilitar la distribución correcta de los datos en memoria, hemos definido en el apartado 8.2.1 el parámetro `piv` que junto con el indicador `ACTS_lib` nos detalla el tipo de distribución en bandas a realizar.

Por otro lado, para todas las rutinas descritas durante este capítulo también se proporciona un ejemplo de utilización en la carpeta `EXAMPLES` de la distribución PyACTS y se detallan los parámetros de entrada y salida en el manual de referencia de PyACTS [34].

Por último, una vez se han presentado los módulos PyBLACS, PyPBLAS y PyScaLAPACK, hemos creido conveniente argumentar las elecciones tomadas ante las preguntas

planteadas en el apartado 5.8. En un entorno que se desea hacer más sencillo para un usuario que se inicie en aplicaciones paralelas de memoria distribuida, la configuración de la malla de procesos `nprow` × `npcol` o el tamaño de bloque en la distribución de datos `MB` × `NB` son parámetros que PyACTS establecerá por defecto. Para verificar que estos valores por defecto son adecuados, se han presentado una serie de pruebas con cada uno de los módulos de PyACTS que nos argumentan y corroboran las configuraciones por defecto realizadas.

En resumen, con PyScaLAPACK hemos completado el conjunto de módulos disponibles en la distribución PyACTS. PyScaLAPACK, a pesar de tener una mayor complejidad en sus parámetros y unos costes computacionales superiores a PyBLACS y PyPBLAS, consigue una sencillez en su utilización (reflejada mediante múltiples ejemplos) sin detrimento ni penalización en la eficiencia de los algoritmos. Además, las pruebas han sido realizadas en plataformas con arquitectura y sistema operativo diferente que confirman la portabilidad de la distribución realizada.

Capítulo 9

PyPnetCDF

En el presente capítulo se presenta la distribución PyPnetCDF [54] como complemento a la distribución PyACTS presentada en capítulos anteriores. Tal y como se tratará en mayor detalle, se proporciona una herramienta capaz de acceder a ficheros de datos en formato netCDF tanto para lectura como para la escritura en paralelo e integrado directamente con los formatos de datos actualmente definidos en la distribución PyACTS.

En la sección 9.1 se presenta el estándar netCDF, donde describiremos el formato del archivo netCDF, evaluaremos su rendimiento comparándolo con otros formatos de almacenamiento de datos y enumeraremos algunas de las múltiples aplicaciones actuales de este tipo de archivos. En la sección 9.2 se introduce al lector la principal herramienta de acceso a ficheros netCDF desde Python (descrita en la sección 3.2.2)) y que será tomada como punto de referencia para el desarrollo de una herramienta paralela. En la sección 9.3 se presenta la librería PnetCDF como una implementación paralela del acceso a datos en este tipo de archivos. A partir de esta librería, presentamos en la sección 9.4 el módulo PyPnetCDF como una distribución de Python que permite el acceso en paralelo desde varios procesos a un mismo origen de datos en formato netCDF. Además de presentar el módulo, en esta sección se presentan las pruebas de rendimiento realizadas comparando PyPnetCDF con la librería para el acceso secuencial que introducida en la sección 9.2.

9.1. El estándar netCDF

La comunidad científica ha reconocido la importancia de un sistema portátil y eficiente para la gestión de datos de grandes dimensiones en sus aplicaciones. El estándar netCDF (*Network Common Data Form*) [94, 89, 88] se define como una interfaz a una librería de funciones de acceso a datos para la lectura y escritura de datos matriciales. En netCDF, se define una matriz como una estructura rectangular de dimensión n que contiene elementos con el mismo tipo de datos (por ejemplo, caracteres de 8 bits, enteros de 32 bits, ...). Un escalar se considera como una matriz de dimensión 0.

Hierarchical Data Format version 5 (HDF5) [64, 47] es otro formato de fichero portátil ampliamente utilizado para almacenamiento de matrices multidimensionales. Además, también dispone de una versión paralela implementada mediante MPI. En este trabajo, nos centramos en los ficheros netCDF debido a su amplia utilización en una gran variedad de aplicaciones, de entre las que destacamos las aplicaciones climáticas que centran algunos de los ejemplos que se mostrarán en el capítulo 10.

NetCDF permite la visualización de datos desde una colección de objetos portátiles autodefinidos a los que se puede acceder de una forma sencilla. Se puede acceder directamente a los valores de los vectores sin necesidad de conocer cómo están almacenados los datos. De manera auxiliar, se puede almacenar información complementaria como el tipo de datos, el significado de los mismos, la escala, etc. El desarrollo de multitud de aplicaciones ha contribuido en la mejora del acceso a los datos y en la reutilización del software para la visualización, análisis y gestión de datos orientados a matrices.

Además, la librería netCDF implementa un tipo de datos abstracto, esto significa que todas las operaciones de acceso y manipulación de datos de un fichero netCDF deben utilizar únicamente el conjunto de funciones proporcionadas por la interfaz. La representación de los datos se oculta a las aplicaciones que utilicen la interfaz de la librería y únicamente se puede acceder a ellos y modificarlos mediante dicha interfaz. Al mantener una misma interfaz ocultando el almacenamiento físico de los datos, también se consigue independencia de la plataforma en la que se esté utilizando la librería.

La distribución estándar de netCDF proporciona las interfaces para C, Fortran 77, Fortran 90 y C++, sin embargo terceros desarrolladores proporcionan herramientas para el acceso desde otros lenguajes (Konrad Hinsen por ejemplo, con el módulo incluido en ScientificPython). La librería distribuida ha sido probada en varios de los sistemas opera-

tivos LINUX, y también en Microsoft Windows. Las fuentes de la librería netCDF están disponibles en un FTP público [90] para la distribución a la comunidad científica.

Un detalle en el que se incide de una manera especial en netCDF, es que no se trata de un sistema de gestión de base de datos. Una base de datos relacional tal y como las conocemos actualmente no se conforma como una solución adecuada para almacenar y gestionar datos de matrices. Alguna de las razones de esta afirmación puede ser que las bases de datos relacionales no soportan los objetos multidimensionales como unidad básica al acceso de datos. Además, la representación de matrices mediante relaciones entre tablas provoca un acceso incómodo con poca maniobra para la abstracción de sistemas multidimensionales. Por tanto, se necesita un modelo significativamente diferente orientado al tratamiento de matrices que permita la obtención, modificación, manipulación matemática y visualización de una forma cómoda y potente.

NetCDF proporciona un método común de acceso a datos a multitud de aplicaciones que requieren de un sistema estructurado de almacenamiento. Por ejemplo, en aplicaciones de análisis de las condiciones atmosféricas, se utiliza netCDF para almacenar una variedad de tipo de datos relativos a un único punto de observación: series temporales, mallas regularmente espaciadas e imágenes por radar o por satélite. Muchas organizaciones, incluyendo la gran parte de las dedicadas al análisis climático, utilizan este estándar para el almacenamiento, la distribución y el análisis de sus datos [92]. De este modo, la comunidad de usuarios de netCDF ha contribuido añadiendo nuevos entornos e interfaces para otros lenguajes de programación. En esta línea, el objetivo que nos hemos marcado se centra en proporcionar una interfaz desde Python para el acceso paralelo a ficheros netCDF.

9.1.1. El modelo de datos de netCDF

El modelo de datos de netCDF consiste básicamente en:

- Variables: Son matrices de datos de dimensión N. Las variables pueden ser de seis tipos diferentes (caracter, byte, entero corto, entero, coma flotante y doble coma flotante). Cada variable viene identificada con un valor entero. De este modo, la primera variable definida tiene un valor 0, la segunda 1, y así sucesivamente. Esta enumeración tiene relación directa con el orden en el cual las variables son definidas.

- Dimensiones: Describen las coordenadas de las matrices de datos de las variables. Cada dimensión posee un nombre y una longitud, sin embargo se permite establecer una longitud ilimitada para permitir la expansión de los datos de una variable. Un ejemplo práctico se corresponde con la dimensión temporal de una variable, puesto que podremos ir añadiendo nuevos datos de forma sucesiva.
- Atributos: Aportan información adicional a los ficheros o a las variables. Estos atributos suelen ser caracteres, escalares o bien vectores. No existe una limitación relativa al tamaño de los atributos, pero generalmente se espera que estos sean de un tamaño moderado.

NetCDF puede almacenar datos de multitud de aplicaciones, sin embargo fue originalmente diseñado para almacenar datos relativos a estudios ambientales. En esta línea vamos a ilustrar un ejemplo de un fichero netCDF que contenga algunas magnitudes físicas (como temperatura y humedad relativa) localizadas en unos determinados puntos indicados por la latitud, longitud, nivel de presión atmosférica y tiempo. Por tanto, en este modelo atmosférico, las magnitudes humedad y temperatura se corresponden con las variables, mientras que la longitud, la latitud, la presión atmosférica y el tiempo son dimensiones. Además, cada variable puede contener información adicional como las unidades de las variables (grados Kelvin o grados Celsius), o el origen de esos datos.

A continuación, mostramos un pequeño ejemplo que ilustra los conceptos del modelo de datos de netCDF. La notación utilizada para describir los objetos netCDF se denomina CDL (*network Common Data form Language*).

```
ejemplo_netCDF {
dimensions: // dimensiones (nombres y longitudes)
     lat = 5, lon = 10, level = 4, time = unlimited;
variables: // variables(tipos, nombres, aspecto, atributos)
     float temp(time,level,lat,lon);
     temp:long_name = "temperatura";
     temp:units = "° celsius";
     float rh(time,lat,lon);
          rh:long_name = "Humedad relativa";
          rh:valid_range = 0.0, 1.0; // min y max
// Atributos globales:
     :source = "Modelo de Salida";
```

```
data: // Asignación de datos opcional
temp = 3.2, ... , 15.1,
            ...
       1.6, ... , 2.4;
rh = .5,.2,.4,.2,.3,.2,.4,.5,.6,.7,
            ...
      0,.1,.2,.4,.4,.4,.4,.7,.9,.9;
}
```

Se observa en este ejemplo cómo se definen primero las dimensiones, y a continuación se definen las variables que dependen de una o varias dimensiones. En el caso particular de la dimensión `time`, se define como una dimensión de tipo ilimitado. De este modo, podremos añadir nuevos datos correlativos en esta dimensión de forma ilimitada. En este punto diferenciaremos entre dos tipos de variables: variables no-registro y variables registro. Las primeras son aquellas con tamaño de datos limitado y fijo, es decir, que todas sus dimensiones están delimitadas. En el caso de las variable registro, una de sus dimensiones es ilimitada y el tratamiento que se hace de estas variables en la estructura del fichero es diferente, tal y como se detallará en la siguiente sección 9.1.2.

En la última parte del ejemplo se han introducido los datos para cada una de las variables. Hemos indicado querido abreviar mediante los puntos suspensivos, indicando que en esa parte del código se introducirían los datos con las dimensiones adecuadas, es decir, con un total de `time` × `level` × `lat` × `lon` valores para la variable `temp` y `timelat` × `lon` para la variable `rh`.

Otra particularidad de netCDF son los datos de relleno. En ocasiones se necesita que determinados datos se queden "en blanco" o sin definir. Por ejemplo, si tenemos una variable que representa la temperatura en la superficie del mar, podemos preguntarnos el valor de esta variable en las zonas terrestres. Por tanto, en netCDF podemos definir un atributo llamado `Fill_Value` que expresa el valor de relleno utilizado en la variable.

En referencia a los datos, los tipos de datos utilizados en netCDF se muestran en la figura 9.1. Estos tipos han sido escogidos para ofrecer un margen adecuado entre la precisión de los datos y el número de bits necesario, es decir, su tamaño. Estos tipos de datos son independientes de la representación utilizada por la máquina y su representación es portátil e independiente de la plataforma donde se vayan a leer. Si un programa lee los datos netCDF a una variable interna, será en este momento cuándo se produce la

conversión (si es necesario), al tipo de dato interno a la máquina. Del mismo modo sucede para el proceso de escritura en un archivo netCDF.

Para realizar el acceso a los datos, en la interfaz para C y Fortran se proporcionan las interfaces de las funciones que realizan cada una de las tareas necesarias. Del mismo modo que sucede con BLACS, PBLAS, ScaLAPACK, el nombre de estas rutinas varía en función del tipo de dato al que deseamos acceder.

Denominación C	***Denominación Fortran***	***Almacenamiento***
NC_BYTE	nf_byte	entero con signo de 8 bits
NC_CHAR	nf_char	entero sin signo de 8 bits
NC_SHORT	nf_short	entero con signo de 16 bits
NC_INT	nf_int	entero con signo de 32 bits
NC_FLOAT	nf_float	coma flotante de 32 bits
NC_DOUBLE	nf_double	coma flotante de 64 bits

Figura 9.1: Tipos de datos definidos en netCDF

9.1.2. Formato del fichero netCDF

En un fichero netCDF, los datos se almacenan orientados al tratamiento de matrices y definiendo las dimensiones, las variables y los atributos. Físicamente, el fichero se encuentra dividido en dos partes: una cabecera y una matriz de datos representados en la figura 9.2. La cabecera contiene toda la información relativa a las dimensiones, los atributos y las variables, exceptuando los datos de las mismas que se almacenan en la segunda parte del fichero.

La cabecera del fichero netCDF define el número de dimensiones y el nombre y longitud de cada una de ellas. Estas dimensiones serán las utilizadas para definir el aspecto de las variables. Sólo una de las dimensiones puede ser ilimitada y será la dimensión más significativa para las variables registro.

A continuación de las dimensiones se sitúan un conjunto de atributos utilizados para describir algunas propiedades del conjunto de datos (por ejemplo, el propósito, el origen

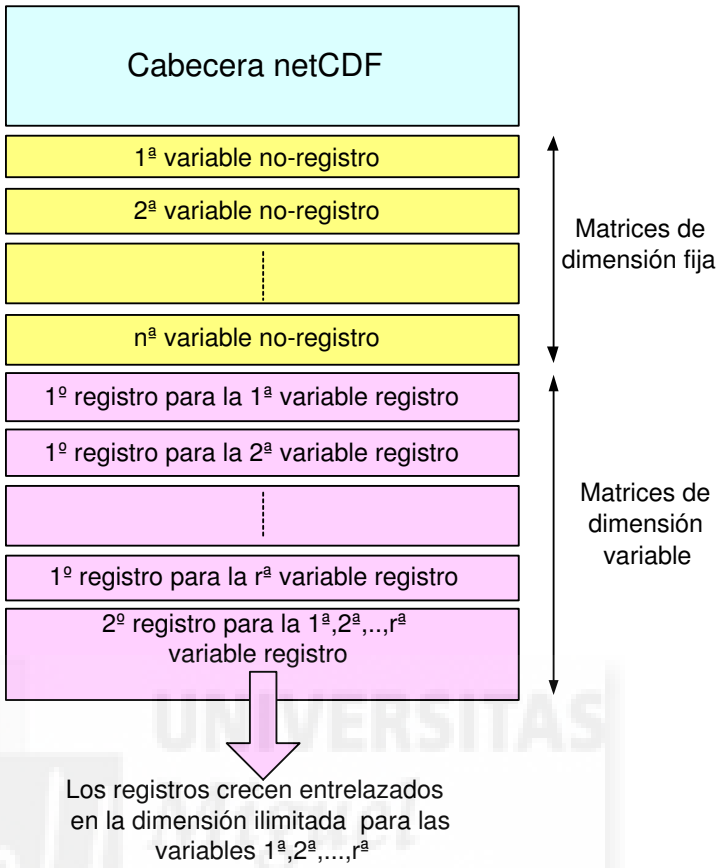


Figura 9.2: Estructura del fichero netCDF

de los datos, las aplicaciones asociadas, . . .). Este tipo de atributos se denominan globales, y se sitúan en el fichero separados de los atributos de cada variable.

Con respecto a las variables, en la cabecera se define para cada variable, su nombre, aspecto, atributos, tipo de datos y desplazamiento de los datos. Sin embargo, los datos de las variables se almacenan en el mismo orden que el definido en la cabecera pero en una parte del fichero destinada a datos.

Para poder implementar matrices de tamaño variable, en netCDF se introducen las variables registro que utilizan una técnica particular para el almacenamiento de los datos. Todas las variables registro comparten la misma dimensión ilimitada como su dimensión más significativa y se espera que el crecimiento de los datos se realice en esa dimensión. Las otras dimensiones menos significativas, definen el aspecto para cada registro de cada variable. En matrices de tamaño variable, netCDF define un primer registro de una matriz

como una submatriz con todas las dimensiones fijas. Todos los primeros registros de las variables ilimitadas se suceden en el orden definido en la cabecera. La figura 9.2 ilustra el almacenamiento tanto para variables de tamaño fijo como ilimitado.

9.1.3. La interfaz secuencial

En su origen, el estándar netCDF y su interfaz para C y Fortran fueron diseñados para realizar operaciones de acceso a datos desde un único proceso. En la librería secuencial netCDF, una secuencia típica de operaciones para escribir en un nuevo fichero netCDF sería la siguiente: crear el fichero, definir las dimensiones, las variables y los atributos, escribir los datos de la variable y finalmente, cerrar el fichero. La lectura de un fichero netCDF existente implica la abertura de la conexión a un fichero netCDF, la consulta de las dimensiones, de las variables y de los atributos, la lectura de los datos de la variable de interés, y finalmente el cierre de la conexión al fichero.

Estas operaciones que se realizan sobre un fichero netCDF pueden ser divididas en las siguientes cinco categorías [94]:

1. Funciones del conjunto de datos.

   Las operaciones que se agrupan en esta categoría son las de crear, abrir, cerrar o eliminar un conjunto de datos o fichero netCDF, establecer el modo de los datos y sincronizar los cambios realizados entre el fichero en memoria y el almacenado en disco.

2. Funciones de definición.

   Permiten la definición de las dimensiones y variables.

3. Funciones de atributos.

   Gestionan la lectura, la modificación o creación de los atributos globales o individuales de cada variable.

4. Funciones de consulta.

   Devuelven información relativa al número o nombre de las variables, las dimensiones o las atributos en el conjunto de datos para poder utilizar un índice a los objetos como si de un vector de objetos se tratase.

5. Funciones de acceso a los datos.

   Proporcionan la capacidad para la lectura o escritura de los datos a través de uno de los cinco métodos posibles:

   - Elemento único.
   - Matriz completa.
   - Submatriz o parte de una matriz.
   - Submatriz de Muestreo.
   - Submatriz referenciada.

La implementación de la interfaz secuencial para netCDF se realiza mediante llamadas de entrada/salida al sistema y posee su propio *buffer* gestionado en el espacio de memoria del usuario. Las técnicas de diseño y optimización implementadas, son útiles en la interfaz secuencial, sin embargo no pueden ser utilizadas para acceso al fichero en modo paralelo.

9.2. Librerías de acceso a netCDF desde Python

Conforme se ha descrito en el apartado anterior, la distribución de la librería netCDF se encuentra disponible mediante sus interfaces para C y Fortran. Sin embargo, puede resultar de utilidad acceder a datos desde otros lenguajes de programación como Java, Matlab, Python, etc. En el sitio de Unidata [93], se proporciona información y enlaces de terceros que han desarrollado aplicaciones o componentes para el acceso a ficheros netCDF desde diferentes lenguajes de programación. De este modo, en dicha página se indica que si deseamos acceder desde el lenguaje de alto nivel Python, podremos hacer uso de la librería Scientific Python de Konrad Hinsen que ya fue presentada en el apartado 3.2.2.

En este apartado nos introduciremos de una manera más profunda en el módulo `Scientific.IO.NetCDF` que es el que implementa el acceso a los ficheros netCDF. A continuación, describiremos cuáles son los objetos y rutinas de dicho módulo mostrando ejemplos de utilización del mismo. Ésto nos permitirá conocer cómo se pueden manejar los ficheros netCDF de una manera cómoda y sencilla y encaminará cómo deberá ser el tratamiento de las dimensiones, variables y atributos que el módulo paralelo de acceso

a ficheros netCDF deba realizar. De este modo, conseguiremos que un usuario acostumbrado a la utilización del módulo `Scientific.IO.NetCDF`, no tenga mayores dificultades en la utilización del módulo `PyPnetCDF` [54] puesto que sus objetos y rutinas son muy similares.

El módulo `Scientific.IO.NetCDF` define dos clases de objetos principales:

- Clase `NetCDFFile`

  Para crear una instancia de esta clase se deberá realizar mediante el constructor `NetCDFFile(filename, mode=‘‘ r’’)`. El primero de los parámetros, `filename`, indica el nombre del fichero al que se está accediendo o creando. El segundo parámetro, `mode`, determinará el modo en el que accedemos a dicho fichero: en modo de sólo lectura (`r`), en modo de escritura (`w`) donde un nuevo fichero será creado, en modo de lectura y escritura (`a`), en el que si no existe será creado y si existe se utilizará para leer y escribir, o en el modo indicado por `r+` muy similar al anterior pero con la exigencia de que el fichero ha de existir previamente.

  Un objeto `NetCDFFile` tiene dos atributos estándar: dimensiones (`dimensions`) y variables (`variables`). Ambos son diccionarios de objetos definidos como dimensiones y como variables. El programa no deberá modificar de forma directa estos diccionarios.

  A continuación mostramos los métodos de los que dispone esta clase:

  - `close()`
    Cierra la conexión con el fichero y por tanto la posibilidad de acceso al mismo una vez ejecutado este método.
  - `createDimension(name, length)`
    Crea una nueva dimensión con el nombre y la longitud indicada. La longitud debe ser un entero positivo o el valor `None`, que indica una dimensión ilimitada.
  - `createVariable(name, type, dimensions)`
    Crea una variable con el nombre, el tipo y las dimensiones indicadas. El parámetro `type` es una letra con el mismo significado que los tipos de datos utilizados en la librería Numeric (apartado 3.2.1). El parámetro `dimensions` es una lista con los nombres de las dimensiones de la variable, que tendrán que haber sido previamente definidas. Este método devuelve un objeto de la clase `NetCDFVariable`.

  - `sync()`
    Escribe el contenido en las memorias intermedias al fichero del disco.

- Clase `NetCDFVariable`

  Estos objetos se crean a partir de la llamada al método `createVariable` del objeto `NetCDFFile`. Los objetos de clase `NetCDFVariable` guardan mucha similitud con las matrices del módulo Númeric, a excepción de que los datos residen en un fichero. La lectura y escritura de los datos se puede realizar por indexación de una parte o la totalidad de la matriz, que puede ser expresada mediante el índice `[:]`. Los objetos `NetCDFVariable` también disponen del atributo `shape` cuyo significado es el mismo que en las matrices pero éste no puede ser modificado. Por otro lado, la propiedad `dimensions` es una lista que indica los nombres de las dimensiones de la variable y tampoco puede ser modificado.

  El resto de propiedades de este objeto se corresponden con atributos de variables (diferenciándolos de los atributos globales).

  La clase `NetCDFVariable` dispone de una serie de métodos que se describen a continuación:

  - `assignValue(value)`
    Asigna valores a una variable. Este método permite la asignación a variables escalares que no pueden utilizar indexación.
  - `getValue()`
    Devuelve los valores de una variable.
  - `typecode()`
    Devuelve el caracter identificador de tipo de dato de una variable.

A continuación, mostramos un ejemplo en el que se crea y posteriormente se lee un fichero netCDF utilizando la librería ScientificPython.

```
1 import sys, string
2 from Numeric import *
3 from Scientific.IO.NetCDF import NetCDFFile
4 #Creamos el archivo netcdf en modo escritura
5 file = NetCDFFile('test.nc', 'w')
6 file.title = "Just some useless junk"
7 file.version = 42
```

```
8  #Definimos las dimensiones
9  file.createDimension('xyz', 3)
10 file.createDimension('n', 20)
11 file.createDimension('t', None) # dimension ilimitada
12 #Definimos las variables
13 foo = file.createVariable('foo', Float, ('n', 'xyz'))
14 foo.units = "arbitrary"
15 #Asignamos datos a las variables
16 foo[:,:] = 1.
17 foo[0:3,:] = [42., 42., 42.]
18 foo[:,1] = 4.
19 foo[0,0] = 27.
20 file.close()
21 # Lectura del fichero
22 file2 = NetCDFFile('test.nc', 'r')
23 print file2.variables.keys()
24 print file2.dimensions.keys()
25 for varname in file2.variables.keys():
26         var1 = file2.variables[varname]
27         print varname,":",var1.shape,";",var1.units
28         foo = file2.variables['foo']
29         data1 = var1.getValue()
30         print "Datos:",data1
31 file2.close()
```

Mediante este sencillo ejemplo hemos podido comprobar la sencillez que aporta el interfaz para Python del módulo de Konrad Hinsen. Deseamos destacar el orden en el que se deben realizar las operaciones durante la definición de una estructura de datos, ya que primero deberemos crear el fichero (línea 5), definir algunos atributos globales (líneas 6 y 7), definir las dimensiones (líneas 9 a 11), definir las variables (líneas 13 y 14). Una vez finalizada la definición de los valores que se encuentran en la cabecera del fichero netCDF, se asignan los datos a los valores de la variable definida. En la línea 16, se observa cómo se asigna a todos los elementos de la matriz el valor 1. Sin embargo, en la línea 17 se asignan valores a las tres primeras filas, en la linea 18 se modifica la segunda columna, y en la línea 19 modificamos el primer elemento de la matriz. Finalmente, en la línea 20 se cierra la conexión con el fichero que fuerza la sincronización de los datos en memorias intermedias al fichero para posteriormente cerrar la conexión al mismo.

A partir de la línea 22, la segunda parte del código muestra la manera en la que se puede acceder a un fichero netCDF ya creado. En esta línea se crea el objeto `NetCDFFile` en modo lectura, y se imprimen las claves de los diccionarios en los que se almacenan las dimensiones y las variables (líneas 23 y 24, respectivamente). En el bucle de la línea 25, recorreremos las variables almacenadas en el fichero, en esta caso únicamente existe la variable `foo`. En la línea 27 imprimimos algunos atributos de la variable, y en la línea 28 accedemos a la variable `foo` mediante su nombre en el diccionario de variables, y copiamos su contenido a una variable Numeric temporal `data1` (línea 29). El resultado de la ejecución de este código es el siguiente:

```
jacin04 PyPNetCDF/test> python read_netcdf.py test.nc
['foo']
['xyz', 't', 'n']
foo : (20, 3) ; arbitrary
Datos: [[ 27.   4.  42.]
 [ 42.   4.  42.]
 [ 42.   4.  42.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 ...
 [  1.   4.   1.]]
```

Los puntos suspensivos mostrados en la penúltima línea representan la repetición de los datos hasta la fila 20 de la matriz `Numeric` impresa.

Podemos comprobar por tanto, la sencillez y comodidad en el acceso a ficheros netCDF mediante el módulo `Scientific.IO.NetCDF`. Para implementar la funcionalidad y la estructura del módulo `PyPnetCDF` hemos utilizado la versión secuencial de Konrad Hinsen como modelo, ya que facilita la migración de aplicaciones secuenciales a paralelas, y no altera la filosofía de acceso a los ficheros desde Python.

9.3. Parallel netCDF

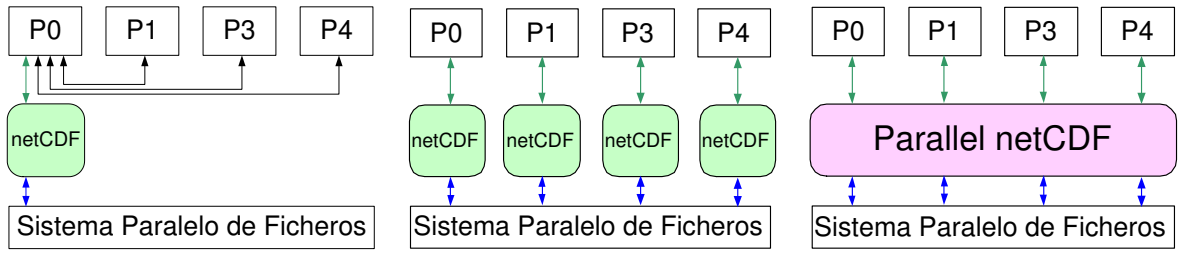
Tal y como se ha comentado en secciones anteriores, los ficheros netCDF se utilizan en un amplio margen de aplicaciones científicas a pesar de que en su origen fueron diseñados para el tratamiento de datos medioambientales. En el caso de las ciencias ambientales, el tamaño de los ficheros a tratar suele ser elevado por lo que en muchas ocasiones los requisitos de memoria de las aplicaciones son exigentes y por tanto se necesita su ejecución en plataformas paralelas que puedan hacer frente a estos requisitos computacionales y de memoria.

La interfaz netCDF fue originalmente desarrollada para aplicaciones secuenciales, no obstante se considera interesante poder acceder a los ficheros netCDF mediante técnicas apropiadas en paralelo. Con tal finalidad se ha desarrollado la librería Pnetcdf [20] que se presenta en esta sección y que utilizaremos como base para crear la librería PyPnetCDF.

A pesar de haber sido desarrollado para aplicaciones secuenciales, en la figura 9.3 se muestran las posibles implementaciones de una aplicación paralela que acceda a los datos desde un fichero netCDF. En la figura 9.3(a), un proceso (en este caso P0) es el encargado de distribuir y recoger los datos para realizar la lectura y escritura en un único fichero netCDF a través de una interfaz secuencial netCDF. Las peticiones de acceso a datos desde los otros procesos son gestionadas por un único proceso que centraliza el acceso al fichero netCDF. Esta configuración conforma un claro cuello de botella en el proceso encargado de la lectura y escritura para su correspondiente distribución de datos.

Para evitar esta limitación, se puede plantear un escenario que se ilustra mediante la figura 9.3(b). En este caso, todas las operaciones de acceso a los ficheros netCDF se realizan de forma paralela pero sobre múltiples ficheros, de modo que se dispone de un fichero netCDF para cada proceso. Esta disposición complica la gestión completa de los datos al tener que distribuirlos en diferentes ficheros con lo que se vulnera uno de los principios de netCDF: la facilidad en la gestión e integración de los datos.

Por último, se introduce una nueva librería optimizada para un entorno paralelo que permita que múltiples procesos puedan acceder y realizar operaciones de entrada y salida de datos de forma cooperativa o colectiva, a un único fichero de datos netCDF. En la figura 9.3(c) se muestra el escenario de esta tercera aproximación. En este caso, cada proceso accede en paralelo a un único fichero netCDF mediante una librería optimizada que permita un acceso concurrente con mejor rendimiento que los escenarios anteriores.



(a) (b) (c)

Figura 9.3: Utilización de netCDF en aplicaciones paralelas

9.3.1. Diseño de la interfaz

Con el objetivo de facilitar y ofrecer un acceso paralelo de alto rendimiento a ficheros netCDF, se define una nueva interfaz paralela en la librería PnetCDF. Se ha de tener en cuenta que un gran número de usuarios han desarrollado sus aplicaciones utilizando la librería secuencial de acceso a ficheros netCDF. Por tanto, la nueva interfaz de la librería paralela debe ofrecer las mismas funcionalidades de diseño tanto en el formato del fichero como en las rutinas proporcionadas en la interfaz. Además, los cambios introducidos debido a un entorno paralelo deberán ser mínimos.

La forma en la que los creadores diferencian las rutinas paralelas de las secuenciales que realizan las mismas funcionalidades se basa en añadir el prefijo `ncmpi` para las llamadas desde C, o `nfmpi` para las llamadas desde Fortran. La librería PnetCDF se construye haciendo uso de las funciones de entrada/salida de MPI (denominadas comúnmente MPI-IO), permitiendo a los usuarios beneficiarse de las optimizaciones incorporadas en las implementaciones de MPI-IO existentes. Algunas de estas implementaciones se refieren al tratamiento de los datos y estrategias de entrada/salida en dos fases [95, 105, 107, 106]. En la figura 9.4 se muestra la arquitectura implementada en la librería PnetCDF. En esta librería paralela, uno o varios de los procesos que intervienen en el grupo de comunicación puede abrir, operar y cerrar la conexión al fichero netcdf. Para permitir a distintos procesos operar sobre el mismo espacio de ficheros, especialmente sobre la información de cabecera del fichero, se han realizado una serie de cambios con respecto a la interfaz secuencial.

En las funciones de creación o abertura de un fichero netCDF, se añade un comunicador MPI en la lista de argumentos para definir los procesos que participan en la operación de entrada/salida. Describiendo el conjunto de procesos mediante el identifica-

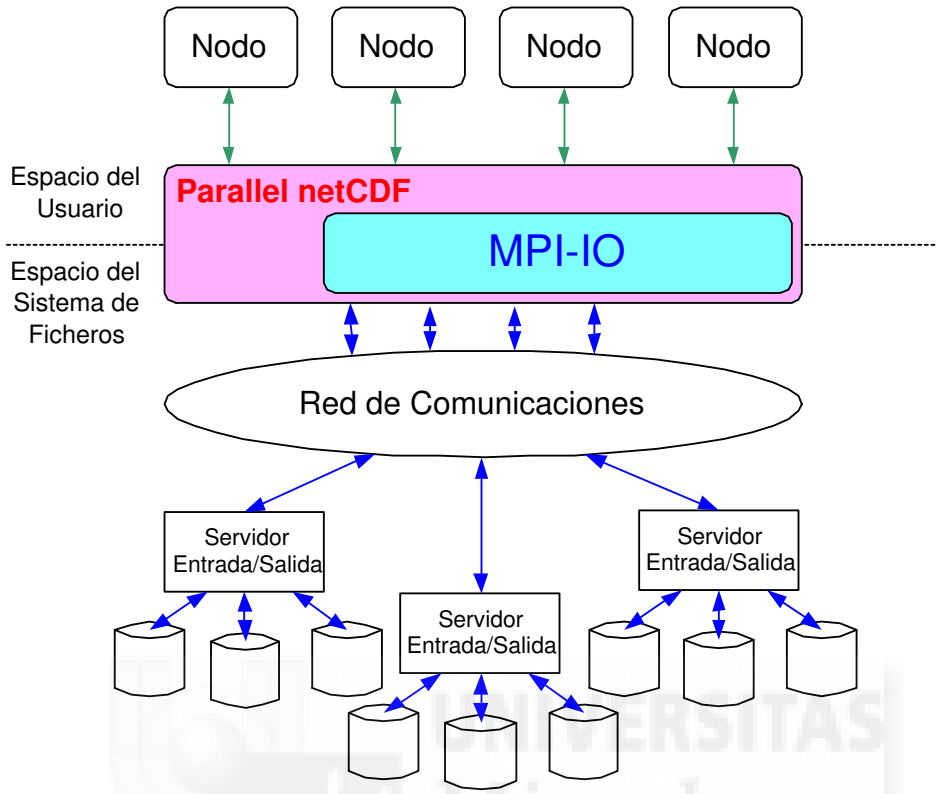


Figura 9.4: Diseño de Parallel netCDF en una arquitectura paralela

dor de comunicador MPI, se proporciona una implementación con información implícita a ese identificador que puede ser utilizada para asegurar una consistencia de los datos durante el acceso paralelo. En este tipo de llamadas también se puede añadir un objeto de información MPI que permita realizar ciertas optimizaciones en la implementación, por ejemplo `MPI_INFO_NULL` indica la ausencia de optimizaciones. Esta posibilidad permite a los usuarios de PnetCDF y de las librerías MPI-IO la posibilidad de proporcionar información con el fin de optimizar la aplicación para determinadas plataformas habilitando o deshabilitando ciertos algoritmos, o bien ajustando el tamaño de los *buffers*.

Del mismo modo que sucede en LAPACK con respecto a ScaLAPACK, entre la librería netCDF y PnetCDF se intenta mantener la misma semántica y sintaxis para las funciones de definición de modos, de atributos y de consulta (introducidas en 9.1.3). Estas funciones actúan en modo colectivo para garantizar la consistencia en la estructura de los datos en todos los procesos participantes del mismo grupo de comunicación MPI. Por tanto, todos los procesos deben llamar a las funciones de definición con los mismos valores para obtener

una definición consistente del conjunto de datos.

El mayor esfuerzo en la librería PnetCDF se centra en la paralelización de las funciones de acceso a los datos. Se proporcionan dos conjuntos de interfaces para el acceso a los datos. Las funciones de la interfaz de alto nivel tiene una gran similitud con las de la interfaz original. De este modo se consigue una fácil migración de las aplicaciones secuenciales a las paralelas. Estas llamadas, toman un único puntero a una región continua de memoria, del mismo modo que sucede con las llamadas en netCDF, y permiten el acceso a elementos de la matriz, la totalidad de la misma, partes de ella o bien múltiples partes no contiguas.

Una mejora de la interfaz paralela con respecto de la original, viene marcada porque la librería paralela proporciona la posibilidad de acceder a posiciones no contiguas de memoria. Estas regiones pueden ser descritas utilizando los tipos de datos de MPI.

El cambio más significativo con respecto a la interfaz netCDF original es la división en dos modos de acceso a datos: colectivo y no colectivo. Las funciones colectivas implican a todos los procesos en un mundo MPI y vienen identificadas con el sufijo `all`. De forma similar a MPI-IO, todos los procesos asociados a un comunicador deben llamar a las funciones colectivas para un fichero netCDF abierto. Esta exigencia no es necesaria para el caso de funciones no colectivas. Mediante la utilización de operaciones colectivas, se proporciona a la librería PnetCDF una optimización en el acceso al fichero netCDF. Los usuarios que hagan uso de la librería podrán elegir entre las funciones colectivas o no colectivas, sin embargo las mayores optimizaciones y mejoras del rendimiento en el acceso masivo a datos se logran en las del primer tipo [106].

En la figura 9.1 se muestra una tabla con una descripción de algunas de las rutinas de la librería PnetCDF. La mayor parte de las rutinas mostradas son colectivas, sin embargo el acceso a datos tanto para lectura o escritura se puede realizar de forma colectiva o no colectiva. Además, según se observa en las funciones descritas, se disponen de rutinas para la definición de la cabecera (configurando dimensiones, variables y atributos). Una vez finalizada la definición del conjunto de datos se ha de finalizar dicha definición mediante la rutina `ncmpi_enddef`. A partir de esta rutina, ya no se podrá alterar la estructura de la cabecera del fichero y deberemos entonces, introducir el contenido de los datos en la parte de datos del fichero netCDF.

De forma interna, sólo un único proceso lee y escribe en la cabecera netCDF. Sin embargo, cada proceso posee una copia de la misma almacenada en su memoria local. Las funciones de definición de modos, de atributos y de consulta trabajan sobre la copia

Escritura	
`ncmpi_create(mpi_comm,filename,0, mpi_info,&file_id)`	Creación del fichero
`ncmpi_def_var(file_id, ...)`	Definición de variables, dimensiones o atributos
`ncmpi_enddef(file_id)`	Fin de la definición de la cabecera
`ncmpi_put_vara_all(file_id,var_id,start[], count[],buffer,bufcount,mpi_datatype)`	Escritura de datos en una variable
`ncmpi_close(file_id)`	Cierre de un fichero
Lectura	
`ncmpi_open(mpi_comm,filename,0, mpi_info, &file_id)`	Abre el fichero netCDF
`ncmpi_inq(file_id, ... );`	Consulta de una variable, dimensión o atributo
`ncmpi_enddef(file_id)`	Fin de la definición de la cabecera
`ncmpi_get_vars_all(file_id,var_id, buffer,bufcount,start[],count[], stride[],mpi_datatype)`	Lectura de datos de una variable

Tabla 9.1: Ejemplos de rutinas de PnetCDF

local de la cabecera del fichero. Estas operaciones se han de realizar en modo colectivo ya que no implican necesariamente un proceso de sincronización. Sin embargo, cuando se modifica la cabecera es recomendable forzar un proceso de sincronización para verificar si los cambios han sido actualizados en todos los procesos. El concepto básico reside en permitir al proceso raíz gestionar la cabecera del fichero y difundirla a todos los procesos durante la apertura de la conexión a un fichero netCDF, durante el cierre o bien durante cualquier modificación de la cabecera. El hecho de que estas funciones de modificación de la cabecera sean colectivas garantiza que todos los procesos tendrán los cambios actualizados en sus respectivas copias.

En el otro lado se encuentran las funciones de datos. Este tipo de funciones han sido completamente modificadas para la librería paralela para conseguir un reparto adecuado de carga. Estas funciones hacen uso de las funciones incorporadas en MPI-IO por lo que se facilita la portabilidad y la optimización de la librería. Debido a que la mayor parte del tiempo de acceso un fichero netCDF se consume en el acceso a los datos, la implementación de las funciones de entrada/salida ha de ser altamente eficiente.

Para cada uno de los cinco métodos de acceso a los datos (vistos en el apartado 9.1.3), se representa el acceso a los datos como una visión de un fichero de MPI construida a partir de los parámetros pasados en las funciones de lectura o escritura. De forma particular para el acceso colectivo, cada proceso tiene una visión diferente del fichero. Todos los procesos pueden realizar un acceso MPI-IO para la transferencia de grandes cantidades de datos contiguos de forma independiente unos de otros. En algunos casos (por ejemplo en el de las variables registro), los datos se almacenan de forma entrelazada por lo que la información de continuidad se pierde. En estos casos, se puede añadir información adicional que mejore el acceso a los datos, como por ejemplo el número, el orden y los índices de los registros a los que se desea acceder para cada variable de tipo registro. De este modo se pueden optimizar las múltiples peticiones y conseguir transferencias más continuas y eficientes. Este tipo de optimizaciones se controlan mediante el parámetro "`MPI Info`" utilizado en la apertura o creación de un fichero netCDF. La experiencia en el acceso a los datos, será la que determine cual de las optimizaciones mediante "`MPI Info`" puede ser más adecuada en una plataforma determinada.

9.3.2. Ventajas y desventajas de PnetCDF

La implementación de la librería PnetCDF ofrece una serie de ventajas con respecto a trabajos similares como HDF5 [20].

En primer lugar, el diseño y la implementación de PnetCDF han sido optimizados para el acceso al formato de fichero netCDF estándar por lo que no existen diferencias entre un fichero escrito con la librería secuencial o la paralela. Además, el fichero netCDF dispone de una configuración lineal de sus datos, en la cual las matrices de datos se almacenan de forma contigua y en un orden predefinido. Esta regularidad en la configuración de los datos permite a la implementación de las funciones de entrada/salida simplificar la descripción de los datos (*buffers*, tipo de datos MPI, etc.) y gestionar todos los procesos de forma similar. Por tanto, en este tipo de operación existe una muy pequeña sobrecarga o penalización comparando el acceso mediante PnetCDF con el acceso a datos planos mediante MPI.

Por otro lado, la implementación paralela de HDF5 utiliza una estructura del fichero en árbol similar al sistema de ficheros de UNIX: los datos se almacenan de forma irregular entrelazando bloques de cabecera, bloques de datos y bloques de extensión (tanto de cabecera como de datos). Este sistema tan flexible es una ventaja para determinadas aplicaciones, sin embargo esta irregularidad dificulta el paso de los parámetros de forma directa a MPI-IO, especialmente en matrices de tamaño fijo. Además, HDF5 utiliza espacio de datos para definir la organización de los datos, y para realizar la transferencia de los datos entre la memoria y el fichero por lo que realiza un empaquetado y desempaquetado de forma recursiva. Por debajo de estas acciones, hace uso de las funciones de MPI-IO, sin embargo, estas operaciones introducen una perdida de eficiencia considerable.

En segundo lugar, la implementación PnetCDF trata de mantener la penalización en la gestión de la cabecera tan baja como sea posible. En un fichero netCDF, una sola cabecera contiene toda la información necesaria para acceder de forma directa a cada matriz de datos cuando ésta sea necesaria. Manteniendo una copia local de la cabecera del fichero netCDF en cada proceso, la librería PnetCDF evita un elevado número de consultas y sincronizaciones entre los procesos. Además, evita el acceso repetido a la cabecera de información cada vez que se necesita acceder a una matriz de datos. Toda la información de la cabecera se encuentra disponible en la copia local, de este modo los procesos de sincronización únicamente serán necesarios durante la definición del conjunto de datos de netCDF. Una vez finalizada la definición, cada matriz viene identificada por

un identificador permanente (`ID`) y puede ser accedida de forma simultánea por cualquier proceso.

Por otro lado, en HDF5 la cabecera está dispersa en bloques separados para cada objeto. En el caso de operar con un determinado objeto, se necesita iterar entre el espacio de nombres de la cabecera para acceder finalmente a ese objeto. Este tipo de acceso generalmente resulta ineficiente en plataformas paralelas. De forma particular, HDF5 define que la apertura y cierre de cada objeto ha de ser una operación colectiva por lo que fuerza a la participación de todos los procesos. Además, en algunos casos la cabecera HDF5 se actualiza durante la escritura de los datos por lo que se necesita de sincronizaciones adicionales que penalizan la eficiencia.

No obstante, la librería PnetCDF también muestra algunas desventajas o limitaciones. Al contrario que HDF5, netCDF no implementa una organización jerárquica basada en los objetos de datos. Debido a que netCDF dispone los datos en orden lineal, el añadir una matriz de datos o extender la cabecera del fichero una vez el fichero ya ha sido creado, puede resultar muy costoso puesto que se debe mover los datos existentes a un área extendida. Además, PnetCDF no implementa la funcionalidad de combinar dos o más ficheros en memoria del modo en el que lo implementa HDF5. Por otro lado, netCDF no implementa compresión de datos aunque esta funcionalidad puede ser realizada por aplicaciones externas.

9.3.3. Evaluación del rendimiento

En el presente apartado, presentamos las pruebas realizadas en las plataformas Cluster-umh y Seaborg que muestran una comparativa entre las versiones secuencial y paralela de las librerías de acceso a ficheros netCDF. Estos experimentos han sido desarrollados por nosotros, pero están basados en las pruebas descritas en [20].

Para comparar el rendimiento obtenido con la librería secuencial y con su versión paralela, se ha desarrollado un código en C que realiza la lectura y escritura de una matriz de tres dimensiones (X,Y,Z) desde y a un fichero netCDF. En este fichero, la dimensión Z es la más significativa, y la dimensión X la menos significativa. Esto indica que los elementos que comparten las mismas coordenadas (Y,Z) son contiguos en memoria. La aplicación de prueba realiza diferentes particiones en la matriz tridimensional en los ejes

X, Y, y Z, tal y como se ilustra en la figura 9.5. Todos las operaciones de lectura y escritura han sido en modo colectivo. En las figura 9.6 se representan los tiempos necesarios para realizar la lectura y escritura de un fichero netCDF de 64MB ($200 \times 200 \times 200$ elementos de coma flotante de doble precisión) en Cluster-umh y en Seaborg. Además, en Seaborg, también hemos obtenido los tiempos de ejecución de la lectura y escritura de un fichero de 512MB ($400 \times 400 \times 400$ elementos de coma flotante de doble precisión).

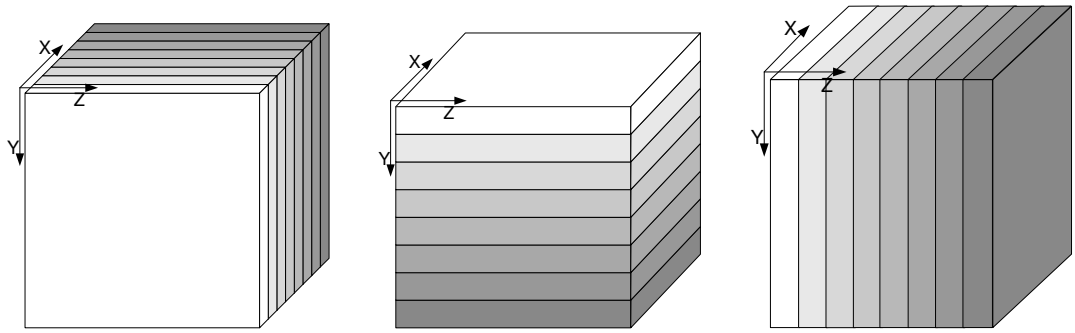
En Seaborg, el rendimiento de PnetCDF escala con el número de procesos. Debido a las optimizaciones colectivas de entrada/salida, la diferencia entre las diferentes distribuciones (desde (c) a (f) en la figura 9.5) es pequeña, aunque se aprecia que la distribución en el eje X suele ser ligeramente mejor que la realizada en el eje Y o Z debido a la continuidad de los datos en la primera distribución. La penalización o carga introducida por la comunicación entre procesos se vuelve casi despreciable cuando el tamaño del fichero es elevado.

Por otro lado, en las figuras 9.6(a) y 9.6(b) se representan los tiempos de lectura y escritura obtenidos para el cluster, respectivamente. Llama la atención que en la lectura, en la distribución en el eje X los tiempos son considerablemente menores que en el resto de ejes. Esto se debe a que el tiempo representado es el obtenido en el `nodo0` que es aquel que tiene acceso al disco y lo comparte con el resto de nodos mediante NFS. Estos detalles se tratarán en mayor profundidad en el apartado 9.4.5

La diferencia entre el rendimiento obtenido en la interfaz serie y la paralela con un procesador se debe a las diferentes implementaciones de entrada/salida y diferentes estrategias de *caching/buffering*. Tal y como muestra la figura 9.3(a), la librería secuencial ya fue utilizada en sistemas multiproceso. Sin embargo, el proceso raíz debe ser el encargado de leer y distribuir los datos o bien de recogerlos para escribirlos al fichero. En este caso, el rendimiento empeoraría de forma considerable con el incremento del número de procesos debido a la carga de comunicaciones adicional en el proceso raíz.

9.4. La distribución PyPnetCDF

En la introducción al estándar netCDF (apartado 9.1) se expone la utilidad y necesidad que cubre dicho estándar en aplicaciones de diversos ámbitos y especialmente meteorológicas. Tal y como se trata en el apartado 9.2, la librería netCDF se encuentra



(a) X (b) Y (c) Z

Figura 9.5: Distribución de los datos en las tres dimensiones (X,Y,Z)

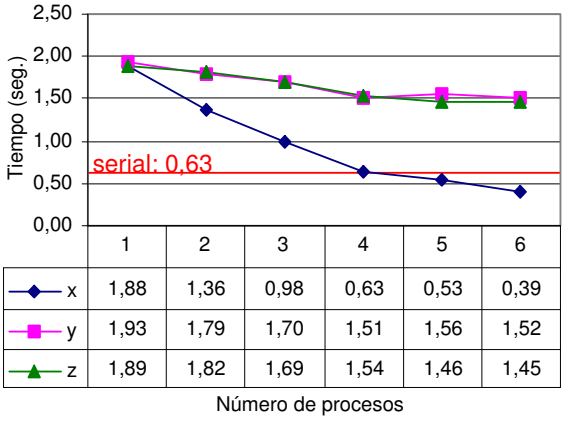
disponible, no sólo para C y Fortran, sino también para otros lenguajes de programación como Python, Java, etc.

Sin embargo, desde sus orígenes, el enfoque de netCDF es claramente secuencial. En el apartado 9.3 se introduce la librería PnetCDF como la implementación de la versión paralela de la librería netCDF. Esta librería paralela hace uso del estándar MPI para implementar un acceso concurrente de varios procesos a un mismo fichero netCDF.

Durante el desarrollo de este trabajo y de la distribución PyACTS, surge la necesidad del acceso a datos en ficheros netCDF desde Python. Hasta el momento, la única herramienta disponible para el acceso a ficheros netCDF desde el lenguaje interpretado Python está disponible en la distribución ScientificPython de Konrad Hinsen (introducida en 3.2.2 y descrita en la figura 9.2). Sin embargo, esta herramienta únicamente permite el acceso secuencial de un único proceso a un fichero netCDF y la única solución planteable hasta el momento es similar a la mostrada en 9.3(a). En esta solución, el proceso 0 será el encargado de acceder a los datos del fichero netCDF (mediante la librería ScientificPython) y de distribuirlos al resto de procesos, por ejemplo mediante `PyACTS.PyBLACS`.

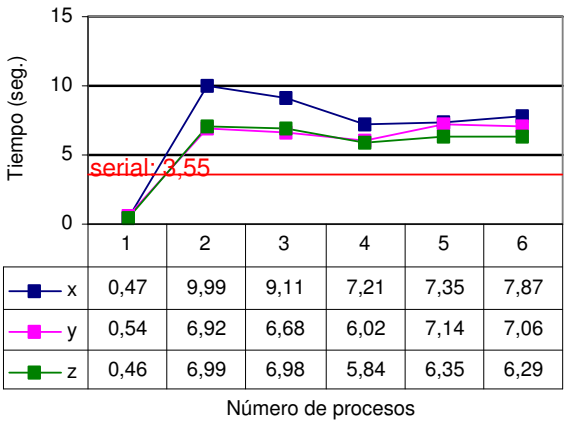
Sin embargo, esta solución no es escalable si el número de procesos crece. A medida que aumenta el número de procesos, mayor es el número de operaciones de distribución y sincronización de datos que deberá realizarse por lo que finalmente, se convierte en un cuello de botella.

En este apartado presentamos PyPnetCDF como un nuevo módulo para Python que permite el acceso concurrente a un mismo fichero netCDF en plataformas paralelas. El módulo PyPnetCDF se distribuye de forma independiente [51], sin embargo éste puede



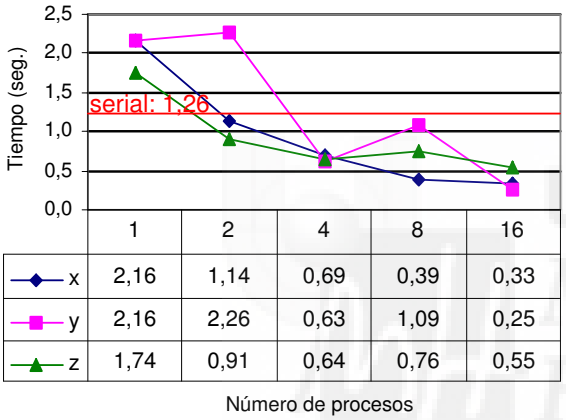
	1	2	3	4	5	6
x	1,88	1,36	0,98	0,63	0,53	0,39
y	1,93	1,79	1,70	1,51	1,56	1,52
z	1,89	1,82	1,69	1,54	1,46	1,45

(a) Cluster-umh, lectura de 64MB



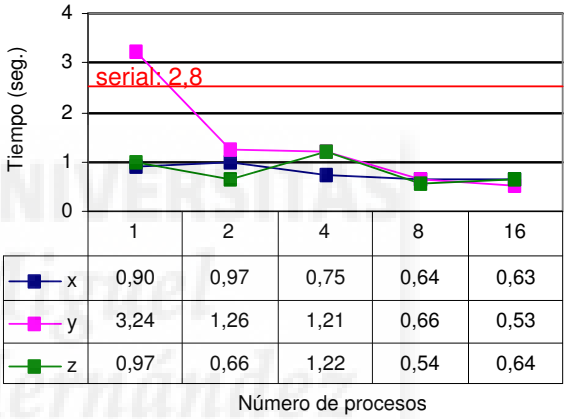
	1	2	3	4	5	6
x	0,47	9,99	9,11	7,21	7,35	7,87
y	0,54	6,92	6,68	6,02	7,14	7,06
z	0,46	6,99	6,98	5,84	6,35	6,29

(b) Cluster-umh, escritura de 64MB



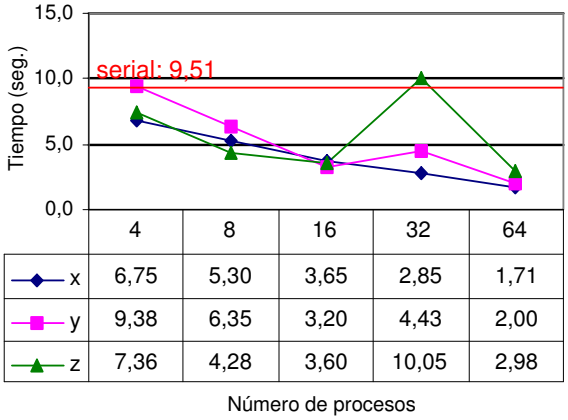
	1	2	4	8	16
x	2,16	1,14	0,69	0,39	0,33
y	2,16	2,26	0,63	1,09	0,25
z	1,74	0,91	0,64	0,76	0,55

(c) Seaborg, lectura de 64MB



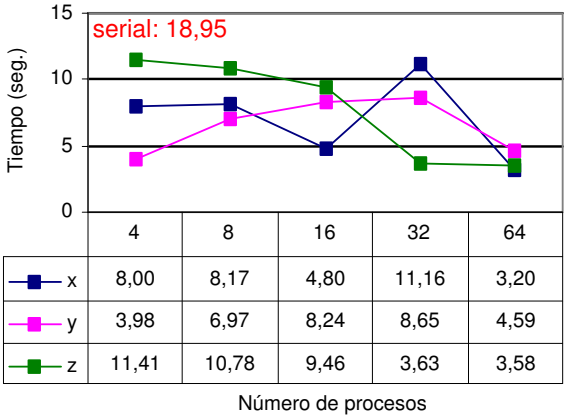
	1	2	4	8	16
x	0,90	0,97	0,75	0,64	0,63
y	3,24	1,26	1,21	0,66	0,53
z	0,97	0,66	1,22	0,54	0,64

(d) Seaborg, escritura de 64MB



	4	8	16	32	64
x	6,75	5,30	3,65	2,85	1,71
y	9,38	6,35	3,20	4,43	2,00
z	7,36	4,28	3,60	10,05	2,98

(e) Seaborg, lectura de 512MB



	4	8	16	32	64
x	8,00	8,17	4,80	11,16	3,20
y	3,98	6,97	8,24	8,65	4,59
z	11,41	10,78	9,46	3,63	3,58

(f) Seaborg, escritura de 512MB

Figura 9.6: Tiempos de lectura y escritura en las tres dimensiones (X,Y,Z)

utilizarse mediante las dos implementaciones de los intérpretes de Python descritos en el apartado 5.4: pyMPI y mpipython.

Además, presentamos el módulo PyPnetCDF como un método eficiente y complementario para la distribución PyACTS. Debido a que el estándar netCDF proporciona un acceso a matrices de datos multidimensionales almacenadas en un fichero portátil e independiente del sistema, si conseguimos que los datos almacenados en un fichero netCDF puedan ser leídos y convertidos a un distribución cíclica 2D, conseguiremos un beneficio mutuo entre ambas distribuciones. Por un lado, aquellos datos distribuidos en ficheros netCDF podrán ser analizados con las rutinas incluidas en la distribución PyACTS. Por otro lado, los ficheros netCDF se conforman como un repositorio útil y sencillo para almacenar los datos manipulados mediante la distribución PyACTS.

En las siguientes secciones, describiremos el módulo PyPnetCDF con sus principales rutinas y se mostrarán ejemplos de utilización. A continuación, describimos el proceso seguido para implementar la interfaz de Python a la librería PnetCDF e introducimos las rutinas `PnetCDF2PyACTS` y `PyACTS2PnetCDF` que implementan la lectura desde un fichero netCDF a una matriz PyACTS. Por último, se exponen las pruebas realizadas para comparar la librería PyPnetCDF con su versión secuencial de ScientificPython.

9.4.1. Descripción del módulo

Durante el presente capítulo, en un primer momento se ha presentado el estándar netCDF y la distribución de la librería con interfaces disponibles para C y Fortran. Para permitir la gestión de ficheros netCDF desde Python, hemos mostrado el módulo que incorpora la distribución ScientificPython. Sin embargo, ante las exigencias de determinadas aplicaciones, el acceso secuencial limita la escalabilidad de las aplicaciones por lo que surge la necesidad que cubre la librería PnetCDF.

Presentamos, en esta sección, el módulo PyPnetCDF que permite el acceso a ficheros netCDF desde el intérprete de Python de forma paralela a varios procesos. Este módulo implementa los interfaces necesarios, algunos de ellos generados mediante la herramienta SWIG (descrita en el apartado 3.3.1). El conjunto de objetos y rutinas que se definen en el módulo guardan un gran parecido con el módulo de acceso a datos creado por Konrad Hinsen y para conseguir tal similitud hemos añadido una "`P`" a las clases descritas en el

apartado 9.2.

Los principales objetos y rutinas que se definen en el módulo PyPnetCDF son los siguientes:

- Clase `PNetCDFFile`
  Similar a la clase `NetCDFFile` de la versión secuencial. Para construir un objeto de este tipo, se utiliza la llamada a `PNetCDFFile(file,mode)` donde ambos parámetros tienen el mismo significado que en la versión secuencial. De forma interna y oculta al usuario, al crear un objeto de esta clase, todos los procesos que pertenecen al mundo de comunicación MPI acceden o crean el mismo fichero netCDF. En el modo de lectura (`mode=r`), cada proceso analiza la estructura del conjunto de datos de netCDF y construye el objeto `NetCDFFile` leyendo y completando sus dos principales diccionarios: `dimensions` y `variables`.

  La clase `PNetCDFFile` ofrece los siguientes métodos, algunos de ellos similares a la versión secuencial:

  - `close()`
    Cierra la conexión y fuerza el sincronismo de los datos entre la memoria de cada proceso y el contenido de disco.
  - `createDimension(name,length)`
    Del mismo modo que en su versión secuencial, se crea una dimensión con el nombre y la longitud indicada. También se define una dimensión ilimitada mediante el valor `None`.
  - `createVariable(self, name, type, dimensions,dist=(0))`
    Crea una variable con el nombre, el tipo y las dimensiones indicadas. Como este método se ejecuta en un objeto de tipo `PNetCDFFile`, el parámetro `self` hace referencia al objeto que llama a dicho método, para asociarle la variable creada. Por otro lado, se dispone de un parámetro adicional y optativo `dist`. Este parámetro es una lista que nos indica cuáles de las dimensiones se utilizarán como eje para realizar la repartición de datos entre los diferentes procesos. Por ejemplo, si se define una variable en función de tres dimensiones (`pos(x,y,z)`), al indicar `dist=(0)` se repartirán los datos sobre el eje X (tal y como se muestra en la figura 9.5(a)). Por otro lado, si `dist=(2)`, la distribución tendría lugar en los ejes Z conforme se muestra en la figura 9.5(c).

- `sync()`
  Sincroniza el contenido de los datos de cada proceso con el contenido del fichero netCDF en disco. Se ha de tener en cuenta que cada proceso almacena una porción de los datos de una variable por lo que cada proceso será el encargado de sincronizar la porción de datos que dispone en memoria con la porción de datos del fichero netCDF.

- `enddef(self)`
  Una vez finalizada la definición del conjunto de datos netCDF, es decir las dimensiones, las variables y atributos, se debe ejecutar este método para sincronizar la cabecera en todos los procesos.

- Clase `PNetCDFVariable`
  Al igual que en el módulo secuencial, para crear una variable se debe utilizar el método `createVariable`. Un objeto de esta clase tiene varios atributos, como por ejemplo `shape`,`vartypecode` que indican el aspecto de los datos y el tipo de los mismos. Se ha de tener en cuenta que `shape` representa el tamaño total de la matriz, y no el tamaño de la porción de matriz que tiene cada proceso. El atributo `dist`, indicará las dimensiones en las cuales se realiza la distribución de carga.

  La clase `PNetCDFVariable` dispone de los siguientes métodos:

  - `getValue(start=0,size=0)`
    Obtiene los valores de una variable de una porción determinada indicada mediante las coordenadas `start` y `size`. Estas coordenadas son realmente listas con tantos elementos como dimensiones tenga la variable. Por ejemplo, según el fichero de datos mostrado en el apartado 9.2, si realizamos una llamada a `getValue` con `start=(10,5,5)` y `size=(10,10,10)`, estaríamos obteniendo el valor de los datos comprendidos en un cubo de arista 10 y con su vértice situado en la coordenada (10,5,5). En el caso de que los valores `start` y `size` no sean indicados, tomarán el valor por defecto y esto implicará que se obtendrán todos los datos disponibles por el proceso conforme a la distribución de datos utilizada (que depende del parámetro `dist`).

  - `setValue(start=0,size=0,data)`
    Asigna valores a una porción determinada o a la globalidad de los datos que almacena cada proceso. Los parámetros `start` y `size` tienen el mismo significado y tratamiento que en el método anterior.

- `typecode(self)`
  Devuelve el tipo de datos que contiene la matriz.
- `getlocalstart()`
  En función de la distribución realizada, obtiene la posición del primer elemento de la matriz local con respecto a la matriz global.
- `getlocalshape()`
  Obtiene el tamaño de la matriz local en función de la distribución realizada.
- `getMPI_Info()`
  Devuelve información relativa a los procesos MPI tal y como el número de procesos y el identificador de cada proceso.

A continuación, mostramos un ejemplo de utilización de PyPnetCDF en el cual volveremos a crear un fichero netCDF y posteriormente realizaremos la lectura. Podemos comparar este ejemplo con el mostrado en el apartado 9.2 y observaremos que las diferencias para el usuario son mínimas.

```
1  from Numeric import *
2  from PyPNetCDF.PNetCDF import *
3  import PyACTS
4  #Creamos el archivo netcdf en modo escritura
5  file = PNetCDFFile('test.nc', 'w')
6  file.title = "Just some useless junk"
7  file.version = 42
8  #Definimos las dimensiones
9  file.createDimension('xyz', 3)
10 file.createDimension('n', 20)
11 file.createDimension('t', None) # dimension ilimitada
12 #Definimos las variables
13 foo = file.createVariable('foo', Float, ('n', 'xyz'))
14 foo.units = "arbitrary"
15 #Asignamos datos a las variables
16 foo[:,:] = 1.
17 foo[0:3,:] = [42., 42., 42.]
18 foo[:,1] = 4.
19 foo.data[0,0] = PyACTS.iam
20 file.enddef()
21 foo.setValue()
```

```
22 file.close()
23 # Lectura del fichero
24 file2 = PNetCDFFile('test.nc', 'r')
25 print "*"*10," Proceso ",PyACTS.iam,"/",PyACTS.nprocs,"*"*10
26 print ncfile1.variables.keys()
27 print ncfile1.dimensions.keys()
28 for varname in file2.variables.keys():
29         var1 = file2.variables[varname]
30         print varname,":",var1.shape,";",var1.units
31         foo = file2.variables['foo']
32         data1 = var1.getValue()
33         print "Datos:",data1
34 file2.close()
```

Si se comparan línea a línea este código y el mostrado en el apartado 9.2, comprobaremos que la diferencia entre ambos se reduce a un par de líneas, especialmente en la importación del módulo (línea 2) y en la creación o apertura del fichero de datos (línea 5 y 24, respectivamente). Por otro lado, desde la línea 25 a 27, cada proceso imprime su identificación y los datos de la variable `foo` que almacena. Sin embargo, para entender el paralelismo de este código preferimos mostrar primero la salida de la ejecución en cuatro procesos.

```
jacin04 PyPNetCDF/test> mpirun -np 4 mpipython .read_pnetcdf.py ./test.nc
 ********** Proceso 0 / 4 **********
['foo']
['xyz', 't', 'n']
foo : (20, 3) ; arbitrary
Datos: [[ 0.  4.  42.]
 [ 42.  4.  42.]
 [ 42.  4.  42.]
 [ 1.  4.  1.]
 [ 1.  4.  1.]]
********** Proceso 3 / 4 **********
['foo']
['xyz', 't', 'n']
foo : (20, 3) ; arbitrary
Datos: [[ 3.  4.  1.]
```

```
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]]
**********  Proceso  2 / 4 **********
['foo']
['xyz', 't', 'n']
foo : (20, 3) ; arbitrary
Datos: [[ 2.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]]
**********  Proceso  1 /4 **********
['foo']
['xyz', 't', 'n']
foo : (20, 3) ; arbitrary
Datos: [[ 1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]
 [  1.   4.   1.]]
```

En este ejemplo se visualiza de una manera clara el paralelismo y la forma de ocultar al usuario este paralelismo. Se ha de tener en cuenta que todos los procesos interpretan este código, pero de forma interna cada uno accede a una parte de los datos de forma transparente al usuario. En este ejemplo, se define una variable `foo` (línea 13) de dimensiones 20×3 a partir de unas dimensiones previamente definidas (líneas 9 a 11). Internamente, a partir del número de nodos que intervienen, PyPnetCDF obtiene el tamaño local de la matriz global asignado a cada proceso, mediante el parámetro opcional `dist` podemos indicar cuál de las dimensiones se utilizará para realizar la distribución. Como por defecto `dist=0`, y en este caso no se ha especificado ningún valor de `dist`, se divide el número de filas de la matriz global entre el número de procesos y de este modo cada nodo almacena una matriz `Numeric` de tamaño 5×3. De este modo, `foo.shape` vale `(20,3)` mientras que `foo.data.shape` vale `(5,3)`.

En la línea 16 se asigna el valor `1` a la segunda columna, como todos los nodos tienen elementos que pertenecen a la segunda columna, deberán modificar el valor de dichos elementos. Por otro lado, en la línea 17 se modifica el valor de las tres primeras filas y en este caso únicamente el proceso `0/4` posee estas filas por lo que debe ser el único en actualizarlas. En resumen, disponemos de una lectura o escritura distribuida de los elementos de una variable netCDF de una forma sencilla y transparente al usuario.

Sin embargo, en determinadas ocasiones es posible que deseemos asignar un valor en las matrices locales en lugar de a la matriz local. Un ejemplo de este tipo de asignación se aprecia en la línea 19, en ella se observa que todos los nodos asignan el valor de su identificador de proceso `PyACTS.iam` al elemento `[0,0]` de su matriz local.

En resumen, consideramos que la distribución PyPnetCDF proporciona una herramienta útil, sencilla, versátil y flexible para el acceso a datos en ficheros netCDF de forma distribuida, evitando problemas de cuello de botella. En apartados siguientes mostraremos pruebas de lectura y escritura de ficheros netCDF de grandes dimensiones para comprobar la eficiencia de PyPnetCDF frente al módulo de acceso netCDF incluido en ScientificPython.

9.4.2. Creación de la interfaz

A partir de la librería PnetCDF, necesitamos crear un nuevo módulo de Python que permita acceder a los objetos y rutinas que dicha librería proporciona. Como ya se trató en el apartado 3.1, uno de los puntos fuertes de Python era su capacidad y facilidad para fusionar o incorporar módulos desarrollados en otros lenguajes de programación, especialmente C y Fortran.

En esta línea, si deseamos crear las interfaces (o comúnmente llamados *wrappers*) podremos hacerlo escribiéndolos manualmente según las líneas marcadas en [30] o bien podremos utilizar una herramienta de automatización como SWIG o F2PY que ya fueron presentadas en los apartados 3.3.1 y 3.3.2, respectivamente.

En la figura 9.7 se representa la estructura del software y de las diferentes librerías de las que hace uso el módulo PyPnetCDF. En el nivel más alto, el intérprete de Python y el usuario interactúan con el módulo PyPnetCDF (implementado mediante el fichero

`PyPnetCDF.py`). Este módulo hace uso de la librería de objetos compartidos `pypnetcdf.so`) que incluye los *wrappers* para la librería PnetCDF, y también la implementación de MPI. Podemos utilizar PyPnetCDF con otras librerías o módulos (PyScaLAPACK, por ejemplo) pero no están diseñados en la misma estructura de software.

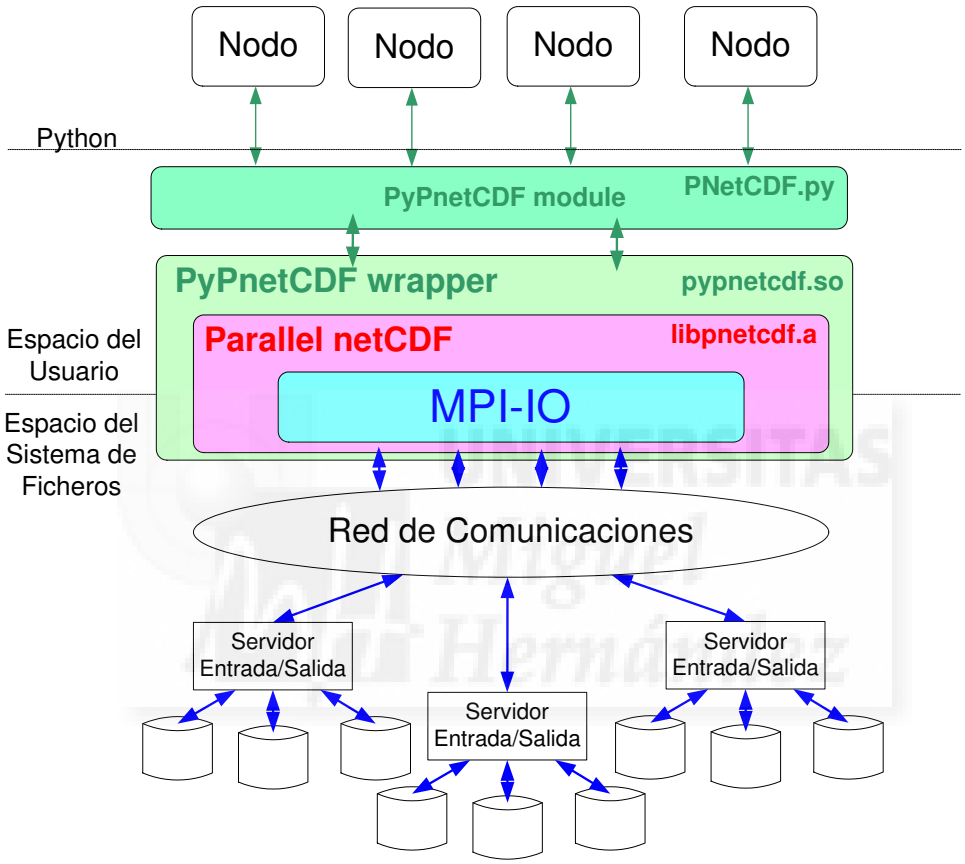


Figura 9.7: Diseño de PyPnetCDF en una arquitectura paralela

En el caso particular de PyPnetCDF, hemos utilizado la herramienta SWIG en lugar de la herramienta F2PY. En la creación de los *wrappers* para la distribución PyACTS habíamos utilizado F2PY, sin embargo en este caso, una facilidad necesaria para crear PyPnetCDF, proporcionada por SWIG, y no implementada por F2PY nos confirma que SWIG deba ser la herramienta a utilizar. Si una aplicación de C define una serie de objetos, puede ser necesario que necesitemos acceder a ellos desde Python, crearlos o bien modificarlos. Por tanto, se necesita crear unas interfaces o *wrappers* a estos objetos, y

además un conjunto de funciones necesarias en su gestión. La creación de las interfaces a objetos definidos por C es una de las ventajas de SWIG frente a F2PY por lo que nos decidimos por la primera opción.

Para crear los *wrappers* necesarios a la librería PnetCDF mediante SWIG, tendremos que escribir el archivo de definición de interfaces (`.i`). Para la definición de las interfaces, es necesario utilizar algunas directivas de SWIG, si se desea más información recomendamos la lectura del manual de usuario [8]. El código de la primera parte de este archivo se muestra a continuación y éste agrupa algunas directivas de SWIG.

```
1 %module pypnetcdf
2 %include "cpointer.i"
3 %include "carrays.i"
4 %array_class(int,intArray);
5 %array_class(char,CharArray);
6 %array_class(size_t,size_tArray);
7 %pointer_functions(int, intp);
8 %pointer_functions(size_t, size_tp);
9 %pointer_functions(nc_type, nc_typep);
```

En la primera línea se observa la definición del nombre del módulo a crear. En las líneas 2 y 3 se incluyen dos archivos de definición (`cpointer.i` y `carrays.i`). El primero de ellos proporciona las rutinas necesarias para el tratamiento de punteros en los parámetros de las rutinas. El segundo proporciona las rutinas que implementan el manejo de matrices de tipos sencillos o de matrices de objetos. En la librería PnetCDF, algunos parámetros de entrada suelen ser vectores de enteros, de caracteres o de objetos (en este caso el objeto definido como `size_t`). Para crear las interfaces adecuadas a estas cadenas, tendremos que definirlas mediante el código que se muestra desde las líneas 4 a 6. Por otro lado, en las líneas 7 a 9 se definen las funciones necesarias para gestionar el tratamiento de los punteros a los objetos definidos en las directivas.

```
1 extern MPI_Comm create_MPI_Comm();
2 extern MPI_Info create_MPI_Info();
3 int MPI_Comm_size(MPI_Comm, int *);
4 int MPI_Comm_rank(MPI_Comm, int *);
5 extern nc_type create_nc_type();
6 extern size_t create_size_t();
7 extern int convert_nc_type(nc_type nctype);
8 extern nc_type convert_int2nc_type(int num);
```

```
9  extern int convert_size_t2int(size_t tam);
10 extern size_t convert_int2size_t(int num);
11 extern size_t create_unlimited();
```

En esta segunda parte se definen las interfaces a crear en un conjunto de funciones que no pertenecen a la propia librería PnetCDF. Estas rutinas se definen en el archivo `pypnetcdftools.c`, y proporcionan funciones auxiliares bien para obtener información del entorno MPI, o bien para convertir o crear los objetos que luego serán utilizados por la rutina PnetCDF.

Por último, mostramos a continuación una muestra reducida de la tercera parte del archivo de definición de interfaces. En este código, se definen las interfaces de todas las rutinas proporcionadas por la librería PnetCDF. Como se puede observar, los nombres de estas rutinas corresponden con los nombres de las interfaces de la librería para C mostrados en la tabla 9.1.

```
1  /* Begin Dataset Functions */
2  extern int ncmpi_create(MPI_Comm comm, ... , int *ncidp);
3  extern int ncmpi_open(MPI_Comm comm, ... , int *ncidp);
4  ...
5  extern int ncmpi_begin_indep_data(int ncid);
6  extern int ncmpi_end_indep_data(int ncid);
7  extern int ncmpi_close(int ncid);
8  /* End Dataset Functions */
9  /* Begin Define Mode Functions */
10 extern int ncmpi_def_dim(int ncid, ... , int *idp);
11 ...
12 extern int ncmpi_rename_var(int ncid, ..., const char *name);
13 /* End Define Mode Functions */
14 /* Begin Inquiry Functions */
15 extern char *ncmpi_inq_libvers(void);
16 extern int ncmpi_inq(int ncid, ... , int *unlimdimidp);
17 ...
18 extern int ncmpi_inq_vardimid(int ncid, int varid, int *dimidsp);
19 extern int ncmpi_inq_varnatts(int ncid, int varid, int *nattsp);
20 /* End Inquiry Functions */
21 /* Begin _att */
22 extern int ncmpi_inq_att(int ncid, ... , size_t *lenp);
```

```
23 extern int ncmpi_inq_attid(int ncid, ... , int *idp);
24 ...
25 extern  int ncmpi_put_att_double(int ncid, ... , const double *op);
26 extern  int ncmpi_get_att_double(int ncid, ... , double *ip);
27 /* End _att */
28 /* Begin {put,get} */
29 extern int ncmpi_put_var1(int ncid, ... , MPI_Datatype datatype);
30 extern int ncmpi_get_var1(int ncid, ... , MPI_Datatype datatype);
31 ...
32 extern int ncmpi_get_vars_double_all(int ncid,... , double *ip);
33 extern int ncmpi_get_vars_double(int ncid, ... , double *ip);
34 /* End {put,get}*/
```

Una vez escrito el archivo de definición `pypnetcdf.i`, generamos el módulo para Python mediante el comando `swig -python pypnetcdf.i`. El resultado es un fichero en C `pypnetcdfmodule.c` y otro en Python `pypnetcdf.py` que definen la librería de objetos compartidos y el módulo Python que hace uso de ésta. Después de este proceso, ya podríamos importar el módulo mediante un simple `import pypnetcdf`. Sin embargo, únicamente tendríamos acceso a las rutinas proporcionadas en la librería PnetCDF sin establecer ninguna estructura de objetos, propiedades y métodos de los mismos.

Para lograr una mayor similitud entre el módulo secuencial y nuestro módulo paralelo se ha modificado el nombre del fichero `PNetCDF.py`, obteniendo así la similitud entre el módulo `NetCDF` de Konrad Hinsen y nuestro módulo `PNetCDF`.

Si deseamos proporcionar un módulo cómodo y sencillo como el proporcionado en la librería ScentificPython, debemos editar el módulo `PNetCDF.py` y crear en él las definiciones de las clases descritas en el apartado 9.4.1. Se definen en este archivo las clases `PNetCDFFile` y `PNetCDFVariable` con sus correspondientes métodos, las funciones de acceso al entorno de MPI (como por ejemplo `getMPI_Info`), las clases que se importan de los objetos definidos en la librería C (`intArray`, `intArrayPtr`, `CharArray`, `size_tArray`, etc.).

En los métodos o eventos de cada clase definida en `PNetCDF.py`, se realiza de forma interna y transparente al usuario la llamada a las rutinas de la librería `libpnetcdf.a` . Por ejemplo, mediante la sencilla asignación de un atributo global (por ejemplo, `file.Titulo = 'Titulo de prueba'`) se activa el método `__setattr__(self,name,value)` de la clase `PNetCDFFile` y será en el código de este método donde se deben realizar las llamadas

a las rutinas de asignación de atributos de la librería PnetCDF (`ncmpi_put_att_text`, `ncmpi_put_att_char`, `ncmpi_put_att_int`, etc.).

Por último, deseamos destacar una modificación adicional que se ha realizado en el código `pypnetcdfmodule.c` generado por SWIG. Una de las principales ventajas de utilización de una herramienta de este tipo es el tiempo que se ahorra en escribir muchas de las rutinas de creación de las interfaces cuya redacción resulta rutinaria y tediosa. Sin embargo, consideramos que SWIG no proporciona una interfaz adecuada en el tratamiento de matrices ni de variables de tipo Numeric. De hecho, el módulo Numeric se considera el estándar para el tratamiento de matrices en Python, y en la librería secuencial se ha visto la relación entre una variable Numeric y otra de tipo NetCDF.

En cualquiera de las rutinas de lectura y escritura de datos, éstos se pasaban mediante un simple puntero a la dirección de memoria en la que se encontraban. Este puntero no resulta una manera cómoda y sencilla de acceder a los datos desde Python. Por tanto, hemos modificado todas las funciones de acceso a datos para definir y tratar la variable de entrada que en lugar de un puntero, el dato utilizado en Python sea una variable de tipo Numeric. Será en la rutina en la que se define la interfaz a cada función, donde se debe tratar dicha variable Numeric, y obtener el puntero a su dirección de memoria para utilizarlo en la rutina de la librería PnetCDF.

A continuación se muestra una de las muchas rutinas en las que se ha realizado dicha modificación. Se ha omitido la declaración de las variables (línea 3) con el fin de no ofrecer un ejemplo demasiado extenso. Deseamos destacar en la línea 4, cómo se *recogen* cada uno de los parámetros pasados por la rutina de Python y se asignan a las variables `arg` y `obj`. Se observa en esta línea la modificación realizada, puesto que el último parámetro `a_capi` hace referencia a la variable Numeric.

```
1  static PyObject *_wrap_ncmpi_get_vara_double_all( PyObject *self,
2          PyObject *args) {
3  ...
4      if(!PyArg_ParseTuple(args,(char *)
5          "iiOOO:ncmpi_get_vara_double_all", &arg1, &arg2, &obj2,
6           &obj3, &a_capi)) goto fail;
7      if ((SWIG_ConvertPtr(obj2,(void **) &arg3, SWIGTYPE_p_size_t,
8          SWIG_POINTER_EXCEPTION | 0 )) == -1) SWIG_fail;
9      if ((SWIG_ConvertPtr(obj3,(void **) &arg4, SWIGTYPE_p_size_t,
10         SWIG_POINTER_EXCEPTION | 0 )) == -1) SWIG_fail;
```

```
11 /* Processing variable a */
12     capi_a_intent |= F2PY_INTENT_IN;
13     capi_a_tmp = array_from_pyobj(PyArray_DOUBLE, a_Dims, a_Rank,
14         capi_a_intent, a_capi);
15     if (capi_a_tmp == NULL) {
16     if (!PyErr_Occurred())
17        PyErr_SetString(pyscalapack_error,"failed in converting
18         4th argument 'data' of open to C/Fortran array" );
19      } else {
20     a = (double *)(capi_a_tmp->data);
21     }
22 /* End Processing variable a */
23     result = (int)ncmpi_get_vara_double_all(arg1, arg2,
24         (size_t const (*))arg3, (size_t const (*))arg4, a);
25     capi_buildvalue = Py_BuildValue("iN",result,capi_a_tmp);
26     return capi_buildvalue;
27     fail:
28     return NULL;
29 }
```

Desde las líneas 12 a 21 se realiza el procesado de la variable Numeric, obteniendo sus propiedades (línea 13), comprobando su validez (línea 16) y obteniendo la dirección de memoria en la que residen los datos (línea 20). Mediante estos pasos conseguimos que las matrices Numeric sirvan de continente en Python para los datos obtenidos de los ficheros netCDF.

También se desea destacar la línea 23 de este ejemplo, en la cual se realiza la llamada a la rutina de la librería PnetCDF a la que queremos acceder. El resultado de la ejecución de la rutina se almacena en la variable `result`, aunque generalmente los datos a los que apuntan algunos de los punteros utilizados modifican sus valores tras la ejecución de la rutina. Por ejemplo, en una lectura de datos desde netCDF, uno de los parámetros será un puntero que indica la dirección de memoria donde guardar esos datos. El puntero de memoria no se modificará pero sí los datos en esas posiciones de memoria.

En resumen, se ha tratado de mostrar el proceso de creación de la librería PyPnetCDF. Se ha de tener en cuenta que la creación de este módulo ha estado condicionada por varias premisas:

- Ofrecer acceso a las rutinas de la librería PnetCDF desde Python.
- Proporcionar una herramienta tan similar como sea posible a su versión secuencial para Python.
- Conseguir un módulo fácil de utilizar y de instalar en cualquier distribución de Python.
- Permitir la interacción de este módulo con el intérprete de Python y con otros módulos (Numeric, PyACTS, etc.).
- Conseguir una escalabilidad similar a la obtenida en PnetCDF sin penalización por utilizar un lenguaje de alto nivel

Los dos primeros puntos han sido logrados mediante los ejemplos y las indicaciones mostrados hasta este punto. Sin embargo, el tercer y cuarto puntos serán tratados en mayor profundidad en los apartados 9.4.4 y 9.4.5, respectivamente. También realizaremos un estudio del rendimiento obtenido comparándolo con su versión secuencial y con programas análogos implementados únicamente en lenguajes de bajo nivel.

9.4.3. Instalación

Como se ha comentado en el presente capítulo, la herramienta utilizada para crear este módulo ha sido SWIG. Conforme a la introducción realizada a SWIG en el apartado 3.3.1, una vez obtenido el archivo en C de definición de las interfaces, se debe compilar éste y todos aquellos necesarios (`pypnetcdfmodule.c`, `pypnetcdftools.c`, `fortranobject.c`) para enlazarlos finalmente en un objeto compartido (`pypnetcdf.so`) mediante un comando del siguiente tipo:

```
mpicc  -shared -o pypnetcdf.so  pypnetcdfmodule.o pypnetcdftools.o
fortranobject.o -L/home/vgaliano/parallel-netcdf-0.9.4/src/lib
-L/usr/local/mpich/lib  -lpnetcdf -lmpi
```

Por tanto, según el proceso indicado por SWIG, el usuario que deba instalar y utilizar PyPnetCDF desde su distribución de Python, deberá compilar manualmente cada uno

de los ficheros incluidos en la distribución y construir el archivo de objetos compartidos (`.so`) mediante el comando mostrado anteriormente. Además, el usuario debe copiar este archivo y los *scripts* de Python al directorio `site-packages` para una instalación completa. Consideramos que este proceso resulta bastante tedioso y en ocasiones complicado para usuarios de Python que no conocen en profundidad los conceptos de compilación y construcción de librerías.

Por esta razón, hemos simplificado considerablemente el proceso de instalación del mismo modo que lo hicimos en la distribución PyACTS, es decir, mediante un archivo de instalación `setup.py`. Este archivo de instalación nos permite, mediante una configuración adecuada de las ubicaciones de los archivos de inclusión y de las librerías, compilar e instalar el módulo PyPnetCDF mediante un simple comando `python setup.py install`.

A continuación se muestra el contenido del archivo `setup.py` que contiene la información necesaria para una correcta compilación e instalación.

```
1  from distutils.core import setup, Extension
2  library_dirs_list=['/usr/local/mpich/lib',
3                     '/usr/local/parallel-netcdf-0.9.4/src/lib',
4  libraries_list = ['pnetcdf','mpi',]
5  include_dirs_list = ['./inc','/usr/lpp/ppe.poe/include',
6          '/usr/local/parallel-netcdf-0.9.4/src/lib',]
7  module_pypnetcdf = Extension( 'PyPnetCDF._pypnetcdf',
8                     [
9                     'src/fortranobject.c',
10                    'src/pypnetcdftools.c','src/pypnetcdfmodule.c'],
11                    libraries = libraries_list,
12                    library_dirs=library_dirs_list,
13                    include_dirs = include_dirs_list,
14                    )
15 setup (name ="PyPnetCDF",
16         version ="1.0.0",
17         description ="Parallel access to NetCDF files from Python",
18         author = "Vicente Galiano, Jose Penades, Violeta Migallon",
19         author_email = "vgaliano@umh.es",
20         url = "http://www.pyacts.org/pypnetcdf",
21         package_dir = { 'PyPnetCDF':'src/PyPnetCDF'},
22         packages = ["PyPnetCDF"],
```

```
23          ext_modules =[module_pypnetcdf]
24          )
```

Tal y como se muestra en el código, aquel usuario que desee instalar el módulo PyPnetCDF, deberá configurar primero los directorios en los que se encuentran las librerías necesarias (línea 2). En la linea 4 se indican el nombre de las librerías necesarias `libpnetcdf.a` y `libmpi.a`. Para el proceso de compilación de las fuentes, también se deberá indicar la localización de los archivos de inclusión en la línea 5. A continuación, en la línea 7 se definen los componentes del módulo en los que el usuario no deberá realizar modificaciones. Finalmente, en la línea 15 se realiza la llamada a la *construcción* e instalación del módulo mediante las rutinas proporcionadas en el módulo `distutils` importado.

9.4.4. Interacción con la distribución PyACTS

En el apartado 5.5.4, se presentaron las rutinas `PNetCDF2PyACTS` y `PyACTS2PNetCDF` como parte de la distribución PyACTS. Se ha de tener en cuenta que los ficheros netCDF permiten el almacenamiento de matrices de datos de tipo denso de grandes dimensiones mediante una interfaz cómoda y sencilla. Como ya hemos visto, la distribución PyACTS incorpora un amplio conjunto de rutinas en sus tres módulos actuales PyBLACS, PyPBLAS y PyScaLAPACK que permiten el intercambio y manipulación de matrices de tipo denso de grandes dimensiones en una plataforma paralela.

Las rutinas que proponemos `PNetCDF2PyACTS` y `PyACTS2PNetCDF` implementan la lectura paralela de datos desde un fichero netCDF a matrices PyACTS y viceversa. En resumen, permiten una interacción entre ambas distribuciones y convierte a la distribución PyPnetCDF en una herramienta cómoda y sencilla para la lectura y escritura de los datos utilizados en los algoritmos que utilicen PyACTS.

A continuación se muestra un ejemplo de utilización de estas rutinas de conversión que nos permiten leer en paralelo desde ficheros netCDF y almacenar el contenido de las variables en matrices PyACTS para poder operar con ellas mediante llamadas a las rutinas incluidas en la distribución PyACTS. La operación realizada en este código será la proporcionada por la rutina `PyPBLAS.pvgemm` ($\alpha op(A)op(B) + \beta C \rightarrow C$), donde los datos

de origen se obtendrán de un fichero netCDF y el resultado se almacenará en otro fichero netCDF.

```
1  from PyACTS import *
2  import PyACTS.PyPBLAS as PyPBLAS
3  from PyPNetCDF.PNetCDF import *
4  from Numeric import *
5  #Inicializamos la malla de procesos
6  PyACTS.gridinit(mb=2,nb=2,nprow=2,npcol=2)
7  #Abrimos el fichero de origen de datos
8  file = PNetCDFFile('data_pvgemm.nc', 'r')
9  a = file.variables['a']
10 b = file.variables['b']
11 c = file.variables['c']
12 #Leemos los datos del fichero netCDF a variables PyACTS
13 ACTS_lib=1 # 1=Scalapack
14 alpha1=PNetCDF2ScalPyACTS(file.alpha[0],ACTS_lib)
15 beta1=PNetCDF2ScalPyACTS(file.beta[0],ACTS_lib)
16 a1=PNetCDF2PyACTS(a,ACTS_lib)
17 b1=PNetCDF2PyACTS(b,ACTS_lib)
18 c1=PNetCDF2PyACTS(c,ACTS_lib)
19 #Llamamos a la rutina de PyPBLAS
20 c1= PyPBLAS.pvgemm(alpha1,a1,b1,beta1,c1)
21 #Creamos el fichero de resultados Results File in NetCDF Format
22 file2 = PNetCDFFile('result_data_pvgemm.nc', 'w')
23 file2.title = "Resultados de la operacion pvgemm"
24 file2.version = 1
25 for dim in file.dimensions:
26         file2.createDimension(dim, file.dimensions[dim])
27 c_result = file2.createVariable('c_result', c.vartypecode,
28         c.dimensions)
29 file2.enddef()
30 #Escribe los resultados en un fichero netCDF
31 c_result=PyACTS2PNetCDF(c1,c_result)
32 file2.close()
33 file.close()
34 PyACTS.gridexit()
```

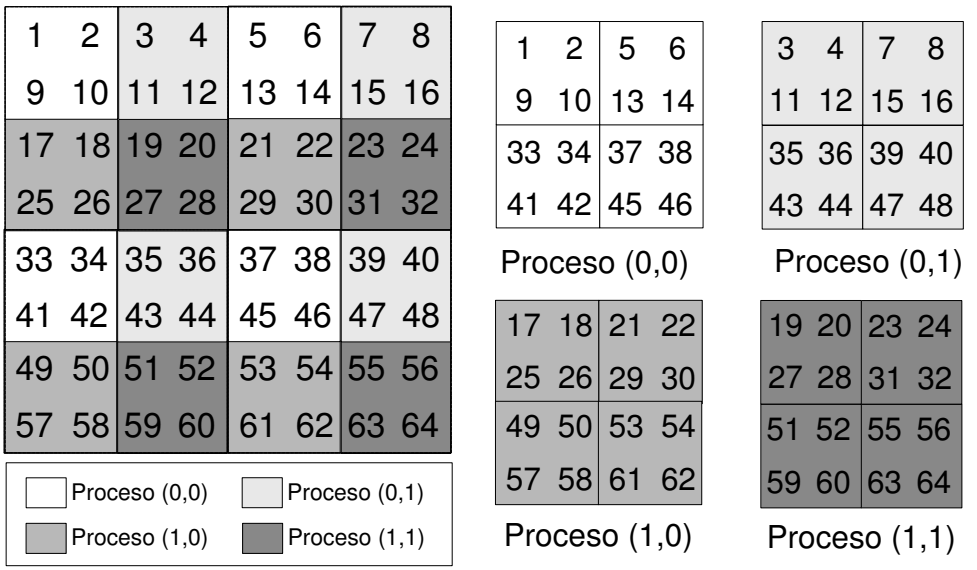
Desde la línea 1 a la 4, observamos la importación de los módulos necesarias en este ejemplo. Se desea destacar que el módulo PyPnetCDF y el módulo PyACTS pertenecen a dos distribuciones diferentes por lo que se debe distinguir entre una matriz PyACTS y una variable netCDF.

Supondremos que este ejemplo lo almacenamos como el archivo `netCDF.py`, y que se ejecuta en 4 procesos mediante el comando `mpirun -np 4 mpipython netCDF.py`. En la línea 6, se inicializa la malla de procesos a un tamaño 2×2, con un tamaño de bloque 2×2. A continuación, leeremos los datos desde un fichero de origen (`data_pvgemm.nc`) y para ello abrimos una conexión de lectura mediante el comando indicado en la línea 8. Los datos de origen están almacenados en el fichero netCDF mediante tres variables (`a`, `b` y `c`). En las líneas 8, 9 y 10 se asignan cada una de estas variables del fichero a un identificador para un tratamiento más cómodo. En las líneas 16 a 18, se realiza la conversión de los datos desde cada variable netCDF a cada matriz PyACTS que se denominarán en este ejemplo como `a1`, `b1` y `c1`. Además, el fichero netCDF de origen de datos contiene dos atributos `alpha` y `beta` que se convierten en dos escalares PyACTS (líneas 14 y 15). Se ha de tener en cuenta que la conversión de las matrices y escalares desde los ficheros netCDF a las matrices PyACTS se realiza de modo distribuido, es decir, no existe ningún proceso dedicado a la lectura de los datos y su posterior distribución.

En la línea 20 se realiza la llamada a la rutina de PyPBLAS, y el resultado se almacena en la variable `c1`. El siguiente paso consiste en almacenar el resultado contenido en la variable PyACTS en un nuevo fichero netCDF denominado `result_data_pvgemm.nc`. Para ello primero se crea dicho fichero en el modo de escritura (línea 22), y se crean las mismas dimensiones que existían en el fichero de origen de datos mediante el bucle de la línea 25 a 26. Una vez definidas las dimensiones, se define la variable `c_result` que almacenará el resultado de la misma. Mediante la llamada a la rutina `PyACTS2PNetCDF`, cada proceso almacena los datos que posee en el fichero netCDF y en su ubicación correspondiente de forma totalmente transparente al usuario.

En la parte final del ejemplo, se deben cerrar las conexiones con los dos ficheros netCDF utilizados, y también se debe liberar la malla de procesos.

Mediante el ejemplo anteriormente mostrado se pretende ilustrar la integración lograda entre PyACTS y PyPNetCDF. El conjunto de operaciones que se realizan de forma transparente al usuario permiten lograr un mecanismo cómodo, sencillo y potente para gestionar la entrada y salida de datos de operaciones implementadas mediante PyACTS.



(a) Asignación de datos cíclica 2D (b) Matriz local de datos en cada proceso

Figura 9.8: Representación de la conversión de NetCDF a PyACTS

Sin embargo, deseamos describir de forma breve e ilustrada mediante la figura 9.8 el proceso de lectura y conversión desde los datos de un fichero netCDF a una matriz PyACTS. Supongamos que los datos almacenados en cualquiera de las variables netCDF (`a`, `b` o `c`) se muestran en la figura 9.8(a). En ella, la variable es una matriz de tamaño 8×8. Conviene recordar que las dimensiones de cada matriz vienen definidas en la cabecera netCDF. Como se ha comentado anteriormente, hemos ejecutado el código en 4 procesos configurando una malla 2×2 y definiendo un tamaño de bloque 2×2. A partir de la distribución cíclica 2D, se asignan los bloques de datos a cada proceso, por tanto cada proceso debe encargarse de leer (mediante las llamadas a la librería `pnetcdf.a`) las porciones de la matriz global. Posteriormente, la lectura de estos datos se almacenará en la posición correspondiente de una matriz local previamente creada y cuyas dimensiones se calculan en función del tamaño global de la matriz, del tamaño de bloque y de la configuración de la malla de procesos.

Por ejemplo, el proceso (1,0) realizará la lectura en la matriz global del bloque de datos a partir de la coordenada (0,2) y de tamaño (2,2) y copiará este bloque a partir de la posición (0,0) de su matriz local. Posteriormente, leerá partir del bloque de datos (0,6) y lo copiará a partir de la posición (0,2) de la matriz local. Este proceso se repite tantas veces como bloques asignados a cada proceso. Una vez finalizado este procedimiento,

cada proceso contiene en su matriz local la parte de datos que le corresponden según la distribución cíclica 2D que se muestra en la figura 9.8(a). Sin embargo, hemos de recordar que la disposición de los datos en memoria para la distribución PyACTS ha de seguir el orden definido en Fortran para las matrices. Es decir, las matrices han de estar ordenadas por columnas y no por filas tal y como se muestra en la figura 9.8(b). Este detalle se puede solucionar de forma sencilla mediante la llamada a la rutina `_fastCopyAndTranspose` incluida en el módulo `Numeric.LinerAlgebra` que permite la obtención de la traspuesta de una matriz de una manera eficiente.

En definitiva, mediante el código mostrado en este apartado se ilustran las facilidades proporcionadas para la gestión de ficheros netCDF desde Python en un entorno paralelo y también para su integración con el resto de módulos de altas prestaciones, especialmente la distribución PyACTS.

9.4.5. Pruebas de rendimiento

Además de las facilidades en el acceso a ficheros netCDF y de la capacidad de interacción entre diversos módulos para Python, debemos cuestionarnos si el rendimiento de PyPnetCDF es adecuado y si es escalable con el número de procesos.

Para comprobar la escalabilidad del módulo, se han realizado una serie de pruebas en dos plataformas con arquitectura diferente en su sistema de ficheros : Cluster-umh y Seaborg. Por un lado, Cluster-umh está compuesto por 6 computadores conectados mediante un conmutador Gigabit Ethernet. En esta configuración, el computador principal (llamado `nodo0`) comparte el directorio `home` con el resto de nodos (`nodo1` a `nodo5`) mediante NFS versión 3. Incidimos en que la versión utilizada de NFS es la número 3 porque las versiones anteriores no permiten la escritura múltiple en un fichero y la librería PnetCDF informa de la necesidad de utilizar la versión 3. En resumen, el `nodo0` lee directamente del disco duro mientras que el resto de nodos acceden o compiten por una unidad compartida de red. Por tanto, durante las pruebas que mostramos, el `nodo0` consume sus recursos en dos tareas: primero, el intérprete mpipython accede al disco duro mediante la librería PnetCDF para leer la información del fichero local, y segundo, el demonio `nfsd` debe atender las solicitudes del resto de nodos que solicitan datos del fichero compartido.

Por otro lado, en Seaborg el sistema de almacenamiento GPFS es distribuido y paralelo

con nodos adicionales exclusivamente destinados a GPFS. Por tanto, todos los nodos que acceden a un fichero netCDF tienen las mismas características y es posible el acceso múltiple en paralelo a un mismo fichero.

Para realizar las pruebas hemos utilizado fundamentalmente dos scripts, uno para la generación y escritura de un fichero netCDF y otro para la lectura de otro fichero. A continuación se muestra el script que genera ficheros netCDF de distinto tamaño.

```
1  from PyPNetCDF.PNetCDF import *
2  from RandomArray import *
3  from Numeric import *
4  import time,sys,os
5  #Dimension of Arrays
6  total=int(sys.argv[1])
7  inc=int(sys.argv[2])
8  eje_dist=int(sys.argv[3])
9  numprocs,iam=getMPI_Info()
10 for i in range(1,total+1):
11   n=i*inc
12   time0=time.time()
13   filenc='testing_write_'+str(n)+'_sec.nc'
14   file = PNetCDFFile(filenc, 'w')
15   file.title = "Data to test writing performance"
16   file.version = 1
17   file.createDimension('x', n)
18   file.createDimension('y', n)
19   file.createDimension('z', n)
20   a = file.createVariable('a',Float,('x','y','z'),dist=eje_dist)
21   file.enddef()
22   time1=time.time()
23   a_temp=Numeric.ones(a[:,:,:].shape,Float)
24   time2=time.time()
25   a[:,:,:]=a_temp[:,:,:]
26   time3=time.time()
27   a.setValue()
28   time4=time.time()
29   file.sync()
30   time5=time.time()
```

```
31    file.close()
32    time6=time.time()
33    print    n,";",numprocs,";",time1-time0,";",time2-time1,";",
34          time3-time2,";",time4-time3,";",time5-time4,";",
35          time6-time5
```

En este primer script se recogen tres parámetros de la línea de comandos en las líneas 6 a 8: `total` indica el número de ficheros con tamaño incremental que deseamos crear, `inc` indica el incremento de tamaño de cada dimensión en los sucesivos ficheros netCDF y `eje_dist` establece el valor al parámetro `dist` de la variable PnetCDF creada en la línea 20. En la línea 10 se define el bucle de los sucesivos tamaños a crear para medir tiempos en cada uno de ellos y en la línea 13 se compone el nombre de dicho fichero en función del tamaño y del número de procesos que hemos inicializado en la ejecución.

En la línea 14 se crea el fichero en modo escritura y en las dos líneas siguientes definimos atributos globales. En la línea 17, 18 y 19 se observa la definición de tres dimensiones `x`, `y`, y `z`. A continuación, se define la variable `a` con las dimensiones creadas, de tipo `Float` (que se corresponde con coma flotante de doble precisión, es decir 8 bytes) y con los datos distribuidos a lo largo de la dimensión indicada. Relacionando el parámetro `eje_dist` con la figura 9.5, la distribución 9.9.5(a) se obtiene mediante `eje_dist=0`, la distribución 9.9.5(b) se consigue con `eje_dist=1` y la distribución 9.9.5(c) la obtenemos con `eje_dist=2`. En el momento de la definición de la variable, además de escribir la información en el fichero, se crea en memoria una matriz de ceros de un tamaño definido a partir de las dimensiones utilizadas y de la distribución realizada. Por ejemplo, si `n=100`, `numprocs=4` y `eje_dist=0`, el tamaño de la matriz multidimensional creada para almacenar los datos en memoria sería de `(25,100,100)` en cada proceso.

En la línea 23 se crea una matriz `a_temp` del tamaño local de `a` pero cuyos elementos valen 1. Llama la atención la forma de obtener el tamaño local de `a` mediante una llamada a todos sus elementos `a[:,:,:]`. Sin embargo, tal y como se ha explicado este tipo de llamadas es similar a una consulta del tipo `a.data.shape`. Si cada proceso quiere consultar el tamaño global de `a` lo puede hacer obteniendo el valor de `a.shape`.

En la siguiente acción (línea 25), se copian los valores de la matriz de unos a la matriz en memoria de NetCDF. Este proceso representa únicamente la copia en memoria de ambas matrices, mientras que la escritura en el fichero de datos se realiza mediante la llamada a `setvalue` de la línea 27. Antes de cerrar el fichero, debemos realizar una llamada a la sincronización del mismo (línea 29) y en caso contrario la propia llamada

a `close` (línea 31) realizaría una sincronización si esta no ha sido realizada. Esta acción comprueba y verifica que todos los procesos han realizado todas las tareas de escritura en el fichero.

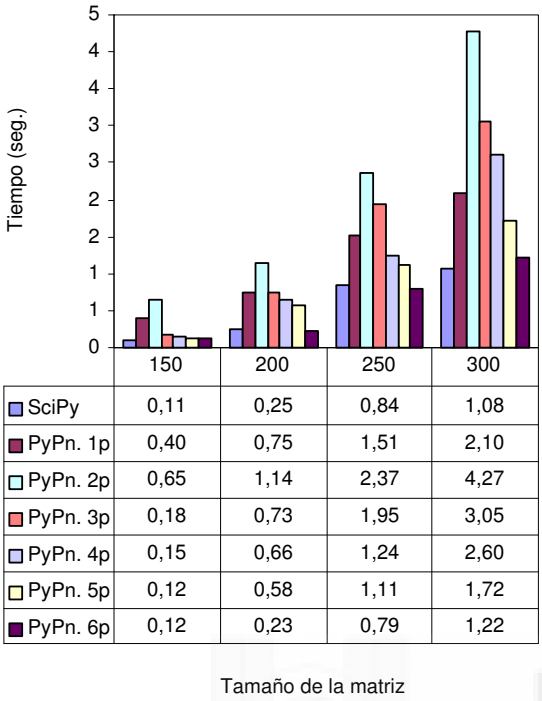
Por último, en la línea 33 cada proceso imprime en la salida los tiempos invertidos en cada uno de los pasos descritos anteriormente y que nos permitirán generar las gráficas que mostraremos a continuación.

En las líneas 12, 22, 24, 26, 28 , 30 y 32 realizamos una llamada a la rutina `time` para poder medir la diferencia de tiempos entre las sucesivas acciones que se desarrollan en el script. En los intervalos de tiempos delimitados por los temporizadores se realizan las siguientes acciones:

- `time1 - time0`: Definición de la cabecera.
- `time2 - time1`: Creación de una variable `Numeric` de unos.
- `time3 - time2`: Copia de los valores de la matriz creada a la variable netCDF que reside en memoria.
- `time4 - time3`: Escritura de los valores de la variable en el fichero netCDF.
- `time5 - time4`: Sincronización de la escritura de todos los procesos.
- `time6 - time5`: Cierre de la conexión al fichero netCDF.

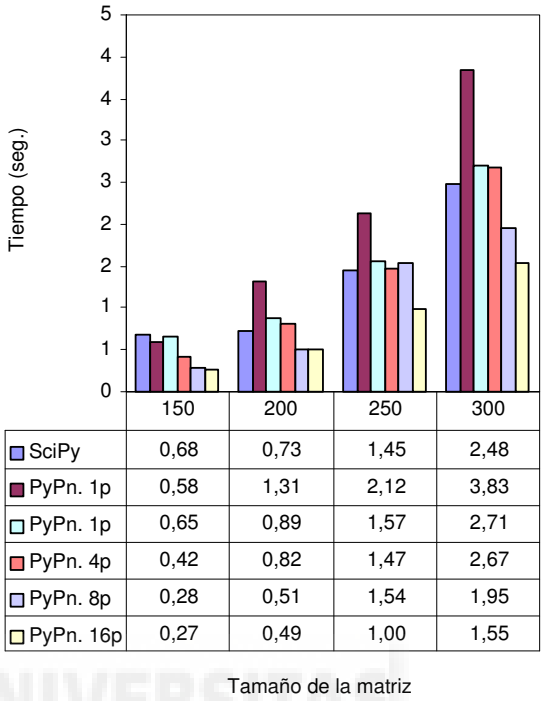
En la figura 9.9 se representan los tiempos invertidos en escribir los datos (es decir, `time4 - time3`) para diferentes tamaños de la matriz multidimensional `a` y diferente número de procesos. Cabe mencionar que para minimizar el espacio en esta figura y en siguientes, hemos utilizado las abreviaturas "SciPy" y "PyPn." que hacen referencia a ScientificPython y PyPnetCDF, respectivamente. En ambos sistemas la distribución de los datos se realiza en la primera dimension `x` (`dist=0`). En la figura 9.9(a) mostramos los tiempos obtenidos en Cluster-umh mientras que en la figura 9.9(b) representamos los tiempos obtenidos en Seaborg. Se observa en ambas figuras que en la primera fila se indican los tiempos obtenidos con la implementación secuencial de acceso a ficheros netCDF para Python proporcionada por Konrad Hinsen y descrita en el apartado 9.2.

En la figura 9.9(a) se observa que en Cluster-umh los tiempos de la herramienta secuencial son mejores que los obtenidos con la herramienta en paralelo e incluso con dos procesos se obtienen tiempos considerablemente superiores en proporción. Estos tiempos



	150	200	250	300
SciPy	0,11	0,25	0,84	1,08
PyPn. 1p	0,40	0,75	1,51	2,10
PyPn. 2p	0,65	1,14	2,37	4,27
PyPn. 3p	0,18	0,73	1,95	3,05
PyPn. 4p	0,15	0,66	1,24	2,60
PyPn. 5p	0,12	0,58	1,11	1,72
PyPn. 6p	0,12	0,23	0,79	1,22

(a) Cluster-umh



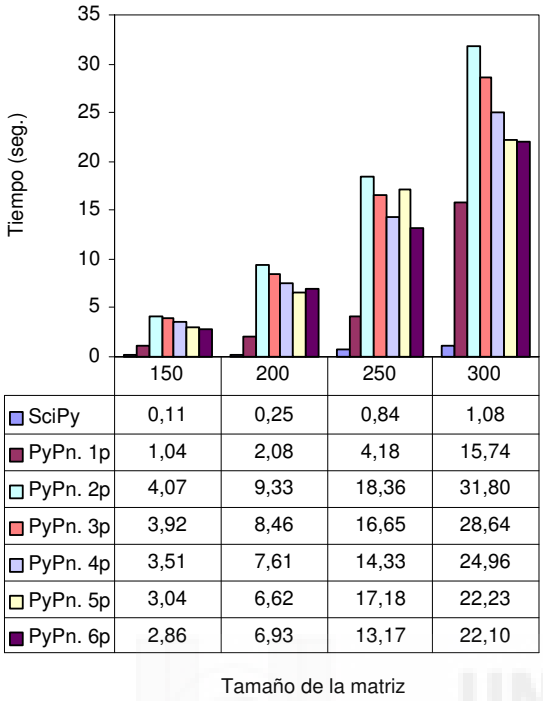
	150	200	250	300
SciPy	0,68	0,73	1,45	2,48
PyPn. 1p	0,58	1,31	2,12	3,83
PyPn. 1p	0,65	0,89	1,57	2,71
PyPn. 4p	0,42	0,82	1,47	2,67
PyPn. 8p	0,28	0,51	1,54	1,95
PyPn. 16p	0,27	0,49	1,00	1,55

(b) Seaborg

Figura 9.9: Tiempos de escritura sobre un fichero netCDF en paralelo

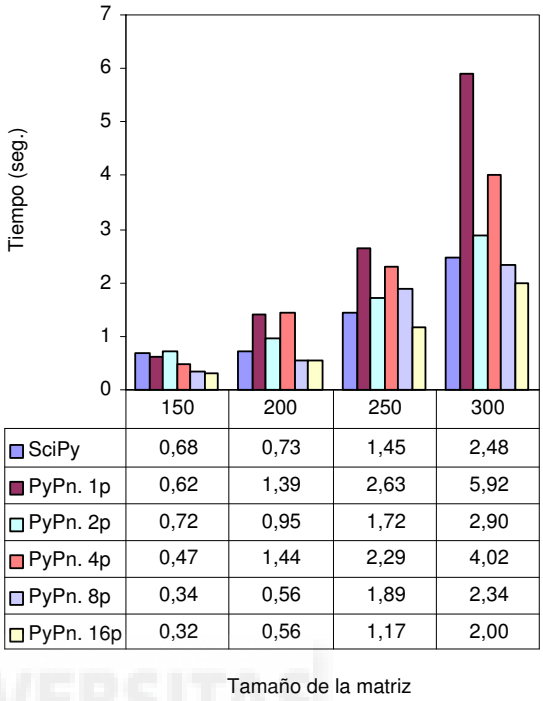
obtenidos con PyPnetCDF disminuyen para un mismo tamaño a medida que aumentamos el número de procesos. Por otro lado, en Seaborg (figura 9.9(b)), los tiempos obtenidos con 2 o 4 procesos son similares a los tiempos obtenidos con la herramienta secuencial, y para un número mayor de procesos obtenemos tiempos de escritura menores. La razón de esta disparidad de comportamientos que se observa entre Cluster-umh y Seaborg se debe a las propias arquitecturas de los sistemas de ficheros anteriormente explicadas.

Como ya se ha comentado, en el Cluster-umh el `nodo0` además de escribir su parte de datos de la matriz distribuida `a`, deberá gestionar las peticiones de acceso al fichero mediante el demonio `nfsd`. Por tanto, el resto de nodos compiten por acceder al sistema de ficheros para escribir los datos. Esta situación puede provocar que unos nodos (fundamentalmente el `nodo0`) finalicen antes el acceso al fichero que otros. Por tanto, la verdadera escritura de los datos del fichero se produce cuando todos los nodos han finalizado el proceso de escritura. Para verificar que ésto ha sucedido llamamos a la rutina `sync` que garantiza la sincronización de cabecera y datos entre todos los nodos que acceden al fiche-



	150	200	250	300
SciPy	0,11	0,25	0,84	1,08
PyPn. 1p	1,04	2,08	4,18	15,74
PyPn. 2p	4,07	9,33	18,36	31,80
PyPn. 3p	3,92	8,46	16,65	28,64
PyPn. 4p	3,51	7,61	14,33	24,96
PyPn. 5p	3,04	6,62	17,18	22,23
PyPn. 6p	2,86	6,93	13,17	22,10

(a) Cluster-umh



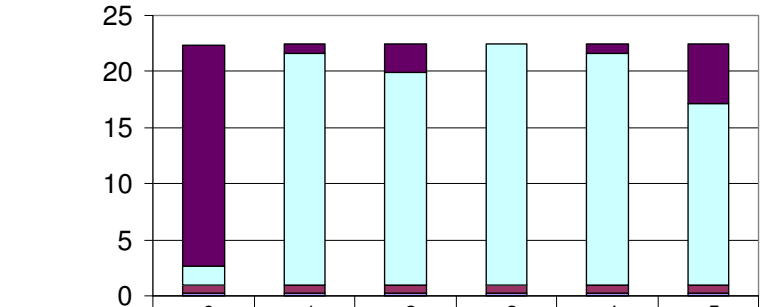
	150	200	250	300
SciPy	0,68	0,73	1,45	2,48
PyPn. 1p	0,62	1,39	2,63	5,92
PyPn. 2p	0,72	0,95	1,72	2,90
PyPn. 4p	0,47	1,44	2,29	4,02
PyPn. 8p	0,34	0,56	1,89	2,34
PyPn. 16p	0,32	0,56	1,17	2,00

(b) Seaborg

Figura 9.10: Tiempos de escritura y sincronización sobre un fichero netCDF en paralelo

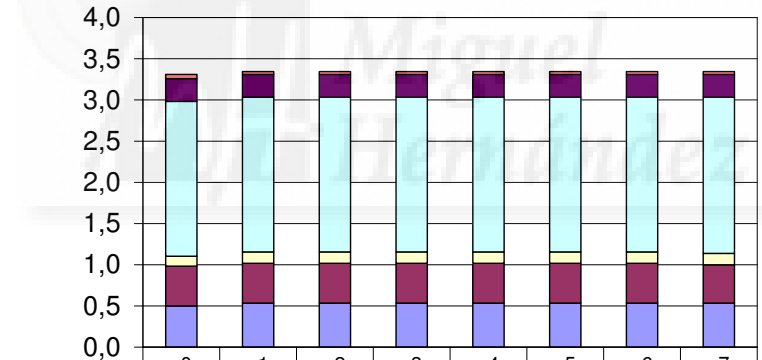
ro netCDF. En la figura 9.10 se muestran estos tiempos globales de escritura (`setvalue` + `sync`) para ambas plataformas. Si se comparan estos tiempos con los obtenidos para la herramienta secuencial se observa que en Cluster-umh son considerablemente superiores. Este comportamiento es normal y esperado en un sistema de ficheros compartidos, puesto que en el caso secuencial el único proceso no debe competir por el acceso al fichero y se realiza la escritura al mismo en un único momento. No obstante, en Seaborg el comportamiento es significativamente diferente y los tiempos globales de escritura de PyPnetCDF son inferiores a la herramienta secuencial para un número de procesos mayor o igual a 4.

Esta diferencia entre los comportamientos de Cluster-umh y Seaborg se ilustra también en la figura 9.11. En ella se muestran los tiempos invertidos por cada proceso (indicado mediante p0,p1, ...) para generar un fichero netCDF donde `n=300` y para cada una de las tareas descritas en el ejemplo mostrado anteriormente: definición de la cabecera, creación de la variable `Numeric` local de unos, copiar los valores a la matriz de la variable netCDF, escribir los datos en el fichero, sincronizar todos los procesos y cerrar el acceso al fichero.



	p0	p1	p2	p3	p4	p5
Close	0,00	0,00	0,00	0,00	0,00	0,00
Sync.	19,74	0,83	2,43	0,00	0,81	5,30
Set Val.	1,67	20,56	18,97	21,40	20,59	16,10
Cop.Var.	0,08	0,08	0,08	0,08	0,08	0,08
Creac.Num.	0,66	0,66	0,66	0,66	0,66	0,66
Def.Hdr.	0,25	0,27	0,27	0,27	0,27	0,27

(a) Cluster-umh con 6 procesos



	p0	p1	p2	p3	p4	p5	p6	p7
close	0,05	0,05	0,05	0,05	0,05	0,05	0,05	0,05
Sync.	0,28	0,28	0,28	0,28	0,28	0,28	0,27	0,28
Set Val.	1,88	1,88	1,88	1,88	1,88	1,88	1,87	1,89
Cop.Var.	0,13	0,13	0,13	0,13	0,13	0,13	0,13	0,14
Creac.Num.	0,49	0,49	0,49	0,49	0,49	0,49	0,49	0,46
Def.Hdr.	0,49	0,54	0,54	0,54	0,54	0,54	0,54	0,54

(b) Seaborg con 8 procesos

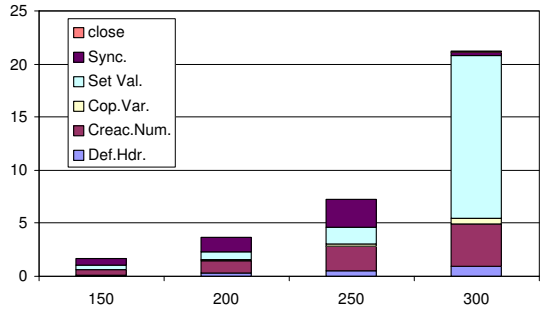
Figura 9.11: Tiempos por tarea en la escritura de un fichero netCDF en todos los procesos

En la figura 9.11(a), el `nodo0` invierte poco tiempo en la escritura del fichero puesto que accede al sistema local de directorios. Sin embargo, en la sincronización debe esperar a que el resto de nodos hayan finalizado su escritura en el fichero netCDF compartido. Una vez que el último nodo (`nodo3`) ha finalizado la escritura, se completa la sincronización y todos los procesos continúan la ejecución. Vale la pena destacar que los nodos que acceden al fichero netCDF mediante la red Gigabit Ethernet compiten por el medio y realizan la escritura en el sistema de ficheros del `nodo0` de un modo desordenado. Por esta razón, se observa en la gráfica que desde que finaliza el `nodo0` hasta que finaliza el `nodo5`, los cinco nodos han tratado de acceder al medio compartido y han enviado sus paquetes de forma desordenada.

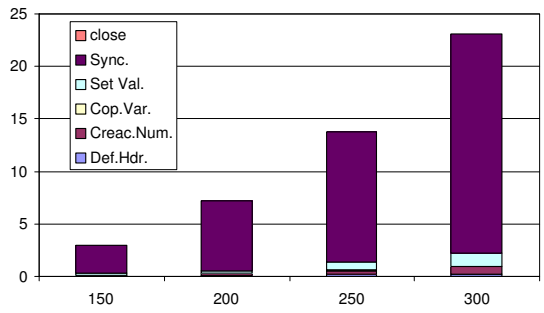
Sin embargo, en la figura 9.11(b) se representa la generación de un fichero del mismo tamaño en Seaborg con 8 procesos. Llama la atención que las acciones se realizan en todos los procesos de forma totalmente concurrente, incluso la escritura de datos en el fichero netCDF. Además, se observa que todos los nodos invierten el mismo tiempo en la escritura de datos al fichero netCDF y no existen unos nodos con más privilegios o con características diferentes al resto. Al tener un acceso paralelo al fichero, el tiempo necesario para realizar el proceso de sincronización es muy reducido y el mismo en todos los nodos.

Otra forma de representar los tiempos consumidos en cada una de las tareas se muestra en la figura 9.12. En Cluster-umh se muestran los tiempos invertidos por el `nodo0` cuando se ejecuta la escritura por sólo un proceso (es decir, sólo `nodo0`) (figura 9.12(a)) o por los seis nodos (figura 9.12(b)). Se observa que el tiempo total no difiere mucho entre 1 o 6 procesos, sin embargo cuando se ejecuta en seis procesos, los tiempos invertidos en generar, copiar y escribir los datos son inferiores pero ha de finalizar la sincronización de una forma correcta y ésto consume la mayor parte del tiempo.

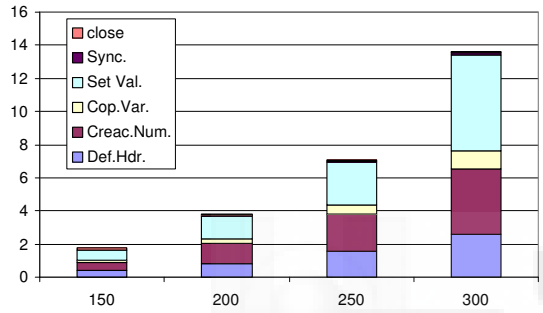
Por otro lado, en las figuras 9.12(c) y 9.12(d) se representan los tiempos invertidos en Seaborg cuando la ejecución se realiza con 1 o con 16 nodos respectivamente. En este caso, los tiempos obtenidos con 16 procesos son considerablemente inferiores a los obtenidos con un único proceso. Además, el tiempo invertido en generar y escribir una matriz `Numpy` del tamaño global es un proceso es muy superior al necesario si esta generación y escritura se realiza en los 16 procesos. Estos tiempos se visualizan de una forma clara comparando ambas gráficas. En definitiva, en Seaborg se consigue una mejora considerable de los tiempos de escritura gracias a que la arquitectura de ficheros está diseñada para un acceso concurrente.



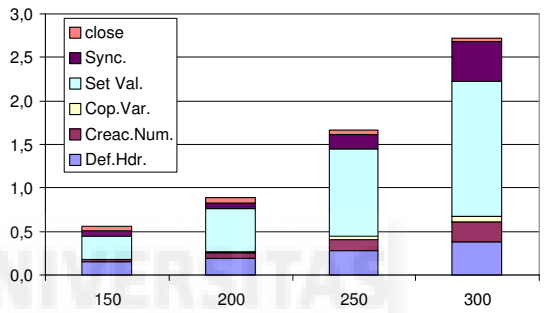
(a) Cluster-umh, 1 proc.



(b) Cluster-umh, 6 proc.



(c) Seaborg, 1 proc.



(d) Seaborg, 16 proc.

Figura 9.12: Tiempo consumido por trabajo para diferente número de procesos en la escritura de un fichero netCDF

Hasta este momento únicamente se ha analizado el comportamiento de la escritura paralela en ficheros netCDF proporcionada por PyPnetCDF. Para obtener el comportamiento de la librería PyPnetCDF en la lectura concurrente de un fichero netCDF hemos implementado el siguiente script.

```
1 from PyPNetCDF.PNetCDF import *
2 from RandomArray import *
3 from Numeric import *
4 import time,sys
5 total=int(sys.argv[1])
6 inc=int(sys.argv[2])
7 eje_dist=int(sys.argv[3])
8 numprocs,iam=getMPI_Info()
9 for i in range(1,total+1):
```

```
10    n=i*inc
11    time0=time.time()
12    filenc='testing_write_'+str(n)+'_sec.nc'
13    file = PNetCDFFile(filenc, 'r',distvars=eje_dist)
14    time1=time.time()
15    a=file.variables['a']
16    time2=time.time()
17    data= a.getValue()
18    time3=time.time()
19    file.close()
20    time4=time.time()
21    if iam==0:
22      print n,";",time1-time0,";",time2-time1,";",time3-time2,
23            ";",time4-time3
```

Este código es considerablemente más sencillo que el visto para la escritura paralela, donde los parámetros pasados por la línea de comandos `total`, `inc` y `eje_dist` tienen el mismo significado que en el código anterior. Se ha de tener en cuenta que los ficheros netCDF han sido previamente generados. En la línea 13 se accede al fichero netCDF en modo lectura y de forma transparente al usuario, la librería PyPnetCDF analiza la cabecera del fichero y obtiene las dimensiones, las variables y los atributos globales de cada variable definida en el archivo. En la línea 15 asignamos al identificador `a` la variable con ese nombre del diccionario de variables, esta asignación no representa coste computacional importante. Será en la línea 17 donde se realice la llamada a la lectura de los datos de la variable netCDF mediante la llamada a su método `getvalue()` y por tanto, se acceda a disco para la lectura de los elementos de la matriz. Este método devuelve una matriz `Numpy` de dimensiones locales conforme a la distribución realizada de la matriz global. En la línea 19 se cierra la conexión al fichero y se finaliza el acceso al mismo.

Del mismo modo que en la escritura de datos, en las líneas 11, 14, 16, 18 y 20 hemos introducido unos temporizadores para poder delimitar el tiempo empleado en cada una de las acciones que se realizan. Estas acciones se pueden resumir del modo siguiente:

- `time1 - time0`: Acceso al fichero netCDF en modo lectura.
- `time2 - time1`: Asignación de una variable del diccionario a un identificador.
- `time3 - time2`: Lectura de los datos de la variable.

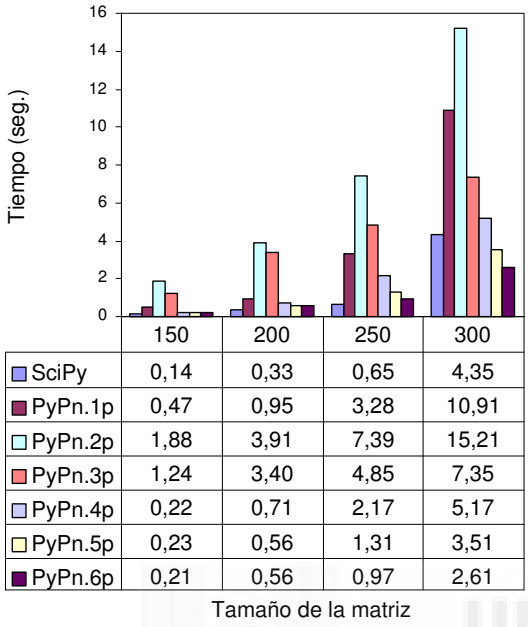
- `time4 - time3`: Cierre del acceso al fichero.

En la figura 9.13 se representan los tiempos de lectura (línea 17) en el `nodo0` y en el `nodo1` del Cluster-umh (figura 9.13(a) y 9.13(b), respectivamente) y en uno de los nodos de Seaborg, para diferentes tamaños de la variable `a` de tres dimensiones escrita en las pruebas anteriores. En este caso, hemos querido mostrar el `nodo1` además del `nodo0` para reflejar los tiempos de acceso a datos de un proceso que accede en red y no sólo en local como ocurre con el `nodo0`. Además, la distribución de datos en la lectura se realiza en el eje X, y por tanto `dist=0`.

Comparando los tiempos de lectura obtenidos entre el `nodo0` y el `nodo1`, observamos que no son excesivamente diferentes aunque el `nodo0` suele obtener tiempos de lectura inferiores en todos los casos. Con respecto a la implementación secuencial, tanto en el `nodo0` como en el `nodo1` se obtienen mejores tiempos con la herramienta secuencial, sin embargo se ha de tener en cuenta que en la figura no se muestra los tiempos necesarios para realizar la distribución de los datos una vez se hayan leído con ScientificPython. Por otro lado, en Seaborg únicamente representamos los tiempos obtenidos en uno de sus nodos puesto que el comportamiento es el mismo en cualquiera de ellos gracias a la arquitectura de su sistema de ficheros. Además, comparando los tiempos obtenidos con respecto a un acceso secuencial al fichero netCDF se obtienen mejores tiempos a partir de dos procesos, llegando incluso a la tercera parte con 16 procesos.

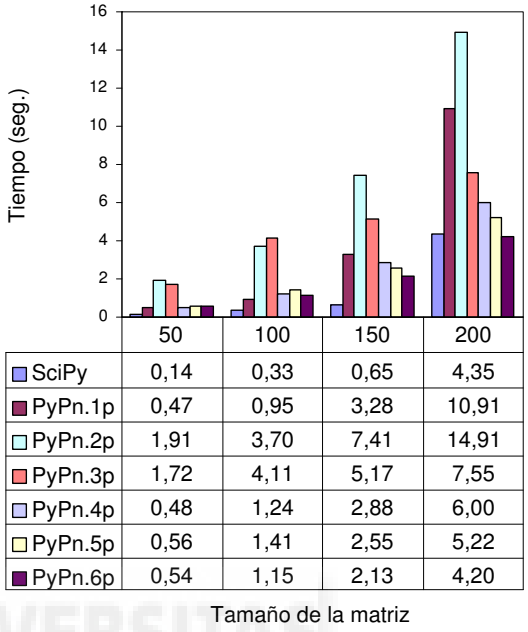
Para reflejar las acciones simultaneas que realiza cada proceso en la lectura a un mismo fichero netCDF de una variable $300 \times 300 \times 300$, mostramos en la figura 9.14 los tiempos invertidos en cada proceso (`p0, p1, ...`) para realizar cada uno de los pasos comentados en el script. Se observa en la figura 9.14(a), que el `nodo0` de Cluster-umh es el primero en finalizar la lectura y el resto de nodos finalizan la lectura de una forma desordenada. Este proceso es no bloqueante, es decir, cuando cualquiera de los procesos ha finalizado la lectura de los datos, puede seguir con la ejecución del script independientemente del resto. Por otro lado, en la figura 9.14(b) se muestran los tiempos invertidos para cada uno de los 8 procesos que ejecutan la lectura de un mismo fichero netCDF en Seaborg. De igual modo que sucedía en la escritura, los procesos acceden de manera simultánea al fichero y finalizan prácticamente en el mismo instante de tiempo.

Hasta el momento hemos supuesto en todos las pruebas mostradas que la distribución de los datos se realizaba en la primera de las dimensiones de la matriz `a`, y por tanto indicábamos dicha distribución mediante `dist=0`, o lo que es lo mismo, la distribución



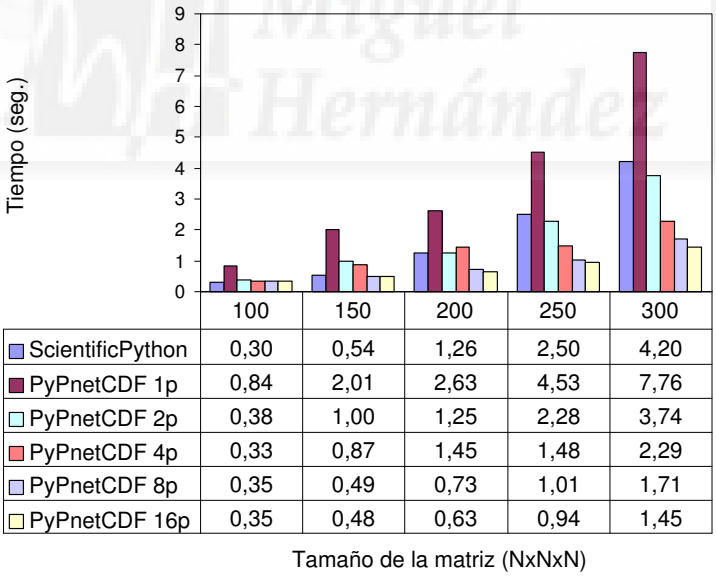
	150	200	250	300
SciPy	0,14	0,33	0,65	4,35
PyPn.1p	0,47	0,95	3,28	10,91
PyPn.2p	1,88	3,91	7,39	15,21
PyPn.3p	1,24	3,40	4,85	7,35
PyPn.4p	0,22	0,71	2,17	5,17
PyPn.5p	0,23	0,56	1,31	3,51
PyPn.6p	0,21	0,56	0,97	2,61

(a) Cluster-umh, `nodo0`



	50	100	150	200
SciPy	0,14	0,33	0,65	4,35
PyPn.1p	0,47	0,95	3,28	10,91
PyPn.2p	1,91	3,70	7,41	14,91
PyPn.3p	1,72	4,11	5,17	7,55
PyPn.4p	0,48	1,24	2,88	6,00
PyPn.5p	0,56	1,41	2,55	5,22
PyPn.6p	0,54	1,15	2,13	4,20

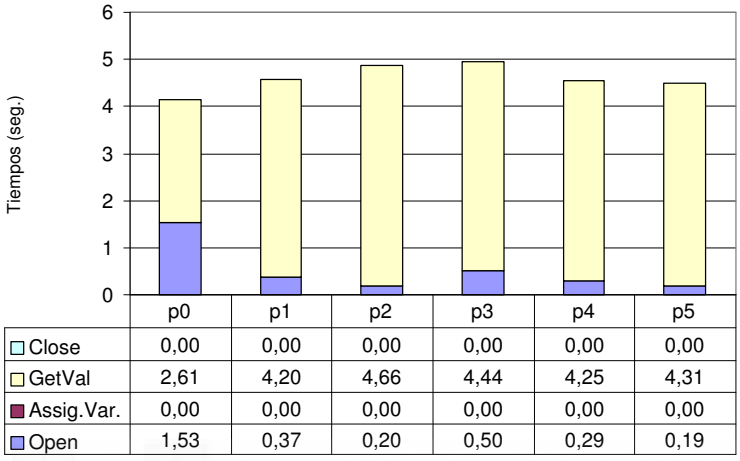
(b) Cluster-umh, `nodo1`



	100	150	200	250	300
ScientificPython	0,30	0,54	1,26	2,50	4,20
PyPnetCDF 1p	0,84	2,01	2,63	4,53	7,76
PyPnetCDF 2p	0,38	1,00	1,25	2,28	3,74
PyPnetCDF 4p	0,33	0,87	1,45	1,48	2,29
PyPnetCDF 8p	0,35	0,49	0,73	1,01	1,71
PyPnetCDF 16p	0,35	0,48	0,63	0,94	1,45

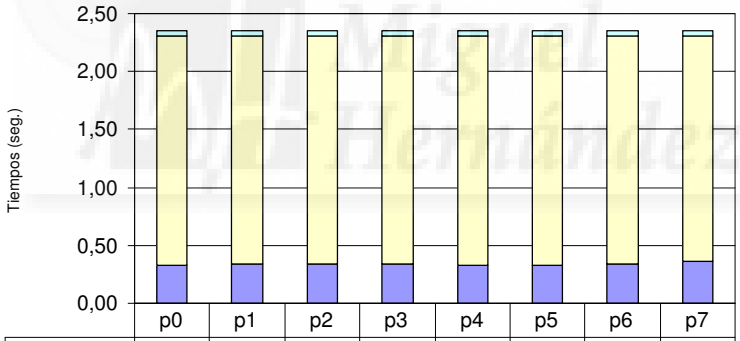
(c) Seaborg

Figura 9.13: Tiempos de lectura de un fichero netCDF en paralelo



	p0	p1	p2	p3	p4	p5
Close	0,00	0,00	0,00	0,00	0,00	0,00
GetVal	2,61	4,20	4,66	4,44	4,25	4,31
Assig.Var.	0,00	0,00	0,00	0,00	0,00	0,00
Open	1,53	0,37	0,20	0,50	0,29	0,19

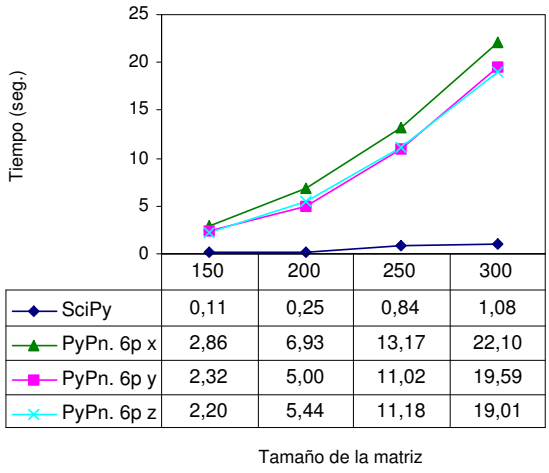
(a) Cluster-umh, 6 procesos



	p0	p1	p2	p3	p4	p5	p6	p7
Close	0,04	0,04	0,04	0,04	0,04	0,04	0,04	0,04
GetVal	1,98	1,97	1,98	1,98	1,98	1,98	1,98	1,96
Assig.Var.	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
Open	0,33	0,34	0,34	0,33	0,33	0,33	0,33	0,36

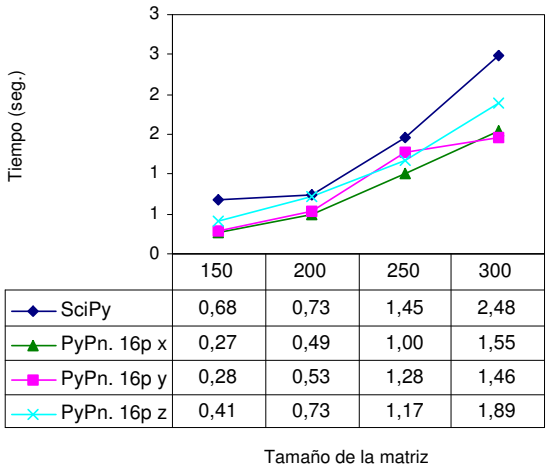
(b) Seaborg, 8 procesos

Figura 9.14: Tiempos por tarea en la lectura de un fichero netCDF en todos los procesos



	150	200	250	300
SciPy	0,11	0,25	0,84	1,08
PyPn. 6p x	2,86	6,93	13,17	22,10
PyPn. 6p y	2,32	5,00	11,02	19,59
PyPn. 6p z	2,20	5,44	11,18	19,01

(a) Cluster-umh



	150	200	250	300
SciPy	0,68	0,73	1,45	2,48
PyPn. 16p x	0,27	0,49	1,00	1,55
PyPn. 16p y	0,28	0,53	1,28	1,46
PyPn. 16p z	0,41	0,73	1,17	1,89

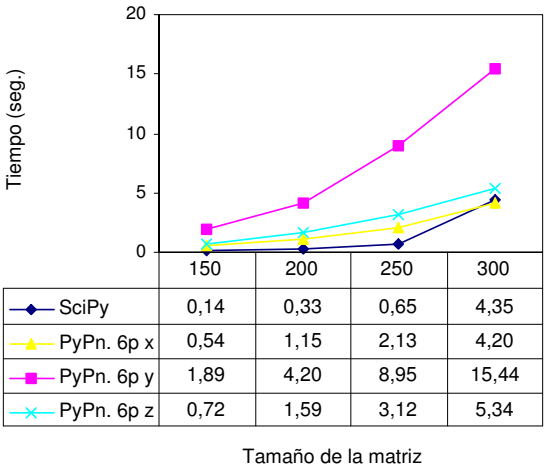
(b) Seaborg

Figura 9.15: Influencia del eje de distribución en la escritura de un fichero netCDF

de los datos se realizaba en el eje `x`. Sin embargo, podemos cuestionarnos cuál de las distribuciones es más adecuada o cuál de ellas proporciona menores tiempos de lectura y escritura. Para tratar de aclarar esta cuestión mostramos en las figuras 9.15 y 9.16, los tiempos de escritura y lectura para un fichero netCDF con una variable `a` con las tres dimensiones (`x,y,z`) donde cada dimensión tiene 300 elementos.

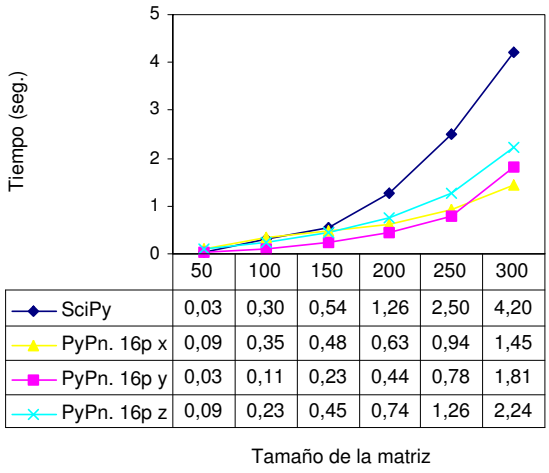
En el caso de la escritura, los tiempos medidos hacen referencia a las instrucciones de escritura y sincronización. De este modo se confirma que la escritura se ha completado en todos los procesos. En la figura 9.15(a) se observa un comportamiento similar entre las tres distribuciones, aunque los tiempos obtenidos en la distribución en la primera dimensión son ligeramente superiores. Por otro lado, PyPNetCDF frente a ScientificPython presenta tiempos considerablemente superiores. Se ha de tener en cuenta que en el sistema de ficheros de Cluster-umh, tendremos varios procesos compitiendo por un único recurso que pueden provocar la aparición de colisiones, ya que se trata de una red Gigabit Ethernet.

Por otro lado, en Seaborg sí se observa un mejor comportamiento de PyPNetCDF frente a ScientificPython de forma general para la distribución en cualquiera de las tres dimensiones. Sin embargo, la dimensión en el eje `x` (`dist=0`) ofrece tiempos ligeramente menores que las otras distribuciones, siendo la distribución en el eje `z` la que presenta peores tiempos.



	150	200	250	300
SciPy	0,14	0,33	0,65	4,35
PyPn. 6p x	0,54	1,15	2,13	4,20
PyPn. 6p y	1,89	4,20	8,95	15,44
PyPn. 6p z	0,72	1,59	3,12	5,34

(a) Cluster-umh



	50	100	150	200	250	300
SciPy	0,03	0,30	0,54	1,26	2,50	4,20
PyPn. 16p x	0,09	0,35	0,48	0,63	0,94	1,45
PyPn. 16p y	0,03	0,11	0,23	0,44	0,78	1,81
PyPn. 16p z	0,09	0,23	0,45	0,74	1,26	2,24

(b) Seaborg

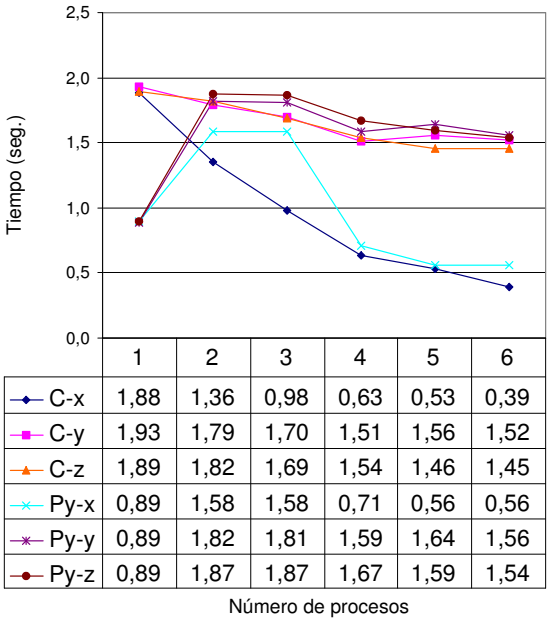
Figura 9.16: Influencia del eje de distribución en la lectura de un fichero netCDF

Con respecto al comportamiento del eje de distribución de datos en la lectura de un fichero netCDF, en la figura 9.16(a) se muestran los tiempos obtenidos en el `nodo1` del cluster, ejecutando la lectura con seis procesos para distintos valores de `dist` y distinto tamaño de la variable a leer. Hemos mostrado el `nodo1` ya que se encuentra en las mismas condiciones de acceso que los nodos 2 a 5.

Por otro lado, en Seaborg se obtienen tiempos mejores en la lectura secuencial desde el tamaño de la dimensión más pequeño. Además de ofrecer tiempos de lectura menores, los datos ya se encontrarían distribuidos entre los procesos por lo que no necesitaremos rutinas auxiliares de distribución de datos. El comportamiento entre los tiempos para cada distribución es muy similar aunque ligeramente superior para la distribución en el eje `y`.

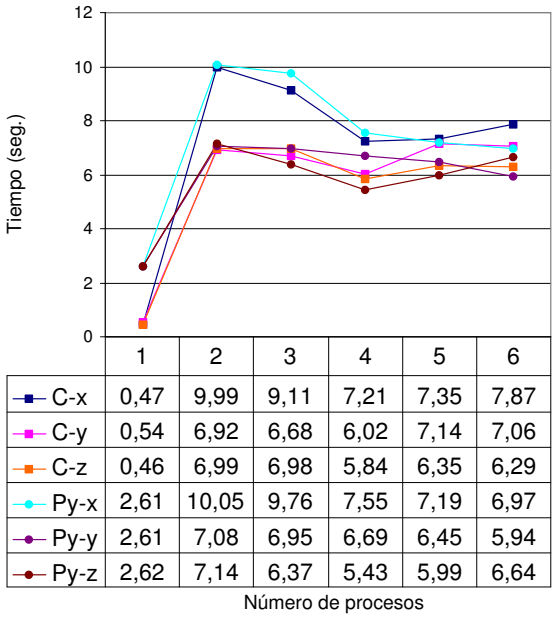
De todos modos, si debemos decidir un eje en el cual realizar la distribución se deberá tener en cuenta fundamentalmente el tratamiento y análisis que se le vaya a dar a los datos en la aplicación.

Hasta el momento, hemos comparado PyPNetCDF con su equivalente secuencial. Además, hemos comprobado la comodidad en el acceso a ficheros netCDF desde Python tanto en su versión secuencial y paralela. En este punto, nos podemos preguntar cuál es la penalización sufrida por utilizar un lenguaje interpretado de alto nivel como Python



	1	2	3	4	5	6
C-x	1,88	1,36	0,98	0,63	0,53	0,39
C-y	1,93	1,79	1,70	1,51	1,56	1,52
C-z	1,89	1,82	1,69	1,54	1,46	1,45
Py-x	0,89	1,58	1,58	0,71	0,56	0,56
Py-y	0,89	1,82	1,81	1,59	1,64	1,56
Py-z	0,89	1,87	1,87	1,67	1,59	1,54

(a) Lectura



	1	2	3	4	5	6
C-x	0,47	9,99	9,11	7,21	7,35	7,87
C-y	0,54	6,92	6,68	6,02	7,14	7,06
C-z	0,46	6,99	6,98	5,84	6,35	6,29
Py-x	2,61	10,05	9,76	7,55	7,19	6,97
Py-y	2,61	7,08	6,95	6,69	6,45	5,94
Py-z	2,62	7,14	6,37	5,43	5,99	6,64

(b) Escritura

Figura 9.17: Comparativa de tiempos entre el PnetCDF y PyPnetCDF para una matriz $200 \times 200 \times 200$

y por hacer uso de la interfaz creada. Para responder esta cuestión, hemos realizado unas pruebas en Cluster-umh que se reflejan en la figura 9.17. En ella podemos ver los tiempos de lectura y escritura para una matriz de tres dimensiones $200 \times 200 \times 200$ y diferente número de procesos. Para comparar los tiempos hemos utilizado las distribuciones en los tres ejes X, Y, y Z tanto desde la interfaz para C ("C-") como para Python ("Py-"). Tanto en la lectura como en la escritura, los tiempos obtenidos para una distribución en el mismo eje son muy similares y siguen la misma tendencia tanto si se utiliza la librería PnetCDF de forma directa mediante C, como si ésta se utiliza mediante la distribución PyPNetCDF. En definitiva, la facilidad proporcionada por PyPNetCDF para el tratamiento de los ficheros netCDF no se ve penalizada por una perdida de eficiencia en el acceso a este tipo de ficheros.

En resumen, PyPNetCDF presenta mejoras considerables en los tiempos de escritura y lectura en plataformas que presentan un sistema paralelo de acceso a ficheros como es el caso de Seaborg. Sin embargo, en Cluster-umh hemos obtenido tiempos considerablemente superiores para PyPNetCDF con respecto a ScientificPython. Esto nos lleva a deducir

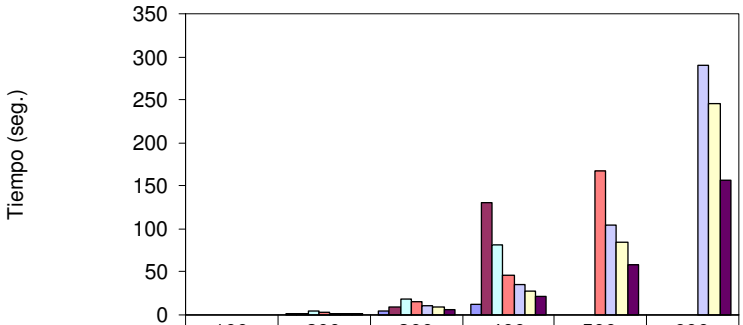
que PyPNetCDF puede no ser una herramienta adecuada en plataformas con un acceso a ficheros similar a Cluster-umh. Sin embargo, no hemos comentado todavía otro de los puntos fuertes de PyPNetCDF: la escalabilidad.

Si, por ejemplo, el `nodo0` realiza la lectura de un fichero netCDF mediante ScientificPython y posteriormente distribuye los datos con rutinas de comunicaciones como las proporcionadas por PyBLACS, dicho proceso deberá tener memoria suficiente para almacenar todo el contenido de la variable. Sin embargo, éste método no es escalable puesto que si aumentan las dimensiones de la variable llegará un punto en el cual el nodo no disponga de suficientes recursos de memoria.

Esta situación la vemos reflejada en las figuras 9.18(a) y 9.18(b) que se corresponden con los tiempos invertidos en Cluster-umh para realizar la lectura y escritura de una variable `a` netCDF de dimensiones `(x,y,z)` donde la longitud de las dimensiones `(x=y=z)` se muestran en cada columna y la distribución de los datos se realiza en el eje `x`. Las celdas en blanco en ambas gráficas significan que la ejecución ha fallado por no disponer de suficientes recursos de memoria. Por tanto, observamos como ScientificPython no puede leer variables de dimensiones superiores a $400 \times 400 \times 400$ con elementos en coma flotante de doble precisión, que en total suman 512 Mbytes. Un tamaño superior $500 \times 500 \times 500$ necesitaría un total 1Gbyte de memoria mientras que, por ejemplo, el `nodo0` mediante el comando `top` nos informa que la memoria total disponible es de 905Mbytes para todas las aplicaciones del sistema. En definitiva, nos resultaría imposible abordar la lectura de variables con dimensiones superiores.

Sin embargo, PyPnetCDF soluciona este problema por ser una herramienta absolutamente escalable en cualquier plataforma. De hecho, cuando se realiza la lectura y escritura del cluster en paralelo, el demonio `nfsd` será quien atienda las solicitudes de acceso a fichero de los intérpretes de Python en los nodos que acceden por red. De este modo, el nodo origen no necesita almacenar toda la variable en memoria y por tanto, disponemos de una solución escalable.

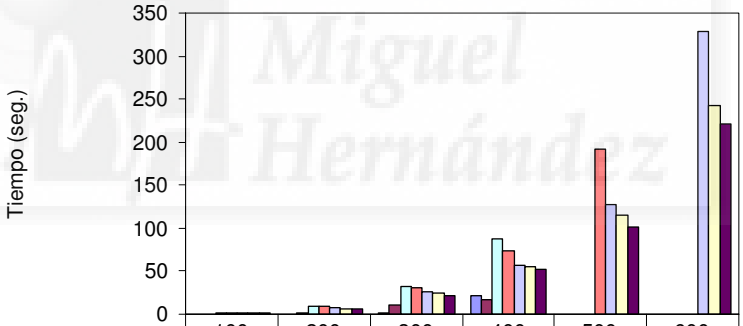
Los tiempos de lectura mostrados en la figura 9.18(a), se corresponden con el `nodo0` y éstos no difieren en gran medida con los tiempos obtenidos en el resto de nodos tal y como vimos para variables de dimensiones más pequeñas. Por otro lado, los tiempos de escritura representados en la figura 9.18(b) se corresponden con los tiempos de escritura y sincronización de los datos, es decir, el tiempo en el cual todos los nodos han finalizado la escritura en el fichero netCDF compartido.



	100	200	300	400	500	600
ScientificPython	0,18	1,37	4,65	11,67		
PyPnetCDF 1p	0,26	1,95	8,89	130,21		
PyPnetCDF 2p	0,14	4,31	17,98	82,12		
PyPnetCDF 3p	0,29	3,30	15,11	45,93	167,11	
PyPnetCDF 4p	0,07	2,22	11,22	34,96	104,40	289,87
PyPnetCDF 5p	0,06	2,25	8,56	27,66	83,82	245,78
PyPnetCDF 6p	0,12	1,61	6,90	21,38	58,57	156,83

Tamaño de la matriz

(a) Lectura



	100	200	300	400	500	600
ScientificPython	0,03	0,25	0,93	21,61		
PyPnetCDF 1p	0,27	2,08	10,20	16,28		
PyPnetCDF 2p	1,36	9,38	32,12	87,71		
PyPnetCDF 3p	1,28	9,50	31,35	73,68	191,96	
PyPnetCDF 4p	1,25	8,43	25,34	57,54	127,05	328,43
PyPnetCDF 5p	1,18	6,32	23,87	54,84	114,56	242,10
PyPnetCDF 6p	1,09	6,61	21,34	51,74	101,94	220,99

Tamaño de la matriz

(b) Escritura

Figura 9.18: Tiempos de lectura y escritura para un fichero netCDF de grandes dimensiones en Cluster-umh

Integración de PyPnetCDF con PyACTS

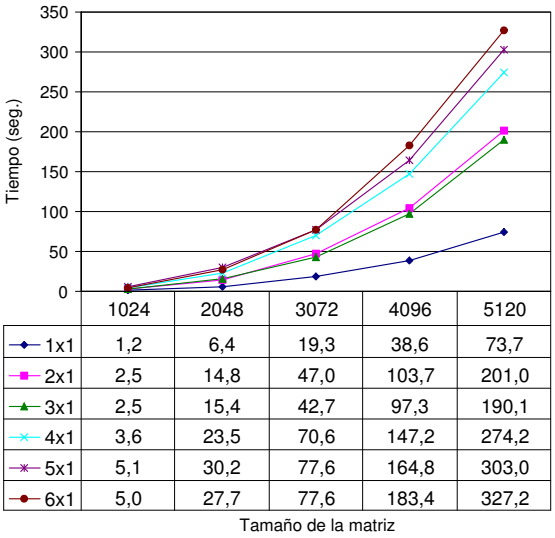
En los ejemplos mostrados hasta el momento, la lectura de los datos se realiza en una única orden, es decir, accede a los datos de una bloque entero de la matriz global. Sin embargo, para la conversión directa de un fichero netCDF a una matriz PyACTS o viceversa descrita en el apartado 9.4.4, el proceso de lectura o escritura se realiza en bloques de tamaño $MB \times NB$. Para comprobar el comportamiento y escalabilidad de este algoritmo hemos realizado unas pruebas de ejecución de las rutina `PyPNetCDF2PyACTS` y `PyACTS2PyPNetCDF` para diferente tamaño de matriz y diferente número de procesos.

En las figura 9.19(a) y 9.19(b) mostramos los tiempos de lectura desde un fichero netCDF a una matriz PyACTS con tamaño de bloque `NB=MB=64` en Cluster-umh y en Seaborg, respectivamente. El fichero de datos utilizado contiene una variable de tres dimensiones X,Y, y Z; todas ellas de la misma magnitud y expresada en cada columna de las figuras. Además, el eje X ha sido el utilizado para realizar la distribución de datos entre los procesos.

En ambas figuras se observa un comportamiento claramente diferenciado debido a la diferente arquitectura de su sistema de ficheros. En Cluster-umh, en una malla 1×1 será un único proceso el que accede de forma repetitiva a la lectura de los bloques; mientras que para una 6×1, todos los nodos intentan acceder a cada bloque de datos del fichero y esto genera un cuello de botella que incrementa los tiempos considerablemente. Por otro lado, en Seaborg el comportamiento es inverso. Los tiempos superiores se consiguen con único proceso y si aumentamos el número de procesos los tiempos de lectura y conversión a PyACTS se decrementan.

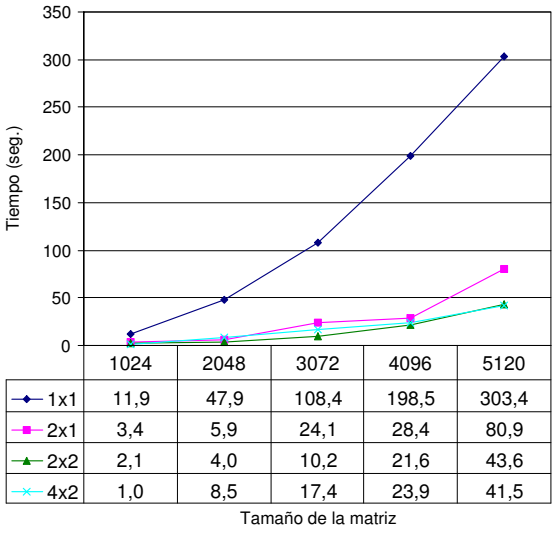
En estas pruebas el tamaño de bloque se ha establecido cuadrado a 64 elementos. Sin embargo, podemos suponer que si aumentamos el tamaño de bloque el número de accesos en paralelo que cada nodo debe efectuar al fichero netCDF será menor y por tanto menor serán los tiempos obtenidos. Para confirmar este comportamiento hemos realizado las pruebas que se representan en la figura 9.20. Apreciamos cómo Cluster-umh es más sensible al aumento del tamaño de bloque que Seaborg ya que el acceso a ficheros en Cluster-umh es secuencial y el aumento del número de accesos provoca un incremento en el número de peticiones a un mismo recurso.

Para comprobar el funcionamiento con respecto a la escritura de un fichero netCDF de similares características a las descritas anteriormente para la lectura, hemos desarrollado



	1024	2048	3072	4096	5120
1x1	1,2	6,4	19,3	38,6	73,7
2x1	2,5	14,8	47,0	103,7	201,0
3x1	2,5	15,4	42,7	97,3	190,1
4x1	3,6	23,5	70,6	147,2	274,2
5x1	5,1	30,2	77,6	164,8	303,0
6x1	5,0	27,7	77,6	183,4	327,2

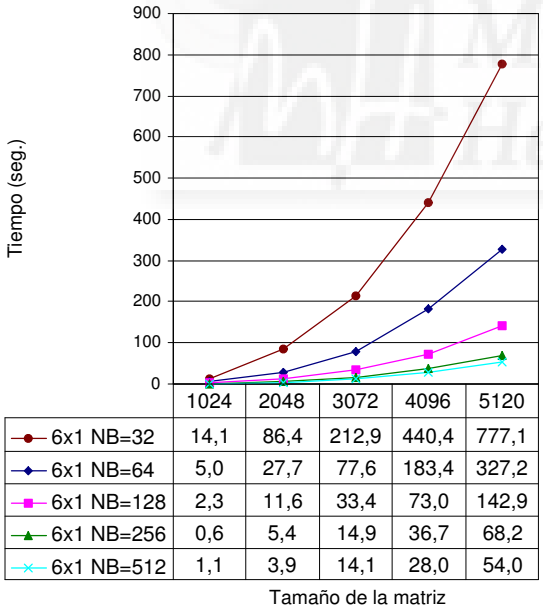
(a) Cluster-umh



	1024	2048	3072	4096	5120
1x1	11,9	47,9	108,4	198,5	303,4
2x1	3,4	5,9	24,1	28,4	80,9
2x2	2,1	4,0	10,2	21,6	43,6
4x2	1,0	8,5	17,4	23,9	41,5

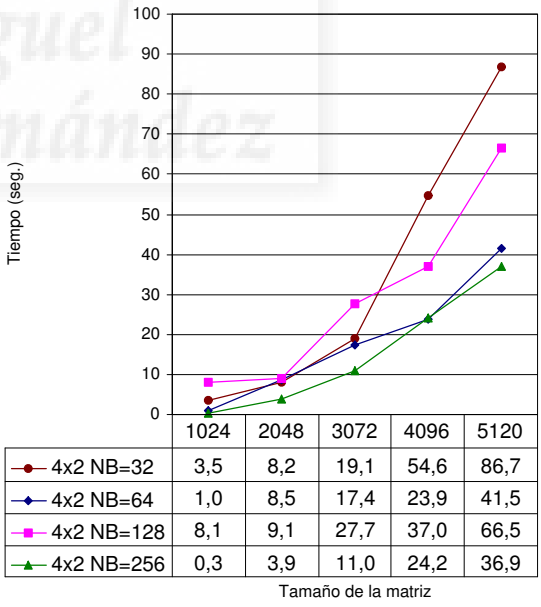
(b) Seaborg

Figura 9.19: Lectura de PyPnetCDF a PyACTS (`PyPNetCDF2PyACTS`) con NB=64



	1024	2048	3072	4096	5120
6x1 NB=32	14,1	86,4	212,9	440,4	777,1
6x1 NB=64	5,0	27,7	77,6	183,4	327,2
6x1 NB=128	2,3	11,6	33,4	73,0	142,9
6x1 NB=256	0,6	5,4	14,9	36,7	68,2
6x1 NB=512	1,1	3,9	14,1	28,0	54,0

(a) Cluster-umh



	1024	2048	3072	4096	5120
4x2 NB=32	3,5	8,2	19,1	54,6	86,7
4x2 NB=64	1,0	8,5	17,4	23,9	41,5
4x2 NB=128	8,1	9,1	27,7	37,0	66,5
4x2 NB=256	0,3	3,9	11,0	24,2	36,9

(b) Seaborg

Figura 9.20: Lectura de PyPnetCDF a PyACTS (`PyPNetCDF2PyACTS`) para diferentes tamaños de bloque

unas pruebas cuyos resultados se reflejan en las figuras 9.21 y 9.22.

En la figura 9.21(a) comprobamos cómo los tiempos de escritura crecen considerablemente a medida que aumenta el número de procesos que compiten por el acceso a disco. Por otro lado, en Seaborg (figura 9.21(b)) los tiempos de escritura disminuyen conforme aumenta el número de procesos aunque en el caso 2×1 los tiempos son superiores a la ejecución 1×1. Se ha de tener en cuenta que en ambas plataformas se ha contabilizado el tiempo de escritura y de sincronización entre todos los procesos.

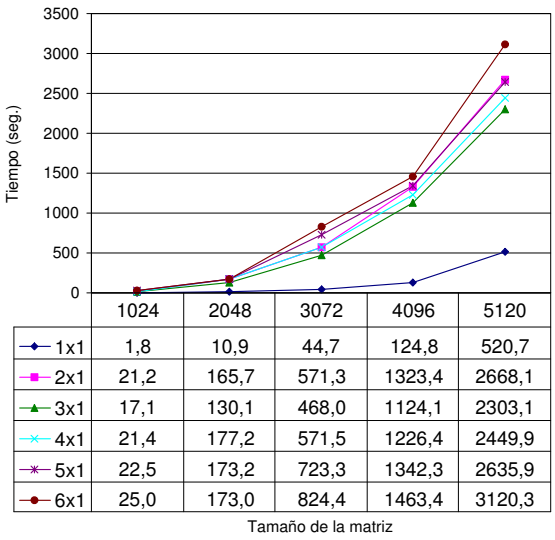
En resumen, hemos presentado una nueva distribución de Python que permite el acceso en paralelo a ficheros netCDF de una manera cómoda, eficiente y escalable. Además, hemos proporcionado una herramienta para la gestión de la entrada y salida de datos a la distribución PyACTS complementándose ambas herramientas.

9.5. Conclusiones

En el presente capítulo, hemos presentado PyPNetCDF como una distribución para Python que permite el acceso en paralelo a un fichero netCDF de una manera sencilla y muy similar a la herramienta secuencial actualmente disponible y utilizada por la comunidad científica.

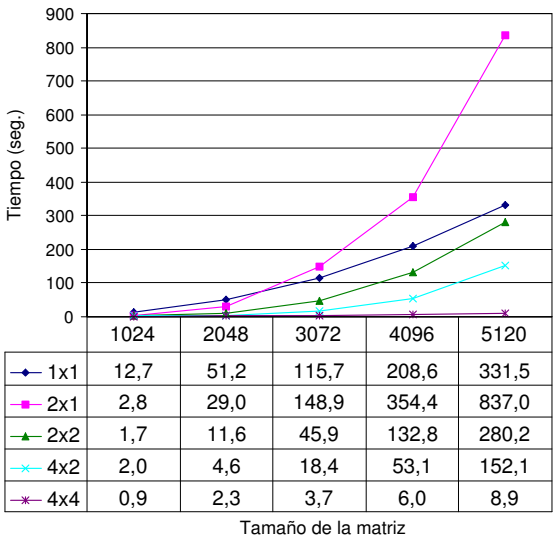
Para poder introducir mejor el funcionamiento de PyPnetCDF, se ha realizado una introducción al estándar netCDF y a las principales características de su modelo de datos y de la estructura de un fichero netCDF. Como parte de este estándar se proporciona la librería para C y Fortran que permite un acceso secuencial a un fichero netCDF.

Para ampliar el espectro de programadores que puedan hacer uso de ficheros netCDF, se han implementado librerías e interfaces que permiten el acceso a ficheros netCDF desde otros lenguajes, incluido Python. La distribución ScientificPython permite el acceso desde Python a ficheros netCDF y define las clases `NetCDFFile` y `NetCDFVariable`, con sus respectivas propiedades y métodos, que consiguen que el manejo de los ficheros netCDF se convierta en una tarea cómoda e intuitiva. Por estos motivos y porque ScientificPython es la herramienta más utilizada en Python para el acceso a ficheros netCDF, la hemos tomado como modelo para crear una distribución con las mismas funcionalidades y sencillez pero



	1024	2048	3072	4096	5120
1x1	1,8	10,9	44,7	124,8	520,7
2x1	21,2	165,7	571,3	1323,4	2668,1
3x1	17,1	130,1	468,0	1124,1	2303,1
4x1	21,4	177,2	571,5	1226,4	2449,9
5x1	22,5	173,2	723,3	1342,3	2635,9
6x1	25,0	173,0	824,4	1463,4	3120,3

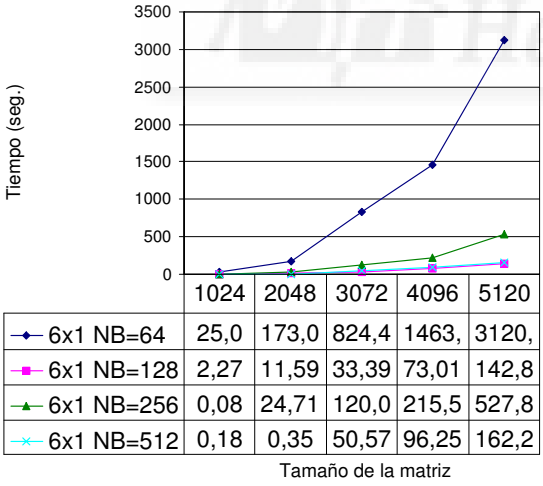
(a) Cluster-umh



	1024	2048	3072	4096	5120
1x1	12,7	51,2	115,7	208,6	331,5
2x1	2,8	29,0	148,9	354,4	837,0
2x2	1,7	11,6	45,9	132,8	280,2
4x2	2,0	4,6	18,4	53,1	152,1
4x4	0,9	2,3	3,7	6,0	8,9

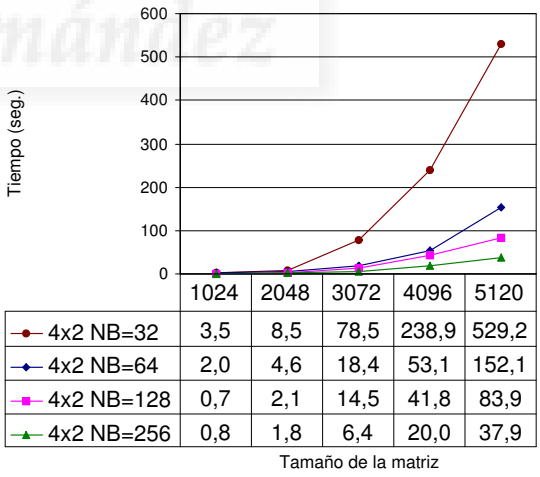
(b) Seaborg

Figura 9.21: Escritura de PyACTS a PyPnetCDF (`PyACTS2PyPNetCDF`) con NB=64



	1024	2048	3072	4096	5120
6x1 NB=64	25,0	173,0	824,4	1463,	3120,
6x1 NB=128	2,27	11,59	33,39	73,01	142,8
6x1 NB=256	0,08	24,71	120,0	215,5	527,8
6x1 NB=512	0,18	0,35	50,57	96,25	162,2

(a) Cluster-umh



	1024	2048	3072	4096	5120
4x2 NB=32	3,5	8,5	78,5	238,9	529,2
4x2 NB=64	2,0	4,6	18,4	53,1	152,1
4x2 NB=128	0,7	2,1	14,5	41,8	83,9
4x2 NB=256	0,8	1,8	6,4	20,0	37,9

(b) Seaborg

Figura 9.22: Escritura de PyACTS a PyPnetCDF (`PyACTS2PyPNetCDF`) para diferentes tamaños de bloque

permitiendo una escalabilidad y un mayor rendimiento ante un volumen elevado de datos.

Por otro lado, hemos introducido la librería PnetCDF que permite el acceso concurrente a ficheros netCDF sin modificación alguna en el formato del propio estándar netCDF. De este modo, la definición de la estructura de un fichero netCDF es la misma aunque su acceso se realice en paralelo. Esta librería se proporciona para su utilización desde C o Fortran. A partir de esta librería, hemos desarrollado la distribución PyPnetCDF que proporciona las mismas funcionalidades que la distribución secuencial pero que en niveles inferiores hace uso de la librería PnetCDF para implementar el acceso en paralelo a ficheros netCDF.

En este sentido, hemos presentado la distribución PyPnetCDF detallando sus módulos principales y las clases `PNetCDFFile` y `PNetCDFVariable` que permiten un acceso sencillo y cómodo a los ficheros netCDF en un entorno distribuido. Para ilustrar la sencillez en su utilización hemos mostrado ejemplos de lectura y escritura detallando, linea a linea, las acciones realizadas. La sintaxis de Python y la sencillez de PyPnetCDF mejoran considerablemente el manejo de ficheros netCDF en un entorno paralelo en comparación con su interfaz a C.

Una vez presentada la distribución, hemos profundizado en el diseño e implementación de PyPnetCDF explicando cuáles han sido los pasos realizados hasta conseguir un paquete de fácil instalación y distribución. Deseamos destacar la complejidad en la realización de los *wrappers* para la librería `libpnetcdf.a`, ya que debido a las limitaciones de SWIG, éste no implementa una interacción cómoda con la matrices `Numpy` y por tanto se han tenido que modificar muchas de las rutinas de acceso a datos dentro de un código C de más de 6000 líneas. Por otro lado, el módulo `PNetCDF.py` (ver figura 9.7) ha permitido la definición de las clases `NetCDFFile` y `PNetCDFVariable` con sus respectivos métodos que permiten al programador un acceso cómodo a los ficheros netCDF e interactúan con las rutinas incluidas en `libpnetcdf.a` en niveles inferiores.

PyPnetCDF se distribuye independientemente de PyACTS, sin embargo, ambas distribuciones pueden complementarse y en conjunto, ofrecen al programador un entorno sencillo, eficiente y escalable para el acceso y análisis de datos de grandes dimensiones. En esta línea, hemos presentado la interacción entre ambos módulos gracias a las rutinas `PNetCDF2PyACTS` y `PyACTS2PNetCDF`.

En el último apartado se ha realizado un análisis en profundidad del rendimiento y comportamiento de PyPnetCDF en dos plataformas con un sistema de acceso a ficheros

completamente diferente. En Cluster-umh, la compartición de una unidad en uno de los nodos con el resto provocaba la aparición de un cuello de botella que incrementaba los tiempos de lectura y escritura con respecto a la herramienta secuencial ScientificPython. Sin embargo, en una arquitectura preparada para el acceso a ficheros en paralelo, se consiguen tiempos de lectura y escritura menores que en el caso secuencial.

Esta disparidad en el comportamiento de ambas plataformas tanto en la escritura como en la lectura en paralelo de ficheros netCDF ha quedado reflejada en las figuras 9.11 y 9.14, respectivamente. En ellas se aprecia cómo los procesos en Seaborg realizan las tareas de una forma concurrente mientras que los nodos del Cluster-umh compiten por el mismo recurso provocando un acceso desordenado y menos eficiente.

Del mismo modo que se realizó en PyACTS, hemos comprobado el rendimiento de PyPnetCDF frente a PnetCDF y se han obtenido unos tiempos similares y un mismo comportamiento en la variación del eje de distribución de datos. Conseguimos, por tanto, una herramienta considerablemente más sencilla y cómoda que nos proporciona el acceso a ficheros netCDF en modo paralelo con eficiencia similar a la obtenida accediendo desde C.

Además, otra importante ventaja que presenta PyPnetCDF frente a ScientificPython es su escalabilidad. El tamaño máximo de lectura de un fichero netCDF en secuencial para su posterior distribución entre los procesos siempre estará condicionado por la memoria disponible del proceso que accede al fichero. Sin embargo, mediante PyPnetCDF cada proceso accede de forma colectiva a su porción de datos del fichero netCDF consiguiendo una solución totalmente escalable, tal y como se ha reflejado en la figura 9.18.

Por último, hemos comprobado el rendimiento de las rutinas `PNetCDF2PyACTS` y `PyACTS2PNetCDF` para ambas plataformas y se ha observado de nuevo un comportamiento dispar. Mientras en Seaborg los tiempos de lectura y escritura disminuyen conforme aumenta el número de nodos, en Cluster-umh estos tiempos aumentan al aumentar el número de nodos que compiten por el mismo recurso. Otro factor decisivo en la realización de la distribución cíclica 2D a partir de un fichero netCDF es el tamaño de bloque. Las figuras 9.20 y 9.22 han reflejado una realidad sencilla: a medida que aumentamos el tamaño de bloque, el número de accesos por proceso al fichero netCDF es menor y por tanto el tiempo total por procesos disminuye. Esto nos lleva a pensar que cuanto mayor sea el tamaño de bloque menores tiempos obtendremos en la distribución. Sin embargo, si posteriormente deseamos hacer uso de los datos distribuidos con rutinas de la distribución PyACTS deberemos encontrar un punto de equilibrio. Según se ha analizado en el

apartado 8.4.1, el tiempo de ejecución de algunas rutinas de PyACTS (especialmente las incluidas en el módulo PyScaLAPACK) se incrementa con el tamaño de bloque definido. Por tanto, debemos encontrar un tamaño de bloque adecuado tanto para la lectura de los datos y su posterior tratamiento con la distribución PyACTS.

En resumen, consideramos que el acceso a ficheros de datos en una arquitectura paralela es un tema complejo y delicado. El estándar netCDF proporciona un fichero de datos portátil, independiente de la plataforma y estructurado. Por otro lado, la librería PnetCDF permite el acceso paralelo a un fichero netCDF estándar mediante la librería MPI. Nuestra aportación PyPnetCDF permite crear una herramienta que abstrae las rutinas que se proporcionan en la librería PnetCDF a un entorno orientado a objetos que mejora la usabilidad y manejo de los ficheros de datos. Si la librería ScientificPython se ha convertido en la librería más utilizada para el acceso a ficheros netCDF, esperamos que PyPnetCDF, al ofrecer una interfaz prácticamente idéntica, se convierta en la primera y más importante herramienta para Python de acceso paralelo a ficheros netCDF para ser utilizada por la comunidad científica y por el conjunto de programadores en otros ámbitos. Además, los ejemplos y las pruebas realizadas confirman su sencillez y eficiencia en el tratamiento de los datos en paralelo con respecto a su versión secuencial.

Capítulo 10

Aplicaciones

En capítulos anteriores hemos introducido las necesidades requeridas en los entornos de computación de alto rendimiento, hemos mostrado un conjunto de librerías pertenecientes a la colección ACTS ampliamente utilizadas en la comunidad científica cuya robustez y fiabilidad han sido respaldadas por la comunidad científica [42]. La mayor parte de estas herramientas de altas prestaciones se encuentran accesibles para lenguajes de bajo nivel como C o Fortran que ofrecen un buen rendimiento pero cuya sintaxis, entornos de edición y depuración pueden ser muy complejos para programadores con poca experiencia en sistemas de computación de memoria distribuida.

Con el objetivo de ofrecer unas herramientas más cómodas, fáciles e igualmente eficientes y escalables, hemos desarrollado las distribuciones para un lenguaje de alto nivel que permiten la utilización de librerías de alto rendimiento desde entornos más cómodos, sencillos e interactivos. Los usuarios finales de estas nuevas herramientas pueden pertenecer a diferentes áreas de conocimiento cuyas necesidades requieran de cálculos o simulaciones con matrices de elevadas dimensiones, y por tanto de la necesidad de utilización de herramientas de computación en paralelo. En el presente capítulo expondremos algunas de las aplicaciones que pueden hacer uso de las distribuciones presentadas y que hemos desarrollado como ejemplos de la facilidad y utilidad que pueden tener estos paquetes en diversas áreas de investigación.

En este sentido, expondremos un conjunto de ejemplos y aplicaciones que tratarán de mostrar la potencia y sencillez de las herramientas que centran nuestro trabajo. Las

áreas de conocimiento en las cuales se pueden utilizar estas herramientas no se limitan a las mostradas en estos ejemplos. Las distribuciones PyACTS y PyPnetCDF pueden resultar de gran utilidad en cualquier aplicación que necesite operar con matrices densas de elevado tamaño, resolver un sistema de ecuaciones, calcular la descomposición en valores singulares, proporcionar un sistema de almacenamiento de datos o cualquier otra funcionalidad descrita en capítulos anteriores.

De este modo, en la sección 10.1 expondremos el método del gradiente conjugado y compararemos la complejidad y rendimiento de este algoritmo utilizando un entorno tradicional, por ejemplo Fortran, y utilizando la distribución PyACTS desde Python. Posteriormente, en la sección 10.2 se presenta una herramienta utilizada en el análisis de la variabilidad climática denominada PyClimate [101] pero cuyos problemas de escalado no permiten el análisis de datos de grandes dimensiones. Mediante la utilización de las distribuciones PyACTS y PyPNetCDF, presentaremos un primer prototipo de una versión paralela ParPyClimate, paralelizando y haciendo escalable algunas de sus rutinas.

10.1. Método del gradiente conjugado

Dado el sistema de ecuaciones lineales

$$Ax = b, \tag{10.1}$$

donde $A \in \mathbb{R}^{n \times n}$ es una matriz simétrica y definida positiva, uno de los métodos más eficientes que existen para resolver el sistema (10.1) cuando la matriz de coeficientes es además grande y dispersa, es el *método del gradiente conjugado*.

Este procedimiento pertenece a una clase de métodos iterativos conocida con el nombre de *métodos de minimización*, cuyo nombre se debe a que resolver simultáneamente un conjunto de n ecuaciones es equivalente a encontrar el mínimo de una función error definida sobre un espacio n-dimensional. En cada paso de un método de este tipo, se usa un conjunto de valores de variables para generar un nuevo conjunto que proporcionan un valor menor en la función error. Veamos de una manera más formal la descripción de este método.

Consideremos la siguiente forma cuadrática

$$Q(x) = \frac{1}{2}x^T Ax - b^T x. \tag{10.2}$$

Si la matriz A es simétrica y definida positiva, el único x que minimiza la forma cuadrática $Q(x)$ es la solución del sistema $Ax = b$, que llamaremos x^*.

La razón de esto es porque el gradiente de la forma cuadrática (10.2) es

$$grad(Q(x)) = \left[\frac{\delta}{\delta x_1}Q(x), \ldots, \frac{\delta}{\delta x_n}Q(x)\right]^T = Ax - b. \tag{10.3}$$

Por tanto, $grad(Q(x^*)) = Ax^* - b = 0$. Por ser la matriz A definida positiva, podemos asegurar que el punto crítico de $Q(x)$ es un mínimo. Luego, para resolver el sistema de ecuaciones (10.1) es suficiente encontrar un vector x^* que minimice la forma cuadrática $Q(x)$.

Hay muchos métodos para minimizar la forma cuadrática $Q(x)$, algunos de ellos se basan en minimizaciones sucesivas sobre rectas. Una iteración de un método de este tipo viene dada por la expresión

$$x^{(l+1)} = x^{(l)} - \alpha_l p^{(l)}, \quad l = 0, 1, \ldots, \tag{10.4}$$

donde $p^{(l)}$ es el vector director de la recta y α_l es el escalar que indica la distancia a la que se encuentra la nueva aproximación de $x^{(l)}$, en la dirección determinada por el vector $p^{(l)}$. Dependiendo de la elección de α_l y de $p^{(l)}$, tenemos distintos métodos, pero quizá la forma más natural de elegir el valor del escalar α_l es tomarlo de tal forma que minimice la función $Q(x)$ en la dirección $p^{(l)}$, es decir,

$$Q(x^{(l)} - \alpha_l p^{(l)}) = \min_{\alpha} Q(x^{(l)} - \alpha p^{(l)}). \tag{10.5}$$

Para $x^{(l)}$ y $p^{(l)}$ fijos, la expresión (10.5) se corresponde con un problema de minimización en una única variable α. Por tanto, puede resolverse explícitamente, concluyendo (ver Ortega [82]), que como A es definida positiva, la forma cuadrática $Q(x)$ en α alcanza el valor mínimo cuando

$$\alpha_l = \frac{(p^{(l)})^T(Ax^{(l)} - b)}{(p^{(l)})^T Ap^{(l)}} = -\frac{(p^{(l)})^T \tau^{(l)}}{(p^{(l)})^T Ap^{(l)}}, \tag{10.6}$$

donde $\tau^{(l)} = b - Ax^{(l)}$ es el vector residual o residuo que se obtiene después de l iteraciones. Este residuo no necesita calcularse explícitamente después de cada iteración ya que puede

ser calculado conociendo el residuo de la iteración anterior. Es decir, en la iteración $(l+1)$, el residuo $\tau^{(l+1)}$ puede ser calculado de la siguiente forma

$$\begin{aligned}\tau^{(l+1)} &= b - Ax^{(l+1)} \\ &= b - A(x^{(l)} - \alpha_l p^{(l)}) \\ &= b - Ax^{(l)} + \alpha_l Ap^{(l)} \\ &= \tau^{(l)} + \alpha_l Ap^{(l)}.\end{aligned}$$

De esta forma, en cada iteración sólo se calcula el producto matriz-vector $Ap^{(l)}$, el cual ya se ha calculado anteriormente en el cálculo de α_l de la expresión (10.6).

Uno de estos métodos de minimización es el llamado método del *descenso más rápido* o método de *Richardson*, que se caracteriza porque para un vector dado $x^{(l)}$, el vector director $p^{(l)}$ se elige en la dirección opuesta a la del máximo crecimiento de $Q(x^{(l)})$, es decir, $p^{(l)} = -grad(Q(x^{(l)})) = b - Ax^{(l)} = \tau^{(l)}$. Esta elección natural de los vectores directores $p^{(l)}$, proporciona el siguiente esquema iterativo

$$x^{(l+1)} = x^{(l)} - \alpha_l(b - Ax^{(l)}), \quad l = 0, 1, \ldots,$$

donde α_l viene definida en (10.6).

Podemos observar, que si se normaliza la matriz A para que tenga elementos diagonales igual a 1 y $\alpha_l = 1$, este método se reduce al método iterativo de Jacobi.

Aunque moviéndonos en la dirección opuesta al gradiente se obtiene un decrecimiento máximo local en el valor de la función, el método de Richardson generalmente converge muy lentamente porque los residuos oscilan.

Otra forma de elegir los vectores $p^{(l)}$, es tomar los ejes de coordenadas como direcciones de búsqueda, es decir, los vectores directores $p^{(l)}$ se eligen de forma cíclica como $p^{(0)} = e_1,\ p^{(1)} = e_2, \ldots,\ p^{(n-1)} = e_n,\ p^{(n)} = e_1$, donde $e_i = [0, \ldots, 1, \ldots, 0]^T$. En este caso, si α_l se elige como en la expresión (10.6), el vector $x^{(l+1)}$ se puede calcular como

$$x^{(l+1)} = x^{(l)} - \alpha_l e_i = x^{(l)} - \frac{1}{a_{ii}} \left(\sum_{j=1}^{n} a_{ij} x_j^{(l)} - b_i \right) e_i. \tag{10.7}$$

Se puede comprobar (ver Ortega [82]), que n iteraciones del esquema (10.7) son equivalentes a una iteración del método de Gauss-Seidel en el sistema $Ax = b$. En este contexto,

el método de Gauss-Seidel se conoce también como método de *relajación univariante*, ya que en cada iteración del esquema (10.7) sólo se cambia una de las variables.

Dentro de los métodos de minimización, cabe destacar los llamados métodos de direcciones conjugadas, que aparecen cuando elegimos n vectores de dirección no nulos $p^{(0)}, p^{(1)}, \ldots, p^{(n-1)}$ que satisfacen la condición

$$(p^{(i)})^T A p^{(j)} = 0, \quad \text{siempre que } i \neq j. \tag{10.8}$$

Estos vectores son ortogonales con respecto al producto escalar definido por A y se dice que son vectores A-ortogonales o conjugados con respecto a A.

Una de las propiedades básicas que verifican los métodos de direcciones conjugadas viene dada por el siguiente teorema.

Teorema 10.1. *(Teorema 3.3.1, [82]). Si $A \in \mathbb{R}^{n\times n}$ es una matriz simétrica y definida positiva y $p^{(0)}, \ldots, p^{(n-1)}$ son vectores no nulos A-conjugados, entonces para cualquier $x^{(0)}$ el vector iterado $x^{(l+1)} = x^{(l)} - \alpha_l p^{(l)}$, donde α_l es elegido como en (10.6), converge a la solución exacta del sistema (10.1), en no más de n etapas.*

Como podemos observar, este teorema asegura no sólo que el método converge, sino que en ausencia de errores de redondeo, converge en un número finito de iteraciones menor o igual que n. Por ello y basándonos en esta propiedad, podemos decir que los métodos de direcciones conjugadas son realmente métodos directos. Pero a causa de los errores de redondeo, esta importante propiedad no deja de ser una propiedad teórica, ya que generalmente lo que se obtiene en muchos menos de n pasos es una buena aproximación. Por tanto, se pueden tratar los métodos de direcciones conjugadas como métodos iterativos.

En 1952, Hestenes y Stiefel [69] diseñaron un método de resolución de sistemas de ecuaciones lineales $Ax = b$, siendo A una matriz simétrica y definida positiva, al que llamaron método del gradiente conjugado.

Este método, es un método de direcciones conjugadas donde las direcciones de búsqueda se construyen haciendo A-ortogonales los vectores residuales $\tau^{(l)}$.

El siguiente algoritmo describe este método. Notamos que aquí usamos el producto interno $\langle x, y\rangle = x^T y$.

Algoritmo 10.1. Algoritmo del Gradiente Conjugado.

Dado un vector inicial $x^{(0)}$.

$p^{(0)} = \tau^{(0)} := b - Ax^{(0)}$
REPETIR
$\alpha_l := \frac{\langle \tau^{(l)}, \tau^{(l)} \rangle}{\langle p^{(l)}, Ap^{(l)} \rangle}$
$x^{(l+1)} := x^{(l)} - \alpha_l p^{(l)}$
$\tau^{(l+1)} := \tau^{(l)} - \alpha_l Ap^{(l)}$
SI test de convergencia = VERDADERO
entonces FIN
$\beta_l := -\frac{\langle \tau^{(l+1)}, \tau^{(l+1)} \rangle}{\langle \tau^{(l)}, \tau^{(l)} \rangle}$
$p^{(l+1)} := \tau^{(l+1)} - \beta_l p^{(l)}$.

Como podemos observar en este algoritmo, las direcciones A-conjugadas se van generando a la vez que avanzamos en el cálculo de la solución. Esto es una de las ventajas de este método, ya que no se tienen que calcular de antemano un conjunto de direcciones A-conjugadas.

El criterio de parada que sugiere Ortega en [82], consiste en asegurar que la norma del vector residual $\tau = b - Ax$, sea menor que una cota de error predeterminada. Si denotamos por $e^{(l)}$ el error que se comete en la iteración l-ésima y x^* es la solución exacta del sistema $Ax = b$, se tiene que

$$e^{(l)} = x^{(l)} - x^* = A^{-1}(Ax^{(l)} - b) = A^{-1}\tau^{(l)}.$$

Por tanto

$$\|e^{(l)}\| \leq \|A^{-1}\|\|\tau^{(l)}\|.$$

Entonces, si se tiene una cota para la norma de A^{-1} exigiendo que $\|\tau^{(l)}\| \leq \epsilon$, podemos asegurar que se verifica $\|e^{(l)}\| \leq \epsilon\|A^{-1}\|$.

Otros criterios de parada que implican otros tipo de cota para el error a priori son:

1. $\|\tau^{(l)}\| \leq \epsilon(\|A\|\|x^{(l)}\| + \|b\|)$.
2. $\|\tau^{(l)}\| \leq \epsilon\|b\|$.
3. $\|\tau^{(l)}\| \leq \epsilon\frac{\|x^{(l)}\|}{\|A^{-1}\|}$.
4. $\|\tau^{(l)}\| \leq \epsilon\|\tau^{(0)}\|$.

Existen dos casos importantes en los cuales la solución se alcanza, suponiendo que se trabaja en aritmética exacta, en un número de iteraciones m menor que n. El siguiente teorema muestra estos dos casos.

Teorema 10.2. *(Teorema 9.1.3, [60]). Los vectores iterados por el método del gradiente conjugado convergen a la solución del sistema en no más de m iteraciones si se cumple cualquiera de las condiciones siguientes:*

1.- $p^{(0)} = \tau^{(0)}$ es combinación lineal de m vectores propios de A.

2.- A tiene sólo m valores propios distintos.

Los siguientes teoremas estudian el error que se comete al usar el método del gradiente conjugado. Así, el teorema que enunciamos a continuación muestra que los errores son decrecientes de una iteración a otra.

Teorema 10.3. *(Teorema A.3.5, [82]). Los vectores iterados $x^{(l)}$ del método del gradiente conjugado satisfacen*

$$\|x^* - x^{(l)}\|_2 < \|x^* - x^{(l-1)}\|_2,$$

excepto cuando $x^{(l-1)} = x^*$.

Por último, damos una cota del error en términos del número de condición de la matriz A, utilizando la A-norma y la norma euclídea.

Teorema 10.4. *(Teorema A.3.4, [82]) El vector iterado $x^{(l)}$ del método del gradiente conjugado satisface*

$$\|x^* - x^{(l)}\|_A \leq 2\mu^l \|x^* - x^{(0)}\|_A,$$

y

$$\|x^* - x^{(l)}\|_2 \leq 2\sqrt{c}\mu^l \|x^* - x^{(0)}\|_2,$$

donde $c = cond(A)$ y $\mu = \frac{\sqrt{c}-1}{\sqrt{c}+1}$.

Teniendo en cuenta que A es una matriz simétrica y definida positiva, y suponiendo que los valores propios de A, ordenados en orden creciente, son $\lambda_n \geq \cdots \geq \lambda_1 > 0$, el número de condición de A puede escribirse

$$c = cond(A) = \|A\|_2 \|A^{-1}\|_2 = \frac{\lambda_n}{\lambda_1}$$

Se observa que si $c = 1$ entonces $\mu = 0$, y cuando c tiende a $+\infty$, μ tiende a 1. De esta forma, cuanto más grande sea c, μ se acerca más a 1 y por tanto, la convergencia será más lenta.

Una vez introducido el método del gradiente conjugado, exponemos a continuación las pruebas realizadas y que implementan dicho algoritmo en paralelo mediante los lenguajes de programación Fortran y Python. Para ello utilizaremos las rutinas proporcionadas en la distribución PBLAS y en PyPBLAS, respectivamente y compararemos los tiempos de ejecución necesarios para la obtención de la solución aproximada.

Debemos destacar que la implementación en Python resulta más sencilla que la programación del algoritmo en Fortran. Esta característica se debe fundamentalmente a la facilidad de la sintaxis en Python y su comodidad en la ejecución del código, sin necesidad de los procesos de edición, compilación y depuración. Del mismo modo, la longitud de ambas implementaciones difiere en gran medida, siendo considerablemente más sencilla la implementación de Python (23 líneas de algoritmo propiamente dicho) que la implementación en Fortran (70 líneas de algoritmo con mayor número de parámetros en cada rutina).

A continuación mostramos el código Python que implementa el algoritmo 10.1. Este código se ha ejecutado en una plataforma multiprocesador de memoria distribuida mediante una instrucción del tipo `mpirun -np 4 mpipython gradconjugado.py 6000 4 1`. Como podemos comprobar, al script de Python le hemos pasado tres parámetros que se corresponden con el tamaño del sistema (línea 6) y la configuración de la malla de procesos `[nprow,npcol]` (líneas 7 y 8).

```
1  from PyACTS import *
2  import PyACTS.PyScaLAPACK_Tools as PyScaTools
3  from PyACTS.PyPBLAS import *
4  from Numeric import *
5  import time,sys
6  n=int(sys.argv[1])          # Mediante la línea de comandos
7  parnprow=int(sys.argv[2])   # recibimos el tamaño del sistema
8  parnpcol=int(sys.argv[3])   # y el aspecto de la matriz de
9  time0=time.time()           # procesos
10 ACTS_lib=1
11 PyACTS.gridinit(npcol=parnpcol, nprow=parnprow)
12 time1=time.time()
13 a=distGCdata(n,n,ACTS_lib)    # Creamos de forma distribuida
```

```
14 b=distGCdata(n,1,ACTS_lib)    # los datos y los vectores
15 x=distGCdata(n,1,ACTS_lib)
16 p=distGCdata(n,1,ACTS_lib)
17 pkp1=distGCdata(n,1,ACTS_lib)
18 r=distGCdata(n,1,ACTS_lib)
19 ap=distGCdata(n,1,ACTS_lib)
20 alpha=Scal2PyACTS(0,ACTS_lib)  # Definimos escalares PyACTS
21 beta=Scal2PyACTS(0,ACTS_lib)   # para asegurar su distribución
22 uno=Scal2PyACTS(1e0,ACTS_lib)  # entre todos los procesos
23 cero=Scal2PyACTS(0,ACTS_lib)
24 menos_uno=Scal2PyACTS(-1,ACTS_lib)
25 time2=time.time()
26 #Inicializacion del Algoritmo
27 x=pvcopy(b,x)                    # x0=b
28 r=pvcopy(b,r)                    # r0=b
29 ap=pvgemv(uno,a,x,cero,ap)       # r0= b- A*x0 (1/2)
30 r=pvaxpy(menos_uno,ap,r)         # r0= b- A*x0 (2/2)
31 p=pvcopy(r,p)                    # p0= r0
32 cond,eps,num_iter=1,1e-15,1
33 time3=time.time()
34 while (cond>=eps) and (iter<n):
35                                  ap=pvgemv(uno,a,p,cero,ap)
36                                  dot_rr=pvdot(r,r)
37                                  dot_app=pvdot(ap,p)
38                                  alpha.data=(dot_rr)/(dot_app)
39                                  x=pvaxpy(alpha,p,x)
40                                  alpha.data=(-1)*alpha.data
41                                  r=pvaxpy(alpha,ap,r)
42                                  cond=pvdot(r,r)
43                                  beta.data=cond/dot_rr
44                                  pkp1=pvcopy(r,pkp1)
45                                  pkp1=pvaxpy(beta,p,pkp1)
46                                  p=pvcopy(pkp1,p)
47                                  num_iter=num_iter+1
48 time4=time.time()
49 print n,";",PyACTS.iam,";",PyACTS.nb,";", PyACTS.nprow,";"
50           ,PyACTS.npcol,";",time1-time0,";",time2 -time1b,";",
```

```
51          time3-time2,";",time4-time3,";"
```

Desde la línea 13 hasta la 19, se realiza la definición de los datos. En este caso la matriz A será una matriz cuadrada $n \times n$ cuyos elementos en la diagonal principal valen n y el resto de elementos vale uno. Esta definición se realiza mediante la rutina que hemos creado exclusivamente para realizar estas pruebas llamada `distGCdata`. Es conveniente remarcar que también se ha implementado una rutina similar en Fortran que define el mismo contenido de la matriz A.

Desde la línea 20 hasta la 24 se definen varios escalares PyACTS que se utilizarán en las sucesivas llamadas a las rutinas de PyPBLAS. A partir de la línea 27 se implementa el algoritmo del método del gradiente conjugado, realizándose la inicialización de los vectores $p^{(0)}, x^{(0)}$ y $r^{(0)}$. A partir de la línea 34, comienza el bucle que finaliza cuando el error sea menor que `eps` o el número de iteraciones mayor que el tamaño del sistema. En dicho bucle se realizan las acciones descritas en el algoritmo 10.1, donde se comprueba la similitud entre el pseudocódigo del algoritmo y el código de Python.

Por otro lado, la implementación de Fortran es considerablemente más compleja y extensa (más de 200 líneas), por lo que preferimos no mostrarla en el presente documento para evitar extendernos en la longitud del mismo. La implementación en Fortran realizará las llamadas en el mismo orden que en este código. Sin embargo, la gestión de las variables es más compleja y el número de parámetros que se le debe pasar a cada rutina es mayor. En ambas implementaciones, las operaciones a realizar son las mismas en Fortran y en Python y requieren el mismo número de iteraciones. Dicho de otro modo, el valor obtenido en cada una de las llamadas a las rutinas de PBLAS y PyPBLAS es el mismo puesto que la rutina que realiza el cálculo y los parámetros pasados son los mismos. La diferencia entre ambas implementaciones radica en el origen de los datos o en cómo estos son tratados en niveles de software superior.

En la figura 10.1 se muestran los tiempos obtenidos en la plataforma Cluster-umh para la resolución de diversos sistemas, variando el orden de la matriz de coeficientes y el número de procesos utilizados. Hemos realizado pruebas para obtener el tiempo de ejecución en sistemas de diferentes tamaños (desde 1000 a 15000) y con diferente número de procesos (1, 2, 4 ó 6). En dicha figura se observa un comportamiento similar en las implementaciones de Fortran y C. Como se puede observar, los tiempos de ejecución son prácticamente similares y ambas implementaciones se comportan del mismo modo frente a diversos tamaños de sistema y número de procesos. Por tanto, la penalización por utilizar

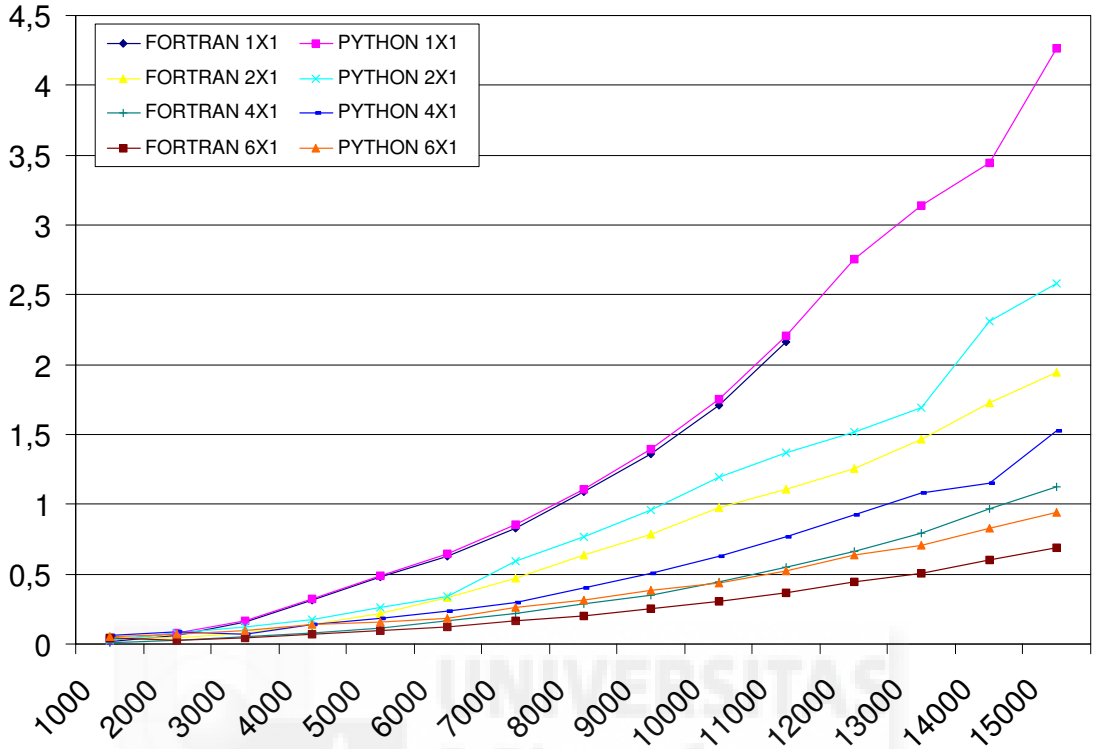


Figura 10.1: Comparativa de tiempos de ejecución en Fortran y Python

los interfaces y programar el algoritmo en un lenguaje de alto nivel es pequeña y podemos considerarla nula frente a la productividad de Python frente a otros lenguajes como C o Fortran. De este modo hemos implementado un código prácticamente igual de eficiente pero mediante un lenguaje cómodo, interactivo y sencillo.

Un detalle importante que se desea resaltar es la mejora en la eficiencia de la memoria desde Python frente a Fortran. Tal y como se observa en la figura 10.1, en las ejecuciones con un único proceso, mediante Python hemos podido resolver el sistema para tamaños superiores a 11000 elementos hasta 15000, mientras que en Fortran no se han podido resolver sistemas superiores a 11000. Este comportamiento se repite para dos procesos y tamaños superiores entre 15000 y 20000 que no se muestran en la figura. Esta facilidad de gestión de memoria en Python se ha observado en la posibilidad de definir variables Numeric de los tamaños necesarios mientras que en Fortran el propio compilador nos informa de la falta de espacio de memoria en la definición de matrices.

En definitiva, utilizando la distribución PyACTS para la implementación del método del gradiente conjugado hemos ahorrado tiempo y esfuerzos en la edición y depuración del algoritmo (siendo ambas características subjetivas y difíciles de medir), y también hemos podido obtener la solución en sistemas de mayor tamaño con el mismo número de procesos. El supuesto inconveniente en la utilización de estos interfaces es la introducción de un tiempo de penalización por la utilización de los mismos, sin embargo las pruebas mostradas en este método nos indican que la penalización es muy pequeña.

10.2. Análisis de componentes principales y EOF

En el presente apartado introduciremos las técnicas de análisis de datos en el campo meteorológico, y mostraremos los conceptos físicos que se enlazan con el tratamiento de las matrices. Posteriormente, se introduce la herramienta PyClimate como distribución disponible para Python que incorpora funciones y rutinas muy útiles en el análisis de la variabilidad climática. Dicha herramienta está implementada para una arquitectura secuencial y por tanto, no es escalable cuando el tamaño de los datos a analizar es elevado. Con el objetivo de conseguir una herramienta paralela y escalable, hemos desarrollado ParPyClimate que incorpora las mismas rutinas que su versión secuencial pero paralelizadas para ser ejecutadas en arquitecturas de memoria distribuida. Mostraremos ejemplos de análisis de componentes principales mediante ParPyClimate y compararemos los resultados y los tiempos de ejecución obtenidos con la herramienta secuencial.

10.2.1. Introducción al análisis y exploración de datos

En este apartado se describen brevemente algunas herramientas estadísticas básicas para la exploración y el análisis de datos meteorológicos y climatológicos. Estas técnicas han sido utilizadas profusamente para explorar, comprender y simplificar los datos disponibles en un problema dado, analizando las relaciones de dependencia entre variables

y eliminando la redundancia. Para una información más detallada de la aplicación de técnicas estadísticas en este ámbito recomendamos la lectura de [5], [116] y [117].

En la mayoría de las ocasiones, además de los factores controlados por el experimentador en un problema dado, existen otros factores desconocidos que dan lugar a una incertidumbre y a un carácter aleatorio en el problema. La probabilidad es una herramienta apropiada en estas situaciones, donde se conocen los posibles resultados, pero no se sabe cuál de ellos ocurrirá en la realización del mismo.

Los sucesos de mayor interés suelen estar asociados a alguna variable numérica relacionada con el experimento (por ejemplo la cantidad de precipitación). El concepto de variable aleatoria permite tratar estas situaciones asignando un número real a cada suceso del espacio muestral (por ejemplo, la cantidad de precipitación, en l/m^2). Cuando el rango de la variable aleatoria es un número finito o contable de elementos, ésta se denomina discreta y, en caso contrario se denomina continua. Dependiendo del tipo de problema, algunas variables pueden considerarse tanto discretas, como continuas. Por ejemplo, la precipitación puede considerarse como variable discreta, con estados (0,1) (correspondiente a la ausencia o presencia del evento de precipitación), pero también puede considerarse como variable continua; en este caso el rango sería el intervalo [0,1] (correspondiente a la cantidad de precipitación).

Para caracterizar la probabilidad de que una variable tome distintos valores (o intervalos de valores) se definen conceptos como funciones de probabilidad, probabilidades conjuntas y probabilidades marginales y condicionadas. Estos conceptos se consideran básicos en estadística y consideramos que no debemos extendernos en una introducción a estos conceptos. Por tanto, recomendamos la lectura de las referencias anteriores si se desea ampliar o recordar dichos conocimientos.

Los conceptos de covarianza y correlación caracterizan la relación lineal existente entre variables continuas de un cierto proceso. La covarianza tiene el defecto de depender de las escalas de los datos. Sin embargo, el coeficiente de correlación es adimensional y caracteriza el grado de la relación lineal existente; el coeficiente de correlación r está relacionado con la covarianza de las dos variables de la siguiente forma:

$$r = \frac{1}{n-1}\sum_{i=1}^{n}(\frac{x_i - \bar{x}}{\sigma_x})(\frac{y_i - \bar{y}}{\sigma_y}) = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

y sus posibles valores están acotados en $-1 \leq r \leq 1$. Los valores extremos corresponden a relaciones lineales perfectas, ya sea en sentido creciente o decreciente, mientras que un

coeficiente de correlación cero es indicativo de que no existe relación lineal entre ambas variables. Por tanto, el concepto de correlación es de gran importancia teórica y práctica para medir la influencia relativa de unas variables sobre otras.

En algunas ocasiones interesa utilizar esta dependencia para poder representar los datos con un número menor de variables, independientes unas de otras. El análisis de Componentes Principales (CPs), también conocido en las ciencias atmosféricas como análisis de Funciones Ortogonales Empíricas (EOF), es una técnica estándar para eliminar la información redundante con la mínima pérdida de variabilidad.

Esto se logra proyectando el conjunto de datos en un nuevo espacio de menor dimensión que el original, donde las nuevas variables (dimensiones) representen aquellas direcciones del espacio donde los datos tienen mayor varianza; en otras palabras, el análisis de CPs es una técnica eficiente para comprimir datos [86]. Este método es especialmente útil en espacios de alta dimensionalidad, donde los datos pueden estar correlacionados en sus distintas componentes y, por tanto, pueden contener mucha información redundante en su descripción.

Un ejemplo típico en meteorología lo constituyen los campos de circulación atmosférica, dados por los valores de una o varias magnitudes (la presión a nivel del mar, etc.) en una rejilla sobre una cierta zona de interés. Dado que estas magnitudes están correlacionadas espacialmente, existirá una gran redundancia en esta forma de expresar los datos. La técnica de componentes principales reduce la dimensión del espacio preservando el máximo de varianza de la muestra. Para ello, la base del nuevo espacio se forma con aquellos vectores donde la muestra proyectada presenta mayor varianza. Los vectores de esta base son de enorme utilidad en meteorología, pues los primeros de ellos pueden corresponder a patrones dominantes como la NAO [72] (en el sentido de la variabilidad de la muestra que representan).

Se parte de una muestra de m datos

$$x_k = (x_1, \ldots, x_{kn})^T,\ k = 1, \ldots, m$$

en un espacio n dimensional con base canónica $\{e_1, \ldots, e_n\}$. Se desea obtener un subespacio de dimension $d < n$ dado por una nueva base $\{f_1, \ldots, f_d\}$ (siendo cada f_j una combinación lineal de los vectores e_i de la base canónica). Para un vector x se tendrá:

$$x = \sum_{i=1}^{n} e_i x_i \approx \bar{x} = \sum_{i=1}^{d} f_i x_i.$$

El criterio para obtener este subespacio es que la muestra proyectada en el nuevo espacio $\{\bar{x}_k, k = 1, ..., m\}$ tenga la máxima varianza posible. Es decir, fijada una dimensión d, el problema consiste en encontrar los vectores f_i que proyectan la muestra con varianza máxima. El cálculo matemático para obtener los vectores óptimos es sencillo y consiste en estimar la matriz de varianzas y covarianzas a partir de la muestra de datos. Los vectores propios (o Funciones Ortogonales Empíricas, EOFs) de esta matriz son los nuevos vectores f_i y los correspondientes valores propios indican la varianza explicada (la varianza de la muestra proyectada sobre el vector). Los coeficientes de un punto (dato) en la nueva base se denominan Componentes Principales (CPs).

Dada la muestra x_k, se puede estimar la matriz de varianzas y covarianzas C_x, donde cada elemento σ_{ij} representa la covarianza de los datos entre la variable i y la j del espacio original:

$$\sigma_{ij} = \langle (x_{ki} - \mu_i)(x_{kj} - \mu_j) \rangle_k \,;\; \mu_i = \langle x_{ki} \rangle_k \,;\; \mu_j = \langle x_{kj} \rangle_k \;\;,$$

donde $\langle\rangle$ denota el promedio aritmético. Esta matriz de varianzas y covarianzas es cuadrada y simétrica por lo que se puede calcular una nueva base ortogonal encontrando sus valores propios λ_i (que serán reales) y los correspondientes vectores propios f_i:

$$C_x f_i = \lambda_i f_i, \;\; i = 1, \ldots, n.$$

Es fácil resolver este problema cuando n es pequeño pero, a medida que aumenta la dimensión, el problema se complica debido al posible mal condicionamiento de la matriz. En estos casos es necesario aplicar métodos numéricos eficientes como la Descomposición en Valores Singulares (SVD), que proporciona una factorización de la matriz C_x de la forma (ver [87] para más detalles):

$$C_x = F \Lambda F^T,$$

donde Λ es una matriz diagonal que contiene los valores propios λ_i (ordenados de forma decreciente) de C_x, y las columnas de F son los vectores propios f_i. Además F es una matriz ortogonal y F^T es su inversa. De esta manera, si hacemos la proyección:

$$\bar{x}_k = F^T x_k = \begin{pmatrix} f_{11} & \cdots & f_{1n} \\ \vdots & \cdots & \vdots \\ f_{n1} & \cdots & f_{nn} \end{pmatrix} \begin{pmatrix} x_{1k} \\ \vdots \\ x_{nk} \end{pmatrix} \;\;,$$

se tendrá el elemento de la muestra proyectado sobre la base de autovectores de C_x, mientras que la proyección inversa se obtendrá mediante $x_k = F\bar{x}_k$. Esta proyección tiene las siguientes propiedades:

- Componentes incorrelacionadas: $\langle \bar{x}_{ki}\bar{x}_{kj} \rangle_k = 0,\ i \neq j$.
- $Var(\bar{x}_{ki}) = \lambda_i,\ i = 1, \ldots, n$.
- $\sum_{i=1}^{n} Var(x_{ki}) = \sum_{i=1}^{n} Var(\bar{x}_{ki}) = \sum_{i=1}^{n} \lambda_i$.

Dado que los vectores se eligen en orden decreciente de varianza, es posible hacer un recorte de dimensión reteniendo la máxima varianza posible (obviamente, si se quiere conservar toda la varianza habrá que tomar $d = n$). Si se toman sólo las d primeras EOFs, cada elemento de la muestra se podrá expresar aproximadamente como:

$$x_k \approx \tilde{F}\tilde{F}_T = \begin{pmatrix} f_{11} & \cdots & f_{d1} \\ \vdots & \cdots & \vdots \\ \vdots & \cdots & \vdots \\ f_{1n} & \cdots & f_{dn} \end{pmatrix} \begin{pmatrix} f_{11} & \cdots & \cdots & f_{1n} \\ \vdots & \cdots & \cdots & \vdots \\ f_{d1} & \cdots & \cdots & f_{dn} \end{pmatrix} x_k \ ,$$

donde $\tilde{F}$ representa a la matriz F truncada a los d primeros vectores propios. El vector $\bar{x}_k = \tilde{F}^T x_k$ de dimensión $d \times 1$ contendrá las CPs del patrón x_k, es decir, las componentes del vector en el nuevo espacio de dimensión d. Para recuperar la dimensión original, el vector de CPs se proyectará mediante $\tilde{F}^T x_k$, obteniendo una aproximación del vector original (mejor cuanto mayor sea la dimensión d del espacio proyector).

Por tanto, la técnica de componentes principales obtiene la proyección lineal óptima en sentido de máxima varianza explicada y de mínimo error cuadrático de reconstrucción.

Para eliminar el efecto de las distintas escalas de cada una de las componentes del vector, es conveniente estandarizar los datos (componente a componente) como paso previo a realizar el análisis. De esta forma se evita que las variables de mayor varianza se hagan dominantes en el análisis. En el caso de datos atmosféricos, se han de estandarizar por separado los valores correspondientes a cada punto de la rejilla, de forma que la variabilidad del patrón en toda la extensión espacial sea homogénea. Otro procedimiento para tener en cuenta este problema consiste en utilizar la matriz de correlaciones en lugar de la de varianza-covarianza para realizar el análisis [76].

10.2.2. PyClimate

PyClimate [98, 101] es un paquete de Python de libre distribución que incorpora rutinas útiles para el análisis de la variabilidad climática. Entre algunas de sus utilidades se incluyen funciones de acceso a datos desde ficheros netCDF [89], análisis de funciones ortogonales empíricas (EOF), análisis de correlación canónica, descomposición en valores singulares (SVD) de conjuntos de datos y un conjunto de filtros digitales.

PyClimate proporciona rutinas para el análisis de componentes principales (CPs) y vectores ortogonales (EOFs) en uno de sus módulos denominado `svdeofs.py`. La utilidad de este análisis ha sido descrita en el apartado 10.2.1 y mostramos en éste una herramienta útil que implementa los cálculos necesarios.

A continuación describimos de forma breve los principales métodos que incorpora el módulo `svdeofs.py`:

- `SVDEOFs.__init__(self, dataset)`

  Dado un conjunto de datos bidimensional $N \times M$, dónde N indica el número de muestras temporales y M el número de puntos de observación, este constructor obtiene Z,Λ,E. $Z = [z_1] \dots [z_L]$ es una matriz $N \times L$ donde cada columna z_k es un vector de longitud N que contiene la k-ésima Componente Principal (CP). Cada elemento λ_k del vector Λ, representa la varianza relacionada para cada k-ésima CP. Finalmente, la matriz $E = [e_1] \dots [e_L]$ contiene un vector ortogonal (EOFs) para cada columna. El máximo de EOFs será el mínimo entre N y M en función del rango de la matriz.

- `SVDEOFs.svdeofs(self, pcscaling=0)`

  Devuelve las funciones empíricas ortogonales (EOFs) de un conjunto de datos. Desde la versión 1.1 se soportan conjuntos de datos multidimensionales siempre y cuando la primera de las dimensiones represente la escala temporal. Por otro lado, si `pcscaling=0` entonces los valores propios están escalados por las varianzas, los componentes principales z_k están escalados de modo que su varianza es la varianza relacionada con cada componente ($\langle z_k, z_j \rangle = \sigma_{kj}\lambda_j,\ e_k \cdot e_j = \delta_{kj}$).

- `SVDEOFs.pcs(self, pcscaling=0)`

  Devuelve los componentes principales.

- `SVDEOFs.eigenvalues(self)`
  Devuelve las varianzas asociadas a cada EOFs.

- `SVDEOFs.eofsAsCorrelation(self)`
  Devuelve las EOFs escaladas o la correlación de los CP a partir de la siguiente fórmula:
  $$Corr(z_i, X_j) = \frac{\sigma(z_i)}{\sigma(X_j)} E_{ij}$$
  donde X_j representa la serie temporal del elemento j. La correlación entre las series temporales originales se devuelven en cada columna.

Además de los descritos, el módulo incorpora otros que no explicaremos por no ser útiles en el ejemplo que deseamos mostrar y para evitar extendernos en el mismo.

Ejemplo de análisis EOFs y CPs

A continuación mostramos una serie de ejemplos en los que se realiza el análisis de Componentes Principales o también llamado análisis de Funciones Ortogonales Empíricas (*EOFs*). El primer ejemplo y más sencillo se muestra como ejemplo 1 en la página oficial de la distribución de PyClimate. A continuación mostramos el código que analizaremos para entender el proceso de obtención de las EOFs.

```
1  from Numeric import *
2  from Scientific.IO.NetCDF import *
3  from pyclimate.svdeofs import *
4  from pyclimate.ncstruct import *
5
6  inc = NetCDFFile("ncepslp.djf.nc")
7  slp = inc.variables["djfslp"]
8  areafactor = cos(inc.variables["lat"][:])**2
9  slpdata = slp[:,:,:,:] * areafactor[NewAxis,NewAxis,:,NewAxis]
10 oldshape = slpdata.shape
11 slpdata.shape = (oldshape[0], oldshape[1]*oldshape[2]*oldshape[3])
12
13 pcs, lambdas, eofs = svdeofs(slpdata)
14 dims = ("lat", "lon")
```

```
15 onc = nccopystruct("firstEOF.nc", inc, dims, dims, dims)
16 eof1 = onc.createVariable("eof1", Float32, dims)
17 eof1[:,:] = ( reshape(eofs[:,0], oldshape[2:]) / areafactor[:,
18         NewAxis] ).astype(Float32)
19 onc.close()
```

Como podemos observar, la longitud del código es bastante reducida pese a la funcionalidad que implementa y que explicaremos a continuación. En las primeras líneas (1 a 4) se importan los módulos necesarios. En la línea 6 abrimos una conexión con un fichero netCDF que contiene una variable (`djfslp`) que representa la presión atmosférica a nivel del mar en un rango de las dimensiones latitud y longitud. Dicha variable netCDF se asigna al identificador `slp`. Este fichero ha sido obtenido desde la propia web de PyClimate para probar este ejemplo y contiene los datos de la presión atmosférica a nivel del mar en el área del atlántico (ver figura 10.2). En las líneas 8 a 11 ponderamos el dato de cada latitud por el cuadrado del coseno de la latitud y convertimos la matriz de cuatro dimensiones (`time, level, lat, lon`) en una matriz con sólo dos dimensiones (`time` y la concatenacion del resto a lo largo de la nueva dimensión). La razón de este cambio de dimensiones viene impuesta por la función `svdeofs` en la línea 13, puesto que la matriz deberá ser bidimensional. Esta llamada devuelve las componentes principales (`pcs`), los valores propios (`lambdas`) y las funciones empíricas ortogonales (`eofs`).

Una vez hemos obtenido las EOFs se procede a guardarlas en un fichero netCDF para su posterior tratamiento y visualización. Para ello se crea un nuevo fichero copiando la estructura (es decir, las dimensiones fundamentalmente) del fichero de entrada (línea 15).

En la línea 16 se define una nueva variable en el fichero netCDF de salida, denominada `eof1` que representa el primer vector ortogonal obtenido. Mediante la indexación `eofs[:,0]` se selecciona el primer vector que recordemos tiene un valor para cada `lat`,`lon` y `level` (es decir, el número total de elementos será `lat`×`lon`×`level`, donde únicamente se ha definido un nivel en este fichero). Mediante el comando `reshape` generamos de nuevo una matriz con dimensiones `lat` y `lon` para almacenarla en el fichero netCDF.

El resultado final es un fichero netCDF que almacena el primer valor EOF para cada punto en función de la latitud y longitud. Para representar este fichero de una manera más visual, utilizaremos la herramienta de visualización y análisis GrADS [1]. Esta herramienta interactiva permite la manipulación y visualización de datos relativos a ciencias ambientales. El formato de los datos puede ser binario, GRIB, netCDF o HDF-SDS. La herramienta GrADS se distribuye de forma libre y es ampliamente utilizada por la

comunidad científica en este ámbito de la investigación.

A continuación mostramos el contenido del *script* `ejemplo1.sh` que contiene las instrucciones necesarias para visualizar el contenido de las EOFs obtenidas anteriormente.

```
1  /usr/local/grads-1.8sl11/bin/gradsnc -pb <<EOF
2  sdfopen firstEOF.nc
3  set grads off
4  set gxout shaded
5  d eof1
6  run cbarn
7  draw title NAO pattern generated by ejemplo1.py
8  enable print eof1.gm
9  print
10 disable print
11 quit
12 EOF
13
14 /usr/local/grads-1.8sl11/bin/gxeps -c -i eof1.gm -o ejemplo1.eps
15 rm -f eof1.gm
```

En la primera parte del *script*, hacemos uso de la utilidad `gradsnc` que abre el archivo netCDF (línea 2), visualiza la variable `eof1` (línea 5), e imprime la gráfica en el archivo `eof1.gm` (líneas 8 a 10). En la segunda parte del *script*, convertiremos mediante la utilidad `gxeps` el archivo en formato .gm a un formato .eps más conocido y estándar.

En la figura 10.2 se representa la imagen generada por el *script*. La asignación de colores para cada valor de la variable `eof1` la ha realizado GrADS, así como la utilización de los valores de longitud y latitud acotados a los valores de las dimensiones definidas en el archivo netCDF. Esta imagen sirve para visualizar cuál es el primer modo que concentra la mayor parte de información de la serie temporal calculada.

En el ejemplo mostrado, el archivo de datos se ha obtenido desde la propia página de la distribución PyClimate. Sin embargo, podemos obtener más datos en formato netCDF del Climate Diagnostics Center (`www.cdc.noaa.gov`) que proporciona un amplio conjunto de datos de diferentes variables climatológicas (temperatura del aire, humedad, salinidad oceánica, etc). Dentro de esta página se proporcionan archivos de diferentes fuentes, en nuestro caso hemos seleccionado la fuente nombrada como *CDC Derived NCEP Reanalysis Products Pressure Level*.

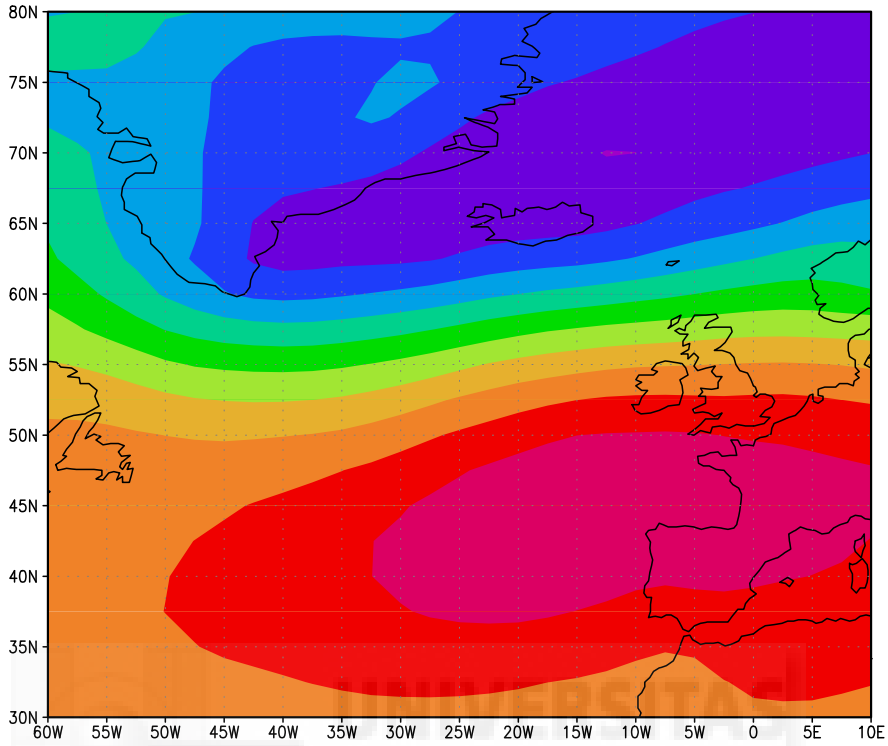


Figura 10.2: Representación del primer EOF obtenido en los datos del Atlántico Norte

En este origen de datos, podemos obtener un archivo netCDF que contiene los datos de todo el globo en una malla cada 2.5 grados de latitud y longitud (144×73 puntos en total, indicado por las variables `lat` y `lon`), para 17 niveles diferentes de presión (`level`) durante un intervalo temporal definido. Este intervalo temporal varía en función del archivo que deseemos descargar. El archivo con el intervalo temporal más reducido con el que hemos realizado pruebas es `air.mon.ltm.nc` que contiene la media de cada mes durante un año para una malla 144×73 que cubre todo el globo y para 17 niveles de presión atmosférica. En total, tendremos $17 \times 12 \times 144 \times 73$ elementos en el archivo netCDF que generan un fichero de 8.5MB.

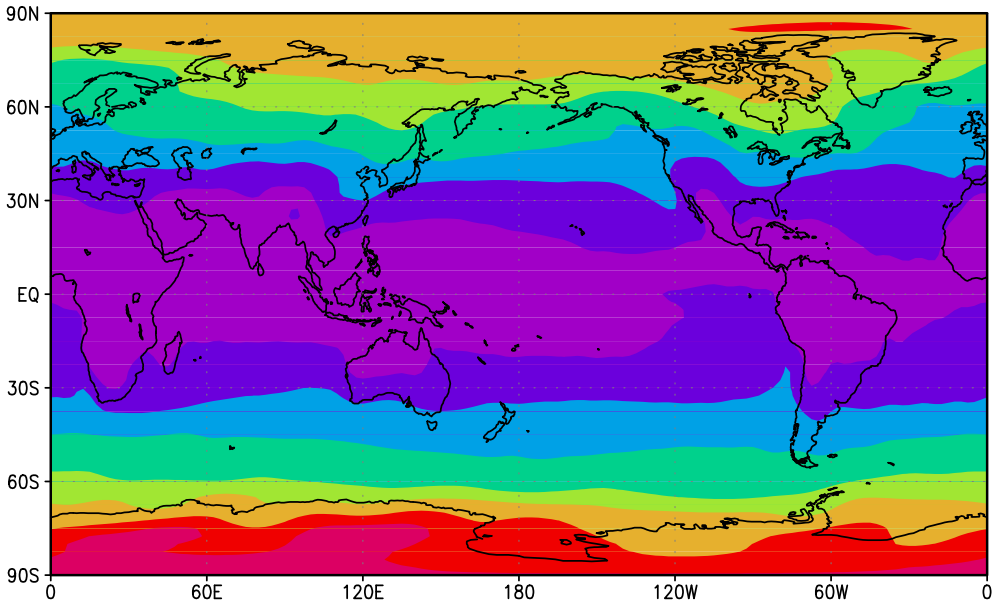
Otro ejemplo, que corresponde con el archivo `air.day.ltm.nc`, contiene la media diaria de la temperatura en cada punto (latitud, longitud y nivel) en el último año, teniendo por tanto 365 muestras en el eje temporal. El tamaño total del archivo netCDF es de 1.3 GB. Por otro lado, el archivo `air.mon.mean.nc` contiene la media de la temperatura para cada mes desde 1948 (formando 703 medidas temporales) en cada punto de una malla

144×73 que cubre todo el globo y para 17 niveles de presión atmosférica. Por tanto, en este caso tendremos una matriz de cuatro dimensiones de tamaño $703 \times 17 \times 144 \times 73$ que genera un fichero netCDF de 2.5 GB.

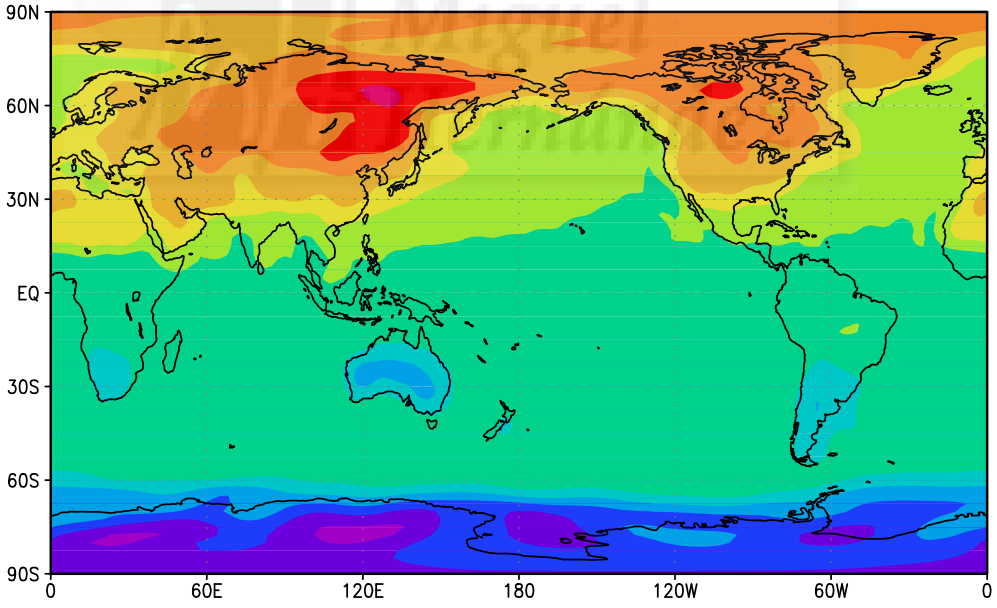
Se desea destacar que la presión atmosférica es un dato comúnmente utilizado en el análisis climático y tiene una alta correlación con la altitud. En los archivos netCDF descargados, se proporcionan 17 niveles de presión atmosférica en lugar de altitud. Por ejemplo, una presión atmosférica de 1000 hPa se corresponde con el nivel del mar, y una presión atmosférica de 200 hPa se corresponde con una altura de 11580 metros aproximadamente.

Modificando ligeramente el código de ambos *scripts* (`.py` y `.sh`) mostrados anteriormente, podemos realizar la descomposición en CP y EOFS a los datos descargados. En las figuras 10.3 y 10.4 se representan los dos primeros EOF obtenidos en el análisis de CPs para los archivos `air.day.ltm.nc` y `air.mon.mean.nc`, respectivamente. Para cada uno de los archivos, mostramos la representación de la primera EOF y de la segunda EOF a nivel 0, es decir a nivel del mar. Debemos destacar que la covarianza entre ambas representaciones es 0, de forma visual, esto se aprecia observando la poca similitud entre ambas imágenes. Además, en ambas figuras se indica el porcentaje de la varianza total que representa cada una de las EOFs representadas. Por otro lado, las EOFs se obtienen ordenadas de forma decreciente según su importancia en la varianza total. Por ejemplo, la figura 10.3(a) representa un 99.72 % de la varianza total de la presión atmosférica a nivel del mar, sin embargo, la segunda EOF representa únicamente el 0.25 % de la varianza total. Esto nos permite obtener información de la importancia de cada una de las funciones empíricas ortogonales obtenidas.

Como ya se ha comentado, el tamaño del archivo de datos `air.mon.mean.nc` es elevado (2.5 GB) y la variable que indica la temperatura del aire tiene unas dimensiones de $703 \times 17 \times 144 \times 73$. Por tanto, se deben realizar 17 análisis de CPs y EOFs para cada malla cuyo requerimiento sea que la primera dimensión se corresponda con el eje temporal. Por tanto, se deberán realizar 17 descomposiciones SVD a una matriz de tamaño 703×10512. Consideramos este tamaño elevado, sin embargo en otro tipo de análisis podemos llegar a utilizar archivos muy superiores a éste. Si deseamos realizar este tipo de analisis con un conjunto mayor de datos, deberemos crearlo previamente puesto que el sitio CDC no distribuye datos superiores a los mostrados anteriormente. De este modo, hemos obtenido los archivos de la temperatura del aire para diferentes años desde 1998 a 2005. Concatenando los datos para el nivel del mar (1000hPa) obtenemos un conjunto de 2922 muestras

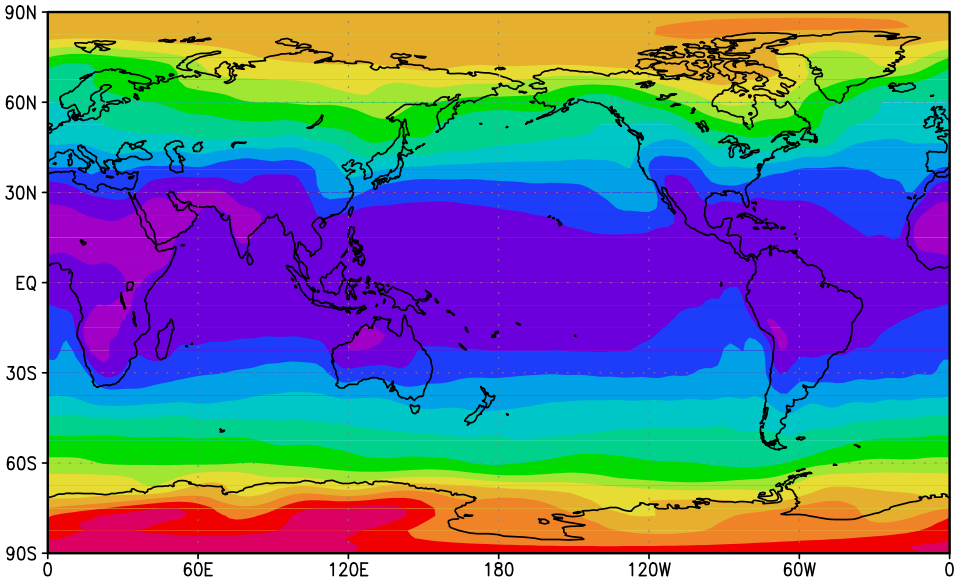


(a) EOF1 99.72 %

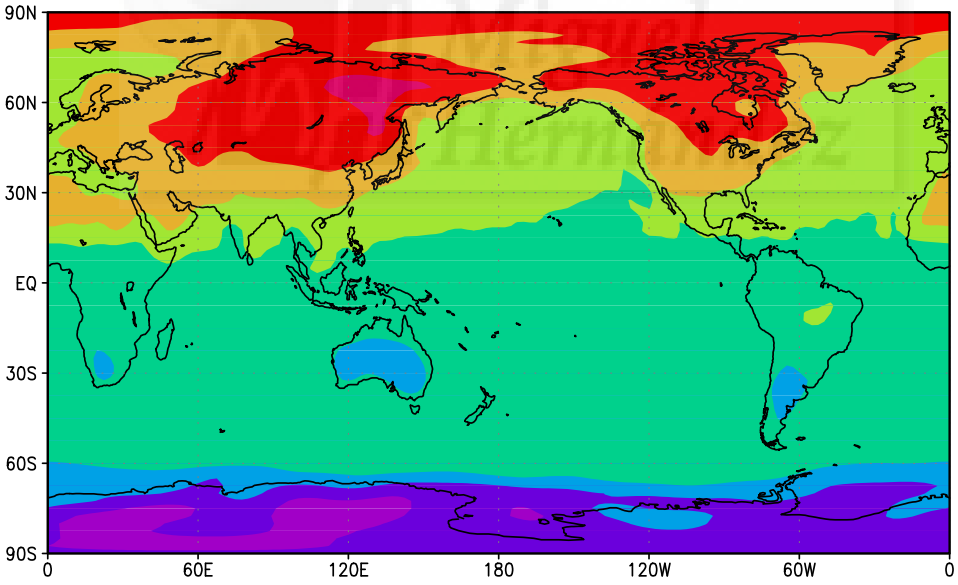


(b) EOF2 0.253 %

Figura 10.3: Representación de las dos primeras EOFs para air.day.ltm.nc



(a) EOF1 99.60 %



(b) EOF2 0.34 %

Figura 10.4: Representación de las dos primeras EOFs para air.mon.mean.nc

temporales en cada punto de la malla, formando una matriz de tamaño $2922 \times 73 \times 144$. Este archivo, que hemos denominado `air.1988to2005.nc`, ocupa 245MB, pero contiene únicamente un nivel de presión; en el caso de concatenar los 17 niveles de presión nuestro archivo ocuparía aproximadamente 4.17GB.

En la figura 10.5 se representan los tiempos necesarios para realizar el análisis EOF en la plataforma Cluster-umh para los diferentes archivos de datos mostrados en cada columna. Debido a los tiempos elevados que se necesitan para el análisis de los archivos grandes, hemos aplicado una escala logarítmica en el eje vertical para poder representar los valores de una manera más cómoda. En definitiva, se observa que para tamaños elevados, el tiempo crece considerablemente y por tanto se plantea la necesidad de una herramienta escalable que disminuya considerablemente estos tiempos.

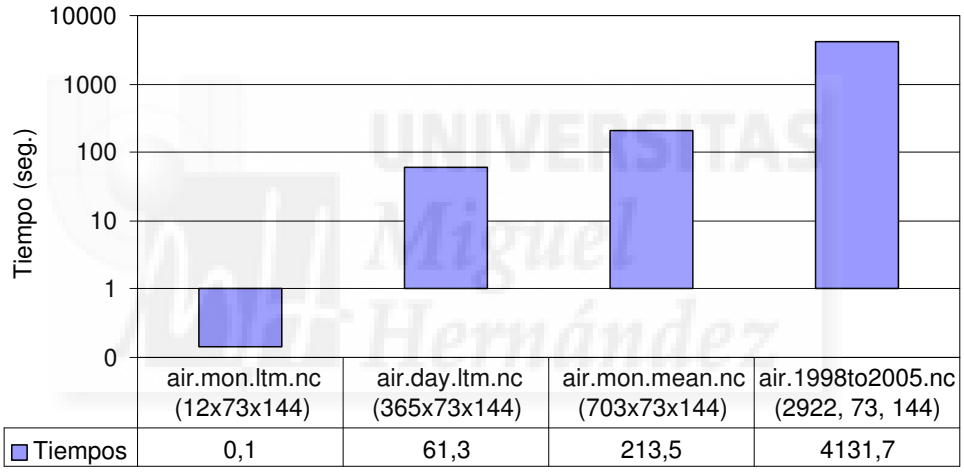


Figura 10.5: Tiempos de ejecución para el análisis de CPs y EOFs

Los tiempos necesarios para realizar el análisis de CPs y EOFs se incrementan de forma considerable con el tamaño de las datos a analizar. Por ejemplo, si en lugar de medias mensuales desde 1948, obtuviéramos las medias diarias de todos los días, el tamaño de la variable multidimensional `air` sería de $21090 \times 17 \times 144 \times 73$. La fuente de datos que hemos utilizado no distribuye estos datos en un único archivo sino que los segmenta por años debido al elevado tamaño que tendría un archivo netCDF con estas características. Este tamaño requiere de una plataforma de altas prestaciones para una ejecución secuencial, tanto por tiempos como por los recursos necesarios de memoria para almacenar los datos. Por este motivo, propondremos en el siguiente apartado, una solución que realizará las

mismas funcionalidades, pero en un entorno paralelo permitiendo una escalabilidad frente al aumento en el tamaño del sistema a analizar.

10.2.3. ParPyClimate

En el apartado anterior hemos descrito la utilidad de los análisis de CPs y EOFs en datos climatológicos y el uso de la herramienta PyClimate que implementa los cálculos necesarios. Sin embargo, hemos observado que el tiempo de cómputo crece considerablemente a medida que aumenta el tamaño de la matriz de datos.

Presentamos en este apartado ParPyClimate como un primer paso para obtener una herramienta capaz de realizar tareas similares a su versión secuencial pero con un comportamiento escalable para poder ser ejecutado en sistemas de memoria distribuida.

Análisis del método secuencial

La distribución PyClimate incorpora diferentes módulos que proporcionan rutinas útiles en el análisis climático y que se describen en su manual de referencia accesible en [101]. En este caso, el módulo que centra nuestro interés es `PyClimate.svdeofs` y la rutina que incorpora `svdeofs()` será analizada para estudiar sus posibilidades de paralelización.

Por este motivo mostramos a continuación el código de la rutina `svdeofs()`, el cual analizaremos para localizar qué acciones pueden ser paralelizables y utilizar las herramientas adecuadas en ese caso. Por tanto, nos centraremos en las operaciones que se realizan y no en las razones por las que se realizan, puesto que nuestro interés es paralelizar el algoritmo sin entrar en detalle en las razones físicas para cada operación. Si se desea más información respecto a las manipulaciones de las matrices recomendamos la lectura de [100].

```
1 def svdeofs(dataset, pcscaling=0):
2   residual = pmvstools.center(dataset)
3   field2d = len(dataset.shape)==2
4   if not field2d:
```

```
5       residual, oldshape = ptools.unshape(residual)
6     A,Lh,E = SVD(residual)
7     normfactor = float(len(residual))
8     L = Lh*Lh/normfactor
9     E = Numeric.transpose(E)
10    pcs = A*Lh
11    if pcscaling == 0:
12      pass
13    elif pcscaling == 1:
14      E = E * Numeric.sqrt(L)[NA,:]
15      pcs = pcs / Numeric.sqrt(L)[NA,:]
16    else:
17      raise pex.ScalingError(pcscaling)
18    if not field2d:
19      E = ptools.deunshape(E, oldshape[1:]+E.shape[-1:])
20    return pcs,L,E
```

Observamos en la línea 2 que se hace uso de la rutina `center` proporcionada en el módulo `pyclimate.mvarstatools`. Esta rutina (cuyo código no mostraremos) obtiene la media temporal para cada dato y resta cada elemento con su media. Se comprueba que la matriz tiene dos dimensiones (línea 3), y en ese caso se llama a la rutina SVD (línea 6) que es un alias de la rutina `singular_value_decomposition` que se encuentra en el módulo `LinearAlgebra` de la librería Numeric. De este modo comprobamos que PyClimate hace uso de la distribución Numeric descrita en el apartado 3.2.1 que a su vez incorpora el archivo `lapack_lite.c` (una versión reducida de la librería LAPACK con sus principales funciones). En definitiva, será la rutina `sgesvd` (o `dgesvd` dependiendo del tipo de dato) quien realice en último término la descomposición en valores singulares.

Una vez obtenida la descomposición en valores singulares mediante la ejecución de la línea 6 (donde `residual`$= A \cdot Lh \cdot E$), se convierten los valores propios en varianzas mediante la operación de la línea 8. Para conseguir las componentes principales (`pcs`), las varianzas (`L`), y las funciones ortogonales (`E`), se realizan una serie de cálculos y normalizaciones que se describen en mayor profundidad en [100].

En resumen, la rutina `svdeofs()` que proporciona PyClimate, prepara la matriz de datos para obtener su descomposición en valores singulares y posteriormente normaliza y convierte los datos a matrices con una representación de análisis más clara. Por tanto, si queremos obtener una rutina paralela deberemos implementar las mismas funcionalidades

mediante rutinas que se ejecuten en plataformas de memoria distribuida.

En la figura 10.6 se representa la estructura del código y de las librerías que hace uso nuestra herramienta ParPyClimate. En un nivel superior, se situa el intérprete de Python en cada nodo, el cual llevará a cabo las tareas programadas mediante un *script* `.py` o bien de forma interactiva. En estas instrucciones se realiza la importación del paquete ParPyClimate y por tanto, de todos los módulos incluidos en el paquete (`svdeofs`, `svd`, `LinearFilter`, ...).

El método `svdeofs.svdeofs()` realiza llamadas a algunos métodos de PyBLACS, PyPBLAS, y PyScaLAPACK. Las capas de software de la distribución PyACTS ya fueron descritas en el apartado 5.2, sin embargo hemos creído convenientes mostrarlas también en la figura 10.6 para proporcionar una visión de conjunto de la pila de software utilizado.

Ejemplo de análisis EOFs y CPs mediante librerías paralelas

La implementación de ParPyClimate se muestra en este capítulo como un ejemplo de aplicación de la distribución PyACTS. Por esta razón, creemos conveniente mostrar el código de la versión paralela de ParPyClimate. A continuación mostramos el *script* análogo al mostrado en el apartado 10.2.2 pero cuya implementación es totalmente escalable gracias a la utilización de librerías paralelas.

En este código se observa en algunos comienzos de línea tres puntos suspensivos ("..."). Estos puntos significan que los comandos en esa línea y sucesivas, son los mismos que en su versión secuencial y hemos preferido no repetirlos para simplificar el ejemplo.

```
1 from Numeric import *
2 from Scientific.IO.NetCDF import *
3 from parpyclimate.svdeofs import *
4 from parpyclimate.ncstruct import *
5 from PyACTS import *
6 nb0=int(sys.argv[2])
7 nb1=nb0 / PyACTS.nprocs
8 PyACTS.gridinit(nb=nb1,nprow=PyACTS.nprocs,npcol=1)
9 if PyACTS.iread:
```

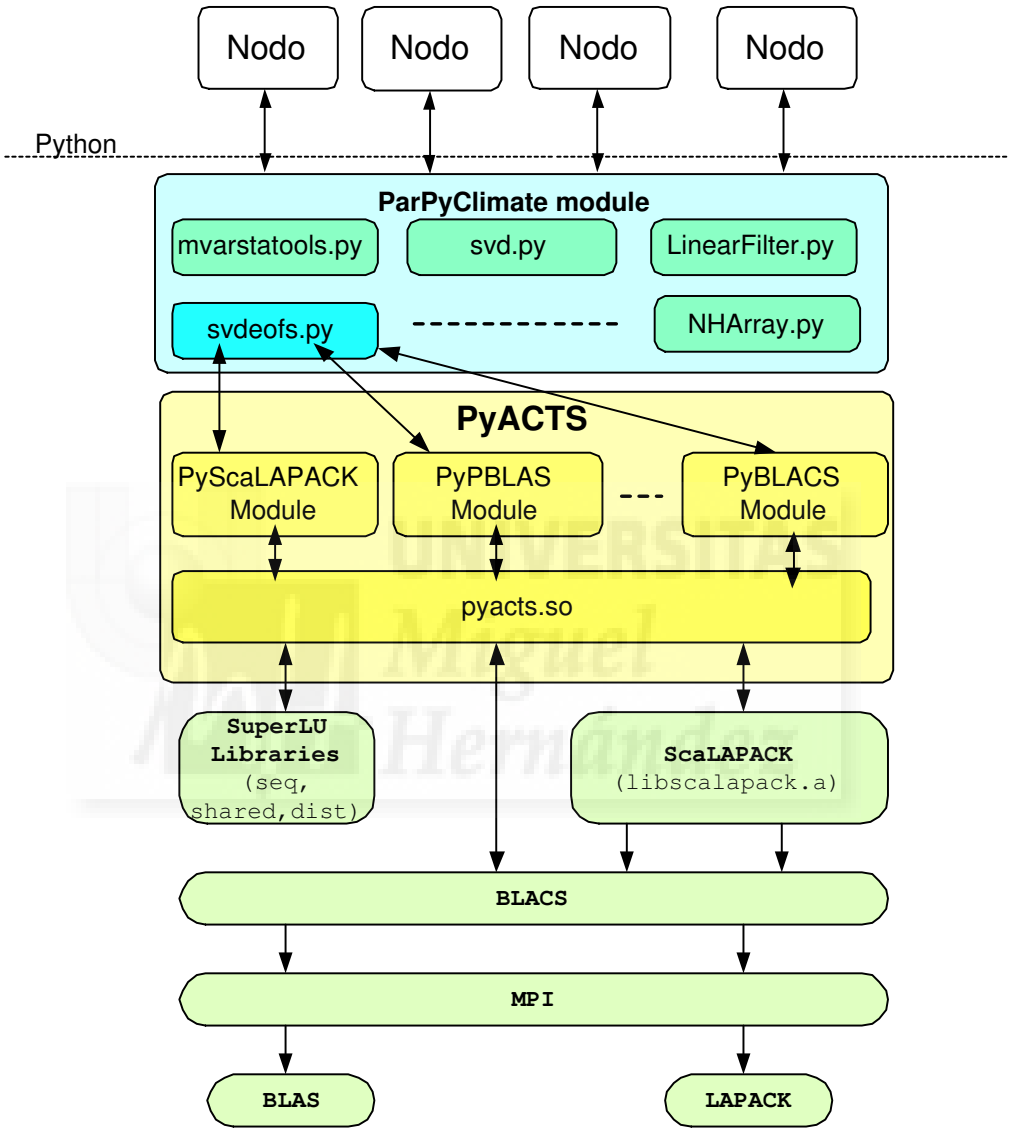


Figura 10.6: Estructura de ParPyClimate y librerías utilizadas en el análisis de CPs y EOFs

```
10          inc = NetCDFFile("air.mon.ltm.nc")
11  ...
12          slpdata.shape = (oldshape[0], oldshape[1]*oldshape[2])
13  else:
14          slpdata=None
15  slpdata=Num2PyACTS(slpdata,1)
16  pcs, lambdas, eofs = svdeofs(slpdata,1)
17  pcs=PyACTS2Num(pcs)
18  eofs=PyACTS2Num(eofs)
19  if PyACTS.iread:
20          dims = ("lat", "lon")
21  ...
22          onc.close()
23  PyACTS.gridexit()
```

De este código deseamos destacar que la inicialización de la malla de procesos (línea 8) se realiza para una configuración `nprocs` $\times$ `1` que será la configuración adecuada para la posterior distribución de datos. En este ejemplo, un único proceso (aquel con `PyACTS.iread=1`) realiza la lectura de la variable del fichero netCDF a una variable Numeric. En la línea 15 se realiza la conversión de la variable Numeric a otra de tipo matriz PyACTS.

En la línea 16 se realiza la llamada a la rutina de ParPyClimate cuyo nombre es el mismo que en la implementación secuencial. Los parámetros devueltos por `svdeofs` son las componentes principales y las funciones empíricas ortogonales en dos matrices PyACTS (`pcs` y `eofs`) y las varianzas mediante una variable Numeric (`lambdas`). Para poder escribir en un fichero netCDF mediante la herramienta de ScientificPython, necesitamos convertir las matrices PyACTS en variables Numeric (línea 17 y 18).

Del mismo modo que en el caso secuencial, el proceso con `iread=1` será el encargado de almacenar los datos en el fichero netCDF del mismo modo como se hizo en el ejemplo secuencial. En este ejemplo, el acceso a los ficheros netCDF se realiza de modo secuencial. En apartados posteriores mostraremos un ejemplo del mismo *script* pero con un acceso a los ficheros netCDF en paralelo mediante la herramienta PyPnetCDF presentada en el capítulo 9.

Implementación del método paralelo

Una vez hemos presentado un ejemplo de utilización de la rutina paralela de ParPyClimate, mostramos a continuación el código de dicha rutina para ilustrar el manejo de las rutinas de PyACTS en una aplicación con una funcionalidad determinada.

En este caso, la matriz de datos utilizada como parámetro de entrada es una matriz PyACTS, es decir, se encuentra distribuida entre los nodos de la malla de procesos y si se desea realizar alguna operación algebraica, debería ser con la rutina de PyPBLAS o PyScaLAPACK correspondiente.

Por ejemplo, en la línea 3 se necesita centrar los datos con respecto a la dimensión temporal. Para ello se hace uso del método `center` del módulo `parpyclimate.mvarstatools`. Este método también ha sido paralelizado aunque no mostraremos el código en este ejemplo.

Una vez centrados, llamamos a la rutina de PyScaLAPACK indicando que también se calculen las matrices U y V^T. El resultado se devuelve mediante dos matrices PyACTS (`A`, `E`) y una matriz Numeric `Lh`. A partir de la línea 5, se realizan una serie de manipulaciones de las matrices para normalizar los datos y utilizar los valores propios como varianzas de las funciones empíricas ortogonales.

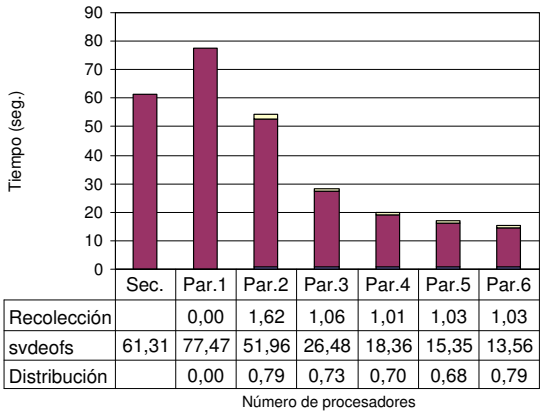
```
1  def svdeofs(dataset, pcscaling=0):
2    ACTS_lib,field2d=1,1
3    residual = pmvstools.center(dataset)
4    Lh,A,E,info= PySLK.pvgesvd(residual,jobu='V',jobvt='V')
5    normfactor = float(residual.desc[PyACTS.m_])
6    L = Lh*Lh/normfactor
7    ACTS_mone=PyACTS.Scal2PyACTS(-1,ACTS_lib)
8    ACTS_one=PyACTS.Scal2PyACTS(1,ACTS_lib)
9    ACTS_zero=PyACTS.Scal2PyACTS(0,ACTS_lib)
10   Array_zeros=PySLKTools.makeACTSArray(residual.desc[PyACTS.n_],
11         residual.desc[PyACTS.m_],dataset.data.typecode())
12   E=PyPBLAS.pvtran(ACTS_mone, E, ACTS_zero,Array_zeros)
13   I=Numeric.identity(len(Lh), Lh.typecode())
14   I=I*Lh
15   I=Num2PyACTS(I,ACTS_lib)
16   pcs=PySLKTools.makeACTSArray(len(Lh),len(Lh),Lh.typecode())
```

```
17    pcs=PyPBLAS.pvgemm(ACTS_mone,A,I,ACTS_zero,pcs)
18    if pcscaling == 0:
19        pass
20    elif pcscaling == 1:
21        IL=Numeric.identity(len(L),L.typecode())
22        IL=IL*Numeric.sqrt(L)
23        IL=Num2PyACTS(IL,ACTS_lib)
24        Epcsca=PySLKTools.makeACTSArray(E.desc[PyACTS.m_],
25          E.desc[PyACTS.n_], E.data.typecode())
26        Epcsca=PyPBLAS.pvgemm(ACTS_one,E,IL,ACTS_zero,Epcsca)
27        L_inv=1/Numeric.sqrt(L)
28        L_inv=L_inv*Numeric.identity(len(L),L.typecode())
29        L_inv=Num2PyACTS(L_inv,ACTS_lib)
30        pcsca=PySLKTools.makeACTSArray(pcs.desc[PyACTS.m_],
31          pcs.desc[PyACTS.n_], pcs.data.typecode())
32        pcsca=PyPBLAS.pvgemm(ACTS_one,pcs,L_inv,ACTS_zero,pcsca)
33        E,pcs=Epcsca,pcsca
34    else:
35        raise pex.ScalingError(pcscaling)
36    if not field2d:
37        E = ptools.deunshape(E, oldshape[1:]+E.shape[-1:])
38    return pcs,L,E
```

El resultado devuelto por el método paralelo es prácticamente igual que en el método secuencial, únicamente aparecen algunas diferencias lógicas de redondeo a partir del sexto número decimal que no se aprecian en una representación gráfica por lo que las figuras obtenidas en las versiones paralelas serán las mismas que las figuras 10.3 y 10.4.

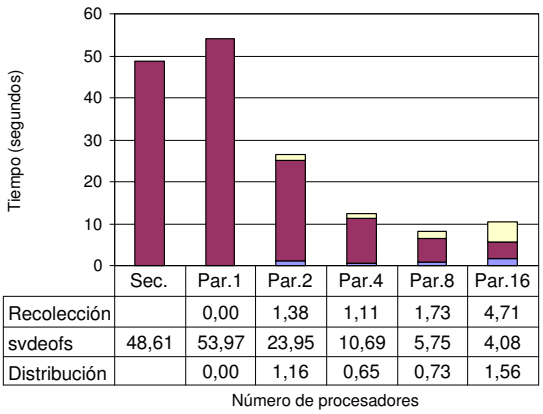
Comparativas de rendimiento

En la figura 10.5 se ha comprobado cómo los tiempos de ejecución para la obtención de las primeras EOFs crecen considerablemente con el tamaño del archivo a analizar. En dicha figura hemos aumentado la escala temporal, pero del mismo modo, si tenemos una malla de puntos más fina provocaría un aumento del tamaño total de la matriz. En el presente apartado compararemos los tiempos de ejecución obtenidos en la versión



	Sec.	Par.1	Par.2	Par.3	Par.4	Par.5	Par.6
Recolección		0,00	1,62	1,06	1,01	1,03	1,03
svdeofs	61,31	77,47	51,96	26,48	18,36	15,35	13,56
Distribución		0,00	0,79	0,73	0,70	0,68	0,79

(a) Cluster-umh



	Sec.	Par.1	Par.2	Par.4	Par.8	Par.16
Recolección		0,00	1,38	1,11	1,73	4,71
svdeofs	48,61	53,97	23,95	10,69	5,75	4,08
Distribución		0,00	1,16	0,65	0,73	1,56

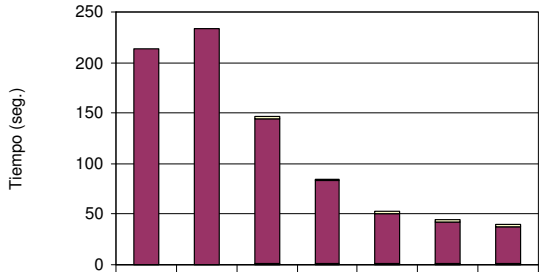
(b) Seaborg

Figura 10.7: Comparativa de tiempos para el archivo `air.day.ltm.nc`

secuencial (PyClimate) y las funciones propuestas (ParPyClimate) para realizar el análisis EOFs a un conjuntos de datos de diferentes dimensiones.

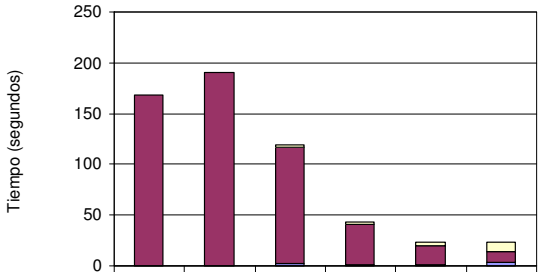
En la figura 10.7 se muestran los tiempos necesarios para calcular las EOFs de la temperatura del aire en el nivel del mar para el archivo `air.day.ltm.nc`. Por tanto, la matriz contiene $365 \times 73 \times 144$ elementos de coma flotante de doble precisión, es decir 8 bytes cada uno. Hemos realizado los cálculos para las plataformas Cluster-umh y Seaborg; los tiempos para cada una de las plataformas se muestran en las figuras 10.7(a) y 10.7(b), respectivamente. En cada uno de los gráficos y en sus correspondientes tablas se representa el tiempo invertido en obtener las EOFs, es decir, el tiempo consumido en la llamada a la rutina `svdeofs` de PyClimate para el caso secuencial o de ParPyClimate en la versión paralela. Además, hemos creído conveniente indicar los tiempos consumidos en la distribución de la matriz de datos (mediante la instrucción `Num2PyACTS`) y la posterior recolección de las EOFs (mediante `PyACTS2Num`). Llama la atención en la figura 10.7(b), que el tiempo invertido en la distribución de la matriz de datos entre 16 procesos supera al tiempo de ejecución de `svdeofs`. Además el tiempo total de ejecución (distribución + svdeofs + recolección) en 16 procesos es superior al tiempo total obtenido con 8 procesos. Por tanto, nos encontramos con un ejemplo que demuestra que aumentando el número de nodos no siempre redunda en tiempos mejores.

De forma similar, en la figura 10.8 se representan los tiempos necesarios para realizar el mismo análisis pero para el archivo `air.mon.mean.nc`. Este archivo contiene la media



	Sec.	Par.1	Par.2	Par.3	Par.4	Par.5	Par.6
Recolección		0,03	2,40	2,06	2,05	2,04	2,04
svdeofs	213,5	233,0	142,3	81,46	49,63	41,36	35,97
Distribución		0,05	1,60	1,42	1,37	1,32	1,55

(a) Cluster-umh



	Sec.	Par.1	Par.2	Par.4	Par.8	Par.16
Recolección		0,08	2,17	2,27	3,53	9,40
svdeofs	167,64	190,43	115,53	39,61	17,87	11,39
Distribución		0,07	1,86	1,32	1,49	3,08

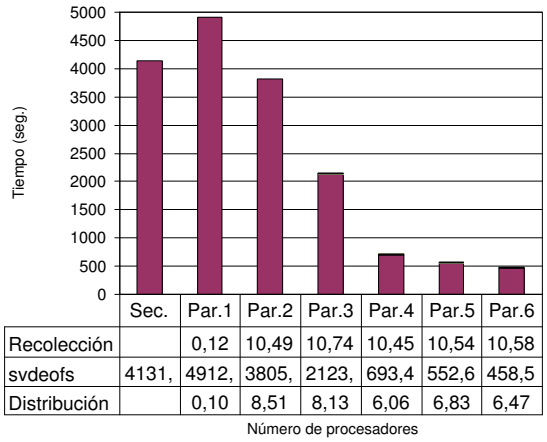
(b) Seaborg

Figura 10.8: Comparativa de tiempos para el archivo `air.mon.mean.nc`

mensual desde 1948, formando una matriz de $703 \times 73 \times 144$ elementos en coma flotante de doble precisión. En las figuras 10.8(a) y 10.8(b) se representan los tiempos obtenidos para obtener las EOFs tanto con PyClimate como con ParPyClimate para diferente número de procesos.

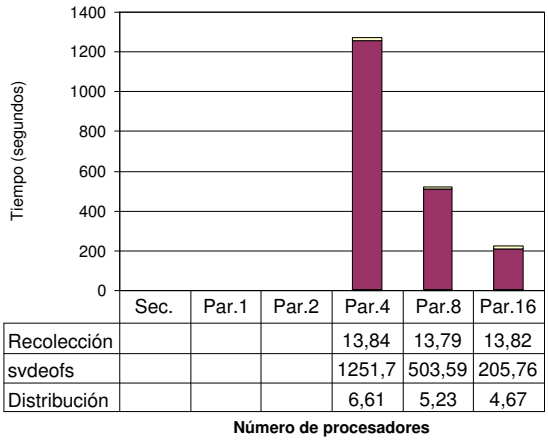
Tanto en la figura 10.7 como en la 10.8, observamos que en si utilizamos ParPyClimate en un sólo proceso los tiempos de ejecución de `svdeofs` son superiores al caso secuencial. La razón de este aumento no se debe a la presencia de interfaces de alto nivel (puesto que PyClimate y Numeric también ofrecen interfaces a las rutinas de LAPACK), sino a que las rutinas de PyScaLAPACK (que hace uso de ScaLAPACK) realizan llamadas a rutinas de comunicaciones MPI. Sin embargo, al ejecutarlo en un sólo proceso realmente no se produce una comunicación entre los procesos pero sí una llamada a la rutina y un tratamiento en memoria de los datos. Por otro lado, para un proceso se observan tiempos de comunicaciones muy bajos, ésto se debe fundamentalmente, a que la rutina `Num2PyACTS` cuando la malla de procesos es 1×1 no realiza comunicación de datos mediante las rutinas de PyBLACS, sino que únicamente debe realizar una traspuesta de la matriz para disponer en memoria los datos ordenados por columnas conforme al orden utilizado en Fortran.

Por último, en la figura 10.9(a) y 10.9(b) se muestran los tiempos de ejecución necesarios para obtener las EOFs en Cluster-umh y Seaborg, respectivamente. En este caso, el tamaño de la matriz a analizar es elevado, con un total de $2922 \times 73 \times 144$ elementos de coma flotante de doble precisión. Comprobamos en esta figura cómo los tiempos



	Sec.	Par.1	Par.2	Par.3	Par.4	Par.5	Par.6
Recolección		0,12	10,49	10,74	10,45	10,54	10,58
svdeofs	4131,	4912,	3805,	2123,	693,4	552,6	458,5
Distribución		0,10	8,51	8,13	6,06	6,83	6,47

(a) Cluster-umh



	Sec.	Par.1	Par.2	Par.4	Par.8	Par.16
Recolección				13,84	13,79	13,82
svdeofs				1251,7	503,59	205,76
Distribución				6,61	5,23	4,67

(b) Seaborg

Figura 10.9: Comparativa de tiempos para el archivo `air.1998to2005.nc`

invertidos en distribuir y recoger la imagen desde y hacia el proceso origen han crecido considerablemente con respecto a las figuras anteriores. Llama la atención, en la figura 10.9(b), la presencia de algunas celdas en blanco. La ausencia de dato es debida a que no se ha podido realizar el análisis de EOFs por falta de recursos de memoria, puesto que en la operación para la rutina de PyClimate como para ParPyClimate (1 y 2 procesos) nos informa de un fallo de segmentación en memoria. En definitiva, en uno o dos procesos el sistema no tiene recursos suficientes. Por tanto, ParPyClimate aporta una solución a un problema que no podía ser resuelto en un caso secuencial en la plataforma Seaborg.

Por otro lado, creemos conveniente destacar que en la figura 10.9(a), el tiempo de ejecución total para obtener las EOFs en 6 procesos es casi 9 veces inferior al tiempo obtenido con la herramienta secuencial. Se obtiene una eficiencia de 1.47 con respecto al caso secuencial. Esta supereficiencia tiene su explicación en una menor paginación de memoria en el caso de tener los datos distribuidos en un mayor número de procesadores.

10.3. Conclusiones

Los problemas analizados en este capítulo han permitido constatar que las distribuciones PyACTS y PyPNetCDF se configuran como unas herramientas escalables, eficientes y de fácil aprendizaje para la implementación de prototipos o simulaciones en las que los requisitos computacionales y de memoria son exigentes.

En el primer ejemplo mostrado en la sección 10.1, la implementación del algoritmo del gradiente conjugado ha exigido la concatenación de sucesivas llamadas a las rutinas de PyPBLAS dentro de un bucle por lo que el número de veces de utilización de los interfaces a Python es elevado. Sin embargo, ésto no se ha traducido en un incremento considerable de los tiempos de ejecución con respecto a la aplicación desarrollada exclusivamente en Fortran. Esto nos lleva a concluir que una aplicación correctamente desarrollada en Python que haga uso de la distribución PyACTS o de alguno de sus módulos, presentaría unos tiempos de ejecución similares que otra aplicación desarrollada en un lenguaje de más bajo nivel como C o Fortran. De forma general, los tiempos de ejecución son menores para aplicaciones desarrolladas en lenguajes como C o Fortran pero los tiempos de desarrollo y edición se reducen considerablemente. Por tanto, pensamos que PyACTS es una herramienta adecuada para el rápido desarrollo de una aplicación por usuarios no avanzados en la computación paralela que se pueden permitir una muy pequeña pérdida de eficiencia frente a la productividad obtenida.

En la segunda aplicación desarrollada en la sección 10.2, hemos presentado una aplicación actual PyClimate, muy útil para el análisis de la variabilidad climática, pero con un problema de escalado cuando el tamaño del conjunto de datos a tratar es elevado. Como el núcleo de los cálculos de la rutina paralelizada se basaba en una rutina proporcionada en LAPACK a través del paquete `Numeric`, hemos conseguido paralelizar la aplicación sustituyendo dicha rutina por la rutina análoga de PyScaLAPACK. Los resultados reflejados en las figuras 10.7, 10.8, y 10.9 han puesto de manifiesto que la aplicación paralela creada presenta reducciones importantes en los tiempos de ejecución y permite además, obtener una herramienta escalable que permita el análisis de un elevado volumen de datos imposible de realizar en una plataforma secuencial.

Los resultados mostrados en ambas aplicaciones confirman a PyACTS como un paquete de Python que incorpora la eficiencia y escalabilidad de las herramientas de altas prestaciones incluidas en ACTS de una forma cómoda, sencilla y flexible.

Capítulo 11

Conclusiones y líneas futuras

En este capítulo, en su sección 11.1 se hace un resumen del trabajo realizado y se exponen las principales conclusiones que se pueden extraer del mismo, haciendo una síntesis de todas las conclusiones incluidas al final de cada capítulo. En la sección 11.2 expondremos cómo se ha realizado la difusión de este trabajo y la respuesta obtenida a dicha difusión. En la sección 11.3, se perfilan algunas posibles líneas de trabajo a desarrollar en el futuro.

11.1. Resumen

En la actualidad, multitud de librerías numéricas se encuentran ampliamente extendidas y han demostrado su robustez, precisión y portabilidad. La colección ACTS engloba un conjunto de herramientas software muy útiles para los desarrolladores de programas en plataformas paralelas y difiere de otras herramientas para la computación paralela en el hecho que enfoca su funcionalidad en los niveles inferiores de una aplicación, proporcionando librerías que pueden ser utilizadas desde C, Fortran 77/90 o C++. Para introducir (en el capítulo 2) las principales características de los computadores paralelos en los que son ejecutadas estas rutinas, hemos revisado previamente, en el capítulo 1, el estado actual de las arquitecturas uniprocesador.

Por otro lado, en el capítulo 3 hemos presentado Python como una herramienta útil y sencilla desde la cual poder implementar las funcionalidades de una aplicación reservando las tareas computacionalmente más pesadas a módulos o librerías programadas a más bajo nivel. Se han introducido sus principales características incidiendo de forma especial en la facilidad que proporciona para integrar librerías desarrolladas por otros lenguajes. Además, Python permite dividir nuestra aplicación en módulos que pueden ser reutilizados en otros programas y dispone de una gran variedad de módulos no incluidos en la distribución oficial y desarrollados por terceros. Algunos de estos módulos, aquellos que han sido de interés en nuestro trabajo, se han presentado en la sección 3.2. En definitiva, Python surge como un lenguaje sencillo que permite un desarrollo rápido y cómodo de un prototipo. Además, la facilidad en la reutilización de librerías desarrolladas en C o Fortran posibilita la utilización del software de computación científica desde un código escrito en Python. Sin embargo, el proceso de creación de los interfaces a librerías en C o Fortran es complejo y tedioso. Además, debido al elevado número de rutinas a las que se pretende acceder desde Python, hemos recurrido a la utilización de herramientas como SWIG o F2PY presentadas en la sección 3.3.

En el capítulo 4 hemos descrito las principales características del software de computación científica y en la sección 4.3 hemos presentado las librerías de paso de mensajes y la colección ACTS como ejemplos de este tipo de software que centran el trabajo presentado. Dentro de la colección ACTS, en los apartados 4.3.3, 4.3.4 y 4.3.5 hemos profundizado con mayor nivel de detalle en las características de las librerías BLACS, PBLAS y ScaLAPACK, respectivamente.

La parte más importante del trabajo realizado ha sido el desarrollo de la distribución PyACTS. En el capítulo 5, hemos proporcionado una descripción detallada de su estructura, así como una definición de la matriz PyACTS y que nos permite una cómoda interacción entre las diferentes librerías de la colección ACTS y entre otros formatos de datos. En este sentido, hemos descrito las funciones auxiliares de inicialización o liberación del entorno paralelo y las funciones de verificación, de consulta o de conversión de matrices PyACTS. Debido a la complejidad de la estructura y al elevado número de rutinas a las que se ha implementado el acceso desde Python, hemos descrito detenidamente este proceso de creación de los módulos de Python (sección 5.6) y de la librería de objetos compartidos (sección 5.7).

En el paquete PyACTS, se han incluido los módulos PyBLACS, PyPBLAS y PyScaLAPACK que se han descrito en profundidad en los capítulos 6, 7 y 8, respectivamente.

En PyACTS, se han intentado respetar los nombres de las rutinas de las librerías análogas de la colección ACTS para facilitar la integración y migración de una rutina en Python a C o Fortran y viceversa. Sin embargo, gracias a las características de `Numeric`, el nombre de cada rutina de PyACTS es independiente del tipo de dato con el que va a trabajar. Además, para minimizar el número de parámetros, en las rutinas de estos módulos hemos establecido parámetros opcionales con valores por defecto que simplifican la ejecución de las rutinas.

Por un lado, el módulo PyBLACS permite la ejecución de rutinas de comunicaciones sin una pérdida considerable del ancho de banda. La inicialización de una malla de procesos, o la difusión de una matriz se convierten en acciones sencillas de realizar desde un intérprete en paralelo de Python, mientras que en Fortran o C representan mayor dificultad.

PyPBLAS implementa diversas operaciones del álgebra lineal con matrices densas clasificadas en tres niveles atendiendo a su complejidad. La mayor sencillez y similar eficiencia de las rutinas para Python en todos sus niveles con respecto a un lenguaje más clásico como C o Fortran, permiten la utilización de PyPBLAS en la implementación de aplicaciones que requieran este tipo de manipulación de matrices de grandes dimensiones. Además, en PyPBLAS (y posteriormente en PyScaLAPACK) apreciamos una mejora en la utilización de los recursos de memoria disponibles.

El módulo PyScaLAPACK ha sido el más complejo de los desarrollados y se divide en dos tipos de rutinas: driver y computacionales. A su vez, cada uno de estos grupos se divide en subgrupos atendiendo al tipo de problema con el que esta relacionado el algoritmo que implementan. En este módulo, muchas de las rutinas requieren de la generación de espacios de memoria de trabajo o de vectores de pivotación cuyos tamaños dependen de las dimensiones del problema. En la implementación para Python, hemos ocultado la existencia de estos requisitos, generándolos la rutina PyScaLAPACK de forma transparente al usuario.

Las comparativas de rendimiento obtenidas en los tres módulos con respecto a sus interfaces para C o Fortran han demostrado que la ganancia en productividad y sencillez no se traduce en una pérdida de eficiencia significativa. El usuario de PyACTS está dispuesto a sacrificar menos de un 5 % de eficiencia si desarrolla su aplicación de una manera mucho más rápida y cómoda.

El número total de rutinas en los tres módulos es superior a 425 y para cada una de ellas se han realizado pruebas de ejecución, se ha documentado su funcionalidad y su

interfaz y se muestra un ejemplo de utilización en el manual de usuario de PyACTS [34] que hemos desarrollado.

Con el objetivo de proporcionar una herramienta potente y a su vez sencilla y transparente a un usuario no avanzado en la computación paralela, hemos optado por determinadas configuraciones y definiciones del entorno paralelo que se establecen por defecto si el programador no indica lo contrario. Las pruebas mostradas en la sección 8.4, confirman la validez en la elección de una malla de procesos tan cuadrada como sea posible y un tamaño de bloque cuadrado 64×64.

Como complemento a PyACTS, hemos presentado en el capítulo 9, la distribución PyPNetCDF. El estándar netCDF establece un sistema portátil y eficiente para la gestión de datos de grandes dimensiones y PyPnetCDF nos permite el acceso en paralelo a ficheros netCDF desde Python. Para mejorar la usabilidad de PnetCDF, en PyPnetCDF hemos definido las clases `PNetCDFFile` y `PNetCDFVariable` que permiten un acceso a los datos mediante un entorno orientado a objetos. En el apartado 9.4.4 hemos comprobado cómo las rutinas `PNetCDF2PyACTS` y `PyACTS2PNetCDF` permiten la lectura de una variable netCDF a una matriz PyACTS y la escritura de una matriz PyACTS a una variable de un fichero netCDF, respectivamente. De este modo hemos conseguido una integración entre ambos paquetes mediante una sencilla línea muy intuitiva para el programador. En el apartado 9.4.5 se muestran las pruebas de eficiencia de PyPnetCDF, comparándolas con la herramienta análoga secuencial para Python y con la versión de PnetCDF disponible para C. Por un lado, los códigos de ejemplo mostrados en la sección 9.4, reflejan la sencillez de uso; por el otro, si comparamos los tiempos de PyPNetCDF con los obtenidos para la herramienta secuencial, la escalabilidad depende en gran medida de la arquitectura del sistema de ficheros de la plataforma donde se ejecuten. Comparando PyPnetCDF con el rendimiento obtenido para PnetCDF desde un programa en C, los incrementos en tiempo de ejecución por la utilización de una interfaz de mayor nivel son muy pequeños, casi despreciables.

En el capítulo 10 se presentan dos aplicaciones que hacen uso de PyACTS. La implementación del algoritmo del gradiente conjugado nos ha permitido comparar la eficiencia de una aplicación completa desarrollada en Python con respecto a la misma aplicación desarrollada con un lenguaje más clásico como Fortran. En este caso, el comportamiento de los tiempos obtenidos ha sido prácticamente similar. La segunda aplicación desarrollada se corresponde con la paralelización de una de las rutinas proporcionadas en la herramienta para el análisis de la variabilidad climática PyClimate. La utilización de PyScaLAPACK

nos ha permitido paralelizar una herramienta secuencial y conseguir una escalabilidad total cuando el volumen de datos climáticos es elevado.

11.2. Difusión de las distribuciones

La principal aportación de esta tesis doctoral es el desarrollo de las distribuciones PyACTS y PyPnetCDF. Ambas librerías se ponen a disposición de la comunidad científica mediante la publicación del código fuente en la dirección `http://www.pyacts.org`. Además de las fuentes de ambas librerías, se proporcionan ejemplos así como un manual de usuario de PyACTS disponible *online* en html o bien en un archivo `pdf`. Por otro lado, se proporciona la dirección de correo `support@pyacts.org`, en la cual los programadores pueden realizar sus consultas o informar de determinados *bugs*. Consideramos por tanto, que los usuarios que desarrollan aplicaciones de computación científica tienen suficiente información para aprender a utilizar estas distribuciones.

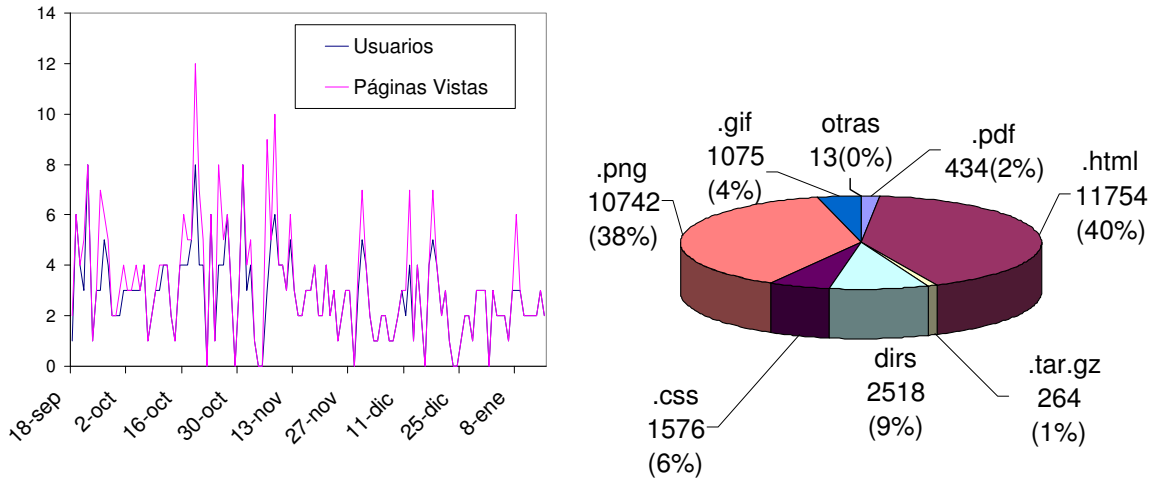
Además del sitio mencionado, también hemos distribuido PyACTS en el repositorio de proyectos más importante actualmente en Internet y cuya dirección es `http://sourceforge.net/projects/pyacts`. Consideramos esta dirección como complementaria a la primera, y garantiza un sitio alternativo en el que el usuario pueda descargar la distribución si la primera no se encuentra disponible.

Con respecto a la distribución de PyPNetCDF, hemos creado la dirección `http://www.pyacts.org/pypnetcdf`. En este enlace se proporciona el paquete, así como ejemplos de utilización del mismo.

Con el objetivo de dar a conocer esta herramienta a los programadores que desarrollen aplicaciones científicas en Python, hemos publicado las distribuciones PyACTS y PyPNetCDF en la "enciclopedia" (conocida como *wiki*) oficial de Python:

- `http://wiki.python.org/moin/NumericAndScientific`.
- `http://wiki.python.org/moin/NumericAndScientific/Formats`.

Para ilustrar el interés de los usuarios por la distribución PyACTS, mostramos en la figura 11.1(a) el número de visitas y de páginas vistas desde el 18 de septiembre de 2006



(a) Visitas y Páginas vistas

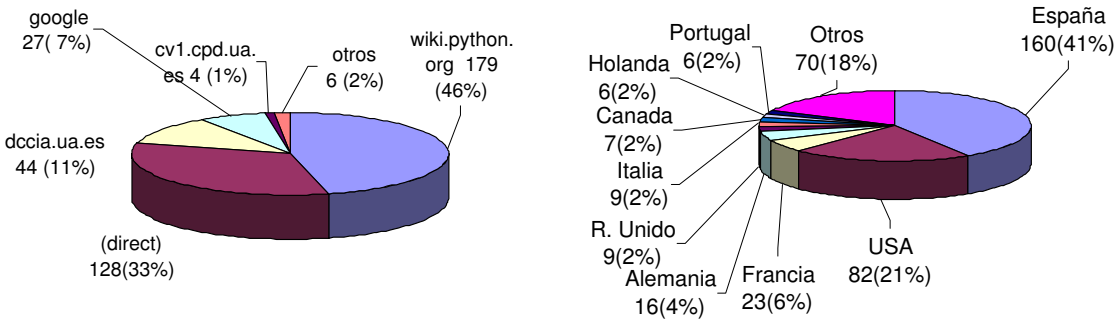
(b) Descargas de ficheros

Figura 11.1: Visitas y Descargas en el sitio `www.pyacts.org`

hasta el 15 de enero de 2007. El sitio oficial de PyACTS ha sido visitado un promedio de 1,8 usuarios/día, sumando un total de 388 visitas. Para obtener estos datos hemos utilizado la herramienta conocida como *Google Analytics*.

En la figura 11.1(b) se representan las descargas de ficheros realizadas desde el 18 de julio de 2006 hasta el 15 de enero de 2007. En este periodo, comprobamos que se han realizado un total de 264 descargas de la distribución PyACTS y 434 descargas del manual de usuario de PyACTS. Deseamos destacar que el intervalo de tiempos para la figura 11.1(b) es diferente al resto puesto que está obtenido con una herramienta diferente (*analog 6.0*) de libre distribución, que permite el análisis de los *logs* del servidor web.

También puede resultar interesante conocer desde qué otras páginas o buscadores han accedido los usuarios al sitio PyACTS (este concepto se denomina "origen de referencia"). Los diferentes orígenes de referencia de PyACTS se muestran en la figura 11.2(a) y en ella se aprecia que un 46 % de usuarios accede por la publicación de las librerías en la enciclopedia oficial de Python. En la figura 11.2(b) y 11.3 se muestran los orígenes por ubicación geográfica de las visitas realizadas. Consideramos que el hecho de que la página principal de PyACTS y la enciclopedia de Python sean en inglés, ha contribuido a que la mayor parte de los accesos se hayan realizado desde países de habla inglesa.



(a) Orígenes de referencia

(b) Orígenes por paises

Figura 11.2: Orígenes por referencia y por países de los usuarios de `www.pyacts.org`

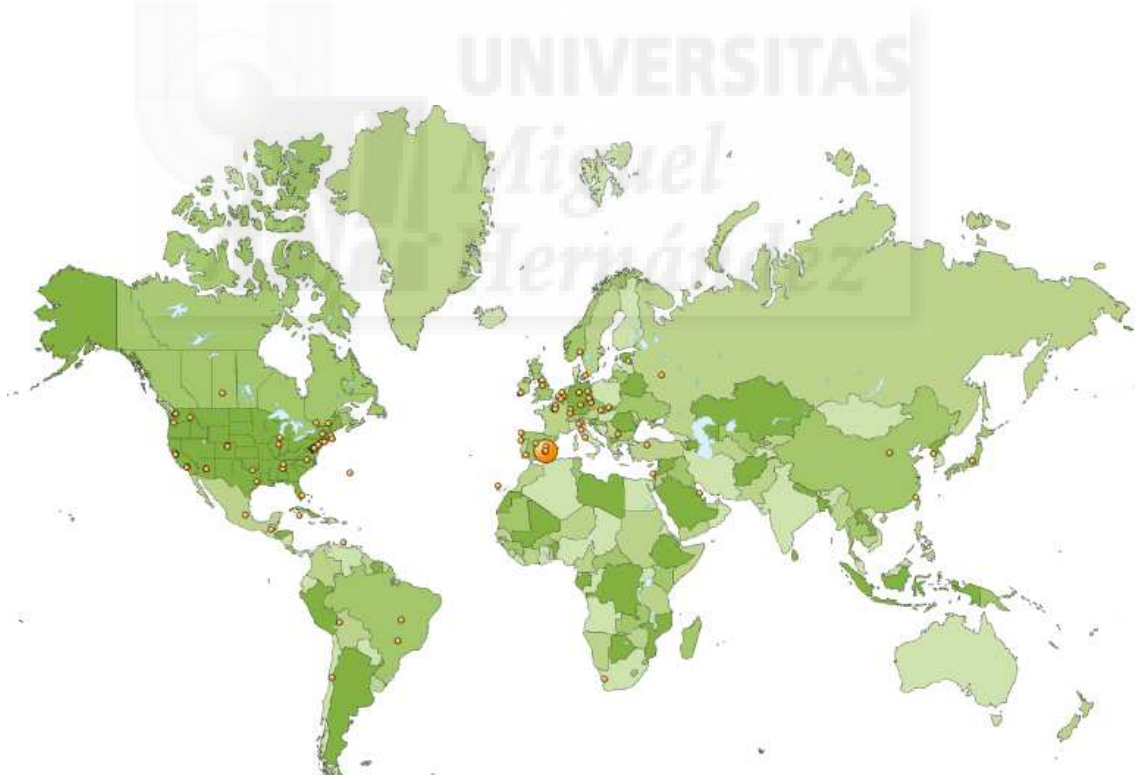


Figura 11.3: Visitas realizadas por origen geográfico

11.3. Líneas futuras

Los resultados obtenidos en este trabajo animan a nuevas investigaciones y desarrollos en este campo, especialmente en la mejora de la librería, así como en la incorporación de nuevos módulos y funcionalidades. En esta sección se enumeran posibles líneas de trabajo a realizar como continuación de esta tesis.

- Actividades de difusión y soporte:

  Es importante dar a conocer PyACTS y PyPnetCDF a la comunidad científica, para que exista un grupo de usuarios estable que la utilice en aplicaciones reales y cuyas experiencias con su uso sirvan para introducir mejoras.

- Incorporar nuevas aplicaciones:

  Existen multitud de aplicaciones reales programadas en Matlab o Python que hacen uso de la librería LAPACK o de las librerías BLAS, con el correspondiente problema de escalado. La paralelización de dichas aplicaciones con PyScaLAPACK y PyPBLAS se propone como solución para la creación de una aplicación robusta y escalable.

- Nuevos módulos de ACTS:

  La incorporación de nuevos módulos de la colección ACTS como SuperLU, Global Arrays, etc. permitirá una mayor integración entre el software científico disponible en la misma. La estructura global de PyACTS ha sido definida y diseñada para facilitar la incorporación de nuevos módulos.

- Nuevos módulos numéricos:

  Si se considera adecuado, PyACTS permite la integración de librerías de computación científica aunque no pertenezcan a la colección ACTS.

- Nuevas funciones de conversión:

  La conversión de datos desde matriz PyACTS a otros formatos de datos no implementados como por ejemplo, un archivo de Excel o un fichero de datos HDF5, puede ampliar el conjunto de usuarios que puedan hacer uso de la distribución.

- Servicio web de computación de altas prestaciones:

  Como Python es un lenguaje interpretado y no se necesita el proceso de compilación y enlazado de librerías, resulta interesante el desarrollo de una aplicación web

que permita a los usuarios introducir sus *scripts* en Python mediante un formulario web y ejecutarlos en una plataforma paralela de memoria distribuida. De este modo, se incorpora a la potencia de la distribución PyACTS, la facilidad en el envío de *scripts* mediante la utilización de un navegador.

- Generación de nueva documentación y traducción al inglés:

  La distribución PyACTS incorpora un manual de usuario escrito en castellano. Sin embargo, hemos comprobado en la figura 11.2(b) que una parte importante de los usuarios provienen de países de habla inglesa. Esto nos motiva a plantearnos la traducción del manual de usuario al idioma inglés para consolidar los usuarios interesados en la misma.

  Por otro lado, en la distribución PyPnetCDF no se proporciona un manual de usuario similar al de PyACTS. La documentación adecuada de PyPnetCDF permitiría ampliar el conjunto de usuarios interesados en la misma.

UNIVERSITAS
Miguel
Hernández

Bibliografía

[1] J. M. Adams, R. Budich, L. Calori, B. E. Doty, W. Ebisuzaki, M. Fiorino, T. Holt, and D. Hooper. Grid Analysis and Display System (GrADS). http://grads.iges.org/grads/grads.html.

[2] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. The Benjaming/Cummings Publishing Company, Inc, 1994. Redwood City.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1995.

[4] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. *Numerical Python*. Lawrence Livermore National Laboratory, Livermore, CA 94566, 2001. UCRL-MA-128569; http://numpy.sourceforge.net.

[5] J. J. Ayuso. *Predicción Estadística Operativa en el INM*. Vol-34 de Monografías del Instituto Nacional de Meteorología. Ministerio de Medio Ambiente, 1997.

[6] S. Balay, K. Buschelman, B. Gropp, D. Kaushik, L. C. McInnes, and B. Smith.

[7] D. M. Beazley. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. Department of Computer Science University of Utah Salt Lake City, Utah.

[8] D. M. Beazley. *Computer Architecture: A Quantitative Approach*. Department of Computer Science, University of Utah, Salt Lake City, Utah 84112, 1996.

[9] S. J. Benson, M. Krishnan, L. McInness, J. Nieplocha, and J. Sarich. Using the GA and TAO for solving large-scale optimization problems on parallel computers. Technical Report Preprint ANL/MCS-P1084-0903, Mathematics and Computer Science Division, Argonne National Laboratory, 2003.

[10] R. H. Bisseling. *Parallel Scientific Computation. A Structured Approach using BSP and MPI.* Oxford University Press, 2004. 324 pages. ISBN 0-19-852939-2.

[11] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. PBLAS Home Page. http://www.netlib.org/scalapack/pblas_qref.html.

[12] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK example programs. http://www.netlib.org/scalapack/scalapackqref.ps.

[13] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK slides. http://acts.nersc.gov/events/Workshop2005/slides/Marques.pdf.

[14] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* ISBN: 0-89871-397-8, SIAM, 3600 University City Science Center, Philadelphia, PA 19104-2688, 1997.

[15] P. Brown, A. Collier, K. Grant, A. Hindmarsh, S. Lee, D. Reynolds, R. Serban, D. Shumaker, and C. Woodward. Sundials. University of Livermore, California, USA. http://www.llnl.gov/CASC/sundials/.

[16] National Energy Resource Center. The ACTS collection. http://acts.nersc.gov.

[17] J. Choi, J. Demmel, I. Dhillon, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In J. Dongarra, K. Madsen, and J. Wasniewski, editors, *Applied Parallel Computing - Computations in Physics, Chemistry and Engineering Science*, pages 95–106. Springer, Berlin, 1996.

[18] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. In J. Dongarra, K. Madsen, and J. Wasniewski, editors, *Applied Parallel Computing - Computations in Physics, Chemistry and Engineering Science*, pages 107–114. Springer, Berlin, 1996.

[19] J. Choi, J. Dongarra, S. Ostrouchov, A. P. Petitet, D.W. Walker, and R.C. Whaley. The design and implementation on the ScaLAPACK SU, QR and Cholesky Factorization Routines. Technical report, University of Tenesse, 1994.

[20] A. Choudhary, B. Gallagher, W. Gropp, J. Li, W. Liao, R. Ross, R. Thakur, R. Latham, A. Siegel, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC2003: High Performance Networking and Computing*, Phoenix, AZ, November 2003. IEEE Computer Society Press.

[21] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1:1–12, 1993.

[22] M. J. Dayde and I. S. Duff. The RISC BLAS: a blocked implementation of level 3 BLAS for RISC processors. *ACM Transactions on Mathematical Software*, 3(25):316–340, 1999.

[23] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. 16(1):1–17, 1990.

[24] J. Dongarra, J. Du Croz, and S. Hammarling. Algorithm 656: An extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Program. 14(1):18–32, 1988.

[25] J. Dongarra, R. Geijn, and C. Whaley. Two dimensional Basic Linear Algebra Communication Subprograms. pages 31–40, 1993.

[26] J. Dongarra, S. Huss-Lederman, S. Otto, M. Snir, and D. Walkel. *MPI: The complete reference*. The MIT Press, 1996.

[27] J. Dongarra and A. Petitet. ScaLAPACK Tutorial. In J. Dongarra, K. Madsen, and J. Wasniewski, editors, *Applied Parallel Computing - Computations in Physics, Chemistry and Engineering Science*, pages 166–176. Springer, Berlin, 1996.

[28] J. Dongarra and R. C. Whaley. *A user's guide to the BLACS v1.1.* Computer Science Dept. Technical Report CS-95-281 University of Tennessee, Knoxville, TN, 1995.

[29] J. J. Dongarra, J. Du Croz, S. Hammarling, J. Wasniewski, and A. Zemla. A Proposal for a Fortran 90 Interface for LAPACK. In J. Dongarra, K. Madsen, and J. Wasniewski, editors, *Applied Parallel Computing - Computations in Physics, Chemistry and Engineering Science*, pages 158–165. Springer, Berlin, 1996.

[30] F. L. Drake and G. Van Rossum. Extending and embedding the Python interpreter. http://docs.python.org/ext/ext.html.

[31] F. L. Drake and G. Van Rossum. Installing Python Modules. http://docs.python.org/inst/inst.html.

[32] F. L. Drake and G. Van Rossum. Python and XML. http://docs.python.org/inst/inst.html.

[33] F. L. Drake and G. Van Rossum. Python Library Reference. http://docs.python.org/lib/lib.html.

[34] L. A. Drummond, V. Galiano, V. Migallón, and J. Penadés. *PyACTS Users' Guide.* Technical Report, Miguel Hernández University, 2005.

[35] L. A. Drummond, V. Galiano, V. Migallón, and J. Penadés. High Level User Interfaces for High Performance Libraries in Linear Algebra: PyBLACS and PyPBLAS. In *12th International Linear Algebra Society Conference*, University of Regina, Regina, Saskatchewan, Canada, June 2005.

[36] L. A. Drummond, V. Galiano, V. Migallón, and J. Penadés. Improving ease of use in BLACS and PBLAS with Python. In G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing (Proceedings of the International Conference ParCo 2005)*, volume 33. NIC series, September 2006. ISBN 3-00-017352-8.

[37] L. A. Drummond, V. Galiano, V. Migallón, and J. Penadés. PyACTS: A High-Level Framework for Fast Development of High Performance Applications. In *Proceedings from Seventh International Meeting on High Performance Computing for Computational Science - VECPAR'06*, pages 373–378, Rio de Janeiro, Brazil, July 2006. Also in *Lectures Notes in Computer Science*, 4395: 417–425, 2007.

[38] L. A. Drummond, V. Galiano, J. Penadés, and V. Migallón. An introduction to PyACTS. In *SIAM Conference on Computational Science and Engineering*. SIAM Activity Group on Computational Science and Engineering, Orlando, Florida, February 2005.

[39] L. A. Drummond, V. Galiano, J. Penadés, and V. Migallón. High Level User Interfaces for Numerical Linear Algebra Libraries: PyScaLAPACK. In *Proceedings of the Fifth International Conference on Engineering Computational Technology*, pages 1–10. Civil-Comp Ltd., September 2006.

[40] L. A. Drummond, V. Galiano, J. Penadés, and V. Migallón. PyACTS: A High-Level Framework for Fast Development of High Performance Applications. In *Proceeding of the SIAM Conference on Parallel Processing for Scientific Computing*. SIAM Activity Group on Supercomputing, San Francisco, California, February 2006.

[41] L. A. Drummond, V. Galiano, J. Penadés, and V. Migallón. High-level User Interfaces for the DOE ACTS Collection. In *PARA06: Proceeding of the Workshop on state-of-the-art in scientific and parallel computing*. Umea University, Suecia, June 2006. To appear in *Lecture Notes in Computer Science*.

[42] L. A. Drummond and O. Marques. The ACTS Collection Robust and High-Performance Tools for Scientific Computing. Guidelines for Tool Inclusion and Retirement. http://acts.nersc.gov/documents/LBNL-PUB-3175.pdf.

[43] Enthought. Scipy: Scientific Tools for Python. http://www.scipy.org/.

[44] R. D. Falgout, J. E. Jones, and U. M. Yang. *The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners, chapter in Numerical Solution of Partial Differential Equations on Parallel Computers*. 2004. Also available as LLNL Technical Report UCRL-JRNL-205459.

[45] M. J. Flynn. Very high–speed computing systems. *IEEE Transactions on Computers*, (12):1901–1909, 1966.

[46] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, (21):948–960, 1972.

[47] M. Folk and E. Pourmal. HDF Software Process. Lessons Learned or Success Factor. Technical report, NCSA HDF Group, 2004.

[48] MPI FORUM. MPI: A message passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8:3–4, February 1994. Special issue on MPI. Also available electronically, the URL is ftp://www.netlib.org/mpi/mpi-report.ps.

[49] I. Foster and C. Kesselman. Provides an overview of the Globus project and toolkit. *Intl J. Supercomputer Applications*, (11):115–128, 1997.

[50] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[51] V. Galiano, V. Migallón, and J. Penadés. PyPnetCDF Project. http://www.pyacts.org/pypnetcdf/.

[52] V. Galiano, J. Penadés, H. Migallón, and V. Migallón. Diseño, creación y aplicaciones de la librería de alto nivel PyScaLAPACK. In *Actas de las XVII Jornadas de paralelismo*, pages 373–378. Universidad de Castilla la Mancha, September 2006.

[53] V. Galiano, J. Penadés, and V. Migallón. Computación en arquitecturas paralelas mediante Python. In *Computación de Altas Prestaciones, Actas de las XV Jornadas de paralelismo*, pages 24–29. Universidad de Almería, September 2004.

[54] V. Galiano, J. Penadés, and V. Migallón. Parallel access for NetCDF files in High Level Languages: PyPNetCDF. In *Proceedings of the Fifth Annual Meeting of the European Meteorological Society*. European Meteorological Society, September 2005.

[55] V. Galiano, J. Penadés, V. Migallón, and V. Migallón. Diseño, creación y evaluación de las interfaces de alto nivel para BLACS y PBLAS de PyACTS. In *Actas de las XVI Jornadas de paralelismo*, pages 511–518. Thomson, September 2005.

[56] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Tennessee, September 1994.

[57] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Workley. A users' guide to PICL: a portable instrumented communication library. Technical report, Oak Ridge National Laboratory, 1990.

[58] P. B. Gibbons. A more practical PRAM model. *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, 1:158 – 168, 1989.

[59] J. Gilbert, S. Li, and J. Demmel. SuperLU. University of California and Lawrence Berkeley National Laboratory. http://crd.lbl.gov/x̃iaoye/SuperLU/.

[60] G. H. Golub and J. M. Ortega. *An Introduction with Parallel Computing.* Academic Press, San Diego, 1993.

[61] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming.* The MIT Press, 1994. Cambridge.

[62] W. R. Hamilton. *Elements of quaternion.* Chelse Pub. Co., 1969. ISBN 0828402191.

[63] R. J. Harrison, R. J. Littlefield, and J. Nieplocha. Global arrays: A portable shared memory model for distributed memory computers. In *Proc. Supercomputing'94*, pages 340–349, 1994.

[64] HDF5 Home Page. The National Center for Supercomputing Applications. http://hdf.ncsa.uiuc.edu/HDF5/.

[65] J. L. Hennessy and D. A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufmann Publishers, 1990. San Francisco.

[66] W. Henshaw, D. J. Quinlan, and D. L. Brown. Automatically Tuned Linear Algebra Software (ATLAS). http://math-atlas.sourceforge.net/.

[67] W. Henshaw, D. J. Quinlan, and D. L. Brown. Overture. http://www.llnl.gov/casc/Overture/.

[68] V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software*, 31(3):351.

[69] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:pp. 409–436, 1952.

[70] K. Hinsen. ScientificPython. http://sourcesup.cru.fr/projects/scientific-py/.

[71] V. E. Howle, S. M. Shontz, and P. D. Hough. Some Parallel Extensions to Optimization Methods in OPT++. Technical Report SAND2000-8877, Sandia National Laboratories Livermore California, October 2000.

[72] I. N. James. *Introduction to Circulating Atmospheres*. Cambridge University Press, Cambridge, 1994.

[73] IBM Corporation. IBM PVMe for AIX User's Guide and Subroutine Reference. Technical Report GC23-3884-00, IBM Corp., Poughkeepsie, NY, October 1995.

[74] A. K. Kaw. *Newton Raphson Method, Numerical Solution of Nonlinear Equations, Mathcad, Maple, Mathematica Simulations*. Holistic Numerical Methods Institute,College of Engineering, University of South Florida, Tampa, FL 33620-5350., 2004. http://numericalmethods.eng.usf.edu/mcd/gen/03nle.

[75] D. J. Kuck. *The structure of computers and computations*. John Wiley, 1978.

[76] M. Noguer. Using statistical techniques to deduce local climate distributions. an application for model validation. 1:277–287, 1994.

[77] D. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for industrial and applied mathematics*, 2(11):431–441, 1963.

[78] MathWorks. Matlab and simulink for technical computing. http://www.matlab.com.

[79] P. Miller. PyMPI - An introduction to parallel Python using MPI. http://www.llnl.gov/computing/develop/python/pyMPI.pdf, 2002.

[80] National Science Foundation under Grant No. ASC-9313958 and DOE Grant No. DE-FG03-94ER25219. The ScaLAPACK Project. http://www.netlib.org/scalapack/.

[81] U.S. Departament of Energy. Office of science. http://www.er.doe.gov/.

[82] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum Press, New York, 1998.

[83] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley Professional, 1994.

[84] P. M. Papadopoulos, J. A. Kohl, and B. D. Semeraro. CUMULVS: Extending a Generic Steering and Visualization Middleware for Application Fault-Tolerance. In *Proceedings of the 31st Hawaii International Conference on System Sciences*, 1998.

[85] P. Peterson. F2PY Users Guide and Reference Manual. http://cens.ioc.ee/projects/f2py2e/.

[86] R. W. Preisendorfer and C. D. Mobley. *Principal component analysis in meteorology and oceanography.* Elsevier, Amsterdam, 1995.

[87] W. H. Press, S. A. Teulosky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipies, 2nd ed.* Cambridge University Press, Cambridge, 1992.

[88] R. Rew and G. Davis. Computer Graphics and Applications, IEEE. 1990. 76-82.

[89] R. Rew and G. Davis. The Unidata netCDF: Software for Scientific Data Access. In *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology*, 2001. Anaheim, CA.

[90] R. Rew, G. Davis, S. Emmerson, and H. Davies. Fuentes de la librería netCDF. Unidata Program Center. ftp://ftp.unidata.ucar.edu/pub/netcdf.

[91] R. Rew, G. Davis, S. Emmerson, and H. Davies. Unidata: netCDF. National Science Foundation. http://my.unidata.ucar.edu/content/software/netcdf/index.html.

[92] R. Rew, G. Davis, S. Emmerson, and H. Davies. Utilización de netCDF. Unidata Program Center. http://www.unidata.ucar.edu/packages/netcdf/usage.html.

[93] R. Rew, G. Davis, S. Emmerson, and H. Davies. Web de Unidata: netCDF. http://ftp.unidata.ucar.edu/software/netcdf/.

[94] R. Rew, G. Davis, S. Emmerson, and H. Davies. NetCDF User's Guige for C. http://www.unidata.ucar.edu/packages/netcdf/guidec, 1997.

[95] J.M. Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *IPPS '93 Parallel I/O Workshop*, 1993.

[96] G. Van Rossum. Python Software Foundation. http://www.python.org.

[97] S. J. Benson and L. C. McInnes and J. J. Moré and J. Sarich. Scalable Algorithms in Optimization: Computational Experiments. Technical Report ANL/MCS-P1175-0604, Mathematics and Computer Science Division, Argonne National Laboratory, 2004.

[98] J. Sáenz, J. Zubillaga, and J. Fernández. Geophysical data analysis using Python. 24(4):457–465, 2002.

[99] Computational Sciences and Mathematics Research Department at Sandia National Laboratories. OPT++. http://acts.nersc.gov/opt++/index.html.

[100] J. Sáenz, J. Fernández, and F. Zubillaga. A fully matricial approach to SVD, EOF and CCA. http://www.pyclimate.org/matrix.pdf.

[101] J. Sáenz, J. Fernández, and F. Zubillaga. Pyclimate. http://www.pyclimate.org.

[102] J. N. Shadid and R. S. Tuminaro. Iterative Methods for Nonsymmetric Systems on MIMD Machines. *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Applications*, 2(11):431–441, 1991.

[103] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145, 1998.

[104] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[105] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION Runtime Library for Parallel I/O. In *Scalable Parallel Libraries Conference*, 1994.

[106] R. Thakur and A. Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[107] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceeding of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, 1999.

[108] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid. *Official Aztec User's Guide: Version 2.1*. Sandia National Laboratory, SAND99-8801J, 1999.

[109] Utrecht University. BSP Worldwide Home Page. http://www.bsp-worldwide.org/.

[110] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, (33):103–111, 1990.

[111] R. C. Whaley. Basic Linear Algebra Communication Subprograms: Analysis and Implementation Across Multiple Parallel Architectures. Technical report, Environments and Tools for Parallel Scientific Computing, 1994.

[112] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[113] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[114] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[115] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[116] D. S. Wilks. Multisite downscaling of daily precipitation with a stochastic weather generator. *Climate Research*, 11:25–136, 1999.

[117] F. W. Zwiers and H. Von Storch. Regime dependent autoregressive time series modelling of the southern oscillation. *Journal of Climate*, 3:1347–1363, 1990.