UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

"ANÁLISIS Y ACELERACIÓN DE ALGORITMOS METAHEURÍSTICOS DE OPTIMIZACIÓN DISCRETA"

TRABAJO FIN DE GRADO

Julio - 2021

AUTOR: Francisco Javier Del Campo Calvo
DIRECTORES: Héctor Francisco Migallón Gomis
Miguel Onofre Martínez Rach

AGRADECIMIENTOS

Quiero dar las gracias:

A la persona más importante de mi vida, mi madre, a la cual le debo más de la mitad de lo que he logrado en esta vida.

A mi padre, por haberme apoyado tanto desde la distancia, en Burgos.

A mi tío Alejandro, principal culpable de que me quisiera dedicar a la Informática.

A mi tutor Héctor Migallón, por haberme enseñado tantas cosas, tanto en su asignatura como en el desarrollo de este trabajo y por haber estado siempre ahí cuando le necesitaba, este trabajo no habría sido posible sin él.

A mis profesores del IES Jaime II Gabriel (Q.E.P.D.), Ismael y Jose Carlos, por haberme enseñado que existía una forma diferente de dar clase y que había profesores que valían realmente la pena, si no fuera por ellos sin duda habría dejado los estudios.

A Luis Eloy, director del IES Jorge Juan, por haberme acogido en su instituto a falta de cinco meses para selectividad tras haber sido expulsado del mío por ejercer mi libertad de expresión y tratar de cambiar un poco las cosas.

A mis profesores de la UMH Guillermo De Scals, Roberto Dale, Antonio Peñalver y otros tantos que me dejo en el tintero, por haber sido tan buenos docentes y mejores personas.

A mis compañeros Ana, Dmitry, Alejandro, Javier, Julio y Daniel por haber hecho de mi paso por la universidad una experiencia mucho más agradable.

A la educación pública, por haberme permitido cursar y finalizar la carrera sin desembolsar un solo euro e independientemente de mi posición socioeconómica.

A mi familia y amigos, en general.

RESUMEN

Este Trabajo de Fin de Grado está estructurado en dos partes, siendo la primera un análisis del rendimiento de tres algoritmos metaheurísticos de optimización discreta: DTSA (Discrete Tree-Seed Algorithm), DJAYA (Discrete Jaya) y DTLBO (Discrete Teaching-Learning-Based Optimization). Para realizar dicho análisis se han implementado los tres algoritmos en lenguaje C y se han probado con 6 problemas del tipo TSP.

El otro objetivo del trabajo es el desarrollo y posterior análisis de eficiencia y escalabilidad de diseños paralelos que permitan acelerar la ejecución de los algoritmos metaheurísticos estudiados en equipos de computación de altas prestaciones con arquitectura de memoria compartida. Han sido implementados tres algoritmos paralelos: un enfoque basado en automatizadores, un enfoque basado en subpoblaciones y finalmente un enfoque híbrido, siendo aplicables todos ellos a cada uno de los tres algoritmos metaheurísticos. Estos algoritmos han sido implementados usando OpenMP.

El análisis de rendimiento de los algoritmos metaheurísticos ha mostrado que los algoritmos DTSA y DJAYA obtienen resultados que DTLBO para los problemas estudiados, teniendo además DJAYA una velocidad de convergencia ligeramente mejor.

Por otra parte, del análisis de eficiencia paralela se ha deducido que el enfoque basado en automatizadores obtiene resultados de eficiencia y escalabilidad mucho más positivos para el algoritmo DTSA que para DJAYA y DTLBO, mientras que el enfoque basado en subpoblaciones ha obtenido unos resultados muy buenos en todos los casos.

Finalmente, los algoritmos híbridos no han mostrado una disminución apreciable en la eficiencia y en la escalabilidad respecto del enfoque con subpoblaciones, lo que sumado a su gran versatilidad supone constatar que son un muy buen diseño para aplicar a este tipo de algoritmos.

ABSTRACT

This Final Degree Project is structured in two parts, the first being a performance analysis of three discrete optimization metaheuristic algorithms: DTSA (Discrete Tree-Seed Algorithm), DJAYA (Discrete Jaya) y DTLBO (Discrete Teaching-Learning-Based Optimization). To perform this analysis, the three algorithms have been implemented in C language and 6 TSP problems have been used to try them.

The other aim of this project is to develop parallel designs to speed up the runtime of the studied metaheuristic algorithms in high-performance computing systems with shared memory architecture. Three parallel algorithms have been implemented: an automatic-parallelization-based approach, a subpopulation-based approach, and a hybrid approach, all of them being applicable to each of the three metaheuristic algorithms. These algorithms have been implemented using OpenMP. An efficiency and scalability analysis of the three parallel algorithms has been performed.

The performance analysis carried out on the metaheuristic algorithms has proved that DTSA and DJAYA algorithms obtain better results than DTLBO for the studied problems, having DJAYA a slightly better convergence speed than DTSA.

On the other hand, the parallel efficiency analysis has shown up that automatic-parallelization-based approach gets much better efficiency and scalability results when applied to DTSA algorithm than when applied to DJAYA and DTLBO, while subpopulation-based approach has obtained very good results in all cases.

Finally, the hybrid algorithms have not shown an appreciable decrease in efficiency and scalability in comparison with subpopulation-based algorithms, which added to their great versatility, makes them a very good design for applying to metaheuristic algorithms similar to those studied.

Índice

CAPÍTULO 1:

INTRODUCCIÓN

En este capítulo se expondrán los dos principales objetivos de este trabajo y se describirá la estructura de éste, explicando el contenido de cada capítulo.

1.1.- OBJETIVOS

Son numerosos los ámbitos del mundo real en los cuales el campo de las matemáticas conocido como optimización es una útil herramienta, ya que ha supuesto grandes avances no sólo en la ciencia y en la ingeniería sino también en campos como la industria o el mundo de los negocios. Desafortunadamente, no siempre es posible encontrar un método que permita resolver un problema de optimización de forma exacta. Además, aunque ese método exista, es posible que conlleve unos requisitos de coste computacional que hagan inviable su uso.

Existen una serie de enfoques, conocidos como algoritmos heurísticos y metaheurísticos, que permiten obtener una solución aceptable (si bien no necesariamente la óptima, sí una lo suficientemente buena) a esta clase de problemas en un coste computacional mucho menor que el que requerirían los métodos exactos tradicionales.

También es conveniente señalar que hoy en día los equipos de computación de altas prestaciones permiten acelerar la velocidad de cómputo hasta niveles insospechados décadas atrás haciendo uso de la computación paralela o distribuida. Por tanto, el desarrollo de programas para esta clase de equipos es un campo clave en la actualidad.

El primero de los dos objetivos principales de este trabajo es estudiar y analizar tres algoritmos metaheurísticos que, si bien fueron originalmente diseñados para la resolución de problemas de optimización continua, han sido adaptados para resolver problemas de optimización discreta o combinatoria. Estos algoritmos son: DTSA (*Discrete Tree-Seed Algorithm*), DJAYA (*Discrete Jaya*) y DTLBO (*Discrete Teaching-Learning-Based Optimization*). Dichos algoritmos han sido implementados en lenguaje C y puestos a prueba con 6 problemas del tipo TSP (*Travelling Salesman Problem* o Problema del Viajante de Comercio). A partir de los datos obtenidos de la resolución de estos problemas, se ha realizado una serie de tablas y se han extraído conclusiones sobre éstas.

Por otra parte, el otro objetivo de este trabajo es diseñar algoritmos paralelos que permitan acelerar la ejecución de estos algoritmos metaheurísticos en equipos de computación de altas prestaciones, concretamente equipos con arquitectura de memoria compartida. En

total han sido implementados tres algoritmos paralelos: un enfoque basado en automatizadores, un enfoque basado en subpoblaciones y finalmente un enfoque híbrido, siendo aplicables todos ellos a cada uno de los tres algoritmos metaheurísticos. Los algoritmos paralelos han sido implementados usando OpenMP. Se han tomado los datos de los tiempos de ejecución, que han sido compilados en tablas y posteriormente analizados para estudiar la eficiencia y escalabilidad de estos algoritmos paralelos.

1.2.- ESTRUCTURA DEL TRABAJO

A continuación, se explica cómo está estructurado este trabajo, explicando a grandes rasgos los objetivos de cada capítulo:

- En el capítulo 2 se presentan y explican los antecedentes necesarios para comprender el marco de trabajo de este proyecto: los problemas de optimización, los algoritmos heurísticos y metaheurísticos y las arquitecturas paralelas.
- En el capítulo 3 se describen los tres algoritmos metaheurísticos de optimización que han sido objeto de estudio en este trabajo: DTSA, DJAYA y DTLBO. También se incluye el pseudocódigo de éstos para ilustrarlos.
- En el capítulo 4 se definen y detallan los tres algoritmos paralelos que han sido diseñados, exponiendo las motivaciones que han llevado al desarrollo de cada uno de éstos y describiendo su funcionamiento, descripción que será ilustrada con la inclusión del pseudocódigo de estos algoritmos paralelos.
- En el capítulo 5 se muestran, después de hacer una descripción del tipo de problema que se ha usado (TSP), los resultados numéricos del proyecto, habiendo en primer lugar un análisis de rendimiento de los algoritmos metaheurísticos y posteriormente un análisis de eficiencia paralela de los algoritmos paralelos.
- En el capítulo 6 se exponen las principales conclusiones que han sido obtenidas al finalizar este trabajo.

CAPÍTULO 2:

PRELIMINARES

En este capítulo se expondrán todos los antecedentes necesarios para poder comprender la temática de este proyecto. En primer lugar, se hará una introducción a los problemas de optimización, para posteriormente centrarse en los algoritmos heurísticos y metaheurísticos que permiten la resolución de los problemas de optimización combinatoria. Finalmente, se tratará el concepto de procesamiento paralelo, describiendo los diferentes tipos de arquitecturas de computación existentes que permiten hacer uso de éste.

2.1.- PROBLEMAS DE OPTIMIZACIÓN

En esta sección se introducirá el marco principal de trabajo de este proyecto, es decir, los problemas de optimización. Se tratarán tanto los problemas de optimización continuos como los problemas de optimización combinatorios. Además, se describirán algunos de los métodos más comunes para su resolución.

2.1.1.- Definición

Se puede considerar que la resolución de los problemas de optimización consiste en tratar de encontrar la mejor solución posible para un determinado problema. Lógicamente, los problemas de optimización pueden ser tanto de maximización como de minimización.

Si se considera que estos problemas están definidos por un conjunto de variables de decisión (o variables de diseño), una definición más científica sería el proceso de hallar los valores de dichas variables de decisión que definen el sistema con el fin de bien maximizar o bien minimizar una determinada función objetivo, que depende, como se ha dicho, de dichas variables de decisión. Estos problemas pueden incluir, además, una serie de restricciones que reduzcan el espacio de las posibles soluciones, es decir no todas las combinaciones posibles de las variables de decisión del problema son aceptables, provocando que existan soluciones factibles y soluciones no factibles.

Según el tipo de las variables de decisión, los problemas de optimización pueden ser continuos o combinatorios. En los problemas continuos, las variables, por lo general, toman valores reales, mientras que, en los problemas de optimización combinatoria, que son los que serán tratados en este trabajo, únicamente podrán tomar valores discretos, y además comúnmente serán valores enteros.

2.1.2.- Optimización continua

Cuando se habla de optimización de continua o de funciones o problemas continuos, se hace referencia a problemas de optimización cuya función objetivo está formada por variables de decisión reales. Cada una de las variables de diseño estará definida en un dominio concreto, que lógicamente puede ser diferente para cada una de ellas. Este es un requisito indispensable en los métodos que serán tratados en este trabajo, ya que acota el dominio de búsqueda, si esto no fuera así el problema no sería abordable. En determinados problemas, las variables, además de tener acotado el dominio de búsqueda algunas de ellas pueden tener restricciones, como por ejemplo ser discretas. Otro aspecto importante que afecta a estos algoritmos es las características matemáticas de la función a optimizar, pudiendo encontrarse problemas lineales, cuadráticos, convexos, cónicos… De hecho, estas características pueden determinar si una determinada función es optimizable mediante métodos matemáticos deterministas o no.

Los métodos de resolución de estos problemas se pueden dividir en dos categorías: deterministas y heurísticos.

Los enfoques deterministas utilizan las propiedades analíticas del problema para generar una secuencia de puntos que converge hacia una solución global óptima. Estos métodos dependen en gran medida del álgebra lineal, ya que suelen estar basados en el cálculo del gradiente. Estos métodos utilizan un enfoque basado en la programación matemática, por lo que sus resultados son unívocos y replicables. Algunos de los métodos más destacados son el método de Newton, el descenso de gradiente y el Broyden-Fletcher-Goldfarb-Shanno (BFGS).

Los métodos deterministas proporcionan herramientas generales que permiten obtener un óptimo global, pero en algunos casos, como en los problemas no convexos o de gran tamaño, estos métodos pueden no ser capaces de derivar una solución óptima global en un tiempo computacional aceptable. Los métodos heurísticos permiten acelerar la convergencia, evitar mínimos locales y evitar restricciones en las funciones a ser optimizadas. Dichos métodos podrían ser definidos como técnicas de búsqueda guiadas que son capaces de generar una solución aceptable, pese a que su adecuación al problema

objetivo no puede ser formalmente probada. Este trabajo se centrará en algoritmos heurísticos.

Los algoritmos heurísticos se suelen clasificar en dos grupos: Algoritmos Evolutivos (EA) y Algoritmos de Inteligencia de Enjambre (SI). Entre los primeros, destacan *Genetic Algorithm* (GA), *Evolutionary Strategy* (ES), *Evolutionary Programming* (EP), *Genetic Programming* (GP), *Differential Evolution* (DE), *Bacteria Foraging Optimization* (BFO) y *Artificial Immune Algorithm* (AIA). Entre los segundos, cabe mencionar *Particle Swarm Optimization* (PSO), *Shuffled Frog Leaping* (SFL), *Ant Colony Optimization* (ACO), *Artificial Bee Colony* (ABC) y *Fire Fly* (FF). Existen además otros algoritmos basados en fenómenos de la naturaleza, como *Harmony Search* (HS), *Lion Search* (LS), *Gravitational Search Algorithm* (GSA), *Biogeography-Based Optimization* (BBO) y *Grenade Explosion Method* (GEM).

El éxito de la mayoría de estos métodos reside en gran medida en los parámetros que emplean, los cuales guían el proceso de búsqueda y contribuyen a la exploración del espacio de búsqueda. Por tanto, el correcto ajuste de los parámetros específicos de un algoritmo supone un factor de éxito crucial para la obtención del óptimo global. Algunos ejemplos de parámetros específicos son: en PSO, tamaño de la población, número máximo de generaciones, tamaño de la élite, peso de la inercia y ratio de aceleración; en ABS, abejas observadoras, trabajadoras y exploradoras; en HS, memoria de la armonía, número de improvisaciones y ratio de ajuste de la altura.

Algunos algoritmos recientemente desarrollados, como Tree Seed Algorithm (TSA), Teacher-Learner Based Optimization (TLBO) y Jaya, permiten prescindir del ajuste de parámetros específicos, ya que únicamente presentan parámetros de carácter general, como el número de iteraciones y el tamaño de la población.

Por lo general, estos algoritmos, diseñados para la resolución de problemas continuos, no pueden ser usados directamente para la resolución de problemas combinatorios, pero pueden desarrollarse algoritmos combinatorios inspirados en ellos. En este trabajo se realizará una adaptación de estos últimos (TSA, TLBO y Jaya) para que puedan ser empleados en la resolución de problemas combinatorios. Hay que tener en cuenta que la

ausencia de parámetros específicos es una característica muy interesante a la hora de la resolución de diferentes tipos de problemas.

2.1.3.- Optimización combinatoria

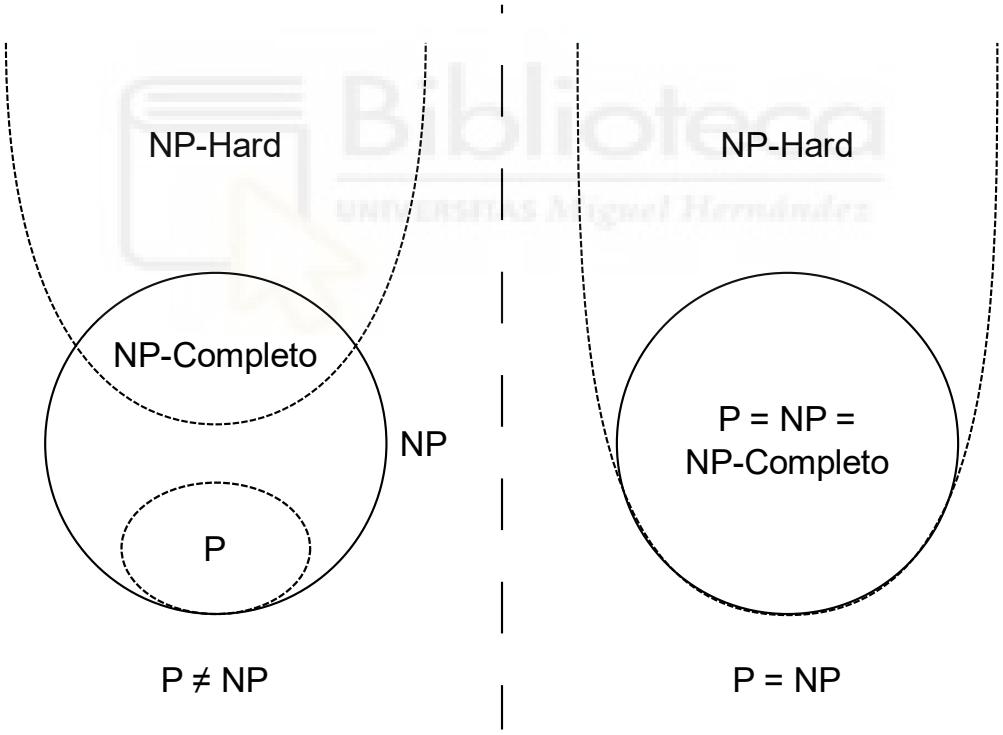
En los problemas de optimización combinatoria, el espacio de posibles soluciones está formado por todas las permutaciones posibles de un subconjunto de números naturales. A diferencia de los problemas de optimización continua, donde las posibles soluciones son infinitas, en la optimización combinatoria existe un número finito de soluciones.

Esta clase de problemas es empleada en diversos campos como informática, logística, telecomunicaciones, ingeniería, etc., y se pueden encontrar fácilmente numerosas aplicaciones de éstos, como son el trazado de redes de comunicaciones, la planificación de inversiones, el diseño de circuitos electrónicos, el enrutamiento de vehículos, el diseño de nuevas moléculas, el posicionamiento de satélites, etc.

Pese a que el espacio de soluciones de estos problemas sea finito, muchos de estos problemas son muy complejos de resolver, perteneciendo a la clase NP-hard. En teoría de complejidad computacional, se define como problemas de clase P a todos aquellos para los cuales existe un algoritmo que permita hallar su solución en un tiempo polinómico. Una definición más técnica es aquellos que pueden ser resueltos por una Máquina de Turing determinista en un tiempo polinómico. Por otra parte, la clase NP engloba a todos aquellos que pueden ser resueltos en tiempo polinómico por una Máquina de Turing no determinista, que se diferencia de la determinista en que para un mismo estado y símbolo puede haber más de una acción. En términos formales, la diferencia es que la transición es una función en vez de sólo una relación. Esta clase de problemas incluye además de a los de la clase P a muchos otros. Una de las grandes preguntas sin responder en la teoría de la computación es si ambas clases son o no equivalentes.

Por otra parte, NP-completo es un subconjunto de NP, este subconjunto está compuesto por todos aquellos problemas tales que todo problema perteneciente a NP puede ser reducido en tiempo polinómico a cualquiera de NP-completo. Finalmente, la clase NP-

hard engloba a los problemas que son al menos tan difíciles como los de la clase NP-completo. Esta última clase no es un subconjunto de NP, sino que únicamente coincide con NP en el subconjunto NP-completo. Un ejemplo de problema NP-hard que no es NP-completo es el problema de la parada, que consiste en desarrollar un programa que sea capaz de determinar si otro programa terminará su ejecución en un número finito de pasos. Se ha demostrado que dicho problema no puede ser resuelto por una Máquina de Turing (y es, por tanto, no computable). Dado que cualquier problema NP-completo se puede reducir a cualquier otro problema NP-completo en tiempo polinómico, todos los problemas NP-completo se pueden reducir a cualquier problema NP-hard en tiempo polinómico. Por tanto, si existiese un algoritmo para resolver un problema NP-hard en tiempo polinómico, habría algoritmos para resolver todos los problemas NP en tiempo polinómico, lo que significaría que P y NP serían equivalentes [1].



Imegen 1. En esta imagen se muestran dos diagramas con las clases de complejidad computacional. El de la izquierda se corresponde con el caso de que la hipótesis P = NP sea falsa, mientras que el de la derecha asume que dicha hipótesis es cierta.

Cabe mencionar que todos los problemas de optimización combinatoria podrían ser resueltos, es decir, obtener el óptimo global, utilizando la técnica de "fuerza bruta" al ser finito el espacio de soluciones posibles. No obstante, cuando el número de variables de

decisión del problema crece, el coste computacional impide que se puedan cumplir los requisitos temporales requeridos.

Algunos de los ejemplos de problemas de optimización combinatoria más conocidos son el problema de la mochila (KP), el problema del viajante de comercio (TSP), el problema de enrutamiento de vehículos (VRP), entre otros.

2.2.- ALGORITMOS DE OPTIMIZACIÓN COMBINATORIA

En esta sección se describirán los algoritmos heurísticos y metaheurísticos de optimización desde el enfoque de la resolución de problemas combinatorios, también denominados problemas discretos.

2.2.1.- Algoritmos heurísticos

Debido a la elevada complejidad que presentan algunos de los problemas anteriormente mencionados, en ocasiones puede no ser viable la utilización de un método exacto (como podrían ser el de fuerza bruta, u otros computacionalmente menos costosos como el *Símplex* [2] o el *Branch & Bound* [3]) que permita obtener la solución óptima, ya sea porque no existe un método para ese problema en particular o porque el tiempo computacional requerido es demasiado elevado. En algunos casos no es necesario obtener el óptimo global, sino que obtener un valor cercano al óptimo global es aceptable si la solución obtenida no se aleja mucho del óptimo global y se cumplen los requisitos temporales requeridos. Para ello, se pueden emplear los llamados algoritmos heurísticos, término que deriva del griego *heuriskein*, que significa hallar, descubrir. Este término fue empleado por primera vez por George Polya en su libro *How to solve it* [4], usándolo para describir las reglas con las que los humanos gestionan el conocimiento común, que se resumen en [5]:

1- Buscar un problema parecido que ya haya sido resuelto.
2- Determinar la técnica empleada para su resolución, así como la solución obtenida.
3- En el caso en el que sea posible, utilizar la técnica y solución descrita en el punto anterior para resolver el problema planteado.

En *Optimización Heurística Y Redes Neuronales* [6] los autores definen un algoritmo heurístico como "*un procedimiento para resolver un problema de optimización bien definido mediante una aproximación intuitiva, en la que la estructura del problema se utiliza de forma inteligente para obtener una buena solución*."

H. Müller-Merbach [7] realizó la siguiente definición: "*En investigación operativa, el término heurístico normalmente se entiende en el sentido de un algoritmo iterativo que no converge hacia la solución (óptima o factible) del problema*".

Esta última definición reafirma la característica más destacada de los métodos heurísticos que los diferencia de los tradicionales, ya que, al no converger, puede obtener una solución distinta a la óptima, pues como se mencionó anteriormente, se suelen emplear cuando no se necesita obtener la solución óptima pero sí se precisa de un buen coste computacional.

Algunas de las razones para utilizar métodos heurísticos son [8]:

- El problema es de una naturaleza tal que no se conoce ningún método exacto para su resolución.
- No existe la necesidad de alcanzar la solución óptima global.
- Aunque existe un método exacto para resolver el problema, su uso es computacionalmente muy costoso.
- Cuando el tiempo dedicado a la resolución del problema presenta una importancia considerable y se precisa de una respuesta rápida.
- Cuando se requiere incorporar condiciones de difícil modelización, ya que el método heurístico es más flexible que un método exacto.
- El método heurístico proporciona más de una solución en muchos de los casos, lo cual permite ampliar el número de opciones del que decide, en especial cuando existe algún factor no cuantificable que merezca la pena ser considerado.

- El método heurístico se utiliza como parte de un procedimiento global que garantiza el óptimo de un problema. Dentro de esto existen dos posibilidades:
    - El método heurístico proporciona una buena solución inicial de partida
    - El método heurístico participa en un paso intermedio del procedimiento, como por ejemplo las reglas de selección de la variable a entrar en la base en el método Simplex.

Dentro de los algoritmos heurísticos se puede realizar la siguiente clasificación [5]:

Algoritmos constructivos: Consisten en construir una solución paso a paso partiendo de una solución vacía que se va completando. Entre ellos destacan enfoques como:

- Estrategia voraz: En cada paso o iteración se añade el elemento que genera una mejora más elevada en la solución parcial para ese paso concreto. Estos algoritmos eligen siempre la mejor opción actual sin tener en cuenta lo que ocurrirá en el futuro, por lo que se dice de éstos que tienen una visión "miope".
- Estrategia de descomposición: Se divide sistemáticamente el problema en subproblemas más pequeños hasta que se tenga un tamaño de subproblema cuya solución sea trivial y posteriormente se combinan las soluciones de éstos para obtener la solución del problema original. Un buen ejemplo son los algoritmos de divide y vencerás.
- Métodos de reducción. Se identifican características que contienen las soluciones buenas conocidas y se asume que la solución óptima también las tendrá. Esto permite reducir drásticamente el espacio de búsqueda.
- Métodos de manipulación del modelo: Se simplifica el modelo del problema original para obtener una solución al problema simplificado. A partir de esta solución aproximada, se extrapola la solución al problema original.

Algoritmos de búsqueda local: Se parte de una solución factible que se mejora progresivamente. Entre ellos están:

- Estrategia de búsqueda local: Parte de una solución factible que la mejora progresivamente. Para ello examina su vecindad y selecciona, o bien el primer movimiento que produce una mejora en la solución actual (first improvement), o aquel que produzca una mejora mayor en la función objetivo (best improvement).

- Estrategia aleatorizada: Para una solución factible dada y una vecindad asociada a esa solución, se seleccionan aleatoriamente soluciones vecinas de esa vecindad.

Estos algoritmos presentan el inconveniente de poder quedar atrapados con facilidad en un óptimo local, lo que puede resultar en una imposibilidad de éstos para alcanzar una solución lo suficientemente buena. Debido a ello, se crearon las llamadas metaheurísticas, que permiten guiar a los métodos heurísticos para impedir que caigan en óptimos locales.

2.2.2.- Algoritmos metaheurísticos

En las últimas décadas han aparecido una serie de métodos bajo el nombre de metaheurísticas con el propósito de obtener mejores resultados que los alcanzados por los métodos heurísticos tradicionales. El término fue acuñado por Fred Glover en 1986 [9] y, desde entonces, es un campo en constante evolución y desarrollo. El sufijo "meta" significa "más allá", haciendo referencia a que se encuentran a un nivel superior respecto de las heurísticas, ya que guían a estas para conseguir la resolución del problema. Se han desarrollado y extendido rápidamente para dar respuesta a una gran variedad de problemas dentro de las áreas de negocio, comercio, ingeniería, industria y muchas otras más.

En 1995 Osman y Kelly definieron los procedimientos metaheurísticos como *una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que los heurísticos clásicos no son efectivos. Los Metaheurísticos proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los mecanismos estadísticos.*

En su artículo *Meta-Heuristics: Theory and Applications* [10], Osman y Kelly definieron la metaheurística como *un procedimiento iterativo con una estructura y unas reglas generales de funcionamiento que lo caracterizan, que guía un método heurístico subordinado combinando inteligentemente diversos conceptos para explorar los espacios*

de búsqueda mediante estrategias aprendidas para conseguir soluciones quasi- óptimas de manera eficiente.

Según Herrera [11], los algoritmos metaheurísticos son *procedimientos iterativos que guían una heurística subordinada combinando de forma inteligente distintos conceptos para explorar y explotar adecuadamente el espacio de búsqueda.*

Para poder entender mejor esta última definición, es preciso definir los términos *exploración* y *explotación:*

- Exploración: Consiste en buscar posibles soluciones en regiones distantes del espacio de búsqueda, evitando reducirse únicamente a la región del óptimo local que se ha alcanzado. También es llamada *diversificación*.

- Explotación: Consiste en intensificar la búsqueda en la región actual con el fin de alcanzar rápidamente la solución óptima de esa región. También es llamada *intensificación*.

Para que un algoritmo funcione exitosamente y permita obtener soluciones cercanas a la óptima en un tiempo reducido, es necesario lograr un equilibrio entre ambas.

Como se ha mencionado en la sección 2.2.1 los algoritmos de optimización metaheurísticos pueden ser clasificados en algoritmos evolutivos o genéticos y en algoritmos basados en enjambres o poblaciones de individuos [12].

- Algoritmos evolutivos: Grupo de técnicas estocásticas que utilizan los conceptos de evolución biológica. Actúan sobre una población de soluciones potenciales aplicando los principios de diversidad de individuos y supervivencia del más fuerte para producir mejores aproximaciones a una solución. En cada generación es creado un nuevo grupo de aproximaciones por el proceso de selección de individuos de acuerdo a su nivel de "desempeño" en el dominio del problema, y se cruzan entre sí utilizando operadores que imitan los conceptos genéticos. Este proceso lleva a la evolución de poblaciones de individuos que están mejor adaptados a su ambiente.

- Algoritmos basados en enjambres: Grupo de técnicas que están basadas en el estudio del comportamiento colectivo en sistemas autoorganizados y descentralizados (distribuidos). Estos sistemas están conformados típicamente por una población de agentes computacionales simples capaces de percibir y modificar su ambiente de manera local. Tal capacidad hace posible la comunicación entre los individuos, que detectan los cambios en el ambiente generado por el comportamiento de sus semejantes. Aunque normalmente no hay una estructura centralizada de control que dictamina cómo los agentes deben comportarse, las interacciones locales entre los agentes usualmente llevan a la emergencia de un comportamiento global. Otra característica adicional es la inexistencia de un modelo explícito del ambiente.

La principal diferencia entre ambas clases de algoritmos es que los algoritmos evolutivos presentan una naturaleza centralizada mientras que los basados en enjambres presentan un comportamiento distribuido. Otra diferencia reside en la interacción con la función objetivo, que en los algoritmos evolutivos corresponde al nivel de "desempeño" o *fitness*, mientras que en los basados en enjambre corresponde al ambiente a ser explorado. Finalmente, cabe mencionar que los agentes en los algoritmos evolutivos no pueden modificar su ambiente, mientras que en los algoritmos basados en enjambre esa característica es fundamental.

Algunos de los algoritmos metaheurísticos continuos más relevantes, entre otros, son: Greedy Randomized Adaptive Search Procedure ,Tabu Search, Simulated Annealing, Ant Colony Optmization, y las distintas variantes de los algoritmos genéticos:

- GRASP: Fue desarrollado por T. Feo y M. Resende al final de la década de los 80 [13]. Su nombre viene de Greedy Randomized Adaptive Search Procedure (procedimiento de búsqueda voraz aleatorizado adaptativo). Este método está formado por dos fases, una de construcción y otra de búsqueda local. Se considera, por tanto, un método multi-arranque. En la primera fase (fase de construcción) se establece una solución factible de alta calidad, que será mejorada en la segunda (fase de mejora) mediante algún procedimiento de intercambios. GRASP se basa en realizar múltiples iteraciones y quedarse con la mejor, por lo que no es

especialmente beneficioso para el método el detenerse demasiado en mejorar una solución dada.

- Tabu Search: Fue introducido en 1989 por Fred Glover [14] y tiene como fin guiar un algoritmo heurístico de búsqueda local para explorar el espacio de soluciones más allá del óptimo local. Este método surge en un intento de dotar de "inteligencia" a los algoritmos de búsqueda local. Tabu Search toma de la Inteligencia Artificial el concepto de memoria con el objetivo de dirigir la búsqueda teniendo en cuenta la historia de ésta. Este método permite moverse a una solución, aunque no sea tan buena como la actual, de modo que se pueda escapar de óptimos locales y continuar estratégicamente la búsqueda de soluciones aún mejores [15]. Para evitar volver a un óptimo local ya visitado, el método clasifica los últimos movimientos como "movimientos tabú", los cuales no será posible repetir durante un cierto horizonte temporal.

- Simulated Annealing: Fue propuesto por Kirpatrick, Gelatt y Vecchi en 1983 [16]. Este método se basa en el proceso físico de recocido de los metales, que es un tratamiento en el que primero se reblandece el sólido calentándolo a una temperatura elevada y luego se va enfriando mientras las partículas se van colocando por sí mismas en el estado fundamental. Este método reduce la probabilidad de quedarse atrapado en un óptimo local gracias a la aleatorización en la selección de la siguiente solución, que permite aceptar soluciones peores. Se ha demostrado que es capaz de hallar la solución óptima tras un número infinito de veces en el peor de los casos. Estableciendo un paralelismo entre el proceso físico del recocido y el algoritmo, una configuración concreta del sólido se correspondería con una solución posible, mientras que su configuración fundamental se correspondería con la solución óptima. La energía de una determinada configuración equivaldría al coste de la solución. La temperatura tendría un paralelismo con la probabilidad de aceptar soluciones que no supongan una mejora. Ésta empieza con un valor alto, que permite diversificar el espacio de búsqueda, y se va reduciendo con el paso de las iteraciones.

- Ant Colony Optimization: Fue desarrollado por Colorni et al. [17] y se inspira en el comportamiento que adoptan las hormigas al buscar alimento y regresar al

hormiguero. Cuando una hormiga encuentra alimento, de camino a casa deja un rastro de feromonas cuya cantidad depende de la calidad del alimento y la distancia a éste. Si las hormigas no detectan feromonas, se mueven aleatoriamente, en caso contrario, tienden a seguir los caminos con mayor concentración de feromonas, lo cual a su vez aumenta la cantidad de feromonas. De este modo llegará un momento en el que todas las hormigas consigan ir por el camino óptimo. Con el fin de evitar una convergencia demasiado temprana de las hormigas hacia un camino subóptimo, lo que supondría que el algoritmo se quedaría atrapado en un óptimo local, se añade un mecanismo de evaporación de feromonas. Al ir disminuyendo con el tiempo la intensidad de las feromonas, se favorecerá la exploración de otras posibles soluciones [18].

- Algoritmos Genéticos: Fueron introducidos por John Holland [19] inspirándose en el proceso de evolución natural de los seres vivos. En la evolución, el problema al que los seres vivos se enfrentan cada día es la supervivencia. Para ello, cuentan con las habilidades innatas provistas en su material genético. Algunos de los principios generales de la evolución biológica que han sido aceptados por la comunidad científica son los siguientes [20]:
    - La evolución opera en los cromosomas en lugar de en los individuos a los que representan.
    - La selección natural es el proceso por el que los cromosomas con "buenas estructuras" se reproducen más a menudo que los demás.
    - En el proceso de reproducción tiene lugar la evolución mediante la combinación de los cromosomas de los progenitores. Llamamos recombinación a este proceso en el que se forma el cromosoma del descendiente. También cabe tener en cuenta las mutaciones que pueden alterar dichos códigos.
    - La evolución biológica no tiene memoria en el sentido de que en la formación de los cromosomas únicamente se considera la información del periodo anterior.

Los algoritmos genéticos representan una posible solución de un problema como un individuo constituido por un conjunto de genes. A partir de estos individuos, se van creando otros nuevos por medio de la selección, recombinación y mutación,

al igual que en la evolución biológica. De esta manera, según se van creando individuos, se van conservando los más aptos y éstos son elegidos para tener descendencia [18].

Algunos de los términos clave de los algoritmos genéticos son [21]:

- *Cromosoma*: Los individuos (que representan las posibles soluciones del problema) se codifican como un conjunto de parámetros (denominados genes) que se agrupan en cadenas de valores típicamente binarios, que se conocen como cromosomas.
- *Fitness* o función de aptitud: Se emplea para seleccionar los individuos de la población que se reproducirán, con el fin de que sobrevivan los más aptos y pasen sus buenos genes a sus descendientes. Se basa en la función objetivo del problema a resolver.
- *Selección*: De todos los individuos de la población, sólo algunos serán elegidos para reproducirse y pasar sus genes a la siguiente generación. Dicha selección se realizará en función de lo aptos que sean los individuos (*fitness*). Algunos operadores de selección son *Roulette Wheel Selection*, *Tournament Selection* o *Rank Selection.*
- *Crossover*: También llamado cruce o recombinación. Consiste en la generación de un nuevo individuo *hijo* a partir de los genes de unos individuos *padres.* Usualmente se empleará a dos padres en la creación de los hijos, aunque este número puede variar. Algunos operadores de cruce son SPX (*Single Point Crossover*), en el que elige un punto de corte y se crea el hijo con los cromosomas de un padre hasta el punto de corte y los del otro a partir de dicho punto; DPX (*Double Point Crossover*), donde se emplean dos puntos de corte o UPX (*Uniform Point Crossover*), donde para cada cromosoma se decide si procederá de un padre o del otro.
- *Mutación*: Consiste en la modificación, normalmente aleatoria, de los genes de un individuo. Permite introducir un factor de diversificación en la población para evitar quedarse atrapado en óptimos locales. Este operador va ganando en importancia a medida que la población va convergiendo. Algunos ejemplos pueden ser sustituir, desplazar o invertir valores aleatorios del cromosoma.

También cabe mencionar que dentro de los algoritmos genéticos se pueden encontrar algoritmos *generacionales* o *estacionarios*, en función de si la nueva población de hijos sustituye por completo a la de los padres o por el contrario la población se conserva y se modifica sustituyendo a los elementos antiguos por aquellos que son más aptos en base a unos criterios de reemplazo.

Finalmente, es conveniente mencionar una variante llamada *algoritmos meméticos*, los cuales son un caso particular en el cual los individuos son mejorados mediante una búsqueda local.

2.3.- ARQUITECTURAS PARALELAS

En esta sección se hablará de las distintas clases de arquitecturas paralelas existentes, haciendo especial énfasis en la clasificación en función de la organización de la memoria, así como de las herramientas más empleadas para trabajar con éstas.

2.3.1.- Introducción al procesamiento paralelo

El procesamiento paralelo es un tipo de procesamiento de información que permite la ejecución concurrente de varios procesos. También se podría definir como el uso de varios procesadores trabajando juntos para resolver una tarea común. Así, un problema es dividido en partes, cada una de las cuales es ejecutada en paralelo en diferentes procesadores. La programación para esta forma de cálculo es conocida como programación paralela, y las plataformas computacionales donde pueden ser ejecutadas estas aplicaciones son los sistemas paralelos y/o distribuidos.

La principal razón para el uso del procesamiento paralelo es la necesidad de una mayor potencia y velocidad de cálculo, ya que, independientemente de que la potencia de los procesadores aumente, siempre habrá un límite que dependerá de la tecnología del momento. La idea que se usa es que al tener **n** computadores se debería tener **n** veces más

poder de cálculo, lo que conllevaría a que el problema pudiera resolverse en **1/n** veces del tiempo requerido por la ejecución con un único proceso, es decir por la ejecución secuencial. Por supuesto, esto se cumpliría bajo condiciones ideales que, como establece la Ley de Amdahl [22], no puede conseguirse en la práctica. Sin embargo, mejoras cercanas a las ideales pueden ser alcanzadas dependiendo del problema, de la cantidad de paralelismo presente en el mismo (es decir, del paralelismo inherente), de la arquitectura, del número de procesos, etc. [23].

Uno de los sistemas más empleados en la clasificación de las arquitecturas paralelas es la llamada taxonomía de Flynn. Esta clasificación se basa en el análisis del flujo de instrucciones y de datos, los cuales pueden ser simples o múltiples. Un flujo de instrucción es una secuencia de instrucciones transmitidas desde una unidad de control a uno o más procesadores. Un flujo de datos es una secuencia de datos que viene desde un área de memoria a un procesador y viceversa. Existen cuatro categorías, las cuales son:

- SISD (*Single Instruction Stream, Single Data Stream*): En esta arquitectura, un único programa es ejecutado usando solamente un conjunto de datos específicos a él. Está compuesto de una memoria central donde se guardan los datos y los programas y de un procesador. En esta plataforma sólo se puede dar un tipo de paralelismo virtual a través del paradigma de multitareas, en el cual el tiempo del procesador es compartido entre diferentes programas. Un ejemplo de esta arquitectura es un computador monoprocesador con arquitectura Von-Neumann.
- SIMD (*Single Instruction Stream, Multiple Data Stream*): Esta arquitectura consiste en un conjunto de elementos de procesamiento que ejecutan la misma instrucción al mismo tiempo. El enfoque de paralelismo usado aquí se denomina *paralelismo de datos*. En estas arquitecturas se envía la misma secuencia de instrucciones a ejecutar a los distintos procesadores, los cuales procesarán sus datos empleando dichas instrucciones. Estos datos pueden estar en un espacio de memoria común o propio de cada procesador. Esta arquitectura hace un uso eficiente de la memoria y facilita la codificación, pues para el programador será como si se estuviera ejecutando de forma secuencial, pero tiene el inconveniente de que no es aplicable a todos los problemas, ya que el problema debe aceptar una descomposición de datos. Un ejemplo de máquina sería el computador CM-2 (*Connection Machine – 2*), del MIT.

- MISD (*Multiple Instruction Stream, Single Data Stream*): En esta arquitectura, cada elemento de procesamiento ejecuta una tarea diferente, de tal forma que todos los datos a procesar deben ser pasados a través de cada elemento de procesamiento. Se considera que esta arquitectura únicamente existe a nivel teórico, aunque algunas arquitecturas específicas como las llamadas *pipeline* podrían clasificarse como MISD.
- MIMD (*Multiple Instruction Stream, Multiple Data Stream*): Es el modelo más general de paralelismo. Las dos ideas principales son que múltiples tareas heterogéneas pueden ser ejecutadas al mismo tiempo y que cada procesador opera independientemente con ocasionales sincronizaciones con otros. Está compuesto por un conjunto de elementos de procesamiento donde cada uno realiza una tarea, independiente o no, con respecto a los otros procesadores. La conectividad entre los elementos no se especifica y puede ser explotado tanto el *paralelismo funcional* (se descompone el problema en funciones que pueden ser ejecutadas por distintos elementos de procesamiento), como el *paralelismo de datos* (el conjunto de datos se descompone en subconjuntos de datos, procesado cada uno de ellos por un elemento de procesamiento). La forma de programación usualmente utilizada es del tipo concurrente, en la cual múltiples tareas, las cuales pueden ser distintas entre ellas, se pueden ejecutar simultáneamente. Muchos multiprocesadores y sistemas de múltiples computadores pertenecen a esta clasificación.

Si nos centramos en la arquitectura MIMD, una clasificación importante, que afecta al diseño paralelo a desarrollar se centra en la organización de la memoria, existiendo dos categorías: *arquitecturas de memoria distribuida* y *arquitecturas de memoria compartida*.

2.3.2.- Arquitecturas de memoria distribuida

En las arquitecturas de memoria distribuida, cada *nodo de procesamiento* está formado por un procesador independiente y su memoria local. Estos nodos están conectados entre sí a través de una red de interconexión, la cual puede tener diferentes formas o topologías.

Al no existir memoria compartida por los nodos, la comunicación más habitual entre éstos se realizará mediante *paso de mensajes*. Estas arquitecturas se desarrollaron con el fin de lograr una arquitectura multiprocesador que fuera escalable, es decir, que pudiera tolerar un incremento en el número de elementos de procesamiento sin que esto supusiera una gran disminución de la eficiencia computacional. El rendimiento de estas arquitecturas está muy ligado al rendimiento de las comunicaciones entre nodos.

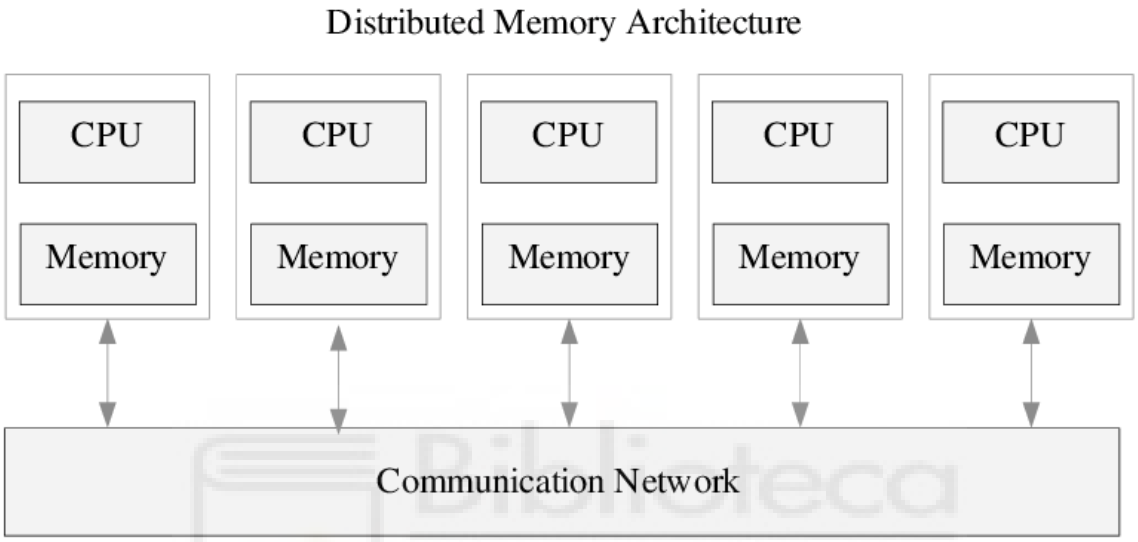


Imagen 2. Diagrama de las arquitecturas de memoria distribuida. Se puede ver que cada nodo tiene su propia memoria y que se comunica con otros nodos a través de la red de comunicaciones.

Como se mencionó en el párrafo anterior, el paradigma empleado para la programación en este tipo de arquitecturas más común es el paso de mensajes, en el cual la comunicación se hace mediante operaciones explícitas de envío y recepción de mensajes. Un mensaje es un conjunto de bytes que usualmente contiene la siguiente información: tamaño, nodo destino y dato. Este paradigma es válido también para las arquitecturas de memoria compartida, pero en éstas se suele emplear un enfoque que aproveche las capacidades que proporciona la existencia de una memoria común. Al no haber datos compartidos, en el paso de mensajes no existe el problema universal del acceso en exclusión mutua. En cambio, es de vital importancia mantener la sincronización entre los distintos nodos, para que las operaciones de envío y recepción puedan funcionar correctamente.

En un sistema de paso de mensajes, cada nodo debe:

- Conocer su dirección (una identificación unívoca empleada para el envío y recepción de mensajes) y la del resto de nodos.

- Poder crear y enviar mensajes.
- Poder recibir mensajes y trabajar con los datos de dichos mensajes.

La comunicación puede ser *síncrona* o *asíncrona* en función de si para poder realizar el envío de un mensaje se requerirá o no que ambos nodos (el emisor y el receptor) estén listos para comunicarse. La comunicación síncrona permite asegurar al emisor que el mensaje llegará al destinatario, pero puede incrementar los tiempos de espera. Por otra parte, la comunicación asíncrona no garantiza que el envío se haya realizado correctamente.

La librería (técnicamente hablando no es una librería, sino una especificación en base a la cual se han desarrollado librerías como OpenMPI, Intel MPI o MVAPICH2) de paso de mensajes más conocida y empleada es MPI (*Message Passing Interface*). MPI fue desarrollado a principios de los 90 por MPI Forum [24]. Sus principales características son estandarización, portabilidad, buenas prestaciones, amplia funcionalidad y existencia de implementaciones libres. Ha sido definida para los lenguajes C, C++ y Fortran, entre otros.

Algunas de las funciones más importantes de MPI son:

- int MPI_Init(int *argc, char ***argv): Esta función inicializa MPI. Es necesario llamarla para poder utilizar las otras funciones.
- int MPI_Finalize(): Finaliza MPI. Después de llamar a esta función, no se pueden utilizar otras funciones de MPI.
- int MPI_Comm_size(MPI_Comm comm, int *size): Devuelve el número de procesos que hay en el *comunicador* (grupo de procesos con una identificación unívoca) que se le pasa por parámetro. Por defecto existe el comunicador (MPI_COMM_WORLD), que contiene a todos los procesos
- int MPI_Comm_rank(MPI_Comm comm, int *rank): Devuelve el identificador del proceso que llama a la función dentro del comunicador que se le pasa por parámetro.
- Algunas funciones para gestionar los comunicadores son MPI_Comm_group para generar un identificador de grupo, MPI_Group_incl para añadir procesos a los grupos y MPI_Comm_create para crear comunicadores.

- int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm): Envía un mensaje. Recibe como parámetro la dirección del mensaje a enviar, el número de elementos a enviar, el tipo de datos de dichos elementos, la identificación del receptor, un identificador del mensaje y el comunicador.
- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status * status): Recibe un mensaje. Recibe como parámetro la dirección donde se recibirá el mensaje, el número de elementos a recibir, el tipo de datos de dichos elementos, la identificación del emisor, el identificador del mensaje, el comunicador y un campo de estado que permitirá conocer si el mensaje se recibió correctamente.
- También existen MPI_ISend y MPI_IRecv para realizar envíos y recepciones asíncronas (MPI_Send y MPI_Recv emplean comunicaciones bloqueantes).
- Finalmente, existe una serie de funciones colectivas en las que participan todos los procesos de un comunicador y facilitan el desarrollo. Algunas de ellas son:
    - MPI_Bcast: Permite que un proceso envíe un mensaje que será recibido por todos los demás.
    - MPI_Reduce: Permite que todos los procesos envíen datos a uno. Éste puede además operar con dichos datos.
    - MPI_Scatter: Funciona como Bcast pero enviando fragmentos del total de datos a los distintos procesos. Permite facilitar la descomposición de datos.
    - MPI_Gather: Funciona a la inversa de Scatter, recogiendo datos de los demás procesos y uniéndolos en uno.

2.3.3.- Arquitecturas de memoria compartida

En esta arquitectura, el conjunto de elementos de procesamiento comparte el mismo espacio de memoria. Debido a esto, todas las comunicaciones entre los nodos se realizarán a través de la memoria. Pese a que la memoria es común, se necesitará definir algunas variables como privadas, las cuales únicamente deberán poder ser accedidas desde un nodo. La correcta gestión de la memoria es un aspecto clave a la hora de desarrollar un código que funcione correcta y eficientemente.

Dentro de las arquitecturas de memoria compartida, se puede distinguir dos tipos de máquinas, las UMA (*Uniform Memory Access*), en las cuales todos los accesos a memoria tardan lo mismo, y las NUMA (*NonUniform Memory Access*), en las que los accesos a memoria pueden tardar tiempos diferentes en función de si el dato en cuestión se encuentra o no en la memoria cercana a ese procesador. Debido a que todos los procesadores comparten una misma conexión, la cual se convierte en un cuello de botella, las arquitecturas UMA no son escalables a un gran número de procesadores. En cambio, resultan más fáciles de programar, ya que, si se trabaja con arquitecturas NUMA, el programador debe ser consciente de dónde se guardarán los datos compartidos para evitar problemas de eficiencia [25].

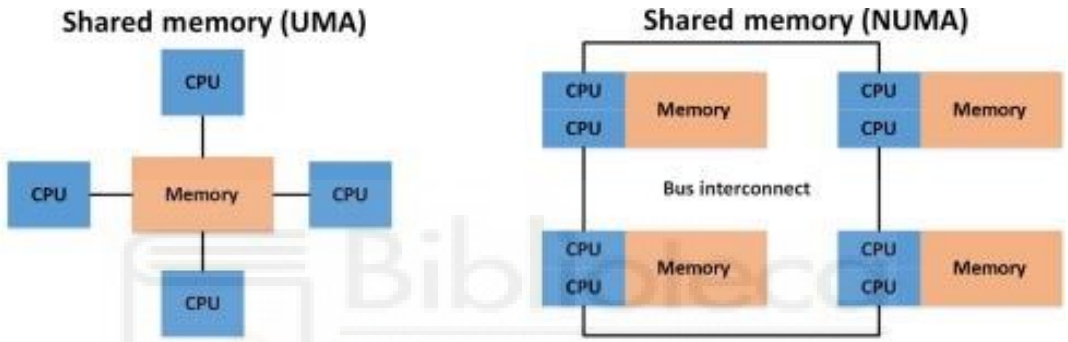


Imagen 3. Diagrama de los dos tipos de arquitecturas de memoria compartida: UMA (Uniform Memory Access) y NUMA (NonUniform Memory Access).. La diferencia entre ambas radica en que en la primera los accesos a cualquier dato en memoria son uniformes para cualquier nodo, mientras que en la segunda, estos datos pueden estar en la memoria cercana a uno u otro nodo.

Un concepto clave en la programación en arquitecturas de memoria compartida es el de *exclusión mutua*. Debido a que todos los nodos tendrán acceso a la misma memoria (salvo por aquellas variables que sean declaradas explícitamente como privadas) resultará imprescindible controlar los casos en los que dos o más procesos puedan acceder al mismo tiempo a una misma posición de memoria y se puedan ocasionar situaciones de inconsistencia (por ejemplo, cuando dos procesos traten de modificar simultáneamente los mismos datos, o cuando un proceso modifique unos datos en mitad de la lectura de éstos por parte de otro proceso). La parte del código en la cual se accede a datos compartidos se conoce como *sección crítica*, y es preciso evitar mediante herramientas de programación que más de un proceso pueda estar ejecutando esta sección crítica a la vez. Las formas más comunes de solventarlo son las empleadas, entre otros, en los Sistemas Operativos: semáforos, monitores, interrupciones, etc.

La API (interfaz de programación de aplicaciones) OpenMP permite la creación de programas que exploten el paralelismo en arquitecturas de memoria compartida. OpenMP quiere decir *Open Multi-Processing* y fue creada a finales de la década de los 90 con especificaciones para Fortran y C/C++.

OpenMP emplea un modelo fork-join. Al inicio del programa se ejecuta un único *thread* (el maestro), hasta que se llega a una región paralela, para la cual se crean *threads* esclavos que se ejecutan en paralelo. Finalmente, se sincronizan los *threads* y el maestro continúa la ejecución. Se define a un *thread* como una entidad de ejecución con una pila y su propia memoria privada. Salvo que se esté empleando *hyperthreading*, un *thread* se corresponderá con un elemento de procesamiento, con el fin de obtener el máximo rendimiento posible.

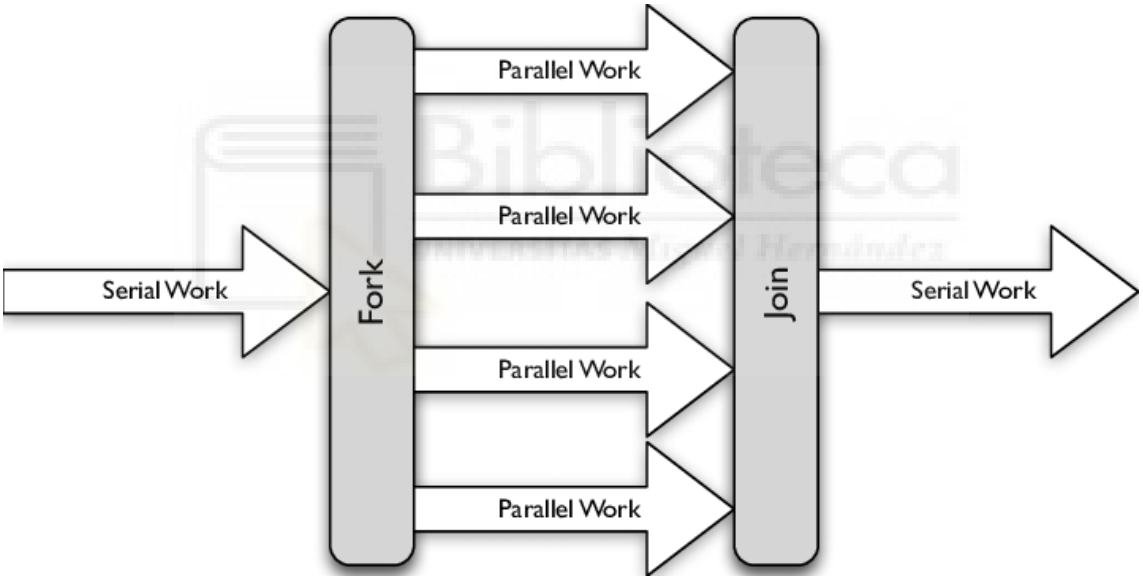


Imagen 4. Esquema del modelo fork-join utilizado por OpenMP para la ejecución en paralelo.

El modelo de memoria de OpenMP es de consistencia relajada, esto quiere decir que cada *thread* tiene una visión propia local en caché de las variables globales y las actualizaciones de éstas por parte de un *thread* no serán siempre visibles a los otros. Es importante, por tanto, gestionar correctamente en qué casos será necesario forzar la realización de procesos de consistencia de memoria.

Para permitir al programador el desarrollo de código paralelo, OpenMP implementa una serie de directivas (sentencias en el código fuente que indican al preprocesador que realice

acciones específicas previamente al proceso de compilación), cláusulas (que especifican el comportamiento de dichas directivas), funciones y variables de entorno [26]. OpenMP es no intrusivo, por lo que, si se compila sin OpenMP, las directivas serán ignoradas. La estructura de las directivas es #pragma omp nombre-directiva [cláusulas].

Se puede dividir a las directivas (o *constructores*) de OpenMP en cuatro grupos:

- Constructores de paralelización
- Constructores de ejecución secuencial
- Constructores de sincronización
- Constructores de gestión de memoria

Dentro de los constructores de paralelización, se pueden encontrar los siguientes:

- #pragma omp parallel: Crea una región paralela. El *thread* que pone en marcha la región actuará como maestro, teniendo 0 como identificador. El código contenido dentro del bloque de esta región será ejecutado por todos los *threads* de ésta, existiendo una barrera implícita al final de ella, lo cual asegurará que no se pueda continuar la ejecución del programa hasta que todos los *threads* hayan finalizado su ejecución del código de la región paralela. Esta directiva acepta la cláusula *num_threads()*, en la cual se podrá especificar el número de *threads* de dicha región. En caso de no especificarse, se utilizará el valor de la variable de entorno OMP_NUM_THREADS. Es posible la generación de una región paralela dentro de otra región paralela (*paralelismo anidado*), pero para poder hacerlo deberá habilitarse esta funcionalidad, que por defecto está desactivada.
- #pragma omp for: Paraleliza automáticamente un bucle for distribuyendo el trabajo a realizar entre los distintos *threads*. Debe encontrarse dentro de una región paralela y presenta una barrera implícita al final a menos que se emplee la cláusula *nowait*. Para poder emplear este constructor es importante que las iteraciones del bucle sean independientes entre sí. Mediante la cláusula *schedule* es posible especificar cómo se realizará el reparto de trabajo del bucle entre los threads, existiendo las siguientes opciones:
    - schedule(static,tamaño): Las iteraciones se dividirán en bloques del tamaño indicado y dichos bloques se repartirán de forma estática a los *threads*. Si no se especifica tamaño se repartirán por igual.

    - schedule(dynamic,tamaño): Igual que *static* pero los bloques se repartirán de forma dinámica, es decir, los nuevos bloques se irán asignando a los *threads* que vayan quedando libres.
    - schedule(guided,tamaño): Se asignarán dinámicamente pero con tamaños decrecientes, comenzando con *numiteraciones/numthreads* hasta alcanzar la variable tamaño.
    - schedule(runtime): Se decide el reparto en tiempo de ejecución a partir de la variable de entorno OMP_SCHEDULE
    - schedule(auto): Se deja la decisión al compilador
- #pragma omp sections: Contiene en su interior varios bloques de código precedidos por la directiva #pragma omp section, los cuales serán realizados cada uno de forma independiente por un *thread*. Debe encontrarse dentro de una región paralela y presenta una barrera implícita al final a menos que se emplee la cláusula *nowait*. Este constructor permite realizar el modelo de paralelismo de *descomposición funcional*.
- #pragma omp parallel for y #pragma omp parallel sections: Permiten crear de forma abreviada una región paralela cuyo único contenido será un for paralelo o un sections. Admite todas sus cláusulas salvo nowait.

En cuanto a los constructores de ejecución secuencial, que permiten que un determinado código no se ejecute de forma paralela, se pueden encontrar los siguientes:

- #pragma omp single: Contiene en su interior un bloque de código, el cual será ejecutado únicamente por un *thread*. Hay una barrera implícita al final a menos que se emplee la cláusula *nowait*.
- #pragma omp master: Contiene en su interior un bloque de código que será ejecutado únicamente por el *thread* maestro (el que tiene identificador = 0). A diferencia de single, no tiene una barrera al final.
- #pragma omp ordered: Se utiliza dentro de un constructor for, el cual debe llevar la cláusula *ordered*. El bloque situado dentro de este constructor se ejecutará en el orden en el que se ejecutaría si estuviese en secuencial.

OpenMP presenta estos constructores para permitir la sincronización:

- #pragma omp critical: El bloque de código contenido dentro de este constructor (llamado sección crítica) será ejecutado por todos los *threads*, pero asegurando la exclusión mutua, es decir, que sólo podrá haber un único *thread* ejecutando dicha sección al mismo tiempo. Puede especificarse un nombre para la sección crítica, permitiendo distinguir distintas secciones críticas que no tienen por qué ser mutuamente excluyentes.
- #pragma omp atomic: Funciona de forma similar a *critical*, pero penalizando mucho menos la eficiencia computacional. Sólo permite realizar una operación sencilla aplicada a una única variable.
- #pragma omp barrier: Permite crear una barrera de sincronización para los *threads*. Cuando un *thread* llegue a una barrera, detendrá su ejecución y no la continuará hasta que todos los *threads* hayan alcanzado dicha barrera. Además, incluye implícitamente un proceso de consistencia, por lo que asegura una visión consistente de la memoria al salir de la barrera.

Finalmente, para gestionar las actualizaciones de los valores de las variables en la memoria (debido a que se emplea el modelo de consistencia relajada descrito anteriormente), se dispone de la siguiente directiva:

- #pragma omp flush (lista): Asegura que las variables de la lista son actualizadas para todos los *threads*. El *thread* que ejecuta un *flush* transmite sus cambios a las variables globales (no a la visión local de los otros *threads*) o actualiza su visión local de la variable global. Existe un *flush* implícito en algunas sentencias como al final de *parallel, for, sections, single, o barrier* o a la entrada y salida de *critical u ordered*, entre otras.

Como se mencionó anteriormente, un correcto uso de la memoria es crucial para el buen desarrollo en OpenMP, ya que es vital definir adecuadamente las variables como públicas o privadas según el uso que se les vaya a dar. Por lo general, es preferible emplear memoria privada cuando se va a usar para lectura y escritura, pero no es conveniente reservar grandes cantidades de memoria como privada. OpenMP dispone de una serie de cláusulas de alcance de datos, que permiten definir cómo deberá ser tratada una determinada variable en memoria. Las cláusulas más destacadas son:

- private (lista): Define las variables como privadas de cada *thread*, no siendo éstas inicializadas antes de entrar en la región que contiene la cláusula y sin guardar su valor al salir de ésta. Cada *thread* tendrá su propia copia independiente de la variable en memoria, a la cual sólo él podrá acceder para lectura y escritura.
- shared (lista): Define las variables como globales, con un espacio en la memoria al cual podrán acceder todos los *threads*, tanto para leer como para escribir. Es conveniente, por tanto, gestionar el correcto acceso tanto para la escritura (*critical*, *ordered*, etc.) como para la lectura (*flush*).
- firstprivate (lista): Funciona de forma similar a *private*, pero inicializando el valor de la variable de cada *thread* al valor que ésta tenía antes de la región que contiene la cláusula. No se guarda el valor al salir de la región, sino que se mantendrá el original.
- lastprivate (lista): Se aplica en *for* paralelos o en *sections*. La variable será privada dentro de la región con la cláusula y, al finalizar ésta, el valor de la variable se actualizará al de la última iteración (*for*) o sección (*section*) para que pueda ser usado fuera de la región.
- default (tipo | none): Permite definir para una región el tipo que tendrán por defecto las variables que no se hayan especificado en dicha región. Empleando *none* se dejará sin especificar el resto de variables, por lo que no podrán ser usadas dentro de la región si no se especifican por otro medio.
- reduction (operador:lista): Se utiliza en variables compartidas. Cada *thread* calcula su valor parcial (como si la variable fuese privada) y posteriormente se realizará la operación especificada con todos los valores de cada uno de los *threads* para obtener el valor que finalmente tendrá esa variable al finalizar la región con la cláusula.
- copyprivate (lista): Se emplea dentro de una región single. Permite copiar el valor de una variable privada de un *thread* (el que ejecuta el single) al resto.
- #pragma omp threadprivate (lista): Técnicamente es una directiva y no una cláusula. Las variables de la lista serán privadas para cada hilo, copiando el valor del master al resto de *threads* y guardando el valor del master al finalizar la región.
- copyin (lista): Permite copiar el contenido de una variable definida como *threadprivate* al resto de hilos. Esto permite emular el comportamiento de la directiva *threadprivate* en regiones paralelas anidadas.

Finalmente, OpenMP incorpora una serie de funciones y variables de entorno que permiten controlar aspectos cruciales en la implementación de un código paralelo. Las más importantes son:

- void omp_set_num_threads(int num_threads): Especifica el número de *threads* que se empleará en la siguiente región paralela.
- int omp_get_num_threads(void): Devuelve el número de *threads* que existen en la región actual.
- int omp_get_max_threads(void): Devuelve el número máximo posible de *threads* para el equipo donde se está ejecutando OpenMP.
- int omp_get_num_procs(void): Devuelve el número máximo de elementos de procesamiento que pueden ser asignados al programa. La diferencia entre *omp_get_max_threads* y esta función es que *omp_get_max_threads* permite obtener el número máximo de *threads* lógicos, mientras que *omp_get_num_procs* obtiene el número de *threads* físicos. Esta diferencia sólo existe cuando se está utilizando *hyperthreading*.
- int omp_get_thread_num(void): Devuelve el identificador del *thread*. Si éste es 0, dicho *thread* actuará como master dentro de esa región.
- int omp_in_parallel(void): Devuelve un valor distinto de 0 si se está ejecutando dentro de una región paralela.
- int omp_set_dynamic(int): Permite activar o desactivar la creación dinámica de *threads*, en la cual se decidirá el número de *threads* de una región en tiempo de ejecución dependiendo del estado de la máquina.
- int omp_get_dynamic(void): Devuelve un valor distinto de 0 si se ha habilitado la creación dinámica de *threads*.
- int omp_set_nested(int): Permite activar o desactivar el paralelismo anidado, es decir, la posibilidad de crear una región paralela dentro de otra región paralela.
- int omp_get_nested(void): Devuelve un valor distinto de 0 si se ha habilitado el paralelismo anidado.
- double omp_get_wtime(void): Devuelve el tiempo de ejecución transcurrido en segundos.
- OMP_SCHEDULE: Controla el tipo de *scheduling* que se empleará por defecto en los for paralelos.

- OMP_NUM_THREADS: Controla el número de *threads* por defecto que tendrá una región paralela.
- OMP_DYNAMIC: Controla la creación dinámica de *threads*. Valdrá 1 si está activada y 0 en caso contrario.
- OMP_NESTED: Controla el paralelismo anidado. Valdrá 1 si está habilitado y 0 en caso contrario.

Para finalizar este capítulo sobre arquitecturas que permiten desarrollar la programación paralela, cabe mencionar que en este trabajo no van a ser considerados los aceleradores hardware como GPU (*Graphics Processing Unit*) o FPGA (*Field Programmable Gate Arrays*), los cuales han experimentado un gran auge en los últimos años y permiten llevar la aceleración de la computación a una nueva dimensión.

CAPÍTULO 3: ALGORITMOS DISCRETOS DE OPTIMIZACIÓN

En este capítulo se describirán y explicarán tres algoritmos metaheurísticos de optimización: TSA (*Tree-Seed Algorithm*), JAYA y TLBO (Teaching-Learning-Based Optimization), así como sus adaptaciones para la resolución de problemas de optimización discretos: DTSA, DJAYA y DTLBO, de los cuales se incluirá un pseudocódigo.

3.1.- ALGORITMO DTSA

En esta sección se describe el algoritmo TSA (*Tree-Seed Algorithm*) del que deriva el algoritmo discreto DTSA (*Discrete TSA*) utilizado en este trabajo.

3.1.1.- Descripción

El algoritmo TSA (*Tree-Seed Algorithm*) es un algoritmo metaheurístico de optimización basado en poblaciones inspirado por la naturaleza. Este algoritmo funciona a partir de las relaciones entre sus semillas.

TSA fue propuesto por Kiran en 2015 para resolver problemas de optimización continuos [27]. En este algoritmo, las posiciones de árboles y semillas en el espacio de búsqueda se corresponden con las posibles soluciones al problema de optimización. En la fase de inicialización los árboles son generados aleatoriamente, y en las iteraciones, se generan semillas para cada árbol. A continuación, se selecciona la mejor semilla de cada árbol, y si ésta es mejor que su árbol, lo sustituye. La generación de semillas se realiza empleando las ecuaciones (1) ó (2). La ecuación (1) emplea el mejor árbol de la población, mientras que la ecuación (2) utiliza el árbol actual. El uso de una u otra ecuación a la hora de generar las semillas es controlado por el parámetro ST (*Search Tendency*), que determina la probabilidad que en la creación de una semilla se use la ecuación (1) (que proporciona mayor explotación) o la ecuación (2) (que proporciona mayor exploración).

Como se mencionó en el apartado sobre algoritmos metaheurísticos, la explotación consiste en mejorar lo máximo posible una posible solución que tenemos, mientras que la exploración consiste en la posibilidad de obtener nuevas soluciones potencialmente mejores en la totalidad del espacio de búsqueda.

$$S_{i,j} = T_{i,j} + \alpha_{i,j} * (B_j - T_{r,j}) \qquad \textbf{(1)}$$

$$S_{i,j} = T_{i,j} + \alpha_{i,j} * (T_{i,j} - T_{r,j}) \qquad \textbf{(2)}$$

Como los problemas que se están tratando aquí son discretos, se ha tenido que implementar una versión del algoritmo TSA que trabaje con variables de decisión discretas (DTSA) [28]. Para generar las semillas en esta versión discreta del TSA, se utilizarán los operadores de transformación *swap*, *shift* y *symmetry*, que serán descritos a continuación. También se ha decidido que, en la generación inicial de los árboles, uno de ellos no sea generado aleatoriamente, sino que se genere a partir del *nearest neighbor tour* (algoritmo voraz que va seleccionando el camino inmediatamente más corto), el cual permite obtener rápidamente una buena solución.

La transformación *swap* selecciona aleatoriamente dos números **x** e **y** entre 0 y N-1, siendo **N** la dimensión del problema, es decir, el número de variables de diseño e intercambia los valores de ambas posiciones del vector. Esta transformación realiza pequeños cambios en el árbol.

La transformación *shift* selecciona aleatoriamente dos números **x** e **y** entre 0 y N-1, siendo **N** la dimensión del problema. El valor de la posición **x** del vector se almacena y los valores situados a la derecha de éste, hasta llegar a **y**, se desplazan una posición hacia la izquierda. Finalmente, se guarda el valor que tenía la posición **x** en la posición **y**. Esta transformación realiza cambios intermedios en el árbol.

Para la operación *symmetry* se han probado 3 posibilidades distintas, y finalmente se ha escogido Symmetry2, que ha sido la que ha ofrecido mejores resultados. Esta transformación realiza grandes cambios en el árbol.

- Symmetry1: selecciona aleatoriamente dos números **x** e **y** entre 0 y N-1, siendo **N** la dimensión del problema y un número **t** menor que N/2. Se toman dos bloques de tamaño **t** que comienzan en las posiciones **x** e **y** respectivamente. El contenido de los dos bloques se invierte dentro del propio bloque y posteriormente los bloques son intercambiados.
- Symmetry2: se selecciona aleatoriamente una posición **x** y un tamaño **t**. Se invierten los **t** valores situados a la izquierda y a la derecha de la posición **x**, que actúa como centro de la simetría. A diferencia de *Symmetry1*, se invierte únicamente un bloque y no dos bloques, los cuales en *Symmetry1* podían ser no contiguos.

- Symmetry3: funciona igual que *Symmetry1* con la diferencia de que las posiciones **x** e **y** no serán las de comienzo de ambos bloques, sino que serán la de fin del bloque situado más a la izquierda y la de comienzo del bloque situado más a la derecha.

3.1.2.- Pseudocódigo

Las siguientes figuras ilustran el pseudocódigo del algoritmo TSA discreto que se ha implementado.

```
Set maxfes (número máximo de evaluaciones de funciones)
Set D (variables de decisión)
Set N (tamaño de la población)
Determinar el parámetro ST (Search Tendency) entre 0 y 1
Generar el primer árbol (X^0) usando nearest neighbor tour
for i=1 to N do
        Generar el árbol (X^i) como una permutación aleatoria
end for
for i=0 to N do
        Calcular F_obj(X^i)
end for
fes = N
Determinar el mejor árbol usando las F_obj (X^best)
while fes < maxfes
        for i=0 to N do
                Set ns (número de semillas)=6
                Determinar un árbol aleatorio de la población (exceptuando al actual) (X^i)
                Generar un número aleatorio entre 0 y 1 (rand)
                if rand < ST
                        Generar una semilla con el operador swap usando X^best (s^1)
                        Generar una semilla con el operador shift usando X^best (s^2)
                        Generar una semilla con el operador symmetry usando X^best (s^3)
                        Generar una semilla con el operador swap usando X^i(s^4)
                        Generar una semilla con el operador shift usando X^i(s^5)
                        Generar una semilla con el operador symmetry usando X^i (s^6)
```

```
                else
                        Generar una semilla con el operador swap usando X^i (s^1)
                        Generar una semilla con el operador shift usando X^i (s^2)
                        Generar una semilla con el operador symmetry usando X^i (s^3)
                        Generar una semilla con el operador swap usando X^i (s^4)
                        Generar una semilla con el operador shift usando X^i(s^5)
                        Generar una semilla con el operador symmetry usando X^i (s^6)
                end if
                for i=0 to 6 do
                        Calcular F_obj (s^i)
                end for
                fes=fes+6
                Determinar la mejor semilla de las 6 (bestseed)
                        Si bestseed < F_obj(X^i)
                                Sustituir al árbol actual por bestseed
                        Fin si
        end for
        Determinar el mejor árbol (tempbest)
        if tempbest < best
                Sustituir a best por tempbest
        end if
end while
```


3.2.- ALGORITMO DJAYA

En esta sección se describe el algoritmo JAYA, del que deriva el algoritmo discreto DJAYA (*Discrete Jaya*) utilizado en este trabajo.

3.2.1.- Descripción

El algoritmo Jaya es un algoritmo metaheurístico de optimización basado en poblaciones que permite resolver problemas continuos de optimización con y sin restricciones diseñado por Rao en 2016 [29]. Jaya se basa en el concepto de que la solución obtenida para un determinado problema debería moverse hacia la mejor solución y evitar la peor.

La característica más destacada de este algoritmo, que lo diferencia de otros similares, es que utiliza la mejor y la peor solución de la población, además de la actual, para generar la solución candidata. Esto permite mejorar la intensificación y la diversificación de la población, proporcionando una amplia área de búsqueda. Jaya es fácilmente adaptable ya que no contiene parámetros de control específicos del algoritmo, sino únicamente los parámetros de control comunes a todos los algoritmos, como el número de iteraciones y el tamaño de la población. El algoritmo Jaya realiza una buena exploración del espacio de posibles soluciones debido al uso de las soluciones mejor y peor en la generación de nuevas soluciones. Jaya utiliza la ecuación (3) para generar nuevas soluciones.

$$X'_{j,k,i} = X_{j,k,i} + r1_{,j,i}\left(X_{j,best,i} - \left|X_{j,k,i}\right|\right) - r2_{,j,i}\left(X_{j,worst,i} - \left|X_{j,k,i}\right|\right) \qquad \textbf{(3)}$$

La versión discreta del algoritmo (DJAYA) [30] que se va a implementar emplea, al igual que DTSA, los operadores de transformación *swap*, *shift* y *symmetry* para la generación de soluciones candidatas. También se utiliza para la generación del primer elemento de la población inicial el *nearest neighbor tour*.

Este algoritmo usa dos parámetros a la hora de generar las soluciones candidatas, *ST1* y *ST2*, que permiten ajustar si se quiere utilizar en mayor medida la solución actual, la mejor o la peor. Esto nos permite balancear entre exploración y explotación.

Para seleccionar uno de los tres operadores de transformación cada vez que se vaya a generar una solución candidata, se empleará la selección *Roulette Wheel*, que permite que tanto las soluciones buenas como las malas puedan ser elegidas. Esto permite que las soluciones aparentemente malas puedan proporcionar mejores soluciones candidatas y no evitar quedarse atrapado en un óptimo local. La forma en la que se ha implementado *Roulette Wheel* en este algoritmo Jaya discreto es la siguiente:

El primer 10% de las generaciones de soluciones candidatas seleccionará con igual probabilidad los tres operadores de transformación y se utilizará para determinar las probabilidades que tendrán en el 90% restante. Cada vez que una solución candidata sea mejor que su predecesora, se incrementará un contador que llevará la cuenta de todas las

veces que ese operador de transformación mejoró la solución, de esta forma el algoritmo va aprendiendo qué opción es mejor y la hace prevalecer sobre el resto.

Se ha decidido dar a los parámetros ST1 y ST2 los valores de 0.4 y 0.7 respectivamente, para que las 3 posibilidades (actual, mejor y peor) tengan lugar con una probabilidad similar, habiendo un pequeño sesgo en favor del uso de la solución actual.

3.2.2.- Pseudocódigo

Las siguientes figuras ilustran el pseudocódigo del algoritmo Jaya discreto que se ha implementado.

```
Set maxfes (número máximo de evaluaciones de funciones)
Set D (variables de decisión)
Set N (tamaño de la población)
Determinar los parámetros ST1 y ST2 entre 0 y 1
Generar el primer elemento (X^0) usando nearest neighbor tour
for i=1 to N do
        Generar el elemento (X^i) como una permutación aleatoria
end for
for i=0 to N do
        Calcular F_obj(X^i)
end for
fes = N
Determinar el mejor elemento usando las F_obj (X^best)
Determinar el peor elemento usando las F_obj (X^worst)
while fes < maxfes
        for i=0 to N do
                Generar un número aleatorio entre 0 y 1 (rand)
                if rand <= ST1
                        Utilizar X^i
                else if rand <= ST2
                        Utilizar X^best
                else
                        Utilizar X^worst
                end if
```

```
                    Seleccionar un operador de transformación usando Roulette Wheel
                    Calcular Xi' usando el operador seleccionado
                    Calcular Fobj(Xi')
                    if Fobj(Xi') < Fobj(Xi)
                            Sustituir Xi por Xi'
                    end if
            end for
end while
```

A continuación, se muestra el pseudocódigo del Roulette Wheel que se ha decidido emplear en este algoritmo para realizar la selección del operador de transformación al generar un nuevo elemento.

```
Set contswap, contshift, contsymmetry = 0
if fes < 0.1*maxfes
            Generar un entero aleatorio entre 0 y 2 (rand2)
            if rand2==0
                        Generar Xi' usando swap
            else if rand2==0
                        Generar Xi' usando shift
            else
                        Generar Xi' usando symmetry
            end if
            Calcular Fobj(Xi')
            fes++
            if Fobj(Xi') < Fobj(Xi)
                        Sustituir Xi por Xi'
                        Incrementar el contador del operador de transformación usado
            (contswap/contshift/contsymmetry++)
            end if
end if
else
            Set porcentajeswap=contswap/(contswap+contshift+contsymmetry)
            Set porcentajeshift=contshift/(contswap+contshift+contsymmetry)
            Set porcentajesymmetry=contsymmetry/(contswap+contshift+contsymmetry)
Generar un número aleatorio entre 0 y 1 (roulette)
if roulette <= porcentajeswap
            Generar Xi' usando swap
else if roulette <= porcentajeswap+porcentajeshift
            Generar Xi' usando shift
else
            Generar Xi' usando symmetry
end if
```

3.3.- ALGORITMO DTLBO

En esta sección se describe el algoritmo TLBO (*Teaching-Learning-Based* Optimization) del que deriva el algoritmo discreto DTLBO (*Discrete TLBO*) utilizado en este trabajo.

3.3.1.- Descripción

El algoritmo TLBO (*Teaching-Learning-Based* Optimization) es un algoritmo metaheurístico de optimización basado en el proceso de enseñanza-aprendizaje desarrollado por Rao et. al en 2011 [31] que involucra a un profesor (*teacher*) y a un conjunto de alumnos (*learners*) de tal forma que cada *learner* aprende, no solo del *teacher*, sino también de otros *learners*. La principal ventaja de TLBO respecto de otros algoritmos metaheurísticos es su ausencia de parámetros específicos, ya que sólo el tamaño de la población y el número de iteraciones han de ser configurados.

Este algoritmo se basa en el efecto de la influencia de un profesor en los resultados (notas) de sus alumnos. Se considera que el profesor es un individuo con elevados conocimientos que comparte sus conocimientos con los alumnos. La calidad de un profesor afecta a los resultados de sus alumnos, ya que un buen profesor enseñará a sus alumnos de tal forma que puedan obtener mejores notas. La labor del profesor será realizada por la mejor solución obtenida hasta el momento. El algoritmo se divide en dos fases: "*Teacher Phase*", donde los *learners* (alumnos) aprenden del *teacher* (profesor) y "*Learner Phase*", donde aprenden en base a la interacción entre ellos.

- Teacher phase

  En esta fase, se calcula la función objetivo de todos los elementos y se designa como *teacher* al más óptimo. El *teacher* es utilizado para crear una nueva versión de cada elemento de la población o *learner* usando la ecuación (4). A continuación se compara la función objetivo del elemento actual y del nuevo y si ha habido una mejora se sustituye por el nuevo.

$$X_{new}(i,j) = X(i,j) + rand(0,1)(X_{best}(j) - TFactor * X_{mean}(j)) \qquad \textbf{(4)}$$

- Learner phase

  Se utiliza el mejor elemento de la población (aquel con un mejor valor de la función objetivo), el peor (aquel con el peor valor de la función objetivo) y uno aleatorio para generar un nuevo elemento empleando la ecuación (5). Si ese elemento es mejor que el anterior, se sustituye.

  $$X_{new}(i,j) = X(i,j) + rand(0,1) * (BestLearner(j) - WorstLearner(j))$$ **(5)**

Para poder tratar con problemas discretos, se ha hecho uso de un algoritmo TLBO discreto (DTLBO) [32][33]. Una característica que cabe destacar del DTLBO es que divide la población en subpoblaciones, con el fin de evitar quedarse atrapado en un mínimo local. Se utilizarán 4 subpoblaciones. El primer elemento de cada subpoblación se generará usando *nearest neighbor tour*, mientras que los otros serán permutaciones aleatorias.

Para cada subpoblación se designará un *Partial Teacher* que se utilizará también en la generación de nuevos individuos. Además, se generará un elemento que contendrá los valores medios de toda la subpoblación (media aritmética de los valores de cada variable de diseño o decisión), que también será usado en la generación.

A diferencia de los otros dos algoritmos, DTLBO no siempre genera individuos factibles (es decir, aquellos en los que no se repita un mismo valor en dos posiciones distintas del vector solución). Por tanto, se tendrá que utilizar una función que transforme un individuo no factible en uno factible, la cual será descrita en la sección 3.3.2.

Para la generación de nuevos elementos se utilizarán las transformaciones *Crossover* y *Mutation*. *Crossover* toma dos elementos **x** e **y**, genera dos números aleatorios entre 0 y N-1 siendo **N** el número de variables de diseño del problema y genera un nuevo elemento que toma de **y** los valores de las posiciones situadas entre ambos números y de **x** el resto. *Mutation* emplea el operador *symmetry2* utilizado en los otros algoritmos para generar un nuevo elemento realizando modificaciones en el anterior.

3.3.2.- Pseudocódigo

Las siguientes figuras ilustran el pseudocódigo del algoritmo Jaya discreto que se ha implementado.

Set maxfes (número máximo de evaluaciones de funciones)
Set D (variables de decisión)
Set N (tamaño de la población)
Set S (tamaño de las subpoblaciones) = N/4
Generar el primer elemento de cada subpoblación (X^0) usando nearest neighbor tour
for i=0 to N do
    Calcular $F_{obj}(X^i)$
end for
fes=N
while fes < maxfes //TEACHER STAGE
    Determinar el mejor elemento usando las F_{obj} (gteacher)
    for i=0 to 4 do
        Calcular $pteacher^i$ (mejor elemento de la subpoblación i)
        Calcular m^i (elemento de medias de la subpoblación i)
        for j=0 to s do
            Generar un entero aleatorio entre 0 y 3 (rand)
            if rand == 0
                $X^{s*i+j'} = X^{s*i+j} \otimes gteacher$
            else if rand == 1
                $X^{s*i+j'} = X^{s*i+j} \otimes pteacher^i$
            else if rand == 2
                $X^{s*i+j'} = X^{s*i+j} \otimes media^i$
            else
                $X^{s*i+j'} = pteacher^i \otimes media^i$
            end if
            Aplicar operación de factibilidad
            Aplicar symmetry2
            Calcular $F_{obj}(X^{s*i+j'})$
            fes++

```
                    if F_obj(X^s*i+j') <  F_obj(X^s*i+j)
                        Sustituir X^s*i+j' por X^s*i+j
                    end if
                end for
            end for
            for i=0 to 4 do //LEARNER STAGE
                for j=0 to s do
                    Determinar un elemento aleatorio de la población (exceptuando al actual) (X^r)
                    X^s*i+j' = X^s*i+j ⊗ X^r
                    Aplicar operación de factibilidad
                    Aplicar symmetry2
                    Calcular F_obj(X^s*i+j')
                    fes++
                    if F_obj(X^s*i+j') <  F_obj(X^s*i+j)
                        Sustituir X^s*i+j' por X^s*i+j
                    end if
                end for
            end for
end while
```

A continuación, se muestra el pseudocódigo de la función de factibilidad, que se encargará de comprobar si una determinada solución es o no factible tomando un vector posible solución, comprobando si hay elementos duplicados y sustituyéndolos por aquellos que no aparezcan en el vector.

```
Definimos un vector recorridos de tamaño N e inicializamos todas sus posiciones a -1.
Para cada ciudad i desde 0 hasta N-1
        if i no se ha guardado aún en recorridos
                Guardamos en la posición i de recorridos la posición en la que aparece la ciudad i en el vector
                    que queremos transformar en factible
                Guardamos en el vector nuevo (el que será factible) la ciudad
        else (cuando una ciudad aparece más de una vez)
                Ponemos a -1 la posición del vector nuevo donde estaba anteriormente esa ciudad
                Guardamos en la nueva posición del vector nuevo la ciudad
                Guardamos en la posición i de recorridos la posición en la que aparece la ciudad i en el vector
        end if
end for
```

```
for i = 0 to N-1 do
        if nuevo(i) == -1
                for i = 0 to N-1 do
                        if ciudad i no aparece en el vector
                                Guardamos en el vector nuevo la ciudad
                                Guardamos en la posición i de recorridos la posición en la que aparece la
                                    ciudad i en el vector
                                Break
                        end if
                end for
        end if
end for
```

CAPÍTULO 4: ALGORITMOS PARALELOS

En este capítulo se describirán los distintos algoritmos que se han implementado en este trabajo con el fin de paralelizar con diferentes enfoques los tres algoritmos metaheurísticos con los que se está trabajando (DTSA, DJAYA y DTLBO).

4.1.- VERSIÓN CON AUTOMATIZADORES

En esta sección se explicará uno de los tres diseños paralelos realizados, la implementación con automatizadores, que utiliza el constructor *for* paralelo para repartir el trabajo entre los *threads*.

4.1.1.- Descripción

La idea principal de la primera propuesta paralela es reducir el tiempo computacional del bucle *for* principal del algoritmo, esta propuesta va a ser aplicada a los tres algoritmos que están siendo estudiados en este trabajo, es decir DTSA, DJAYA y DTLBO.

En el esquema iterativo de estos tres algoritmos, la tarea de dicho bucle consiste en recorrer uno a uno los individuos de la población para generar nuevos individuos empleando el método concreto de cada algoritmo, para posteriormente comparar al nuevo individuo con el antiguo y sustituirlo si ha habido una mejora en la función de coste. Para el caso del DTLBO son en realidad dos bucles *for*: el de la *teacher phase* y el de la *learner phase*, pero eso no afecta a la idea de este enfoque, ya que es análogo a procesar la población 2 veces consecutivas con dos métodos diferentes.

Para reducir el tiempo computacional se generará una región paralela y se empleará el constructor *#pragma omp for* para repartir el trabajo del bucle entre los distintos *threads* de la región paralela. Además de ese bucle *for*, que es sin duda alguna la sección de código que supone un mayor coste computacional, se paralelizarán usando el mismo constructor todos aquellos bucles *for* cuyo trabajo pueda ser repartido.

Uno de los cambios más notables respecto de la versión secuencial es el cálculo del mejor individuo, en el caso del DJAYA también el peor individuo y en el caso del DTLBO también el mejor individuo de cada subpoblación. Para obtener estos individuos será necesario definir un *array* donde cada *thread* guarde su resultado parcial, siendo posteriormente calculado el individuo global buscado, esta última tarea puede realizarse bien fuera de la región paralela o bien por un único *thread* dentro de ésta. Debido a las

características concretas de cada algoritmo, se ha optado por la primera opción en DTSA y DJAYA, mientras que en DLTBO se ha empleado la segunda.

Además, este procedimiento sólo se empleará en el cálculo inicial realizado antes de entrar al bucle iterativo principal (salvo por el caso del cálculo del peor individuo en DJAYA), ya que en vez de tener que volver a calcular el mejor elemento una y otra vez en cada iteración, únicamente se realizará una actualización de éste al mejor elemento de aquellos que hayan sido creados en la iteración, siempre y cuando éste último sea mejor que el mejor elemento que había al comienzo de la iteración (que estaba guardado en la variable **Xbest** que emplean los tres algoritmos para generar nuevos elementos). Para realizar esta tarea, que disminuye el coste computacional, se empleará una variable denominada **marcador**, que se pondrá a 1 en el caso de que algún *thread* haya generado algún elemento que mejore la mejor solución global actual, y únicamente cuando al final de la iteración esta variable valga 1 se realizará la actualización la mejor solución global actual. Con el DTLBO se realizará lo mismo también para los mejores elementos de cada subpoblación. Esto permite ahorrar una gran cantidad de cálculos (y por tanto coste computacional) innecesarios.

Hay que remarcar que los algoritmos han sido también optimizados computacionalmente en su versión secuencial. Además, hay que señalar que, tanto en este como en los otros algoritmos paralelos, para la obtención de números aleatorios (operación bastante frecuente) ha sido necesario emplear la función *rand_r*, ya que la función comúnmente usada, *rand*, ocasiona problemas de eficiencia cuando es llamada de forma concurrentemente por varios *threads*, al generar contenciones en el acceso a la variable global en la que se almacena la semilla.

Un problema surgido durante el desarrollo paralelo con automatizadores del algoritmo DTSA fue el acceso simultáneo a un mismo árbol (individuo de la población) por parte tanto del *thread* al que le corresponde trabajar con dicho árbol, y está escribiendo en él para actualizarlo al haberse encontrado una semilla mejor que él, como de otro *thread* que accede al árbol para leerlo, ya que al seleccionar un árbol aleatorio en su generación de semillas ha coincidido con el mismo árbol que el otro estaba modificando. Esto suponía la corrupción del árbol leído y la generación de un árbol no factible, es decir con algunas variables de diseño repetidas.

Este error no sucede en la paralelización del DJAYA, ya que este algoritmo únicamente trabaja con el árbol actual, el mejor y el peor, y basta con crear *arrays* para estos dos últimos donde se copie el mejor y el peor individuo de la población al finalizar cada iteración del *for* paralelo, habiendo una barrera al final de éste.

Por otra parte, el DTLBO, por su propia naturaleza puede generar individuos no factibles, es por tanto necesario desarrollar la denominada *función de factibilidad*, que transforma individuos no factibles en factibles. Esta función es llamada después de la realización de la operación crossover, que es la que puede generar individuos no factibles, éste es el punto crítico donde podría existir corrupción en la lectura, por lo que posibles errores serían solventados en la llamada a la *función de factibilidad*.

Volviendo al DTSA, dicho problema de posible no factibilidad fue solucionado definiendo un *array* de árboles para cada *thread* donde éste fuese guardando aquellos árboles que fuese sustituyendo (que realmente no serían sustituidos en el *array* común de árboles), además de un *array* de enteros donde fuese guardando la posición que tenían dichos árboles en el *array* común de árboles. Dichos *arrays* tendrían de tamaño el número máximo de árboles que un mismo *thread* podría llegar a procesar. Al finalizar el bucle principal del algoritmo (y, por tanto, fuera de la sección donde podría producirse un error por acceso simultáneo), cada *thread* actualiza en el *array* común aquellos árboles que deberían haber sido sustituidos.

4.1.2.- Pseudocódigo

A continuación se muestra el pseudocódigo de esta propuesta paralela, hay secciones que se han ilustrado por separado para no dificultar la comprensión. Se señala en azul el código correspondiente a la paralelización; en morado, las diferencias entre DTSA, DJAYA y DTLBO; y en verde, las secciones ilustradas por separado.

(1) Código de la versión con automatizadores

```
//Código común a los tres algoritmos
        Set numhilos (Número de threads que tendrán las regiones paralelas que se creen)
        Inicio de la región paralela
                //El master (hilo con id 0) obtendrá el número de threads por defecto, variable que será necesaria más adelante
                Inicio región master
                        numhilos = omp_get_num_threads();
                Fin región master
        Fin de la región paralela
        // Definir las variables necesarias para el algoritmo que deban ser comunes para todos los threads (población, variables que son de sólo lectura dentro del bucle principal…)
//En DTSA y DJAYA
        Generar el primer individuo (X0) usando nearest neighbor tour
        For paralelo
        for i=1 to N do
                Generar el árbol (Xi) como una permutación aleatoria
        end for
//En DTLBO
        For paralelo
        for i=0 to N do
                if i % s == 0 //Primer elemento de una subpoblación
                        Generar el elemento (Xi) usando nearest neighbor tour
                else
                        Generar el elemento (Xi) como una permutación aleatoria
                End if
        end for
//Código común a los tres algoritmos
        For paralelo
        for i=0 to N do
                Calcular Fobj(Xi)
        end for
        Código (2) //Cálculo del mejor elemento de la población (lo mismo también con el peor en el caso de DJAYA)
        //Parte principal del algoritmo
        Inicio de la región paralela
                Set id = omp_get_thread_num() // Guardar el identificador del thread
                //Definir las variables que deban ser privadas de cada thread (el array que contenga el nuevo elemento que se genere, variables necesarias para el algoritmo que se modifiquen durante las iteraciones del for…) En el caso de DTSA también los arrays auxiliares de árboles y posiciones
//Sólo en DTLBO
                Código (3) //Cálculo del mejor elemento de cada subpoblación (partial teacher) y el mejor de toda la población (global teacher)
```

```
//Código común a los tres algoritmos
        while fes < maxfes
//Sólo en DTSA y DJAYA
            For paralelo
            for i=0 to N do
                //Código del algoritmo (NOTA: cuando uno de los threads genere un elemento con una mejor función de coste que la de Xbest, pondrá una variable "marcador" a 1. En caso contrario, valdrá 0)
            end for
            //En el caso del DTSA, aquí cada thread actualiza el array común de árboles con aquellos que sustituyó, habiendo una barrera de sincronización después de la actualización
//Sólo en DTLBO
            Código (4) //Cálculo de las medias de cada subpoblación
            for i=0 to 4 do
                For paralelo
                for j=0 to s do
                    //Código teacher phase (NOTA: se empleará un array "marcadorsub" en el que cada posición "i" se pondrá a 1 en el caso de que uno de los threads genere un elemento mejor que el partial teacher de la subpoblación "i". En caso contrario valdrá 0)
                end for
            end for
            for i=0 to 4 do
                For paralelo
                for j=0 to s do
                    //Código learner phase (NOTA: se utiliza marcadorsub de la misma manera)
                end for
            end for
//Sólo en DTSA y DJAYA
            Inicio región master
                if marcador == 1 //Cuando se ha encontrado un elemento mejor que Xbest
                    Sustituir Xbest por Xbestpos, siendo bestpos una variable donde se guardó la posición del mejor elemento de los que se generaron en esta iteración
                end if
            Fin región master
//Sólo en DJAYA
            Código (5) //Cálculo de Xworst
//Sólo en DTLBO
            Código (6) //Actualización de los partial teachers y el global teacher
//Código común a los tres algoritmos
            Barrera de sincronización
        end while
    Fin de la región paralela
```

(2) Cálculo del mejor elemento (en DJAYA se realizará el mismo procedimiento para calcular el peor)

```
Definir unos vectores bestpos_par (de enteros) y fopt_par (de reales) de tamaño numhilos
for i=0 to numhilos do
        bestpos_par(i) = 0
        fopt_par(i) = FLT_MAX //Inicializar al mayor valor posible para un real
end for
//Cada thread calculará el mejor de los elementos que procese
Inicio de la región paralela
        Set id = omp_get_thread_num() // Guardar el identificador del thread
        Set posopt = 0
        For paralelo
        for i=0 to N do
                if Fobj(Xi) < Fobj(Xposopt)
                        posopt = i
                end if
        end for
        bestpos_par(id) = posopt
        fopt_par(id) = Fobj(Xposopt)
Fin de la región paralela
Set fopt = fopt_par(0)
Set bestpos = bestpos_par(0)
for i=1 to numhilos do
        if fopt_par(i) < fopt
                fopt = fopt_par(i)
                bestpos = bestpos_par(i)
        end if
end for
for i=0 to D do
        Xbest(i) = X(bestpos)(i)
end for
```

(3) Cálculo del mejor elemento de cada subpoblación (*partial teacher*) y el mejor de toda la población (*global teacher*), sólo utilizado en el algoritmo DTLBO

```
//Cálculo del global teacher
bestpos_par(id) = 0
fopt_par(id) = FLT_MAX //Inicializar al mayor valor posible para un real
Set posopt = 0
For paralelo
for i=0 to N do
        if Fobj(Xi) < Fobj(Xposopt)
                posopt = i
        end if
end for
bestpos_par(id) = posopt
fopt_par(id) = Fobj(Xposopt)
Inicio región master
        fopt = fopt_par(0)
        bestpos = bestpos_par(0)
        for i=1 to numhilos do
                if fopt_par(i) < fopt
                        fopt = fopt_par(i)
                        bestpos = bestpos_par(i)
                end if
        end for
        for i=0 to D do
                Xbest(i) = X(bestpos)(i)
        end for
Fin región master

//Cálculo de los partial teachers
Definir pteacherpos como un array de enteros de tamaño 4
for i=0 to 4 do
        Inicio región master
                pteacherpos(i) = 0
                Fopt= FLT_MAX //Inicializar al mayor valor posible para un real
        Fin región master
        bestpos_par(id) = s * i
        fopt_par(id) = FLT_MAX //Inicializar al mayor valor posible para un real
        Set s = N / 4
        posopt = s * i
```

```
        For paralelo
        for j=0 to s do
                if Fobj(X s * i + j)  < Fobj(Xposopt)
                        posopt = s * i + j
                end if
        end for
        bestpos_par(id) = posopt
        fopt_par(id) = Fobj(Xposopt)
        Barrera de sincronización
        Inicio región master
                Fopt = fopt_par(0)
                for j=1 to numhilos do
                        if fopt_par(j) < fopt
                                fopt = fopt_par(j)
                                pteacherpos(i) = bestpos_par(j)
                        end if
                end for
                for j=0 to D do
                        pteacher(i)(j) = X(pteacherpos(i))(j)
                end for
        Fin región master
end for
```

(4) Cálculo de las medias de cada subpoblación, sólo utilizado en el algoritmo DTLBO

```
for i=0 to 4 do
        For paralelo
        for j=0 to D do
                Set suma = 0
                for k=0 to s do
                        suma += X(s * i + k)(j)
                end for
                media(i)(j) = suma / s
        end for
end for
```

(5) Cálculo del del peor individuo en las iteraciones del algoritmo DJAYA

```
Definir unos vectores worstpos_par (de enteros) y fpes_par (de reales) de tamaño numhilos (realmente se definieron antes de entrar en la región paralela, cuando se calculó Xworst)
worstpos_par(i) = 0
fpes_par(i) = FLT_MIN //Inicializar al menor valor posible para un real
Set pospes = 0
For paralelo
for i = 0 to N do
        if Fobj(Xi) > Fobj(Xpospes)
                pospes = i
        end if
end for
worstpos_par(id) = pospes
fpes_par(id) = Fobj(Xpospes)
Inicio región master
        Set fpes = fpes_par(0)
        Set worstpos = worstpos_par(0)
        for i=1 to numhilos do
                if fpes_par(i) > fpes
                        fpes = fpes_par(i)
                        worstpos = worstpos_par(i)
                end if
        end for
        for i=0 to D do
                Xworst(i) = X(worstpos)(i)
        end for
Fin región master
```

(6) Actualización de los *partial teachers* y el *global teacher*, sólo utilizado en el algoritmo DTLBO

```
Inicio región master
        for i=0 to 4 do
                if marcadorsub(i) == 1 //Cuando se ha encontrado un elemento mejor que el partial teacher de
                la subpoblación i
                        Sustituir pteacher(i) por X(pteacherpos(i)), siendo pteacherpos(i) una variable
                        donde se guardó la posición del mejor elemento que se generó en esta iteración para
                        la subpoblación i.
                        if fopts(i) < fopt
                                //fopts(i) contiene la mejor función de coste de los elementos que se
                                generaron en esta iteración para la subpoblación i.
                                Sustituir Xbest por X(pteacherpos(i))
                        end if
                end if
        end for
Fin región master
```


4.2.- VERSIÓN CON SUBPOBLACIONES

En esta sección se explicará el segundo de los tres diseños paralelos realizados, la implementación basada en subpoblaciones, que divide la población para que cada *thread* trabaje de forma independiente con una parte de ésta.

4.2.1.- Descripción

Un factor que puede disminuir la eficiencia de los algoritmos paralelos es el acceso continuado para escritura a variables compartidas por los *threads*. En el diseño con automatizadores presentado en el apartado anterior esto sucede en gran medida, ya que a la hora de sustituir los elementos de la población por aquellos que han sido generados y presentan una mejor función de coste, se está accediendo a una variable que está definida como una matriz de enteros (o doble puntero a entero, para ser exactos), la cual está

situada en memoria compartida, y dicha sustitución se realiza en un gran número de ocasiones, lo cual puede generar contenciones y hacer bajar la eficiencia paralela.

Para paliar este problema inherente a la versión con automatizadores, se ha decidido realizar un diseño que se basa en aplicar una característica que el algoritmo DTLBO ya realiza en su versión secuencial, la división de la población en subpoblaciones, al diseño paralelo. La idea principal de esta versión es que cada *thread* disponga de los elementos de la población con los que va a trabajar en memoria privada y no necesite, por tanto, escribir continuamente en memoria compartida. Para ello, se dividirá la población en tantas subpoblaciones de igual tamaño como *threads* haya, y estas subpoblaciones se guardarán en matrices almacenadas en la memoria privada de cada *thread*.

A diferencia de la versión con automatizadores, el reparto de trabajo del bucle *for* principal del algoritmo no se realizará de forma automática empleando el for paralelo de OpenMP, sino que todos los hilos ejecutarán el bucle al completo, trabajando cada uno con su propia matriz de elementos de la población, que lógicamente es de tamaño inferior a la población inicial.

Por otra parte, y fundamentalmente debido a su menor tamaño, al ser únicamente un *array* y no una matriz (*array* doble), y que por tanto implica un menor número de accesos a memoria compartida, no se ha considerado necesario definir de forma privada a cada *thread* el *array* donde se guarda el valor de la función de coste de cada elemento de la población. Como este array tiene el tamaño completo (**N**) y no el tamaño de la subpoblación (**tam**), cuando sea necesario acceder a una posición de él dentro de un bucle en el que un *thread* trabaje con su subpoblación, se accederá a la posición **tam * id + j**, siendo **tam** el tamaño de la subpoblación, **id** el identificador del *thread* (obtenido mediante la función *omp_get_thread_num()* y que identificará a la subpoblación) y **j** el identificador del individuo dentro de la subpoblación.

En las situaciones en las que sea necesario disponer de memoria compartida, como por ejemplo a la hora de guardar el mejor elemento de la población (**Xbest**), se tratará de disminuir al mínimo los accesos para escritura a memoria compartida. Para el cálculo de **Xbest**, cada *thread* tendrá su propia variable **marcador**, que pondrá a 1 si ha generado un elemento mejor que **Xbest**, y al final de la iteración, si la variable **marcador** vale 1,

actualizará **Xbest** dentro de una sección crítica, ya que podría haber varios *threads* con su variable **marcador** activada. En el caso del DTLBO, los *partial teacher* y las medias de cada subpoblación pueden ser definidos como variables privadas sin que eso afecte al funcionamiento del algoritmo.

Debido a que el peor elemento (**Xworst**), que únicamente es utilizado en el algoritmo DJAYA, no presenta tanta relevancia a la hora de que el algoritmo obtenga un buen resultado, y a que su cálculo no puede ser simplificado como el de **Xbest** usando la variable **marcador**, se ha decidido prescindir del uso de un **Xworst** global, teniendo cada subpoblación su propio **Xworst** donde en cada iteración guardará el peor elemento de dicha subpoblación.

También se ha realizado la misma modificación en el algoritmo DTSA cuando éste utiliza un elemento aleatorio de la población para la generación de uno nuevo. En este diseño paralelo se seleccionará aleatoriamente un elemento que obligatoriamente debe pertenecer a su subpoblación y no uno de la población completa.

Como un *thread* no podrá en ningún momento modificar un elemento de la población al que otro *thread* pueda acceder, ya no será necesario contar con el *array* de árboles auxiliar propio del DTSA con automatizadores, que solucionaba el problema surgido cuando un *thread* escribía al mismo tiempo que otro leía el mismo árbol.

Finalmente, cabe destacar que debido a que el planteamiento del DTLBO lo hace trabajar siempre con 4 subpoblaciones, y en este enfoque el número de subpoblaciones dependerá del número de *threads* con el que se desee ejecutar el algoritmo, esta versión únicamente podrá mostrar todo su rendimiento para los algoritmos DTSA y DJAYA, pues la versión de DTLBO con subpoblaciones estará limitada a usar 4 *thread*s, es decir ajustando el número de *threads* al número de subpoblaciones y no a la inversa. En cambio, la versión híbrida (descrita en el apartado 4.3) presenta muchas características comunes con esta versión y permite incrementar el número de *threads* a usar.

4.2.2.- Pseudocódigo

A continuación se muestra el pseudocódigo, se señala en negrita las diferencias respecto de la versión con automatizadores.

(7) Código de la versión con subpoblaciones

```
//Código común a los tres algoritmos
        Set numhilos (Número de threads que tendrán las regiones paralelas que se creen)
        Inicio de la región paralela
                //El master (hilo con id 0) obtendrá el número de threads por defecto, variable que será necesaria más adelante
                Inicio región master
                        numhilos = omp_get_num_threads();
                Fin región master
        Fin de la región paralela
        // Definir las variables necesarias para el algoritmo que deban ser comunes para todos los threads (población, variables que son de sólo lectura dentro del bucle principal…)
        Set tam = N / numhilos
//En DTSA y DJAYA
        Generar el primer individuo (X0) usando nearest neighbor tour
        For paralelo
        for i=1 to N do
                Generar el árbol (Xi) como una permutación aleatoria
        end for
//En DTLBO
        For paralelo
        for i=0 to N do
                if i % s == 0 //Primer elemento de una subpoblación
                        Generar el elemento (Xi) usando nearest neighbor tour
                else
                        Generar el elemento (Xi) como una permutación aleatoria
                End if
        end for
```

```
//Código común a los tres algoritmos
        For paralelo
        for i=0 to N do
                Calcular Fobj(Xi)
        end for
        Código (2) //Cálculo del mejor elemento de la población (lo mismo también con el peor en el caso de DJAYA)
        //Parte principal del algoritmo
        Inicio de la región paralela
                Set id = omp_get_thread_num() // Guardar el identificador del thread
                //Definir las variables que deban ser privadas de cada thread (el array que contenga el nuevo elemento que se genere, variables necesarias para el algoritmo que se modifiquen durante las iteraciones del for…)
                //Definir poblacionpriv como una matriz de tamaño (tam, D)
                for i=0 to tam do
                        for j=0 to D do
                                poblacionpriv(i)(j) = X(tam*id+i)(j)
                        end for
                end for
//Sólo en DTLBO
                Código (8) //Cálculo del mejor elemento de cada subpoblación (partial teacher) y el mejor de toda la población (global teacher)
//Código común a los tres algoritmos
                while fes < maxfes
//Sólo en DJAYA
                        Código (9) //Cálculo del peor elemento de cada subpoblación (Xworst)
//Sólo en DTSA y DJAYA
                        for i=0 to tam do
                                //Código del algoritmo (NOTA: cuando uno de los threads genere un elemento con una mejor función de coste que la de Xbest, pondrá una variable "marcador" a 1. En caso contrario, valdrá 0. En esta versión la variable marcador será privada de cada thread)
                        end for
//Sólo en DTLBO
                        //Cálculo de las medias de cada subpoblación
                        for j=0 to D do
                                Set suma = 0
                                for k=0 to s do
                                        suma += poblacionpriv(k)(j)
                                end for
                                media(j) = suma / s
                        end for
```

```
                    for j=0 to s do
                        //Código teacher phase (NOTA: se utiliza la variable marcador igual
                        que en DTSA y DJAYA)
                    end for
                    for j=0 to s do
                        //Código learner phase (NOTA: se utiliza la variable marcador igual
                        que en DTSA y DJAYA)
                    end for
//Código común a los tres algoritmos
                    Código (10) //Actualización de Xbest
                    //No es necesario calcular el worst en DJAYA o el partial teacher en DTLBO
                    ya que cada thread calcula el suyo
                    Barrera de sincronización
                end while
            Fin de la región paralela
```

(8) Cálculo del mejor elemento de cada subpoblación (*partial teacher*) y el mejor de toda la población (*global teacher*), sólo utilizado en el algoritmo DTLBO

```
Inicio región master
        fopt = FLT_MAX //Inicializar al mayor valor posible para un real
Fin región master
foptpriv = FLT_MAX //Inicializar al mayor valor posible para un real
Set posopt = 0
for i=0 to s do
        //Cada thread trabaja con su subpoblación
        if Fopt(s * id + i) < Fopt(s * id + posopt)
                posopt = i
        end if
end for
foptpriv = Fopt(s * id + posopt)
if foptpriv < fopt
        Inicio región crítica
                fopt = foptpriv
                for i=0 to i=D do
                        Xbest(i) = poblacionpriv(posopt)(i)
                end for
        Fin región crítica
end if
```

(9) Cálculo del **Xworst** de cada subpoblación, sólo utilizado en el algoritmo DJAYA

```
worstpospriv = 0
fpespriv = FLT_MIN //Inicializar al menor valor posible para un real
for i=0 to tam do
        if Fopt(tam * id + i) > fpespriv
                worstpospriv = i
                fpespriv = Fopt(tam * id + i)
        end if
end for
for i=0 to i=D do
        Xworst(i) = poblacionpriv(worstpospriv)(i)
end for
```

(10) Actualización del mejor elemento (**Xbest**)

```
if marcador == 1 //Cuando se ha encontrado un elemento mejor que Xbest
        if foptpriv < fopt
                //En foptpriv se guardó la mejor función objetivo de los elementos que calculó ese thread
                Inicio región critica
                        Sustituir Xbest por poblacionpriv(bestpos), siendo bestpos una variable donde se
                        guardó la posición del mejor elemento de los que calculó ese thread
                Fin región crítica
        end if
end if
```

4.3.- VERSIÓN HÍBRIDA

En esta sección se explicará la tercera propuesta paralela diseñada, que consiste en una versión híbrida. Es decir, este enfoque combina las ideas de los dos anteriores utilizando el paralelismo anidado.

4.3.1.- Descripción

Este diseño paralelo trata de aunar las ideas de los dos códigos anteriores para conseguir una mayor eficiencia al hacer uso de una característica de OpenMP llamada *paralelismo anidado*, que consiste en crear una región paralela dentro de otra. Para poder hacer uso de esta función (debido a que se encuentra deshabilitado por defecto), es necesario modificar el valor de la variable de entorno OMP_NESTED o llamar a la función *omp_set_nested()*.

Uno de los principales motivos que ha llevado al desarrollo de este algoritmo paralelo es la problemática que surge al tratar de adaptar el diseño basado en subpoblaciones descrito anteriormente al DTLBO, pues al estar basado en la idea de utilizar 4 subpoblaciones, esa versión quedaría limitada al uso de cuatro nodos de procesamiento.

Otra razón para implementarlo es que la versión basada en subpoblaciones presenta algunas modificaciones en el código original de los algoritmos, ya que el DJAYA deja de disponer de un peor elemento global y emplea en su lugar el peor elemento de cada subpoblación y el DTSA sufre cambios en la selección de árboles aleatorios para la generación de semillas, pues sólo se toma en cuenta a la subpoblación a la que pertenece el elemento original. Estos cambios a priori no parecen suponer una pérdida de rendimiento apreciable en los algoritmos, pero si se trabaja con poblaciones de pequeño tamaño, es decir si se trabaja con un gran número de *threads* (y por tanto de subpoblaciones), sí podría suponer una diferencia notable. Con este enfoque híbrido se puede incrementar el número de *threads*, y por tanto acelerar la ejecución, sin que eso suponga grandes cambios que puedan desvirtuar a los algoritmos.

En esta versión habrá dos regiones paralelas, una dentro de la otra, presentando la externa un funcionamiento idéntico al de la versión con subpoblaciones (repartir el trabajo dividiendo la población en subpoblaciones almacenadas en memoria privada), mientras que la región paralela interna empleará automatizadores para paralelizar el trabajo del bucle en el que cada *thread* trabajaba con su subpoblación.

En el caso de los algoritmos DTSA y DJAYA se podrá decidir en el momento de la ejecución tanto el número de *threads* de la región paralela externa como el de la región interna, mientras que en el DTLBO el número de *threads* de la región externa (que define el número de subpoblaciones) estará fijado a 4 y sólo se podrá definir el número de *threads* de la región interna (que se reparten el trabajo del bucle *for* de una misma subpoblación).

Cabe destacar que en las regiones paralelas situadas fuera de la parte principal del algoritmo (como la generación aleatoria de la población o el cálculo inicial de las funciones de coste y del mejor elemento) se empleará como número total de *threads* el producto del número de *threads* especificado para la región externa por el de la interna, es decir el número total de *threads* disponibles en la ejecución paralela.

En el algoritmo DTSA será necesario emplear de nuevo los *arrays* de árboles auxiliares para impedir la corrupción de los datos que se produce cuando un *thread* escribe en un árbol al mismo tiempo que otro accede a dicho árbol para leerlo. En este diseño híbrido, los *arrays* auxiliares serán privados de cada *thread* de la región interna, ya que es entre éstos entre los que se puede producir la corrupción (*threads* pertenecientes a regiones externas distintas nunca podrán trabajar con los mismos datos, pues estarán limitados a sus subpoblaciones).

4.3.2.- Diagrama y pseudocódigo

A continuación se muestran un diagrama que ilustra el funcionamiento de este diseño y paralelo y el pseudocódigo de éste, señalando en negrita las diferencias respecto de la versión con subpoblaciones.

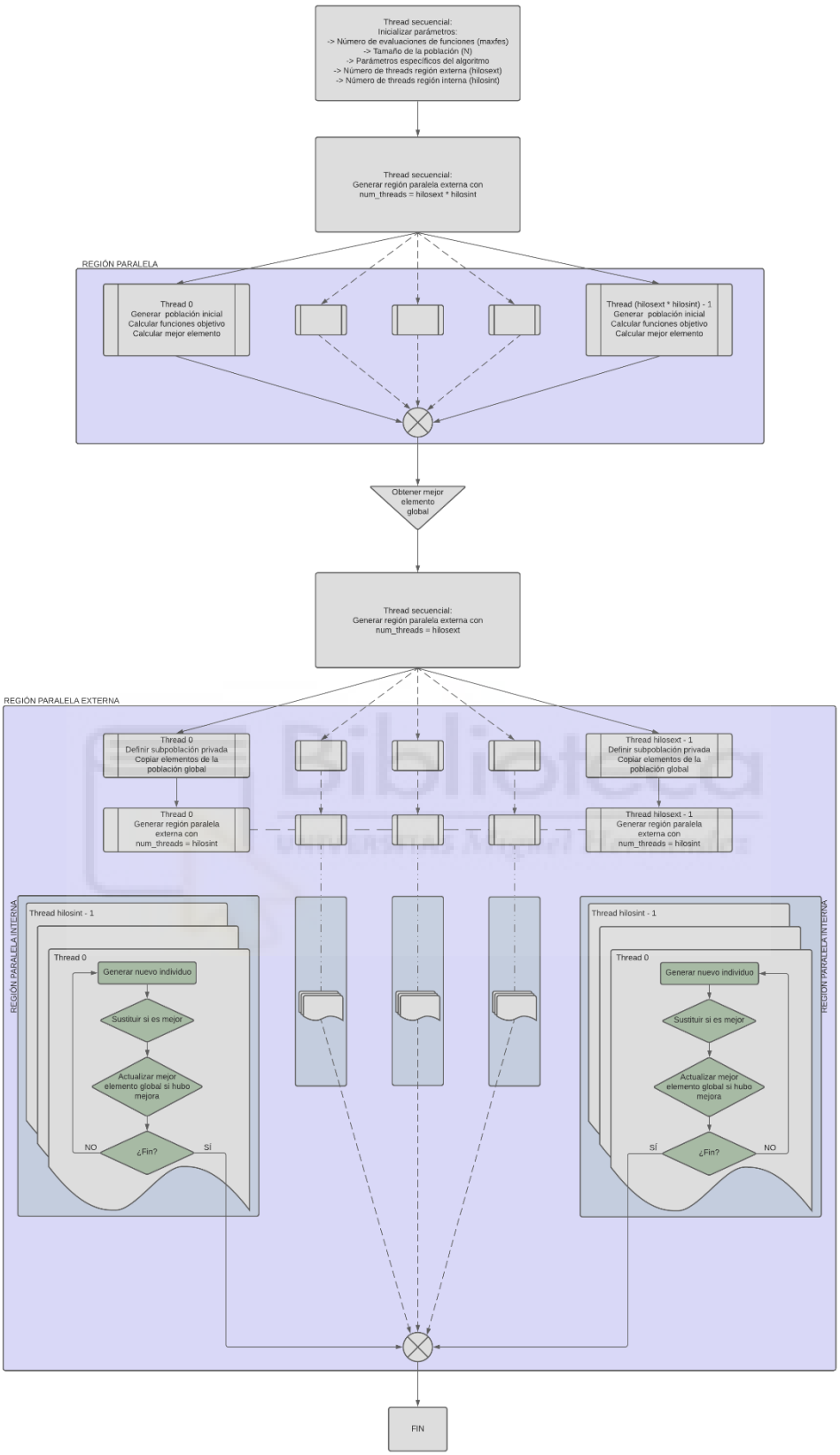


Imagen 5. En este diagrama se ilustra el funcionamiento de los algoritmos híbridos. Se muestra el caso del algoritmo DTSA, pero los otros dos presentan la misma estructura.

(11) Código de la versión híbrida

```
//Código común a los tres algoritmos
        omp_set_nested(1) //Habilitar paralelismo anidado
        Set hilosext (Número de threads de la región paralela externa)
        Set hilosint (Número de threads de la región paralela interna)
//En DTSA y DJAYA
        Inicio de la región paralela
                //El master (hilo con id 0) obtendrá el número de threads por defecto, variable que será necesaria más adelante
                Inicio región master
                        hilosext = omp_get_num_threads();
                Fin región master
        Fin de la región paralela
        //El número de hilos de la región interna viene determinado por un parámetro
//En DTLBO
        Hilosext = 4
//Código común a los tres algoritmos
        Set fullhilos = hilosext * hilosint
        // Definir las variables necesarias para el algoritmo que deban ser comunes para todos los threads (población, variables que son de sólo lectura dentro del bucle principal…)
        Set tam = N / numhilos
//En DTSA y DJAYA
        Generar el primer individuo (X0) usando nearest neighbor tour
        For paralelo con num_threads = fullhilos
        for i=1 to N do
                Generar el árbol (Xi) como una permutación aleatoria
        end for
//En DTLBO
        For paralelo con num_threads = fullhilos
        for i=0 to N do
                if i % s == 0 //Primer elemento de una subpoblación
                        Generar el elemento (Xi) usando nearest neighbor tour
                else
                        Generar el elemento (Xi) como una permutación aleatoria
                End if
        end for
//Código común a los tres algoritmos
        For paralelo con num_threads = fullhilos
        for i=0 to N do
                Calcular Fobj(Xi)
        end for
        Código (2) //Cálculo del mejor elemento de la población (lo mismo también con el peor en el caso de DJAYA)
```

```
//Parte principal del algoritmo
Inicio de la región paralela externa con num_threads = hilosext
    Set id = omp_get_thread_num() // Guardar el identificador del thread
    //Definir poblacionpriv como una matriz de tamaño (tam, D)
    for i=0 to tam do
        for j=0 to D do
            poblacionpriv(i)(j) = X(tam*id+i)(j)
        end for
    end for
    Inicio de la región paralela interna con num_threads = hilosint
        //Definir las variables que deban ser privadas de cada thread (el array que contenga el nuevo elemento que se genere, variables necesarias para el algoritmo que se modifiquen durante las iteraciones del for…) En el caso de DTSA también los arrays auxiliares de árboles y posiciones
//Sólo en DTLBO
        Código (12) //Cálculo del mejor elemento de cada subpoblación (partial teacher) y el mejor de toda la población (global teacher)
//Código común a los tres algoritmos
        while fes < maxfes
//Sólo en DJAYA
            Código (13) //Cálculo del peor elemento de cada subpoblación
//Sólo en DTSA y DJAYA
            for i=0 to tam do
                //Código del algoritmo (NOTA: se usa la variable marcador igual que en la versión con subpoblaciones)
            end for
            //En el caso del DTSA, aquí cada thread actualiza el array común de árboles con aquellos que sustituyó, habiendo una barrera de sincronización después de la actualización
//Sólo en DTLBO
            //Cálculo de las medias de cada subpoblación
            For paralelo
            for j=0 to D do
                Set suma = 0
                for k=0 to s do
                    suma += poblacionpriv(k)(j)
                end for
                media(j) = suma / s
            end for
            for j=0 to s do
                //Código teacher phase (NOTA: se utiliza la variable marcador igual que en DTSA y DJAYA)
            end for
```

```
                                        for j=0 to s do
                                                //Código learner phase (NOTA: se utiliza la variable marcador
                                                igual que en DTSA y DJAYA)
                                        end for
//Sólo en DTSA y DJAYA
                                        Código (10) //Actualización de Xbest
//Sólo en DTLBO
                                        Código (14) //Actualización de Xbest
//Código común a los tres algoritmos
                                        Barrera de sincronización
                                end while
                        Fin de la región paralela interna
                Fin de la región paralela externa
```

(12) Cálculo del mejor elemento de cada subpoblación (*partial teacher*) y el mejor de toda la población (*global teacher*), sólo utilizado en el algoritmo DTLBO

```
Set foptpriv = FLT_MAX //Inicializar al mayor valor posible para un real
Set bestpospriv = 0
For paralelo
for i=0 to s do
        if Fopt(s * id + i) < Fopt(s * id + bestpospriv)
                bestpospriv = i
        end if
end for
foptpriv = Fopt(s * id + posopt)
if foptpriv < foptsub //Actualizar partial teacher (bestpossub guarda la posición del partial teacher)
        Inicio región crítica
                foptsub = foptpriv
                bestpossub = bestpospriv
        Fin región crítica
end if
if foptpriv < fopt //Actualizar global teacher (Xbest)
        Inicio región crítica
                fopt = foptpriv
                for i=0 to i=D do
                        Xbest(i) = poblacionpriv(bestpospriv)(i)
                end for
        Fin región crítica
end if
```

(13) Cálculo del **Xworst** de cada subpoblación, sólo utilizado en el algoritmo DJAYA

```
worstpospriv = 0
fpespriv = FLT_MIN //Inicializar al menor valor posible para un real
For paralelo
for i=0 to tam do
        if Fopt(tam * id + i) > fpespriv
                worstpospriv = i
                fpespriv = Fopt(tam * id + i)
        end if
end for
if Fopt(tam * id + worstpospriv) > worstsub
        Inicio región crítica
                worstsub = Fopt(tam * id + i)
                for i=0 to i=D do
                        Xworst(i) = poblacionpriv(worstpospriv)(i)
                end for
        Fin región crítica
end if
```

(14) Actualización del mejor elemento (**Xbest**)

```
if marcador == 1
        if foptpriv < foptsub
                //En foptpriv se guardó la mejor función objetivo de los elementos que calculó ese thread
                Inicio región critica
                        foptsub = foptpriv
                        bestpossub = bestpospriv
                Fin región crítica
        end if
        if foptpriv < fopt
                Inicio región critica
                        Sustituir Xbest por poblacionpriv(bestpospriv), siendo bestpospriv una variable
                        donde se guardó la posición del mejor elemento de los que calculó ese thread
                Fin región crítica
        end if
end if
```

CAPÍTULO 5: RESULTADOS NUMÉRICOS

En este capítulo se expondrán los resultados de los experimentos realizados con los algoritmos descritos anteriormente. En primer lugar, se explicará el problema de optimización que se va a utilizar, que es el TSP (*Travelling Salesman Problem*). A continuación, se caracterizarán los algoritmos DTSA, DJAYA y DTLBO, obteniendo soluciones a problemas de este tipo. Posteriormente, se medirá el rendimiento paralelo de los tres diseños paralelos descritos en el capítulo 4. Todos los códigos han sido desarrollados en lenguaje C, en particular se ha usado la versión v.4.4.7. La API v3.1 de OpenMP ha sido utilizada para el desarrollo de los códigos paralelos. La plataforma utilizada para ejecutar los experimentos está equipada con dos procesadores Intel Xeon E5-2620 v2, cada uno de ellos con 6 cores a 2.1Ghz. El sistema operativo es CentOS Linux 6.6 no estando activado el hyperthreading.

5.1.- PROBLEMA A UTILIZAR

En esta sección se describirá el problema TSP (*Travelling Salesman Problem*), que es un problema de optimización combinatoria perteneciente a la clase NP-completo.

5.1.1.- Definición del problema TSP

El problema TSP, conocido en español como problema del viajante de comercio, es un problema de optimización combinatoria de origen incierto (una guía alemana para viajantes de 1832 menciona el problema, pero sin realizar un tratamiento matemático de éste), pero que recibe su nombre de la comunidad científica de la Universidad de Princenton (1931-1932), que fue donde se consideró la forma general del problema desde un punto de vista matemático.

El problema TSP consiste en, dada una serie de ciudades y las distancias entre cada par de ellas, hallar la ruta más corta posible que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades [34].

La definición formal de este problema es [35]:

- Dado: Un grafo completo no dirigido **G = (V, E)** con costes enteros no negativos **c (u, v)** para cada arista **(u, v) ∈ E**.
- Objetivo: Encontrar el ciclo hamiltoniano de **G** de menor coste.

En teoría de grafos se define camino hamiltoniano como una sucesión de aristas adyacentes que pasa por todos los vértices del grafo una sola vez. Si es además un ciclo (el primer y el último vértice coinciden), se llama ciclo hamiltoniano. Una definición equivalente más acorde con el problema TSP es la de una trayectoria que empieza y termina en el mismo vértice, no tiene aristas repetidas y pasa por cada vértice una única vez.

Algunas variantes del problema TSP son [36]:

- MAX-TSP: Consiste en encontrar el ciclo hamiltoniano de coste máximo.
- TSP con cuello de botella: Consiste en encontrar el ciclo hamiltoniano que minimice el mayor coste de entre todas las aristas, en vez de minimizar el coste total.
- TSP gráfico: Consiste en encontrar un circuito de coste mínimo tal que se visiten todas las ciudades al menos una vez.
- SCP (*Simple Cycle Problem*): En el ciclo solución no tienen por qué estar incluidas todas las ciudades. Esto se debe a que, para esta variante, no solo existen costes por visitar las ciudades, sino que también existen beneficios por visitar cada ciudad.
- GTSP (*Generalized Travelling Salesman Problem*): El conjunto de ciudades está dividido en regiones, y hay que visitar una ciudad de cada región minimizando el coste.
- HSP (*Hamiltonian cycle problem*): Consiste en determinar si existe o no un ciclo hamiltoniano en un grafo dado.
- TSP con múltiples viajantes: Existe un número de determinado de viajantes que deben visitar ciertas ciudades, sin visitar las que ya han sido visitadas por el resto de viajantes.
- TSP asimétrico: En este caso, la distancia (o el coste) de **i** a **j** no coincide con la distancia de **j** a **i**.

El problema TSP es un problema perteneciente a la clase NP-completo, esto significa que no existe un algoritmo tal que pueda encontrar la solución a este problema en tiempo polinómico en función del tamaño de la entrada, en este caso, el número de ciudades (vértices) a recorrer. Sin embargo, sí existen métodos capaces de obtener la solución óptima para este problema.

Algunos de los más destacados son [37]:

- Fuerza bruta: Consiste en probar todas las posibles soluciones y quedarse con la mejor. Este método tiene un orden de complejidad O(n!) (factorial), por lo que el coste computacional es excesivamente elevado incluso para un número no demasiado grande de ciudades.

- *Branch and Bound*: Este método, conocido en español como ramificación y poda, permite obtener la solución óptima al problema. Este método está considerado una generalización del *backtracking*. Ambos interpretan el problema como un árbol de soluciones, donde cada rama lleva a una posible solución posterior a la actual. La diferencia es que *Branch and Bound* permite eliminar (podar) una determinada rama cuando se determina que no puede llevar a una solución óptima.
- *Nearest Neighbor*: Es un algoritmo heurístico que fue de los primeros en ser diseñados para resolver el problema TSP. Este algoritmo no asegura una solución óptima, pero suele obtener buenas soluciones y, sobre todo, tiene un buen coste computacional, con una complejidad de $O(n^2)$, que sí es un tiempo polinómico. Este algoritmo se emplea en los algoritmos desarrollados en este trabajo para generar uno de los elementos de la población.
- También los algoritmos genéticos y los basados en enjambres, descritos en el capítulo 2, son aplicables a este problema, permitiendo la obtención de un buen resultado (si bien no siempre el óptimo) con un buen coste computacional.

TSP tiene múltiples aplicaciones en el mundo real más allá de encontrar la ruta más corta (para optimizar un transporte de mercancías o similar). Una de las más destacadas es su uso para calcular la forma más óptima de taladrar placas de circuito impreso (PCB), por ejemplo, o para determinar el orden de recogida de unas determinadas mercancías en un almacén. También se emplean variantes del problema en campos como la secuenciación del ADN o la cristalografía de rayos X.

5.1.2.- Problemas TSP escogidos

Para realizar la caracterización de los algoritmos DTSA, DJAYA y DTLBO, se ha utilizado 6 problemas TSP perteneciente a la librería TSPLIB [38], creada por Gerhard Reinelt en 1990 y que recoge numerosos problemas tanto de TSP como de algunas de sus variantes. Concretamente, contiene problemas de las siguientes clases [39]:

- TSP simétrico
- HSP (*Hamiltonian cycle problem*)
- TSP asimétrico

- SOP (*Sequential ordering problem*): Introduce restricciones de precedencia, es decir, que algunos vértices han de ser visitados antes que otros.
- CVRP (*Capacitated vehicle routing problem*): Consiste en determinar los recorridos de **k** vehículos de capacidad **c(k)** que, partiendo de un origen común, deben pasar por una serie de ciudades para recoger o distribuir mercancías según una demanda **d(i)** y volver de nuevo al lugar de origen de forma que la distancia recorrida sea mínima.

En este proyecto se trabajará únicamente con problemas TSP simétricos. Esta librería tiene problemas con tamaños que van desde 14 (el problema *burma14*) hasta 85900 (el problema *pla85900*). Se ha decidido trabajar con problemas de un tamaño medio bajo para caracterizar en condiciones no muy favorables los algoritmos paralelos. En particular, se han usado problemas del orden de las centenas, siendo el de menor tamaño kroA150 (con 150 vértices) y el de mayor, rat575 (con 575 vértices). En los artículos consultados [28][30][32][33] se trabajó con problemas de un tamaño similar, salvo por el caso de los de mayor tamaño, pues en estos artículos el mayor tamaño empleado fue 225. Se ha decidido ampliar el rango para ver cuál es el desempeño de los algoritmos, de los que ya se sabe que obtienen buenos resultados en los problemas estudiados en los artículos, en problemas de tamaño algo mayor y que no han sido probados en éstos.

Los problemas que se utilizarán en la caracterización de los algoritmos son:

- kroA150: Un problema de 150 ciudades creado por Krolak, Felts y Nelson. Su solución óptima es 26524.
- kroB200: Un problema de 200 ciudades creado por Krolak, Felts y Nelson. Su solución óptima es 29437.
- tsp225: Un problema de 225 ciudades creado por Reinelt, que es el creador de la librería. En la documentación de la librería [39] se muestra como solución óptima 3916, que es el valor con el que se ha trabajado en este proyecto, pero algunos estudios han logrado obtener una solución con coste 3859 [40].
- pr226: Un problema de 226 ciudades creado por Padberg y Rinaldi. Su solución óptima es 80369.
- pr439: Un problema de 439 ciudades creado por Padberg y Rinaldi. Su solución óptima es 107217.

- rat575: Un problema *rattled grid* (que es un problema semejante al *drilling*, pero tipo marcado, sin llegar a perforar por completo el material) de tamaño 575 creado por Pulleyblank. Su solución es 6773.

5.2.- ANÁLISIS DEL COMPORTAMIENTO DE LA OPTIMIZACIÓN

En esta sección se realizará la caracterización de los algoritmos DTSA, DJAYA y DTLBO con el fin de evaluar su capacidad para resolver problemas del tipo TSP.

5.2.1.- Caracterización de los algoritmos

Para realizar la caracterización, se utilizarán los 6 problemas TSP mencionados en la sección anterior, los cuales presentan un tamaño que va entre 150 y 575. Para cada medición se realizarán 30 ejecuciones y se obtendrán: la mejor solución obtenida, la peor, la media de todas las soluciones y la desviación estándar.

En cuanto al análisis de los parámetros, se ha decidido realizar el experimento con unos valores de **N** (tamaño de población) de 50, 100 y 150 y de **maxfes** (número máximo de evaluaciones de función) de 10^5, 10^6, 10^7 y 10^8.

Para el tamaño de la población se han escogido unos valores similares a los empleados en los artículos [28][30][32][33], además de que el mayor tamaño de población utilizado se corresponde con el menor tamaño de los 6 problemas estudiados. En el caso del algoritmo DTLBO, estos tamaños serán ajustados para que sean divisibles entre 4.

Por otra parte, el número de evaluaciones elegido es bastante mayor al que aparece en los artículos, pues en [28][30] se utiliza un valor fijo del orden de 10^5, que en este análisis es el menor valor estudiado. Esto es debido a que en los artículos originales se empleaba el

algoritmo de búsqueda local 2-opt [41] una vez alcanzado el número máximo de evaluaciones de función para obtener el óptimo local de la mejor solución generada por el algoritmo metaheurístico (DTSA, DJAYA o DTLBO). El uso del algoritmo 2-opt ha sido descartado en este trabajo debido al gran coste computacional que tiene y a que no supone una mejora muy apreciable en el resultado obtenido cuando se emplea un número máximo de evaluaciones de función lo suficientemente grande.

A continuación, se muestran las tablas del rendimiento de los algoritmos para la resolución de los problemas TSP y las conclusiones extraídas del análisis de éstas.

Tabla 1. Rendimiento de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema kroA150. Solución óptima del problema = 26524.

Algoritmo	Maxfes	N	Best	Worst	Mean	Std. Dev.
DTSA	10^5	50	29273.98	29757.37	29564.01	236.81
		100	30072.32	30596.35	30281.93	256.72
		150	29864.88	30304.82	30064.36	205.19
	10^6	50	27874.27	29168.17	28416.46	311.13
		100	27815.32	28876.60	28321.18	271.93
		150	27714.50	29048.93	28380.93	308.12
	10^7	50	27803.13	28889.99	28157.30	252.27
		100	27686.32	28923.75	28288.20	322.27
		150	27695.43	28739.36	28159.46	313.06
	10^8	50	27479.03	28922.61	28235.08	291.17
		100	27675.70	28737.94	28097.68	276.32
		150	27505.46	28766.71	28009.07	288.60
DJAYA	10^5	50	28832.39	29883.05	29324.09	288.54
		100	29085.64	30737.02	29749.76	507.58
		150	28720.14	30196.43	29619.14	432.82
	10^6	50	27693.45	29106.82	28324.02	369.34
		100	27607.33	29216.78	28312.65	377.18
		150	27544.23	29164.81	28324.99	351.83
	10^7	50	27559.96	28818.36	28069.53	308.63
		100	27267.79	28577.55	27919.39	295.98
		150	27493.74	28681.63	27946.82	248.75
	10^8	50	27642.11	28640.05	28075.65	258.68
		100	27545.01	28513.08	27957.16	255.14
		150	27133.50	28624.79	27813.88	387.77
DTLBO	10^5	52	30908.61	32072.35	31585.07	364.62
		100	31921.32	32503.86	32278.25	175.52
		152	31480.66	32804.96	32491.38	311.59
	10^6	52	28912.93	29964.63	29253.34	239.09
		100	28647.38	30627.39	29507.49	386.31
		152	28734.77	30050.44	29542.95	294.84
	10^7	52	28498.05	29617.21	28999.43	273.29
		100	28417.16	29564.84	28895.56	291.30
		152	28153.87	29418.69	28750.77	273.11
	10^8	52	28411.69	29748.04	29012.91	314.50
		100	28258.18	29324.90	28855.76	225.91
		152	28192.18	29148.33	28707.94	242.10

Tabla 2. Rendimiento de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema kroB200. Solución óptima del problema = 29437.

Algoritmo	Maxfes	N	Best	Worst	Mean	Std. Dev.
DTSA	10^5	50	33848.70	34300.87	34067.60	171.85
		100	33526.52	34250.97	33945.71	312.08
		150	34105.47	35141.90	34403.51	445.67
	10^6	50	30368.29	32058.98	31316.96	449.93
		100	30590.54	32266.60	31236.28	362.78
		150	30387.41	32212.14	31369.71	475.09
	10^7	50	30385.71	31872.12	30974.42	379.44
		100	30286.40	31820.11	30932.77	379.24
		150	30492.31	32326.53	30908.50	399.10
	10^8	50	30376.97	32109.38	30972.75	430.66
		100	30407.21	31744.61	30945.73	402.85
		150	30148.91	31558.89	30835.62	384.31
DJAYA	10^5	50	32778.64	34196.76	33248.74	447.34
		100	32288.13	34159.99	33191.08	584.15
		150	32341.24	34365.96	33429.93	499.67
	10^6	50	30580.01	31930.67	31161.23	352.13
		100	30428.07	32483.16	31269.52	449.23
		150	30698.88	32089.13	31258.51	357.88
	10^7	50	30169.10	31751.00	30790.57	354.13
		100	30091.26	31615.81	30767.12	372.50
		150	30172.71	31521.34	30750.65	371.52
	10^8	50	30254.69	31946.25	30894.52	373.56
		100	30084.46	31170.31	30590.93	297.09
		150	30098.25	31575.62	30730.97	367.59
DTLBO	10^5	52	34639.38	35787.64	35404.87	322.92
		100	34916.36	35967.29	35639.69	299.42
		152	35443.27	36189.48	35918.91	227.04
	10^6	52	31964.65	34058.36	32981.27	427.57
		100	32236.74	34072.93	33380.71	388.14
		152	32826.05	34240.07	33657.34	311.31
	10^7	52	31077.37	32053.25	31532.72	272.57
		100	30825.42	32111.99	31471.77	284.23
		152	31059.25	32016.92	31607.80	240.00
	10^8	52	30989.84	32384.44	31560.17	325.08
		100	30910.57	31915.84	31451.64	248.74
		152	31012.50	31940.68	31393.81	247.73

Tabla 3. Rendimiento de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema tsp225. Solución óptima del problema = 3916. Algunos estudios [40] han demostrado que existen soluciones mejores, habiéndose encontrado una de 3859.

Algoritmo	Maxfes	N	Best	Worst	Mean	Std. Dev.
DTSA	10^5	50	4336.20	4393.52	4357.28	24.32
		100	4398.03	4437.55	4424.79	17.54
		150	4333.68	4434.70	4400.42	47.21
	10^6	50	3969.65	4176.93	4048.66	49.52
		100	3963.19	4158.45	4050.20	42.43
		150	3949.41	4127.02	4052.35	41.35
	10^7	50	3921.32	4056.87	3987.00	37.00
		100	3940.95	4038.63	3986.84	28.06
		150	3907.82	4059.93	3984.57	35.91
	10^8	50	3931.11	4075.61	3981.41	31.14
		100	3914.64	4053.39	3993.07	28.06
		150	3906.19	4043.16	3984.27	33.51
DJAYA	10^5	50	4226.58	4341.11	4279.97	35.46
		100	4238.59	4373.43	4295.15	38.75
		150	4221.42	4457.68	4333.00	63.35
	10^6	50	3957.81	4158.60	4042.66	53.37
		100	3948.21	4128.75	4045.76	46.22
		150	3973.44	4112.03	4051.88	38.10
	10^7	50	3918.25	4055.37	3983.52	34.99
		100	3922.56	4064.83	3983.29	36.35
		150	3924.95	4072.30	3975.92	32.41
	10^8	50	3939.57	4064.40	3987.86	30.68
		100	3909.09	4044.99	3976.75	36.63
		150	3919.75	4023.48	3970.80	24.59
DTLBO	10^5	52	4591.60	4708.70	4647.05	28.43
		100	4610.32	4740.93	4693.03	32.85
		152	4660.15	4748.33	4715.73	21.90
	10^6	52	4161.85	4476.16	4270.24	58.23
		100	4241.93	4484.27	4373.09	62.48
		152	4300.96	4550.19	4434.38	61.05
	10^7	52	4045.28	4217.49	4129.84	45.73
		100	4023.98	4211.73	4104.82	45.57
		152	4033.76	4209.85	4127.16	45.09
	10^8	52	3995.69	4177.51	4069.87	41.58
		100	3982.63	4165.96	4069.67	57.50
		152	3990.04	4180.94	4055.18	50.25

Tabla 4. Rendimiento de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema pr226. Solución óptima del problema = 80369.

Algoritmo	Maxfes	N	Best	Worst	Mean	Std. Dev.
DTSA	10^5	50	86078.87	88390.37	87465.02	988.49
		100	86133.17	87388.87	86584.30	531.31
		150	86457.15	89844.08	88582.80	1440.84
	10^6	50	81316.56	83465.48	82335.75	611.02
		100	80857.05	83951.01	82328.17	908.60
		150	80940.30	84019.97	82386.08	878.95
	10^7	50	81029.02	84106.11	81802.78	725.22
		100	80886.66	82857.06	81462.72	518.29
		150	80415.76	84532.95	81412.24	805.10
	10^8	50	80980.24	83515.19	81820.45	837.59
		100	80686.63	83748.72	81539.73	658.42
		150	80749.96	83962.32	81621.75	844.29
DJAYA	10^5	50	83778.09	86600.45	85048.29	919.79
		100	83146.51	86151.20	85048.28	930.96
		150	84431.44	88640.62	86177.65	1185.25
	10^6	50	80940.30	84127.27	82332.82	803.13
		100	80849.20	83915.64	82482.40	722.27
		150	81389.69	84126.80	82575.24	816.45
	10^7	50	81163.86	83876.85	81664.35	672.62
		100	80737.86	83716.99	81498.49	624.50
		150	80787.41	83778.60	81949.83	777.61
	10^8	50	80737.86	84084.04	81609.79	853.79
		100	80729.79	83680.31	81493.00	708.11
		150	80607.03	83601.37	81365.47	810.48
DTLBO	10^5	52	84913.55	87688.70	86497.93	969.80
		100	87232.49	89624.33	88417.67	838.98
		152	87684.41	90229.34	88777.66	766.76
	10^6	52	81167.02	84211.04	82781.32	613.27
		100	81693.77	84603.64	82976.60	651.49
		152	81840.52	84908.76	83293.50	698.46
	10^7	52	80955.48	83464.33	82177.65	592.80
		100	81077.92	83852.75	82097.99	566.90
		152	80955.48	84707.87	82257.73	786.34
	10^8	52	80929.80	83358.99	81833.33	499.37
		100	80916.43	82932.88	81882.85	508.34
		152	80936.72	83433.28	82022.27	639.48

Tabla 5. Rendimiento de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema pr439. Solución óptima del problema = 107217.

Algoritmo	Maxfes	N	Best	Worst	Mean	Std. Dev.
DTSA	10^5	50	126151.32	129052.51	127931.36	809.17
		100	125270.56	129316.57	127371.48	1607.24
		150	127320.78	129126.77	127975.24	678.96
	10^6	50	114872.84	121097.63	118110.49	1790.05
		100	115455.00	121125.92	118382.08	1442.75
		150	114790.55	121385.19	118547.19	1683.70
	10^7	50	112377.90	117121.99	114940.13	1144.48
		100	112201.38	118130.56	114947.62	1506.83
		150	112385.05	117528.06	114820.43	1340.67
	10^8	50	112572.09	118212.78	115638.11	1461.46
		100	112474.81	118072.08	114837.73	1480.70
		150	112071.24	116768.93	114599.54	1192.41
DJAYA	10^5	50	120771.73	127568.29	124090.64	1861.63
		100	123425.23	128169.92	125106.99	1228.78
		150	121311.48	128006.77	125132.15	1925.91
	10^6	50	113605.23	119553.15	117084.09	1550.15
		100	114116.81	120479.35	117308.75	1593.45
		150	115380.22	120489.19	117515.25	1456.52
	10^7	50	112097.14	117572.48	114687.65	1354.72
		100	112411.64	117357.57	115001.05	1280.94
		150	111928.74	116796.00	114425.30	1116.28
	10^8	50	112616.32	118109.82	115647.84	1443.62
		100	111466.00	118765.13	114272.41	1509.30
		150	113006.02	116822.07	114367.13	959.49
DTLBO	10^5	52	123336.63	129484.33	126906.35	2095.51
		100	124727.00	130369.07	127488.88	1575.97
		152	125484.33	130523.01	127836.56	1643.77
	10^6	52	116291.31	121967.45	118379.96	1713.75
		100	117628.66	124196.74	120620.60	1805.24
		152	118811.77	124281.11	121242.35	1601.81
	10^7	52	113837.57	118230.58	115374.00	1105.60
		100	113732.37	117580.79	115015.86	960.72
		152	113824.97	117898.50	114912.36	851.59
	10^8	52	113262.28	116825.06	114804.28	891.07
		100	113329.35	116245.48	114317.62	727.65
		152	113237.87	116604.08	114395.48	861.95

Tabla 6. Rendimiento de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema rat575. Solución óptima del problema = 6773.

Algoritmo	Maxfes	N	Best	Worst	Mean	Std. Dev.
DTSA	10^5	50	8111.87	8254.35	8196.27	36.83
		100	8122.54	8301.79	8214.94	46.97
		150	8188.27	8288.45	8237.37	31.84
	10^6	50	7436.26	7732.10	7529.50	64.56
		100	7411.19	7672.64	7548.12	63.34
		150	7467.38	7657.75	7571.45	44.26
	10^7	50	7083.61	7240.30	7135.83	33.87
		100	7092.34	7238.80	7146.80	33.52
		150	7082.22	7257.35	7157.35	42.99
	10^8	50	7047.92	7177.58	7109.69	33.69
		100	7045.17	7151.80	7094.87	31.50
		150	7033.49	7170.13	7086.48	31.33
DJAYA	10^5	50	7962.12	8168.76	8060.27	56.26
		100	7995.26	8210.39	8108.10	58.81
		150	8005.80	8244.67	8126.64	54.10
	10^6	50	7301.27	7530.31	7398.53	47.53
		100	7314.84	7533.25	7423.79	54.45
		150	7324.93	7529.57	7425.89	51.41
	10^7	50	7076.15	7246.02	7148.25	38.74
		100	7053.15	7230.64	7151.06	43.27
		150	7059.54	7228.65	7157.13	46.52
	10^8	50	7018.83	7160.08	7085.71	37.51
		100	7000.31	7178.81	7087.50	38.98
		150	7015.23	7136.08	7081.67	33.87
DTLBO	10^5	52	8284.45	8385.31	8340.83	26.98
		100	8325.94	8400.76	8376.21	17.82
		152	8308.85	8416.96	8379.92	25.63
	10^6	52	7690.24	8108.84	7892.81	123.54
		100	7854.48	8220.09	8031.93	88.34
		152	7986.78	8252.20	8119.50	73.49
	10^7	52	7248.90	7439.28	7334.42	54.75
		100	7282.64	7470.25	7370.50	49.06
		152	7334.12	7513.98	7402.34	49.61
	10^8	52	7199.07	7342.44	7288.37	34.19
		100	7204.14	7352.39	7275.60	36.89
		152	7193.12	7335.35	7258.10	29.18

Observando los datos de las tablas 1-6, lo primero que se puede apreciar es que el tamaño de población usado (mostrado como **N** en las tablas) tiene un impacto mucho menor en los resultados que el número máximo de evaluaciones de función (mostrado como **maxfes**).

También se puede destacar que cuando el número de evaluaciones de función es 10^5, utilizar un tamaño de población más pequeño supone en la mayoría de los casos una pequeña mejora en los resultados, y que esta tendencia se va invirtiendo conforme se incrementa este número, habiendo en el caso de 10^8 una ligera ventaja para las ejecuciones con un tamaño de población de 150. Esto seguramente sea debido a que el número de iteraciones que realiza el algoritmo viene determinado por la relación entre **maxfes** y **N**, y por tanto, cuando **maxfes** es pequeño, emplear un **N** grande ocasiona que se realicen pocas iteraciones y el algoritmo no pueda mostrar todo su potencial, mientras que cuando **maxfes** es lo suficientemente grande, el efecto de **N** sobre el número de iteraciones es menos notable y destaca más la mejora que supone contar con una población formada por un número mayor de individuos.

En cuanto al parámetro **maxfes**, se puede ver que el valor 10^5 se queda bastante corto, habiendo una diferencia notable entre los resultados obtenidos con 10^5 y con 10^6, notándose esto sobre todo en los algoritmos DTSA y DTLBO. Esta diferencia se acrecienta conforme aumenta el tamaño del problema, lo cual es lógico si se tiene en cuenta que las operaciones básicas que emplean los algoritmos (*swap*, *shift*, *symmetry* y *crossover*) pierden efectividad conforme aumenta el tamaño.

Entre 10^6 y 10^7 las diferencias son mucho menores pero apreciables en todos los casos, siendo más notorias en los problemas de mayor tamaño; y finalmente, entre 10^7 y 10^8, los resultados no varían prácticamente, siendo inapreciables en los algoritmos DTSA y DJAYA salvo por el caso del problema rat575 (tabla 6) y suponiendo en el algoritmo DTLBO una pequeña mejora, de menor grado que entre 10^6 y 10^7, para los problemas tsp225, pr226, pr439 y rat575 (tablas 2-6).

Por tanto, exceptuando casos puntuales, el orden de magnitud más conveniente para el número máximo de evaluaciones suponiendo problemas TSP similares a los aquí estudiados y un tamaño de población similar sería 10^7. Posiblemente para afrontar

problemas de mayor tamaño sería conveniente incrementar el número de evaluaciones de función, especialmente en el caso del algoritmo DTLBO.

Analizando los resultados obtenidos por los tres algoritmos se puede inferir que DTSA y DJAYA obtienen unos resultados bastante similares, teniendo DJAYA unos valores ligeramente mejores, mientras que DTLBO obtiene peores resultados, salvo por el caso del problema pr439 (tabla 5), donde obtiene iguales resultados (incluso mejores si se utiliza un tamaño de población de 50) cuando se incrementa el número máximo de evaluaciones de función hasta 10^8. Sería interesante experimentar con problemas de un tamaño algo mayor (del orden de las 1000 ciudades) para ver si se igualan más los resultados de los tres algoritmos.

Finalmente, comparando las desviaciones estándar de los resultados, se puede ver que el algoritmo DTLBO obtiene por lo general unos resultados menos dispersos que DTSA y DJAYA, invirtiéndose esa tendencia en el problema tsp225 (tabla 3), que es uno de los problemas donde más diferencia de resultados hay entre DTLBO y los otros dos.

5.2.2.- Estudio de la convergencia

Ahora se va a analizar la velocidad de convergencia de los tres algoritmos, calculando el promedio de evaluaciones de función empleado por cada algoritmo para conseguir una solución con un error menor al 20%, 15%, 10% y 5% respecto de la solución óptima del problema en cuestión. Con el fin de que este valor sea representativo, cada vez que un algoritmo no consiga llegar a un determinado porcentaje de error, se utilizará el valor máximo de evaluaciones de función, lo que hará subir este promedio. También se obtendrá el porcentaje de veces (respecto de las 30 ejecuciones) que el algoritmo no consiga obtener una solución con un error inferior al 5%.

Estos experimentos se realizarán con los mismos valores de **N** (50, 100 y 150 que en el caso de DTLBO son 52, 100 y 152) y con un valor fijo de **maxfes** de 10^8, para dar más posibilidades a los algoritmos de poder converger y obtener soluciones con un error menor al 5%.

Tabla 7. Convergencia de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema kroA150.

Algoritmo	N	< 20%	< 15%	< 10%	< 5%	Error > 5%
DTSA	50	25060	69390	211400	96724418	96.67%
	100	30640	77240	208520	75847120	73.33%
	150	32670	93990	290490	74897213	73.33%
DJAYA	50	20116	50246	176256	70609805	70%
	100	20170	52116	212970	65551936	60%
	150	21555	61325	194555	39231393	33.33%
DTLBO	52	97771	242222	30624147	100000000	100%
	100	194340	437333	11302583	100000000	100%
	152	230003	509615	2317839	100000000	1005

Tabla 8. Convergencia de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema kroB200.

Algoritmo	N	< 20%	< 15%	< 10%	< 5%	Error > 5%
DTSA	50	38200	98860	258020	47692683	46.67%
	100	38420	101620	252580	38307930	36.67%
	150	41430	110370	262380	15053873	13.33%
DJAYA	50	28013	84206	284483	38669820	36.67%
	100	31096	90640	245973	18187733	13.33%
	150	32135	85110	219800	25805471	16.67%
DTLBO	52	103566	455378	2105192	100000000	100%
	100	163386	607940	3182726	100000000	100%
	152	209017	788003	3721235	100000000	10%

Tabla 9. Convergencia de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema tsp225.

Algoritmo	N	< 20%	< 15%	< 10%	< 5%	Error > 5%
DTSA	50	18380	67650	161590	543220	0%
	100	19300	69060	167220	556120	0%
	150	22800	74130	200640	663750	0%
DJAYA	50	15995	60490	146750	514326	0%
	100	18743	59433	145616	459656	0%
	150	18115	61150	148420	585175	0%
DTLBO	52	55970	207541	548226	38979059	33.33%
	100	81786	335066	825073	36408596	30%
	152	94475	397164	1048601	36203067	26.67%

Tabla 10. Convergencia de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema pr226.

Algoritmo	N	< 20%	< 15%	< 10%	< 5%	Error > 5%
DTSA	50	50	12840	37170	135560	0%
	100	100	19080	53980	275360	0%
	150	150	19230	68610	189450	0%
DJAYA	50	50	20501	54928	184995	0%
	100	100	17613	65366	368396	0%
	150	150	17965	69560	268920	0%
DTLBO	52	50	15586	53893	270322	0%
	100	100	20960	76380	374760	0%
	152	150	27243	97178	406479	0%

Tabla 11. Convergencia de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema pr439.

Algoritmo	N	< 20%	< 15%	< 10%	< 5%	Error > 5%
DTSA	50	57790	276140	5322121	96792998	96.67%
	100	54640	268860	5803510	93517760	93.33%
	150	59220	295650	1502463	97237211	96.67%
DJAYA	50	42700	213733	4910698	100000000	100%
	100	46760	219746	4235123	96730046	96.67%
	150	38210	215680	867060	100000000	100%
DTLBO	52	68994	277880	888818	100000000	100%
	100	76820	401853	1378526	100000000	100%
	152	99142	510335	1827308	100000000	100%

Tabla 12. Convergencia de los algoritmos DTSA, DJAYA y DTLBO en la resolución del problema rat575.

Algoritmo	N	< 20%	< 15%	< 10%	< 5%	Error > 5%
DTSA	50	161690	461580	1261670	51834041	36.67%
	100	176820	540320	1441480	50580360	40%
	150	168720	581460	1556550	49591335	30%
DJAYA	50	126170	393278	1059701	39969925	23.33%
	100	134046	433383	1127743	46829390	23.33%
	150	145065	442845	1196170	52135896	26.67%
DTLBO	52	627701	1923800	5829564	100000000	100%
	100	1023971	3042171	8273258	100000000	100%
	152	1263582	3671571	10404644	100000000	100%

Al analizar las tablas 7-12, se puede corroborar que los algoritmos DTSA y DJAYA tienen un mejor desempeño que el algoritmo DTLBO en la resolución de los problemas estudiados, pues este último sólo logra superar la barrera del 5% de error en los problemas tsp225 (tabla 3) y pr226 (tabla 4), además de que en el caso del tsp225 hay ejecuciones en las que no logra superar ese 5% mientras que DTSA y DJAYA lo consiguen en las 30.

Centrándose en los promedios de iteraciones, se puede ver que DJAYA es el que más rápido converge. Esto coincide con los datos de las tablas 1-6, que mostraban que cuando se usaba un número máximo de evaluaciones de función reducido, la diferencia de resultados entre DJAYA y DTSA era mayor. DTLBO, en cambio, requiere muchas más evaluaciones de función que los otros dos para todos los valores de error estudiados.

El caso del problema pr226 (tabla 4) es un caso especial, pues la generación inicial de soluciones genera en todos los casos un individuo con menos del 20% de error (por eso los promedios son iguales al tamaño de la población), sin ser necesaria ninguna iteración de los algoritmos. Esto es debido a que el individuo 0, que se genera con el *nearest neighbor tour* y no aleatoriamente, ya tiene un error menor que el 20%. Tal vez esta pueda ser una de las causas del desempeño marcadamente bueno de los tres algoritmos en este problema respecto de los otros, pues es el único caso en el que todas las ejecuciones siempre bajan del 5% de error en todos los casos.

Por otra parte, también destaca el problema pr439 (tabla 5) pero por causas opuestas, al ser el problema en el que peores resultados obtienen los algoritmos, llegando DTSA y DJAYA únicamente 1 ó 2 veces de cada 30 a obtener un error menor que el 5%. Además, los grandes valores de los promedios para bajar del 10% de error (DTSA en todos los casos y DJAYA con un tamaño de población de 50) indican que ha llegado a haber ejecuciones en las que los algoritmos no han llegado a obtener un error menor del 10%. Siendo la solución óptima 107217, se puede comprobar que, efectivamente, DTSA y DJAYA en esos casos llegaron a obtener al menos una solución con un error mayor del 10%, esto es, peor que 117938.

Finalmente cabe destacar que el rendimiento de los algoritmos depende más del problema que del tamaño de éste, pues por ejemplo kroA150 (tabla 7) presenta peores resultados que rat575 (tabla 12), siendo el problema más pequeño y más grande respectivamente.

5.3.- ANÁLISIS DEL RENDIMIENTO PARALELO

En esta sección se estudiará el rendimiento de los algoritmos paralelos diseñados, descritos en capítulo 4, es decir los tres diseños, los dos primeros basados en automatizadores y subpoblaciones respectivamente, y por último el híbrido, en la resolución de los 6 problemas TSP.

5.3.1.- Análisis de la eficiencia paralela

Para analizar el rendimiento paralelo de los tres diseños que han sido implementados, como se ha dicho, la versión basada en automatizadores, la versión basada en subpoblaciones y la versión híbrida, en cada uno de los tres algoritmos (DTSA, DJAYA y DTLBO), se medirá el tiempo medio de ejecución de cada uno de ellos y se estudiarán los datos de **speedup**, que es el cociente entre el tiempo de ejecución secuencial y el tiempo de ejecución paralelo. También se analizará la **eficiencia**, que es el cociente entre el speedup y el número de *threads* utilizados en la ejecución.

Todas estas mediciones se realizarán, de la misma manera que en el análisis anterior, con los 6 problemas TSP seleccionados (kroA150, kroB200, tsp225, pr226, pr439 y rat575) y con unos tamaños de población de 50, 100 y 150 (adaptando estos valores en el algoritmo DTLBO para que sean divisibles entre 4). En lo que respecta al número máximo de evaluaciones de función, éste quedará fijado en 10^7. Esta decisión se ha tomado debido a que este parámetro en principio no afecta al rendimiento paralelo, y se ha escogido 10^7 porque el análisis de rendimiento de los algoritmos determinó que era el valor más adecuado para utilizar a la hora de resolver problemas similares a los aquí utilizados, ya que incrementarlo a 10^8 no suponía, salvo excepciones puntuales, cambios apreciables.

El número de *threads* con los que se realizará el experimento en las versiones de automatizadores y de subpoblaciones será igual a 1, 2, 3, 4, 6, 8 y 10, exceptuando el algoritmo DTLBO basado en subpoblaciones, donde sólo se hará con 4 *threads*, debido a que este algoritmo lleva implícito el uso de 4 subpoblaciones. Se han elegido estos valores debido a que la máquina donde los algoritmos van a ser probados dispone de 12 *cores*

físicos sin *hyperthreading*, y no es recomendable utilizar el máximo número de *cores* disponibles cuando se busca la máxima eficiencia.

Para el algoritmo DTLBO con automatizadores se prescindirá del uso de 8 y 10 *threads* cuando el tamaño de población sea 52 y del uso de 10 *threads* cuando el tamaño sea 100. Esto se hace porque en el DTLBO el trabajo a repartir en los *for* paralelos va en función del tamaño de las subpoblaciones y no del de la población global, por lo que esas cantidades de *threads* serían excesivamente grandes para repartirse un trabajo de esas características.

Respecto de los algoritmos híbridos, se utilizarán valores que van desde 2 hasta 5 para cada una de las regiones interna y externa, no pudiendo ser nunca el número total de *threads* mayor que 12. Se usarán, por tanto, 2 *threads* externos con 2, 3, 4 y 5 internos, 3 *threads* externos con 3 y 4 internos, 4 *threads* externos con 2 y 3 internos y finalmente 5 *threads* externos con 2 internos. En esta ocasión si se probará a utilizar la totalidad de los *cores* físicos, ya que la finalidad de los algoritmos híbridos es poder funcionar eficientemente en un número elevado de *cores*.

Hay que remarcar que el objetivo de los algoritmos híbridos es poder incrementar el número total de *threads* con la posibilidad de determinar el mejor tamaño de las subpoblaciones, que viene determinado por el número de *threads* externos.

A continuación, se muestran las tablas del rendimiento paralelo de los algoritmos y las conclusiones extraídas de su análisis.

Tabla 13. Rendimiento paralelo de la versión con automatizadores en la resolución del problema kroA150.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	3.63021	1.00000	1.00000	50	1	4.16987	1.00000	1.00000	52	1	13.38078	1.00000	1.00000
	2	1.95226	1.85949	0.92974		2	2.68836	1.55109	0.77554		2	8.51332	1.57175	0.78587
	3	1.41397	2.56738	0.85579		3	2.29368	1.81798	0.60599		3	6.69083	1.99987	0.66662
	4	1.14162	3.17988	0.79497		4	2.02372	2.06050	0.51512		4	5.62874	2.37723	0.59431
	6	0.84719	4.28498	0.71416		6	2.11075	1.97554	0.32926		6	4.40942	3.03459	0.50576
	8	0.73916	4.91129	0.61391		8	1.80057	2.31587	0.28948		8	*	*	*
	10	0.64298	5.64594	0.56459		10	1.98466	2.10105	0.21010		10	*	*	*
100	1	3.64535	1.00000	1.00000	100	1	4.15925	1.00000	1.00000	100	1	15.80850	1.00000	1.00000
	2	1.89018	1.92857	0.96429		2	2.41641	1.72125	0.86063		2	8.97232	1.76192	0.88096
	3	1.37174	2.65747	0.88582		3	1.81994	2.28537	0.76179		3	6.40255	2.46909	0.82303
	4	1.04222	3.49769	0.87442		4	1.57550	2.63995	0.65999		4	5.73713	2.75547	0.68887
	6	0.76391	4.77198	0.79533		6	1.39394	2.98380	0.49730		6	4.21996	3.74613	0.62435
	8	0.62889	5.79652	0.72456		8	1.60586	2.59005	0.32376		8	3.48320	4.53850	0.56731
	10	0.52677	6.92014	0.69201		10	1.55803	2.66955	0.26695		10	*	*	*
150	1	3.66233	1.00000	1.00000	150	1	4.19145	1.00000	1.00000	152	1	17.92621	1.00000	1.00000
	2	1.92651	1.90102	0.95051		2	2.37833	1.76235	0.88117		2	9.43580	1.89981	0.94990
	3	1.33735	2.73850	0.91283		3	1.83474	2.28449	0.76150		3	6.89107	2.60137	0.86712
	4	1.05448	3.47313	0.86828		4	1.46675	2.85765	0.71441		4	5.83029	3.07467	0.76867
	6	0.73157	5.00613	0.83435		6	1.24519	3.36611	0.56102		6	4.51161	3.97335	0.66223
	8	0.59378	6.16784	0.77098		8	1.11612	3.75539	0.46942		8	3.72382	4.81393	0.60174
	10	0.48967	7.47926	0.74793		10	1.10504	3.79305	0.37930		10	3.25153	5.51316	0.55132

Tabla 14. Rendimiento paralelo de la versión basada en subpoblaciones en la resolución del problema kroA150.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	3.63021	1.00000	1.00000	50	1	4.16987	1.00000	1.00000	52	1	13.38078	1.00000	1.00000
	2	1.88005	1.93091	0.96546		2	2.34749	1.77631	0.88815		2	*	*	*
	3	1.28139	2.83301	0.94434		3	1.55869	2.67525	0.89175		3	*	*	*
	4	0.98386	3.68976	0.92244		4	1.21951	3.41931	0.85483		4	3.83601	3.48820	0.87205
	6	0.67973	5.34066	0.89011		6	0.92778	4.49448	0.74908		6	*	*	*
	8	0.52770	6.87929	0.85991		8	1.03530	4.02771	0.50346		8	*	*	*
	10	0.46571	7.79496	0.77950		10	0.81560	5.11266	0.51127		10	*	*	*
100	1	3.64535	1.00000	1.00000	100	1	4.15925	1.00000	1.00000	100	1	15.80850	1.00000	1.00000
	2	1.89484	1.92384	0.96192		2	2.28683	1.81878	0.90939		2	*	*	*
	3	1.29366	2.81785	0.93928		3	1.50405	2.76536	0.92179		3	*	*	*
	4	0.98466	3.70216	0.92554		4	1.16528	3.56930	0.89232		4	4.43877	3.56146	0.89037
	6	0.65844	5.53634	0.92272		6	0.82063	5.06835	0.84472		6	*	*	*
	8	0.50729	7.18587	0.89823		8	0.70038	5.93860	0.74233		8	*	*	*
	10	0.42874	8.50239	0.85024		10	0.64386	6.45988	0.64599		10	*	*	*
150	1	3.66233	1.00000	1.00000	150	1	4.19145	1.00000	1.00000	152	1	17.92621	1.00000	1.00000
	2	1.87642	1.95177	0.97588		2	2.19827	1.90671	0.95335		2	*	*	*
	3	1.31171	2.79202	0.93067		3	1.51928	2.75885	0.91962		3	*	*	*
	4	0.96916	3.77889	0.94472		4	1.14310	3.66673	0.91668		4	4.82509	3.71520	0.92880
	6	0.68064	5.38074	0.89679		6	0.81694	5.13069	0.85511		6	*	*	*
	8	0.49794	7.35498	0.91937		8	0.62285	6.72945	0.84118		8	*	*	*
	10	0.42011	8.71766	0.87177		10	0.55666	7.52969	0.75297		10	*	*	*

Tabla 15. Rendimiento paralelo de la versión híbrida en la resolución del problema kroA150.

DTSA						DJAYA						DTLBO					
N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia
50	2	2	1.02304	3.54844	0.88711	50	2	2	1.25884	3.31247	0.82812	50	2	2	*	*	*
		3	0.76670	4.73487	0.78914			3	1.03346	4.03487	0.67248			3	*	*	*
		4	0.63121	5.75118	0.71890			4	1.07890	3.86492	0.48312			4	*	*	*
		5	0.52007	6.98021	0.69802			5	1.03027	4.04736	0.40474			5	*	*	*
	3	2	0.72323	5.01946	0.83658		3	2	1.07997	3.86109	0.64351		3	2	*	*	*
		3	0.56884	6.38178	0.70909			3	1.01658	4.10185	0.45576			3	*	*	*
		4	0.46638	7.78388	0.64866			4	0.95784	4.35340	0.36278			4	*	*	*
	4	2	0.56386	6.43814	0.80477		4	2	0.82738	5.03986	0.62998		4	2	2.59363	5.15910	0.64489
		3	0.44977	8.07125	0.67260			3	0.90382	4.61359	0.38447			3	2.34924	5.69578	0.47465
	5	2	0.48350	7.50817	0.75082		5	2	0.79042	5.27553	0.52755		5	2	*	*	*
100	2	2	1.00879	3.61360	0.90340	100	2	2	1.15956	3.58693	0.89673	100	2	2	*	*	*
		3	0.70121	5.19869	0.86645			3	0.95352	4.36198	0.72700			3	*	*	*
		4	0.56034	6.50564	0.81320			4	0.79827	5.21030	0.65129			4	*	*	*
		5	0.44992	8.10215	0.81021			5	0.75395	5.51659	0.55166			5	*	*	*
	3	2	0.71443	5.10246	0.85041		3	2	0.94183	4.41611	0.73602		3	2	*	*	*
		3	0.52949	6.88466	0.76496			3	0.76371	5.44611	0.60512			3	*	*	*
		4	0.42328	8.61219	0.71768			4	0.74410	5.58967	0.46581			4	*	*	*
	4	2	0.54536	6.68434	0.83554		4	2	0.68409	6.07996	0.76000		4	2	2.65013	5.96519	0.74565
		3	0.41960	8.68764	0.72397			3	0.67666	6.14676	0.51223			3	2.33348	6.77465	0.56455
	5	2	0.44520	8.18817	0.81882		5	2	0.64945	6.40427	0.64043		5	2	*	*	*
150	2	2	0.97781	3.74546	0.93637	150	2	2	1.15557	3.62718	0.90680	150	2	2	*	*	*
		3	0.68413	5.35325	0.89221			3	0.87838	4.77181	0.79530			3	*	*	*
		4	0.54280	6.74710	0.84339			4	0.71866	5.83235	0.72904			4	*	*	*
		5	0.43294	8.45925	0.84593			5	0.66753	6.27904	0.62790			5	*	*	*
	3	2	0.68860	5.31853	0.88642		3	2	0.88123	4.75637	0.79273		3	2	*	*	*
		3	0.48790	7.50626	0.83403			3	0.69628	6.01978	0.66886			3	*	*	*
		4	0.40029	9.14931	0.76244			4	0.61267	6.84131	0.57011			4	*	*	*
	4	2	0.53493	6.84634	0.85579		4	2	0.65922	6.35822	0.79478		4	2	2.85530	6.27822	0.78478
		3	0.37377	9.79828	0.81652			3	0.56492	7.41957	0.61830			3	2.24257	7.99359	0.66613
	5	2	0.42705	8.57595	0.85759		5	2	0.59339	7.06352	0.70635		5	2	*	*	*

Tabla 16. Rendimiento paralelo de la versión con automatizadores en la resolución del problema kroB200.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	4.71281	1.00000	1.00000	50	1	5.26050	1.00000	1.00000	52	1	20.87816	1.00000	1.00000
	2	2.54954	1.84849	0.92425		2	3.23823	1.62450	0.81225		2	11.40015	1.83139	0.91570
	3	1.79154	2.63060	0.87687		3	2.60949	2.01591	0.67197		3	9.59019	2.17703	0.72568
	4	1.41272	3.33598	0.83400		4	2.24102	2.34737	0.58684		4	9.59283	2.17643	0.54411
	6	1.06912	4.40814	0.73469		6	2.20953	2.38082	0.39680		6	7.44319	2.80500	0.46750
	8	0.92739	5.08182	0.63523		8	2.25892	2.32877	0.29110		8	*	*	*
	10	0.74623	6.31549	0.63155		10	1.98409	2.65135	0.26513		10	*	*	*
100	1	4.72828	1.00000	1.00000	100	1	5.22754	1.00000	1.00000	100	1	25.09250	1.00000	1.00000
	2	2.54106	1.86075	0.93038		2	3.00969	1.73690	0.86845		2	15.43604	1.62558	0.81279
	3	1.77997	2.65638	0.88546		3	2.24473	2.32880	0.77627		3	10.85757	2.31106	0.77035
	4	1.34128	3.52519	0.88130		4	1.85157	2.82330	0.70582		4	9.28926	2.70124	0.67531
	6	0.96776	4.88582	0.81430		6	1.59858	3.27012	0.54502		6	7.25754	3.45744	0.57624
	8	0.75827	6.23566	0.77946		8	1.52229	3.43399	0.42925		8	5.99504	4.18554	0.52319
	10	0.65660	7.20120	0.72012		10	1.52275	3.43295	0.34330		10	*	*	*
150	1	4.75760	1.00000	1.00000	150	1	5.27725	1.00000	1.00000	152	1	27.86434	1.00000	1.00000
	2	2.51367	1.89269	0.94635		2	3.05233	1.72893	0.86446		2	15.77397	1.76648	0.88324
	3	1.73787	2.73760	0.91253		3	2.17333	2.42819	0.80940		3	11.73841	2.37377	0.79126
	4	1.34194	3.54532	0.88633		4	1.75259	3.01111	0.75278		4	9.63025	2.89342	0.72335
	6	0.94183	5.05142	0.84190		6	1.39122	3.79326	0.63221		6	7.09789	3.92572	0.65429
	8	0.74329	6.40075	0.80009		8	1.24270	4.24660	0.53082		8	5.93266	4.69677	0.58710
	10	0.61596	7.72389	0.77239		10	1.24691	4.23226	0.42323		10	5.29197	5.26540	0.52654

Tabla 17. Rendimiento paralelo de la versión basada en subpoblaciones en la resolución del problema kroB200.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	4.71281	1.00000	1.00000	50	1	5.26050	1.00000	1.00000	52	1	20.87816	1.00000	1.00000
	2	2.76353	1.70536	0.85268		2	2.84518	1.84891	0.92446		2	*	*	*
	3	1.65225	2.85236	0.95079		3	1.86821	2.81580	0.93860		3	*	*	*
	4	1.25627	3.75143	0.93786		4	1.50632	3.49228	0.87307		4	5.86772	3.55814	0.88953
	6	0.86881	5.42445	0.90408		6	1.10716	4.75133	0.79189		6	*	*	*
	8	0.67710	6.96027	0.87003		8	1.02432	5.13560	0.64195		8	*	*	*
	10	0.56590	8.32804	0.83280		10	0.93036	5.65426	0.56543		10	*	*	*
100	1	4.72828	1.00000	1.00000	100	1	5.22754	1.00000	1.00000	100	1	25.09250	1.00000	1.00000
	2	2.46660	1.91693	0.95846		2	2.66735	1.95982	0.97991		2	*	*	*
	3	1.67996	2.81452	0.93817		3	1.88039	2.78002	0.92667		3	*	*	*
	4	1.27845	3.69846	0.92462		4	1.47076	3.55430	0.88858		4	6.99306	3.58820	0.89705
	6	0.85114	5.55526	0.92588		6	1.02600	5.09505	0.84917		6	*	*	*
	8	0.64892	7.28637	0.91080		8	0.85862	6.08831	0.76104		8	*	*	*
	10	0.59285	7.97552	0.79755		10	0.70081	7.45923	0.74592		10	*	*	*
150	1	4.75760	1.00000	1.00000	150	1	5.27725	1.00000	1.00000	152	1	27.86434	1.00000	1.00000
	2	2.65927	1.78906	0.89453		2	2.84367	1.85579	0.92790		2	*	*	*
	3	1.71113	2.78038	0.92679		3	1.88083	2.80582	0.93527		3	*	*	*
	4	1.26758	3.75331	0.93833		4	1.42077	3.71436	0.92859		4	7.97215	3.49521	0.87380
	6	0.87782	5.41976	0.90329		6	1.01820	5.18292	0.86382		6	*	*	*
	8	0.63996	7.43421	0.92928		8	0.77411	6.81717	0.85215		8	*	*	*
	10	0.53811	8.84126	0.88413		10	0.70575	7.47752	0.74775		10	*	*	*

Tabla 18. Rendimiento paralelo de la versión híbrida en la resolución del problema kroB200.

DTSA						DJAYA						DTLBO				
N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia
50	2	2	1.37043	3.43894	0.85973	50	2	2	1.65593	3.17677	0.79419	50	2	2	*	*	*
		3	1.00952	4.66839	0.77807			3	1.36282	3.86002	0.64334			3	*	*	*
		4	0.81456	5.78571	0.72321			4	1.27965	4.11089	0.51386			4	*	*	*
		5	0.62897	7.49289	0.74929			5	1.14380	4.59916	0.45992			5	*	*	*
	3	2	0.92895	5.07324	0.84554		3	2	1.31998	3.98528	0.66421		3	2	*	*	*
		3	0.71984	6.54705	0.72745			3	1.23042	4.27538	0.47504			3	*	*	*
		4	0.54453	8.65481	0.72123			4	1.20120	4.37938	0.36495			4	*	*	*
	4	2	0.67148	7.01857	0.87732		4	2	0.94846	5.54637	0.69330		4	2	4.37486	4.77230	0.59654
		3	0.53111	8.87355	0.73946			3	1.01508	5.18235	0.43186			3	3.71615	5.61823	0.46819
	5	2	0.59382	7.93650	0.79365		5	2	0.97139	5.41546	0.54155		5	2	*	*	*
100	2	2	1.27421	3.71076	0.92769	100	2	2	1.45235	3.59936	0.89984	100	2	2	*	*	*
		3	0.91708	5.15579	0.85930			3	1.10380	4.73596	0.78933			3	*	*	*
		4	0.70426	6.71379	0.83922			4	0.98684	5.29726	0.66216			4	*	*	*
		5	0.57376	8.24088	0.82409			5	0.94047	5.55845	0.55584			5	*	*	*
	3	2	0.92134	5.13196	0.85533		3	2	1.14622	4.56066	0.76011		3	2	*	*	*
		3	0.62080	7.61644	0.84627			3	0.94570	5.52768	0.61419			3	*	*	*
		4	0.52322	9.03682	0.75307			4	0.83886	6.23172	0.51931			4	*	*	*
	4	2	0.72092	6.55868	0.81984		4	2	0.86265	6.05986	0.75748		4	2	4.39895	5.70420	0.71302
		3	0.51136	9.24644	0.77054			3	0.77774	6.72145	0.56012			3	3.41968	7.33768	0.61147
	5	2	0.56068	8.43315	0.84331		5	2	0.77225	6.76921	0.67692		5	2	*	*	*
150	2	2	1.30468	3.64658	0.91164	150	2	2	1.44682	3.64748	0.91187	150	2	2	*	*	*
		3	0.87949	5.40951	0.90158			3	1.12899	4.67432	0.77905			3	*	*	*
		4	0.69208	6.87436	0.85929			4	0.86498	6.10100	0.76263			4	*	*	*
		5	0.56487	8.42242	0.84224			5	0.78461	6.72600	0.67260			5	*	*	*
	3	2	0.88276	5.38944	0.89824		3	2	1.09286	4.82886	0.80481		3	2	*	*	*
		3	0.61980	7.67607	0.85290			3	0.82755	6.37694	0.70855			3	*	*	*
		4	0.48771	9.75499	0.81292			4	0.72172	7.31209	0.60934			4	*	*	*
	4	2	0.68372	6.95839	0.86980		4	2	0.86033	6.13397	0.76675		4	2	4.59955	6.05806	0.75726
		3	0.49293	9.65177	0.80431			3	0.66725	7.90901	0.65908			3	3.47939	8.00841	0.66737
	5	2	0.54959	8.65660	0.86566		5	2	0.71071	7.42538	0.74254		5	2	*	*	*

Tabla 19. Rendimiento paralelo de la versión con automatizadores en la resolución del problema tsp225.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	5.21894	1.00000	1.00000	50	1	5.75684	1.00000	1.00000	52	1	40.67535	1.00000	1.00000
	2	2.87921	1.81263	0.90631		2	3.64345	1.58005	0.79003		2	25.20484	1.61379	0.80690
	3	1.97828	2.63812	0.87937		3	2.73000	2.10873	0.70291		3	20.50003	1.98416	0.66139
	4	1.56973	3.32473	0.83118		4	2.39725	2.40144	0.60036		4	17.95717	2.26513	0.56628
	6	1.17292	4.44952	0.74159		6	2.20147	2.61500	0.43583		6	13.42041	3.03086	0.50514
	8	0.98375	5.30513	0.66314		8	2.39390	2.40480	0.30060		8	*	*	*
	10	0.80742	6.46374	0.64637		10	2.38288	2.41592	0.24159		10	*	*	*
100	1	5.23798	1.00000	1.00000	100	1	5.72931	1.00000	1.00000	100	1	38.81623	1.00000	1.00000
	2	2.80358	1.86832	0.93416		2	3.36753	1.70134	0.85067		2	24.46401	1.58667	0.79333
	3	1.96213	2.66954	0.88985		3	2.38845	2.39876	0.79959		3	17.61968	2.20300	0.73433
	4	1.46821	3.56760	0.89190		4	1.97405	2.90232	0.72558		4	15.15143	2.56189	0.64047
	6	1.05699	4.95557	0.82593		6	1.65832	3.45490	0.57582		6	11.49509	3.37677	0.56279
	8	0.84452	6.20229	0.77529		8	1.75314	3.26804	0.40850		8	9.57380	4.05442	0.50680
	10	0.68561	7.63989	0.76399		10	1.57129	3.64624	0.36462		10	*	*	*
150	1	5.25851	1.00000	1.00000	150	1	5.77193	1.00000	1.00000	152	1	43.66335	1.00000	1.00000
	2	2.77133	1.89747	0.94873		2	3.21717	1.79410	0.89705		2	22.75442	1.91889	0.95945
	3	1.92985	2.72483	0.90828		3	2.35200	2.45406	0.81802		3	17.44077	2.50352	0.83451
	4	1.47357	3.56855	0.89214		4	1.94262	2.97121	0.74280		4	14.11977	3.09236	0.77309
	6	1.03597	5.07591	0.84599		6	1.47169	3.92197	0.65366		6	10.48005	4.16633	0.69439
	8	0.81436	6.45724	0.80716		8	1.35843	4.24899	0.53112		8	8.48075	5.14853	0.64357
	10	0.66753	7.87757	0.78776		10	1.26119	4.57657	0.45766		10	7.59612	5.74811	0.57481

Tabla 20. Rendimiento paralelo de la versión basada en subpoblaciones en la resolución del problema tsp225.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	5.21894	1.00000	1.00000	50	1	5.75684	1.00000	1.00000	52	1	40.67535	1.00000	1.00000
	2	2.68772	1.94177	0.97088		2	3.03710	1.89551	0.94775		2	*	*	*
	3	1.81727	2.87186	0.95729		3	2.10596	2.73360	0.91120		3	*	*	*
	4	1.35800	3.84311	0.96078		4	1.61510	3.56438	0.89110		4	12.11071	3.35863	0.83966
	6	0.95860	5.44432	0.90739		6	1.27394	4.51894	0.75316		6	*	*	*
	8	0.74066	7.04635	0.88079		8	1.03167	5.58013	0.69752		8	*	*	*
	10	0.63004	8.28355	0.82836		10	0.90241	6.37942	0.63794		10	*	*	*
100	1	5.23798	1.00000	1.00000	100	1	5.72931	1.00000	1.00000	100	1	38.81623	1.00000	1.00000
	2	2.81797	1.85878	0.92939		2	3.03717	1.88640	0.94320		2	*	*	*
	3	1.86365	2.81061	0.93687		3	2.09046	2.74069	0.91356		3	*	*	*
	4	1.40709	3.72257	0.93064		4	1.61520	3.54713	0.88678		4	11.44061	3.39285	0.84821
	6	0.93809	5.58366	0.93061		6	1.10459	5.18681	0.86447		6	*	*	*
	8	0.71488	7.32712	0.91589		8	0.88849	6.44836	0.80605		8	*	*	*
	10	0.59899	8.74474	0.87447		10	0.80834	7.08779	0.70878		10	*	*	*
150	1	5.25851	1.00000	1.00000	150	1	5.77193	1.00000	1.00000	152	1	43.66335	1.00000	1.00000
	2	2.72666	1.92855	0.96428		2	2.99657	1.92618	0.96309		2	*	*	*
	3	1.90143	2.76555	0.92185		3	2.05612	2.80720	0.93573		3	*	*	*
	4	1.39999	3.75611	0.93903		4	1.56885	3.67908	0.91977		4	11.24326	3.88351	0.97088
	6	0.96690	5.43851	0.90642		6	1.11267	5.18747	0.86458		6	*	*	*
	8	0.70239	7.48661	0.93583		8	0.83741	6.89262	0.86158		8	*	*	*
	10	0.59406	8.85178	0.88518		10	0.73287	7.87584	0.78758		10	*	*	*

Tabla 21. Rendimiento paralelo de la versión híbrida en la resolución del problema tsp225.

DTSA						DJAYA						DTLBO					
N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia
50	2	2	1.49708	3.48607	0.87152	50	2	2	1.76909	3.25412	0.81353	50	2	2	*	*	*
		3	1.08581	4.80650	0.80108			3	1.41976	4.05479	0.67580			3	*	*	*
		4	0.88023	5.92903	0.74113			4	1.37478	4.18745	0.52343			4	*	*	*
		5	0.68025	7.67213	0.76721			5	1.30069	4.42598	0.44260			5	*	*	*
	3	2	1.00448	5.19564	0.86594		3	2	1.38653	4.15199	0.69200		3	2	*	*	*
		3	0.78375	6.65890	0.73988			3	1.27692	4.50839	0.50093			3	*	*	*
		4	0.56223	9.28256	0.77355			4	1.23630	4.65649	0.38804			4	*	*	*
	4	2	0.76265	6.84319	0.85540		4	2	1.00752	5.71386	0.71423		4	2	7.33751	5.54348	0.69293
		3	0.56336	9.26394	0.77200			3	0.99934	5.76063	0.48005			3	6.32362	6.43228	0.53602
	5	2	0.68864	7.57867	0.75787		5	2	1.02810	5.59948	0.55995		5	2	*	*	*
100	2	2	1.44749	3.61866	0.90466	100	2	2	1.70909	3.35225	0.83806	100	2	2	*	*	*
		3	0.98797	5.30179	0.88363			3	1.29418	4.42698	0.73783			3	*	*	*
		4	0.79369	6.59956	0.82494			4	1.09639	5.22560	0.65320			4	*	*	*
		5	0.63605	8.23521	0.82352			5	0.97010	5.90591	0.59059			5	*	*	*
	3	2	0.99752	5.25098	0.87516		3	2	1.27045	4.50966	0.75161		3	2	*	*	*
		3	0.67651	7.74262	0.86029			3	0.96880	5.91385	0.65709			3	*	*	*
		4	0.56752	9.22960	0.76913			4	0.95712	5.98597	0.49883			4	*	*	*
	4	2	0.78588	6.66510	0.83314		4	2	0.91846	6.23797	0.77975		4	2	6.71422	5.78120	0.72265
		3	0.56152	9.32825	0.77735			3	0.87105	6.57747	0.54812			3	5.21053	7.44958	0.62080
	5	2	0.64196	8.15936	0.81594		5	2	0.84507	6.77966	0.67797		5	2	*	*	*
150	2	2	1.42963	3.67824	0.91956	150	2	2	1.65470	3.48820	0.87205	150	2	2	*	*	*
		3	0.97683	5.38325	0.89721			3	1.22400	4.71562	0.78594			3	*	*	*
		4	0.79230	6.63701	0.82963			4	1.02461	5.63330	0.70416			4	*	*	*
		5	0.61299	8.57852	0.85785			5	0.88632	6.51228	0.65123			5	*	*	*
	3	2	0.97774	5.37824	0.89637		3	2	1.19568	4.82731	0.80455		3	2	*	*	*
		3	0.69466	7.56992	0.84110			3	0.90905	6.34938	0.70549			3	*	*	*
		4	0.58442	8.99780	0.74982			4	0.78547	7.34838	0.61236			4	*	*	*
	4	2	0.73724	7.13274	0.89159		4	2	0.91073	6.33767	0.79221		4	2	6.38572	6.83766	0.85471
		3	0.56317	9.33741	0.77812			3	0.74360	7.76218	0.64685			3	6.10134	7.15636	0.59636
	5	2	0.60313	8.71876	0.87188		5	2	0.79724	7.23987	0.72399		5	2	*	*	*

Tabla 22. Rendimiento paralelo de la versión con automatizadores en la resolución del problema pr226.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	5.22018	1.00000	1.00000	50	1	5.76750	1.00000	1.00000	52	1	34.37192	1.00000	1.00000
	2	2.92450	1.78498	0.89249		2	3.65233	1.57913	0.78956		2	19.40979	1.77085	0.88543
	3	2.03143	2.56971	0.85657		3	2.74731	2.09933	0.69978		3	17.55861	1.95755	0.65252
	4	1.58030	3.30328	0.82582		4	2.37202	2.43147	0.60787		4	14.22740	2.41590	0.60397
	6	1.18402	4.40888	0.73481		6	2.25796	2.55430	0.42572		6	11.63384	2.95448	0.49241
	8	0.98342	5.30818	0.66352		8	2.39952	2.40361	0.30045		8	*	*	*
	10	0.86469	6.03702	0.60370		10	2.39656	2.40657	0.24066		10	*	*	*
100	1	5.24818	1.00000	1.00000	100	1	5.74545	1.00000	1.00000	100	1	36.69903	1.00000	1.00000
	2	2.84686	1.84350	0.92175		2	3.39113	1.69426	0.84713		2	20.11181	1.82475	0.91238
	3	1.96411	2.67204	0.89068		3	2.40525	2.38871	0.79624		3	16.37533	2.24112	0.74704
	4	1.48371	3.53721	0.88430		4	1.98912	2.88845	0.72211		4	12.08909	3.03571	0.75893
	6	1.05505	4.97435	0.82906		6	1.70583	3.36813	0.56135		6	9.69904	3.78378	0.63063
	8	0.84812	6.18805	0.77351		8	1.77171	3.24289	0.40536		8	7.46119	4.91865	0.61483
	10	0.67960	7.72247	0.77225		10	1.52749	3.76136	0.37614		10	*	*	*
150	1	5.27893	1.00000	1.00000	150	1	5.83630	1.00000	1.00000	152	1	36.95447	1.00000	1.00000
	2	3.01636	1.75010	0.87505		2	3.28885	1.77457	0.88729		2	20.47812	1.80458	0.90229
	3	1.90779	2.76705	0.92235		3	2.34490	2.48893	0.82964		3	15.24488	2.42406	0.80802
	4	1.47234	3.58540	0.89635		4	1.94308	3.00364	0.75091		4	12.95329	2.85290	0.71323
	6	1.03860	5.08276	0.84713		6	1.48642	3.92642	0.65440		6	11.23281	3.28987	0.54831
	8	0.81367	6.48778	0.81097		8	1.34947	4.32488	0.54061		8	9.15114	4.03824	0.50478
	10	0.66490	7.93942	0.79394		10	1.32173	4.41566	0.44157		10	7.43459	4.97061	0.49706

Tabla 23. Rendimiento paralelo de la versión basada en subpoblaciones en la resolución del problema pr226.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	5.22018	1.00000	1.00000	50	1	5.76750	1.00000	1.00000	52	1	34.37192	1.00000	1.00000
	2	2.75910	1.89198	0.94599		2	3.03982	1.89732	0.94866		2	*	*	*
	3	1.78258	2.92844	0.97615		3	2.08434	2.76706	0.92235		3	*	*	*
	4	1.35313	3.85787	0.96447		4	1.64122	3.51416	0.87854		4	10.41478	3.30030	0.82508
	6	0.96105	5.43176	0.90529		6	1.24919	4.61699	0.76950		6	*	*	*
	8	0.73672	7.08575	0.88572		8	1.04282	5.53065	0.69133		8	*	*	*
	10	0.62548	8.34594	0.83459		10	0.90382	6.38122	0.63812		10	*	*	*
100	1	5.24818	1.00000	1.00000	100	1	5.74545	1.00000	1.00000	100	1	36.69903	1.00000	1.00000
	2	2.68588	1.95399	0.97699		2	2.97731	1.92975	0.96487		2	*	*	*
	3	1.83275	2.86355	0.95452		3	2.05227	2.79956	0.93319		3	*	*	*
	4	1.41749	3.70245	0.92561		4	1.58566	3.62338	0.90584		4	10.78269	3.40351	0.85088
	6	0.93593	5.60743	0.93457		6	1.09956	5.22522	0.87087		6	*	*	*
	8	0.71216	7.36937	0.92117		8	0.86756	6.62258	0.82782		8	*	*	*
	10	0.60340	8.69774	0.86977		10	0.77310	7.43176	0.74318		10	*	*	*
150	1	5.27893	1.00000	1.00000	150	1	5.83630	1.00000	1.00000	152	1	36.95447	1.00000	1.00000
	2	2.79696	1.88738	0.94369		2	2.98020	1.95836	0.97918		2	*	*	*
	3	1.87608	2.81381	0.93794		3	2.08297	2.80191	0.93397		3	*	*	*
	4	1.41163	3.73961	0.93490		4	1.56946	3.71868	0.92967		4	10.74564	3.43902	0.85975
	6	0.96841	5.45113	0.90852		6	1.11158	5.25045	0.87508		6	*	*	*
	8	0.70728	7.46374	0.93297		8	0.83194	7.01534	0.87692		8	*	*	*
	10	0.59287	8.90400	0.89040		10	0.72446	8.05605	0.80561		10	*	*	*

Tabla 24. Rendimiento paralelo de la versión híbrida en la resolución del problema pr226.

DTSA						DJAYA						DTLBO				
N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia
50	2	2	1.46665	3.55925	0.88981	50	2	2	1.74695	3.30147	0.82537	50	2	2	*	*	*
		3	1.06560	4.89880	0.81647			3	1.43352	4.02331	0.67055			3	*	*	*
		4	0.84171	6.20185	0.77523			4	1.39354	4.13875	0.51734			4	*	*	*
		5	0.70410	7.41399	0.74140			5	1.28822	4.47709	0.44771			5	*	*	*
	3	2	0.99451	5.24901	0.87483		3	2	1.42909	4.03579	0.67263		3	2	*	*	*
		3	0.77611	6.72607	0.74734			3	1.28442	4.49035	0.49893			3	*	*	*
		4	0.58856	8.86943	0.73912			4	1.21516	4.74629	0.39552			4	*	*	*
	4	2	0.76578	6.81685	0.85211		4	2	1.01838	5.66343	0.70793		4	2	6.28147	5.47196	0.68399
		3	0.54442	9.58851	0.79904			3	1.03266	5.58509	0.46542			3	5.06936	6.78033	0.56503
	5	2	0.65208	8.00547	0.80055		5	2	1.04218	5.53407	0.55341		5	2	*	*	*
100	2	2	1.39428	3.76409	0.94102	100	2	2	1.67998	3.41996	0.85499	100	2	2	*	*	*
		3	0.99662	5.26599	0.87766			3	1.27714	4.49870	0.74978			3	*	*	*
		4	0.78215	6.70995	0.83874			4	1.00668	5.70731	0.71341			4	*	*	*
		5	0.62628	8.37987	0.83799			5	1.06689	5.38524	0.53852			5	*	*	*
	3	2	1.00543	5.21985	0.86997		3	2	1.27167	4.51804	0.75301		3	2	*	*	*
		3	0.69882	7.51007	0.83445			3	0.96702	5.94142	0.66016			3	*	*	*
		4	0.58704	8.94009	0.74501			4	0.91768	6.26083	0.52174			4	*	*	*
	4	2	0.77928	6.73467	0.84183		4	2	1.00850	5.69705	0.71213		4	2	6.24920	5.87260	0.73408
		3	0.58477	8.97479	0.74790			3	0.80633	7.12543	0.59379			3	4.98173	7.36673	0.61389
	5	2	0.61976	8.46804	0.84680		5	2	0.86636	6.63174	0.66317		5	2	*	*	*
150	2	2	1.40317	3.76215	0.94054	150	2	2	1.67239	3.48979	0.87245	150	2	2	*	*	*
		3	0.98548	5.35673	0.89279			3	1.19271	4.89332	0.81555			3	*	*	*
		4	0.76224	6.92556	0.86570			4	1.01091	5.77330	0.72166			4	*	*	*
		5	0.61091	8.64107	0.86411			5	0.86269	6.76524	0.67652			5	*	*	*
	3	2	0.97587	5.40945	0.90157		3	2	1.19979	4.86445	0.81074		3	2	*	*	*
		3	0.69631	7.58130	0.84237			3	0.97064	6.01285	0.66809			3	*	*	*
		4	0.54996	9.59885	0.79990			4	0.82469	7.07696	0.58975			4	*	*	*
	4	2	0.74121	7.12202	0.89025		4	2	0.91165	6.40193	0.80024		4	2	6.10134	6.05678	0.75710
		3	0.58340	9.04861	0.75405			3	0.77858	7.49612	0.62468			3	4.58417	8.06133	0.67178
	5	2	0.60188	8.77074	0.87707		5	2	0.77453	7.53526	0.75353		5	2	*	*	*

Tabla 25. Rendimiento paralelo de la versión con automatizadores en la resolución del problema pr439.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	10.19347	1.00000	1.00000	50	1	10.77666	1.00000	1.00000	52	1	98.19260	1.00000	1.00000
	2	5.50332	1.85224	0.92612		2	6.65493	1.61935	0.80968		2	62.94992	1.55985	0.77993
	3	3.74715	2.72033	0.90678		3	4.59295	2.34635	0.78212		3	47.60221	2.06277	0.68759
	4	2.97502	3.42636	0.85659		4	3.87001	2.78466	0.69616		4	37.39942	2.62551	0.65638
	6	2.16806	4.70164	0.78361		6	3.25492	3.31089	0.55181		6	33.20420	2.95723	0.49287
	8	1.70123	5.99183	0.74898		8	3.02064	3.56767	0.44596		8	*	*	*
	10	1.37251	7.42688	0.74269		10	2.70269	3.98738	0.39874		10	*	*	*
100	1	10.20897	1.00000	1.00000	100	1	10.72990	1.00000	1.00000	100	1	91.44064	1.00000	1.00000
	2	5.29178	1.92921	0.96461		2	6.31194	1.69994	0.84997		2	57.22867	1.59781	0.79891
	3	3.74135	2.72869	0.90956		3	4.33873	2.47305	0.82435		3	42.61864	2.14556	0.71519
	4	2.86256	3.56637	0.89159		4	3.36149	3.19200	0.79800		4	36.46668	2.50751	0.62688
	6	2.01687	5.06178	0.84363		6	2.61808	4.09838	0.68306		6	26.34011	3.47154	0.57859
	8	1.56070	6.54127	0.81766		8	2.34623	4.57326	0.57166		8	23.41590	3.90507	0.48813
	10	1.26209	8.08897	0.80890		10	2.10611	5.09465	0.50947		10	*	*	*
150	1	10.27495	1.00000	1.00000	150	1	10.81281	1.00000	1.00000	152	1	84.94206	1.00000	1.00000
	2	5.37195	1.91270	0.95635		2	6.29266	1.71832	0.85916		2	51.45602	1.65077	0.82539
	3	3.70178	2.77568	0.92523		3	4.13774	2.61322	0.87107		3	38.45488	2.20888	0.73629
	4	2.84651	3.60966	0.90242		4	3.32477	3.25220	0.81305		4	32.31532	2.62854	0.65713
	6	1.97803	5.19454	0.86576		6	2.42905	4.45146	0.74191		6	24.91680	3.40903	0.56817
	8	1.54237	6.66180	0.83273		8	2.04705	5.28214	0.66027		8	19.58691	4.33668	0.54208
	10	1.26650	8.11286	0.81129		10	1.85877	5.81718	0.58172		10	17.59809	4.82678	0.48268

Tabla 26. Rendimiento paralelo de la versión basada en subpoblaciones en la resolución del problema pr439.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	10.19347	1.00000	1.00000	50	1	10.77666	1.00000	1.00000	52	1	98.19260	1.00000	1.00000
	2	5.30348	1.92203	0.96102		2	5.89549	1.82795	0.91398		2	*	*	*
	3	3.52841	2.88897	0.96299		3	3.82553	2.81704	0.93901		3	*	*	*
	4	2.65510	3.83920	0.95980		4	2.97451	3.62300	0.90575		4	31.09538	3.15779	0.78945
	6	1.83937	5.54183	0.92364		6	2.14773	5.01770	0.83628		6	*	*	*
	8	1.38774	7.34539	0.91817		8	1.74064	6.19120	0.77390		8	*	*	*
	10	1.23145	8.27760	0.82776		10	1.56017	6.90739	0.69074		10	*	*	*
100	1	10.20897	1.00000	1.00000	100	1	10.72990	1.00000	1.00000	100	1	91.44064	1.00000	1.00000
	2	5.27719	1.93455	0.96727		2	5.71352	1.87798	0.93899		2	*	*	*
	3	3.57296	2.85728	0.95243		3	3.80916	2.81687	0.93896		3	*	*	*
	4	2.71271	3.76338	0.94084		4	2.99856	3.57835	0.89459		4	28.59375	3.19792	0.79948
	6	1.80589	5.65315	0.94219		6	2.01120	5.33506	0.88918		6	*	*	*
	8	1.37561	7.42142	0.92768		8	1.56501	6.85614	0.85702		8	*	*	*
	10	1.16179	8.78728	0.87873		10	1.36472	7.86237	0.78624		10	*	*	*
150	1	10.27495	1.00000	1.00000	150	1	10.81281	1.00000	1.00000	152	1	84.94206	1.00000	1.00000
	2	5.27018	1.94964	0.97482		2	5.59996	1.93087	0.96544		2	*	*	*
	3	3.64012	2.82270	0.94090		3	3.85954	2.80158	0.93386		3	*	*	*
	4	2.69865	3.80744	0.95186		4	2.89818	3.73089	0.93272		4	25.50811	3.33000	0.83250
	6	1.87094	5.49186	0.91531		6	2.03962	5.30140	0.88357		6	*	*	*
	8	1.36148	7.54691	0.94336		8	1.51719	7.12685	0.89086		8	*	*	*
	10	1.13824	9.02706	0.90271		10	1.29728	8.33496	0.83350		10	*	*	*

Tabla 27. Rendimiento paralelo de la versión híbrida en la resolución del problema pr439.

DTSA						DJAYA						DTLBO					
N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia
50	2	2	2.83990	3.58938	0.89734	50	2	2	3.17450	3.39476	0.84869	50	2	2	*	*	*
		3	2.06389	4.93896	0.82316			3	2.43750	4.42120	0.73687			3	*	*	*
		4	1.65477	6.16007	0.77001			4	2.24985	4.78996	0.59874			4	*	*	*
		5	1.21647	8.37958	0.83796			5	1.80607	5.96693	0.59669			5	*	*	*
	3	2	1.96744	5.18109	0.86351		3	2	2.31281	4.65956	0.77659		3	2	*	*	*
		3	1.43265	7.11513	0.79057			3	1.90678	5.65177	0.62797			3	*	*	*
		4	0.98989	10.29757	0.85813			4	1.56884	6.86921	0.57243			4	*	*	*
	4	2	1.39709	7.29621	0.91203		4	2	1.68224	6.40615	0.80077		4	2	17.34967	5.65962	0.70745
		3	1.01511	10.04178	0.83681			3	1.62847	6.61766	0.55147			3	13.97007	7.02878	0.58573
	5	2	1.19296	8.54469	0.85447		5	2	1.61838	6.65891	0.66589		5	2	*	*	*
100	2	2	2.79241	3.65597	0.91399	100	2	2	3.09493	3.46693	0.86673	100	2	2	*	*	*
		3	1.93764	5.26877	0.87813			3	2.23710	4.79635	0.79939			3	*	*	*
		4	1.50664	6.77600	0.84700			4	1.86284	5.75997	0.72000			4	*	*	*
		5	1.20676	8.45979	0.84598			5	1.53993	6.96780	0.69678			5	*	*	*
	3	2	1.91743	5.32430	0.88738		3	2	2.25167	4.76531	0.79422		3	2	*	*	*
		3	1.28592	7.93905	0.88212			3	1.59591	6.72336	0.74704			3	*	*	*
		4	1.06859	9.55365	0.79614			4	1.49663	7.16937	0.59745			4	*	*	*
	4	2	1.49115	6.84638	0.85580		4	2	1.74199	6.15957	0.76995		4	2	15.68133	5.83118	0.72890
		3	1.06103	9.62173	0.80181			3	1.32328	8.10854	0.67571			3	11.47337	7.96982	0.66415
	5	2	1.17361	8.69876	0.86988		5	2	1.41191	7.59958	0.75996		5	2	*	*	*
150	2	2	2.76709	3.71326	0.92832	150	2	2	3.06615	3.52651	0.88163	150	2	2	*	*	*
		3	1.91563	5.36375	0.89396			3	2.19525	4.92554	0.82092			3	*	*	*
		4	1.47131	6.98354	0.87294			4	1.71198	6.31597	0.78950			4	*	*	*
		5	1.16657	8.80786	0.88079			5	1.41757	7.62770	0.76277			5	*	*	*
	3	2	1.88906	5.43918	0.90653		3	2	2.14714	5.03592	0.83932		3	2	*	*	*
		3	1.30726	7.85991	0.87332			3	1.55489	6.95405	0.77267			3	*	*	*
		4	1.01105	10.16266	0.84689			4	1.25371	8.62465	0.71872			4	*	*	*
	4	2	1.43429	7.16380	0.89548		4	2	1.65962	6.51523	0.81440		4	2	14.04742	6.04681	0.75585
		3	1.00418	10.23222	0.85268			3	1.18926	9.09208	0.75767			3	10.85250	7.82696	0.65225
	5	2	1.14722	8.95640	0.89564		5	2	1.33181	8.11889	0.81189		5	2	*	*	*

Tabla 28. Rendimiento paralelo de la versión con automatizadores en la resolución del problema rat575.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	31.17499	1.00000	1.00000	50	1	32.35326	1.00000	1.00000	52	1	189.85478	1.00000	1.00000
	2	17.13141	1.81976	0.90988		2	17.72436	1.82536	0.91268		2	106.31492	1.78578	0.89289
	3	12.02979	2.59148	0.86383		3	12.62372	2.56289	0.85430		3	87.29944	2.17475	0.72492
	4	9.32495	3.34318	0.83579		4	10.32794	3.13259	0.78315		4	81.21955	2.33755	0.58439
	6	6.71135	4.64512	0.77419		6	7.83288	4.13044	0.68841		6	59.31722	3.20067	0.53344
	8	5.39472	5.77879	0.72235		8	6.66508	4.85414	0.60677		8	*	*	*
	10	3.95584	7.88076	0.78808		10	5.37228	6.02225	0.60223		10	*	*	*
100	1	32.35797	1.00000	1.00000	100	1	32.12896	1.00000	1.00000	100	1	192.72886	1.00000	1.00000
	2	17.19455	1.88187	0.94094		2	17.57264	1.82835	0.91418		2	109.25100	1.76409	0.88205
	3	12.11876	2.67007	0.89002		3	12.32419	2.60698	0.86899		3	82.08612	2.34789	0.78263
	4	9.02316	3.58610	0.89652		4	9.55648	3.36201	0.84050		4	70.28780	2.74200	0.68550
	6	6.32005	5.11989	0.85332		6	6.87150	4.67569	0.77928		6	54.96696	3.50627	0.58438
	8	4.89875	6.60535	0.82567		8	5.59707	5.74032	0.71754		8	47.30776	4.07394	0.50924
	10	3.84671	8.41186	0.84119		10	4.57472	7.02316	0.70232		10	*	*	*
150	1	33.25501	1.00000	1.00000	150	1	32.10102	1.00000	1.00000	152	1	192.09789	1.00000	1.00000
	2	17.13242	1.94106	0.97053		2	17.11976	1.87509	0.93754		2	109.99494	1.74642	0.87321
	3	11.63084	2.85921	0.95307		3	12.13877	2.64450	0.88150		3	79.04380	2.43027	0.81009
	4	9.14113	3.63796	0.90949		4	9.52734	3.36936	0.84234		4	68.28462	2.81319	0.70330
	6	6.23743	5.33153	0.88859		6	6.72871	4.77075	0.79513		6	54.26275	3.54014	0.59002
	8	4.82576	6.89114	0.86139		8	5.32605	6.02717	0.75340		8	44.22449	4.34370	0.54296
	10	3.87902	8.57305	0.85730		10	4.40041	7.29501	0.72950		10	39.63070	4.84720	0.48472

Tabla 29. Rendimiento paralelo de la versión basada en subpoblaciones en la resolución del problema rat575.

DTSA					DJAYA					DTLBO				
N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia	N	Threads	Tiempo	Speed-up	Eficiencia
50	1	31.17499	1.00000	1.00000	50	1	32.35326	1.00000	1.00000	52	1	189.85478	1.00000	1.00000
	2	16.89187	1.84556	0.92278		2	17.03532	1.89919	0.94959		2	*	*	*
	3	11.20469	2.78232	0.92744		3	11.51612	2.80939	0.93646		3	*	*	*
	4	8.50371	3.66605	0.91651		4	8.90649	3.63255	0.90814		4	55.73847	3.40617	0.85154
	6	5.91197	5.27320	0.87887		6	6.37971	5.07127	0.84521		6	*	*	*
	8	4.49579	6.93426	0.86678		8	4.89489	6.60960	0.82620		8	*	*	*
	10	3.76209	8.28661	0.82866		10	3.95915	8.17177	0.81718		10	*	*	*
100	1	32.35797	1.00000	1.00000	100	1	32.12896	1.00000	1.00000	100	1	192.72886	1.00000	1.00000
	2	17.38298	1.86147	0.93074		2	16.97834	1.89235	0.94617		2	*	*	*
	3	11.49443	2.81510	0.93837		3	11.67476	2.75200	0.91733		3	*	*	*
	4	8.78649	3.68270	0.92067		4	9.01699	3.56316	0.89079		4	59.50703	3.23876	0.80969
	6	5.98399	5.40742	0.90124		6	6.18004	5.19883	0.86647		6	*	*	*
	8	4.48355	7.21704	0.90213		8	4.65380	6.90381	0.86298		8	*	*	*
	10	3.74457	8.64130	0.86413		10	3.73028	8.61302	0.86130		10	*	*	*
150	1	33.25501	1.00000	1.00000	150	1	32.10102	1.00000	1.00000	152	1	192.09789	1.00000	1.00000
	2	17.13224	1.94108	0.97054		2	17.17440	1.86912	0.93456		2	*	*	*
	3	11.72049	2.83734	0.94578		3	11.68477	2.74725	0.91575		3	*	*	*
	4	8.89096	3.74032	0.93508		4	8.50022	3.77649	0.94412		4	59.77851	3.21349	0.80337
	6	5.99946	5.54300	0.92383		6	6.28193	5.11006	0.85168		6	*	*	*
	8	4.42891	7.50863	0.93858		8	4.63547	6.92508	0.86564		8	*	*	*
	10	3.70997	8.96368	0.89637		10	3.84423	8.35045	0.83504		10	*	*	*

Tabla 30. Rendimiento paralelo de la versión híbrida en la resolución del problema rat575.

DTSA						DJAYA						DTLBO				
N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia	N	Th. ext.	Th. int.	Tiempo	Speed-up	Eficiencia
50	2	2	8.99054	3.46753	0.86688	50	2	2	8.40553	3.84904	0.96226	50	2	2	*	*	*
		3	6.54319	4.76450	0.79408			3	6.18008	5.23508	0.87251			3	*	*	*
		4	5.17801	6.02066	0.75258			4	5.18953	6.23433	0.77929			4	*	*	*
		5	3.81393	8.17399	0.81740			5	4.08181	7.92621	0.79262			5	*	*	*
	3	2	5.87231	5.30881	0.88480		3	2	5.76540	5.61163	0.93527		3	2	*	*	*
		3	4.55146	6.84945	0.76105			3	4.45845	7.25661	0.80629			3	*	*	*
		4	3.15212	9.89016	0.82418			4	3.49447	9.25842	0.77153			4	*	*	*
	4	2	4.42071	7.05203	0.88150		4	2	4.36114	7.41853	0.92732		4	2	31.43252	6.04008	0.75501
		3	3.06513	10.17085	0.84757			3	3.23112	10.01302	0.83442			3	25.30274	7.50333	0.62528
	5	2	3.74803	8.31770	0.83177		5	2	3.77468	8.57112	0.85711		5	2	*	*	*
100	2	2	8.87296	3.64681	0.91170	100	2	2	8.21428	3.91135	0.97784	100	2	2	*	*	*
		3	6.29768	5.13808	0.85635			3	5.89615	5.44914	0.90819			3	*	*	*
		4	4.85202	6.66897	0.83362			4	4.65854	6.89679	0.86210			4	*	*	*
		5	3.77090	8.58096	0.85810			5	3.76862	8.52539	0.85254			5	*	*	*
	3	2	6.22929	5.19449	0.86575		3	2	5.71416	5.62269	0.93712		3	2	*	*	*
		3	3.89117	8.31574	0.92397			3	3.96657	8.09994	0.89999			3	*	*	*
		4	3.37556	9.58597	0.79883			4	3.31682	9.68668	0.80722			4	*	*	*
	4	2	4.70832	6.87251	0.85906		4	2	4.40347	7.29629	0.91204		4	2	31.37852	6.14206	0.76776
		3	3.34400	9.67643	0.80637			3	3.22378	9.96623	0.83052			3	24.19601	7.96532	0.66378
	5	2	3.71777	8.70360	0.87036		5	2	3.55781	9.03055	0.90306		5	2	*	*	*
150	2	2	9.02296	3.68560	0.92140	150	2	2	8.11682	3.95488	0.98872	150	2	2	*	*	*
		3	6.22580	5.34149	0.89025			3	5.76848	5.56490	0.92748			3	*	*	*
		4	4.66616	7.12685	0.89086			4	4.42433	7.25556	0.90694			4	*	*	*
		5	3.75614	8.85352	0.88535			5	3.67252	8.74087	0.87409			5	*	*	*
	3	2	6.13071	5.42433	0.90406		3	2	5.74261	5.58997	0.93166		3	2	*	*	*
		3	4.28290	7.76459	0.86273			3	3.80130	8.44475	0.93831			3	*	*	*
		4	3.23644	10.27518	0.85626			4	3.11273	10.31281	0.85940			4	*	*	*
	4	2	4.55062	7.30779	0.91347		4	2	4.24376	7.56428	0.94554		4	2	29.83553	6.43856	0.80482
		3	3.21191	10.35366	0.86280			3	3.09547	10.37031	0.86419			3	22.76234	8.43929	0.70327
	5	2	3.65944	9.08746	0.90875		5	2	3.47626	9.23436	0.92344		5	2	*	*	*

A partir de los datos de las tablas, lo más destacable que se puede inferir es que los algoritmos basados en automatizadores presentan una escalabilidad notablemente peor, lo que significa que sufren mayores pérdidas de eficiencia conforme se incrementa el número de *threads*. Esto justifica la necesidad de desarrollar las otras propuestas paralelas. Esta deficiente escalabilidad es debida en parte al gran número de accesos de escritura a memoria compartida (cada vez que se sustituye un individuo de la población), lo que genera contenciones que son mayores al incrementar el número de *threads*. Esto contrasta con las versiones basadas en subpoblaciones, que presentan una eficiencia muy buena al escalar perfectamente, salvo puntuales excepciones, hasta alcanzar 8 *threads* en el caso del DJAYA y hasta el total de 10 en el DTSA.

Otro dato a tener en cuenta es que el tamaño de la población afecta al rendimiento de los algoritmos paralelos, habiendo diferencias bastante mayores entre emplear 50 y 100 que entre emplear 100 y 150. Posiblemente un tamaño de 50 sea demasiado pequeño para poder paralelizar eficientemente este tipo de algoritmos, ya que al dividir el trabajo a realizar sobre una población de este tamaño entre un número elevado de *threads*, el coste computacional de lo que cada *thread* ejecute será excesivamente pequeño y la eficiencia bajará debido a que, aunque las sincronizaciones no se incrementan su coste relativo sí lo hace, además los procesos a realizar con exclusión mutua sí se ven incrementados al aumentar el número de *threads*.

En cuanto a los tres algoritmos DTSA, DJAYA y DTLBO, es importante mencionar que DTSA consigue una eficiencia mucho mayor que los otros dos cuando se utiliza el enfoque basado en automatizadores, escalando sin problemas exceptuando algunos problemas de tamaño pequeño cuando se usa un tamaño de población de 50. Retomando lo dicho anteriormente, esto puede deberse al menor uso de memoria compartida, pues este algoritmo realiza las sustituciones de los individuos de una manera distinta a los otros dos, como se mencionó en capítulo 4, es decir utilizando memoria temporal para almacenar los nuevos individuos a insertar en la población.

El algoritmo DJAYA es el que presenta una mayor tendencia a mejorar su eficiencia conforme aumenta el tamaño del problema. Con los problemas pequeños (kroA150 y kroB200) no presenta una buena escalabilidad, teniendo problemas de eficiencia en algún caso con 4 *threads* en su versión con automatizadores. Pero en cambio, en los problemas

grandes (pr439 y rat575) sus diferencias con el algoritmo DTSA se reducen mucho, especialmente en la versión basada en subpoblaciones (en el caso de rat575 la versión híbrida también llega a igualar a DTSA). DTLBO tiene un comportamiento bastante similar a DJAYA en términos de eficiencia paralela, aunque presenta menos dependencia del tamaño del problema. Hay que tener en cuenta que el algoritmo DTLBO tiene bastante más coste computacional que los otros dos, debido, entre otras cosas, a la utilización de la función de factibilidad, que implica recorrer el individuo de la población buscando valores duplicados.

Los algoritmos híbridos, por otra parte, presentan unos buenos resultados de eficiencia cuando el tamaño de población es 150, salvo por el DTSA que siempre obtiene buenos resultados. La eficiencia de estos algoritmos es bastante superior a la de la versión con automatizadores y ligeramente inferior a la de la versión basada en subpoblaciones, exceptuando cuando el tamaño de población es 50, pues en este caso sus diferencias con la versión de subpoblaciones se acrecentan.

En cuanto a la distribución de los *threads* en los algoritmos híbridos, se puede ver que es ligeramente más eficiente para un mismo número total de *threads* asignar un mayor número de *threads* a la región externa que a la región interna, siendo esta diferencia no demasiado importante.

Hay que remarcar que las propuestas híbridas ofrecen una mayor versatilidad manteniendo un muy buen comportamiento paralelo. Además, en el caso de que se ejecutaran en una máquina con *hyperthreading*, estos algoritmos permitirían tener mapeados diferentes threads en un mismo *core* compartiendo la memoria caché y por tanto acelerando el código.

Por tanto, las conclusiones de este análisis serían que, al ser ejecutados en unas condiciones de hardware y datos de problema similares a éstas:

- El enfoque basado en automatizadores presenta una eficiencia y escalabilidad buenas únicamente para el algoritmo DTSA.
- El enfoque basado en subpoblaciones escala de forma muy eficiente en la práctica totalidad de los casos estudiados. Pero hay que tener en cuenta que por un lado el

algoritmo DTLBO sólo permite trabajar con 4 *threads*, y por otro que los algoritmos DTSA y DJAYA han sido ligeramente modificados.

- El enfoque híbrido permite una mayor versatilidad que el enfoque basado en subpoblaciones sin suponer una merma significativa en la eficiencia y en la escalabilidad.

CAPÍTULO 6:

CONCLUSIONES

En este capítulo se expondrán detalladamente todas las conclusiones obtenidas tras la realización de trabajo.

En este trabajo se han implementado tres algoritmos metaheurísticos de optimización discreta: DTSA (*Discrete Tree-Seed Algorithm*), DJAYA (*Discrete Jaya*) y DTLBO (*Discrete Teaching-Learning-Based Optimization*), y dichos algoritmos han sido caracterizados tomando los datos de su rendimiento a la hora de resolver 6 problemas TSP (*Travelling Salesman Problem*) provenientes de la librería TSPLIB de Gerhard Reinelt, los cuales son kroA150, kroB200, tsp225, pr226, pr439 y rat575, siendo problemas de tamaño medio-bajo, al estar comprendidos entre 150 y 575 variables de diseño o ciudades si consideramos el problema TSP.

De este análisis se puede concluir que, para los problemas considerados, el número óptimo de evaluaciones de función está alrededor de 10^7, ya que, analizando las curvas de convergencia, a partir de este número de evaluaciones de función la mejora en la solución, si existe, no es relevante respecto al incremento de coste computacional asociado. También se ha constatado que los algoritmos DTSA y DJAYA permiten obtener mejores resultados que DTLBO para los problemas estudiados, siendo la velocidad de convergencia del algoritmo DJAYA ligeramente mejor a la del algoritmo DTSA.

Además, han sido desarrollados tres algoritmos paralelos para permitir acelerar la ejecución de estos algoritmos metaheurísticos en un equipo de computación de altas prestaciones con arquitectura de memoria compartida. Los tres enfoques han sido: un diseño basado en automatizadores, un diseño basado en subpoblaciones y un diseño híbrido, que permite aprovechar las ventajas de ambos, estas tres propuestas han sido aplicadas a cada uno de los tres algoritmos estudiados.

El análisis del rendimiento paralelo de estos algoritmos ha permitido determinar que el enfoque basado en automatizadores obtiene mejores resultados de eficiencia y escalabilidad cuando es aplicado al algoritmo DTSA, mientras que en DJAYA y DTLBO ~~los~~ resultados obtenidos, en especial de escalabilidad, no han sido excelentes. Por otra parte, el enfoque basado en subpoblaciones ha obtenido unos buenos resultados de rendimiento paralelo en todos los casos. También cabe destacar que en el caso de DJAYA, el tamaño del problema influye mucho en el rendimiento del código paralelo, siendo penalizado en los problemas de pequeño tamaño, mientras que esta característica es menos apreciable en DTSA y DTLBO.

Finalmente, en lo que respecta a los algoritmos híbridos, éstos no suponen una disminución apreciable en la eficiencia y en la escalabilidad respecto de los algoritmos basados en subpoblaciones, y tienen además una gran versatilidad, ya que permiten decidir en los algoritmos DTSA y DJAYA el tamaño de las subpoblaciones sin depender del rendimiento paralelo, centrándose únicamente en el rendimiento de la optimización, minimizando las modificaciones requeridas respecto al código original. Además, para el algoritmo DTLBO, en el cual la versión con subpoblaciones sólo permite emplear 4 *threads*, permite incrementar eficientemente el número de *threads* a utilizar.

Este trabajo me ha resultado muy interesante y enriquecedor, pues me ha permitido profundizar más en campos como la algoritmia y la computación paralela, que son áreas que siempre me han llamado la atención y que eché en falta haber podido tocar más en la carrera. Todavía no tengo decidido mi futuro después del grado, pero pienso que me gustaría especializarme en esto o en áreas relacionadas.

BIBLIOGRAFÍA

[1] ¿Cuáles Son Las Diferencias Entre NP, NP-Complete y NP-Hard?. QA Stack. www.qastack.mx/programming/1857244/what-are-the-differences-between-np-np-complete-and-np-hard

[2] Dantzig, G. B. (1956). The Simplex Method. Santa Monica, CA: RAND Corporation, 1956. https://www.rand.org/pubs/papers/P891.html

[3] Land, A. H., & Doig, A. G. (1960). An automatic method of solving discrete programming problems. Econometrica, 28(3), 497. doi:10.2307/1910129

[4] Pólya, G. (1945). How to solve it a new aspect of mathematical method. New York: Princeton University Press.

[5] Rodríguez Ortiz, C. (2010). Algoritmos heurísticos y metaheurísticos para el problema de localización de regeneradores. Escuela Técnica Superior de Ingeniería Informática. Universidad Rey Juan Carlos.

[6] Díaz, V. A., & González, F. J. L. (1996). Optimización heurística y redes neuronales. Madrid: Paraninfo.

[7] Müller-Merbach, H. (1981). Heuristics and their design: A survey. European Journal of Operational Research, 8(1), 1-23. doi:10.1016/0377-2217(81)90024-2

[8] Fernández Hernández, A. (2016). Algoritmos heurísticos y metaheurísticos basados en búsqueda local aplicados a Problemas de Rutas de Vehículos. Departamento de Estadística e Investigación Operativa. Universidad de Valladolid.

[9] Glover, F. (1986) "Future Paths for Integer Programming and Links to Artificial Intelligence," Computers and Operations Research, Vol. 13, pp. 533-549. doi: 10.1016/0305-0548(86)90048-1

[10] Osman, I. H., & Kelly, J. P. (1997). Meta-Heuristics theory and applications. Journal of the Operational Research Society, 48(6), 657-657. doi:10.1057/palgrave.jors.2600781

[11] Herrera, F. (2009). Introducción a los algoritmos metaheurísticos. https://sci2s.ugr.es/sites/default/files/files/Teaching/OtherPostGraduateCourses/Metaheuristicas/Int-Metaheuristicas-CAEPIA-2009.pdf

[12] Muñoz, M. A., Lopez, J., & Caicedo Bravo, E. (2008). Inteligencia de enjambres: sociedades para la solución de problemas (una revisión). Ingenieria e Investigación. 28. 119-130. http://www.scielo.org.co/pdf/iei/v28n2/v28n2a15.pdf

[13] Feo, T. A., & Resende, M. G. (1989). A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters, 8(2), 67-71. doi:10.1016/0167-6377(89)90002-3

[14] Glover, F. (1989). Tabu search—part I. ORSA Journal on Computing, 1(3), 190-206. doi:10.1287/ijoc.1.3.190

[15] Riojas Cañari, A. C. (2005). Conceptos, algoritmo y aplicación al problema de las N – reinas Capítulo3. Búsqueda de tabú. https://sisbib.unmsm.edu.pe/bibvirtualdata/monografias/basic/riojas_ca/cap3.pdf

[16] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. Science, 220(4598), 671-680. doi:10.1126/science.220.4598.671

[17] Colorni, A., Dorigo, M., & Maniezzo, M. (1991). Distributed Optimization by Ant Colonies. Proceedings of the First European Conference on Artificial Life.

[18] Vargas Paredes, J. & Penit Granado, V. (2016). Estudio y aplicación de metaheurísticas y comparación con métodos exhaustivos. Facultad de Informática. Universidad Complutense de Madrid. https://eprints.ucm.es/id/eprint/39844/1/MEMORIA.pdf

[19] Holland, J. H. (2010). Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. MIT Press

[20] Martí, R. Procedimientos metaheurísticos en optimización combinatoria. Departament d'Estadística i Investigació Operativa. Universitat de València.

[21] Pérez, E. (2010). Guía para recién llegados a los algoritmos genéticos. Universidad de Valladolid.

[22] Amdahl, G. M. (1967). Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Conference Proceedings (30): 483–485. doi:10.1145/1465482.1465560.

[23] Aguilar J. Introducción a la computación paralela. http://www.ing.ula.ve/~aguilar/publicaciones/objetos/libros/ICP.pdf

[24] MPI forum. https://www.mpi-forum.org/

[25] Jiménez-González, D. (2013). Introducción a las arquitecturas paralelas. https://www.exabyteinformatica.com/uoc/Informatica/Arquitecturas_de_computadores_avanzadas/Arquitecturas_de_computadores_avanzadas_(Modulo_1).pdf

[26] OpenMP 5.1 Specification. OpenMP. https://www.openmp.org/spec-html/5.1/openmp.html

[27] Kiran, M. S. (2015). TSA: Tree-seed algorithm for continuous optimization. Expert Systems with Applications, 42(19), 6686-6698. doi:10.1016/j.eswa.2015.04.055

[28] Cinar, A. C., Korkmaz, S., & Kiran, M. S. (2020). A discrete tree-seed algorithm for solving symmetric traveling salesman problem. Engineering Science and Technology, an International Journal, 23(4), 879-890. doi:10.1016/j.jestch.2019.11.005

[29] Venkata Rao, R. (2016). Jaya: A simple and new optimization algorithm for Solving constrained and unconstrained optimization problems. International Journal of Industrial Engineering Computations, 19-34. doi:10.5267/j.ijiec.2015.8.004

[30] Gunduz, M., & Aslan, M. (2021). DJAYA: A discrete Jaya algorithm for solving traveling salesman problem. Applied Soft Computing, 105, 107275. doi:10.1016/j.asoc.2021.107275

[31] Rao, R., Savsani, V., & Vakharia, D. (2011). Teaching–learning-based optimization: A novel method for constrained mechanical design optimization problems. Computer-Aided Design, 43(3), 303-315. doi:10.1016/j.cad.2010.12.015

[32] Rico-Garcia, H., Sanchez-Romero, J., Migallon Gomis, H., & Rao, R. V. (2020). Parallel implementation of metaheuristics for optimizing tool path computation on CNC machining. Computers in Industry, 123, 103322. doi:10.1016/j.compind.2020.103322

[33] Wu, L., Zoua, F., & Chen, D. (2017). Discrete Teaching-learning-based optimization Algorithm for Traveling Salesman Problems. MATEC Web of Conferences, 128, 02022. doi:10.1051/matecconf/201712802022

[34] Yann Picand, D. D. Problema del viajante: definición de Problema del viajante y sinónimos de Problema del viajante (español). http://diccionario.sensagent.com/Problema%20del%20viajante/es-es/

[35] Sauerwald, T. (2020). V. Approx. Algorithms: Travelling Salesman Problem. University of Cambridge. https://www.cl.cam.ac.uk/teaching/1920/AdvAlgo/tsp.pdf

[36] Calviño Martínez, A. (2011). Cooperación en los problemas del viajante (TSP) y de rutas de vehículos (VRP): una panorámica. Universidad de Santiago de Compostela. http://eio.usc.es/pub/mte/descargas/ProyectosFinMaster/Proyecto_762.pdf

[37] Ma, S. Understanding The Travelling Salesman Problem (TSP). https://blog.routific.com/travelling-salesman-problem.

[38] TSPLIB. Discrete and Combinatorial Optimization. Ruprecht-Karls-Universität Heidelberg. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[39] Reinelt, G. TSPLIB 95. Ruprecht-Karls-Universität Heidelberg. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf

[40] Guangzhi, M., Yansheng, L., Enmin, S., & Wei, Z. (2009). Introducing Gene Clusters into a P2P Based TSP Solving Algorithm. 2009 WRI World Congress on Computer Science and Information Engineering. doi:10.1109/csie.2009.468

[41] Croes, G. A. (1958). A Method for Solving Traveling-Salesman Problems. Operations Research, 6(6), 791–812. doi:10.1287/opre.6.6.791