ANÁLISIS DE RENDIMIENTO Y OPTIMIZACIÓN DE ALGORITMOS PARALELOS BEST-FIRST SEARCH SOBRE MULTICORE Y CLUSTER DE MULTICORE

Tesis presentada para obtener el grado de
Doctor en Ciencias Informáticas

Autor: Esp. Victoria María Sanz.

Director: Ing. Armando De Giusti.

Co-Director: Dr. Ricardo Marcelo Naiouf

FACULTAD DE INFORMÁTICA - UNIVERSIDAD NACIONAL DE LA PLATA

Diciembre de 2014

ÍNDICE

RESUMEN

El objetivo general de esta tesis se centra en la investigación y desarrollo de algoritmos paralelos de búsqueda en grafos *best-first search* para arquitecturas multicore y cluster de multicore, que mejoran los existentes y se utilizan para resolver problemas de optimización combinatoria y de planificación, acompañado de un análisis de rendimiento (speedup, eficiencia, escalabilidad) de los mismos.

La temática propuesta es de interés en la actualidad por la complejidad computacional de dichos algoritmos de búsqueda y las posibilidades que brindan las arquitecturas mencionadas. Los algoritmos presentados en esta tesis pueden aplicarse para resolver problemas reales como planificación de rutas óptimas, navegación automática de un robot o vehículo, alineamiento óptimo de secuencias, entre otros.

Los temas de investigación derivados son múltiples y se refieren tanto a la paralelización de algoritmos sobre (a) arquitecturas de memoria compartida, como son los multicore (b) arquitecturas de memoria distribuida, como son los clusters (c) y también sobre arquitecturas híbridas, tal es el caso de los clusters de multicore.

El aporte de la tesis es el desarrollo de dos algoritmos paralelos *best-first-search* propios, uno apto para su ejecución sobre máquinas de memoria compartida (multicore) y otro apto para máquinas de memoria distribuida (cluster), basados en el algoritmo HDA* (*Hash Distributed A**), en los cuales se incluyen técnicas originales que optimizan su rendimiento.

Asimismo, se presenta un análisis de rendimiento de los algoritmos desarrollados a medida que escala la carga de trabajo y la arquitectura paralela subyacente.

Para finalizar, se compara la memoria consumida por ambos algoritmos y el rendimiento alcanzado cuando se los ejecuta sobre una máquina multicore; estos análisis presentan originalidad en el área. Los resultados arrojados indican que se obtendría un beneficio al convertir HDA* en una aplicación híbrida, cuando la arquitectura subyacente es un cluster de multicore, por lo que se sientan las bases para éste algoritmo híbrido.

CONTENIDO

A continuación se detalla el contenido de la tesis:

- *Capítulo 1: Introducción.*
  Este capítulo presenta la motivación de la tesis, los objetivos planteados y las contribuciones aportadas. Asimismo, enumera las publicaciones derivadas de éste trabajo y se citan trabajos previos del tesista y sus directores siguiendo la misma temática.

- *Capítulo 2: Algoritmos secuenciales de búsqueda.*
  Este capítulo describe cómo transformar el problema a resolver en un problema de búsqueda, estudia los algoritmos de búsqueda más utilizados y presenta una comparación de los mismos con el objetivo de seleccionar el más apropiado para resolver el caso de estudio.

- *Capítulo 3: Caracterización del caso de estudio.*
  Este capítulo presenta el problema *15-Puzzle* seleccionado como caso de estudio, describe sus características, especifica un algoritmo para decidir cuándo una configuración inicial del problema puede resolverse en una configuración final, describe heurísticas desarrolladas por distintos autores para este problema y menciona problemas del mundo real que pueden resolverse con los algoritmos presentados en esta tesis.

- *Capítulo 4: Arquitecturas paralelas y herramientas de programación.*
  Este capítulo repasa las clasificaciones de las arquitecturas paralelas, describe las características de los clusters y máquinas multicore, analiza las herramientas de programación paralela más utilizadas hoy en día y presenta una discusión acerca de los retos que debe afrontar el programador para sacarle provecho a la arquitectura.

- *Capítulo 5: Algoritmos paralelos Best-First Search*
  Este capítulo presenta las métricas utilizadas para analizar el rendimiento de los algoritmos paralelos de búsqueda y las causas comunes de *overhead*, sintetiza el estado del arte de los algoritmos paralelos *Best First Search* con énfasis en la búsqueda de soluciones óptimas mediante A*, estudia estrategias simples para la detección de terminación en aplicaciones con cómputo distribuido, analiza distintas bibliotecas de gestión de memoria dinámica y presenta posibles mejoras a los algoritmos paralelos de búsqueda existentes.

- *Capítulo 6: Implementaciones*
  Este capítulo presenta implementaciones propias de los algoritmos que permiten resolver el caso de estudio: algoritmo secuencial A*, algoritmo HDA* para arquitecturas de memoria distribuida y algoritmo HDA* para arquitecturas de memoria compartida. Luego, plantea una discusión sobre los factores a analizar de cada algoritmo.

- *Capítulo 7: Resultados*
  Este capítulo analiza los resultados obtenidos de la ejecución del algoritmo A* secuencial utilizando distintos gestores de memoria dinámica y diferentes heurísticas. A continuación, muestra los resultados experimentales para el algoritmo HDA* para memoria compartida cuando se ejecuta sobre una máquina multicore, analizando las técnicas y valores de parámetros que optimizan su rendimiento. Luego, estudia el comportamiento del algoritmo HDA* para memoria distribuida cuando se ejecuta sobre una máquina multicore y sobre un cluster de multicore, analizando el impacto de sus parámetros sobre el rendimiento obtenido. Por último, presenta comparaciones de consumo de memoria y rendimiento para ambos algoritmos paralelos cuando se los ejecuta sobre una máquina multicore.

- Capítulo 8: Algoritmo HDA* Híbrido
  Este capítulo estudia distintos modelos de aplicaciones híbridas y sienta las bases para el algoritmo HDA* Híbrido propuesto.

- *Capítulo 9: Conclusiones y Líneas de Trabajo Futuro.*
  Este capítulo resume los resultados obtenidos y plantea posibles trabajos a realizar en un futuro.

- *Apéndice A: Correctitud del Algoritmo de Verificación de Solubilidad para el N^2-1 Puzzle*
  Este apéndice analiza el algoritmo utilizado para decidir cuándo una configuración inicial del problema N-Puzzle puede resolverse en una configuración final.

- *Apéndice B: Función de Zobrist*
  Este apéndice describe el cálculo realizado por la Función de Zobrist, utilizada por el algoritmo HDA* para balancear la carga de trabajo.

- *Apéndice C: Configuraciones Utilizadas del 15-Puzzle*
  Este apéndice enumera las configuraciones del 15-Puzzle utilizadas en el Capítulo 7.

CAPÍTULO 1: INTRODUCCIÓN.

1.1 Motivación

En el área de Inteligencia Artificial los algoritmos de búsqueda informada son utilizados como base para resolver problemas de optimización combinatoria, para los cuales se requiere encontrar una secuencia de acciones que minimice una función objetivo y permita transformar una configuración inicial, que representa la instancia del problema a resolver, en una configuración final, que representa la solución.

Uno de los algoritmos de búsqueda más utilizados para dicho propósito es conocido como Best-First Search (*BFS*) (Russel & Norvig, 2003), que permite explorar el grafo que representa el espacio de estados del problema utilizando una función de costo $\hat{f}$ para valuar los nodos, conformada en parte por información heurística, la cual permitirá guiar la búsqueda más rápido hacia la solución reduciendo la cantidad de nodos a considerar. El algoritmo difiere de los métodos convencionales en que el grafo es implícito y se genera dinámicamente, es decir los nodos se crean a medida que la búsqueda avanza. Durante el proceso mantiene dos estructuras de datos: una con los nodos que no fueron explorados al momento (*Lista Abierta*), ordenados por la función $\hat{f}$, y otra con los nodos ya explorados (*Lista Cerrada*), utilizada para evitar procesar varias veces un mismo estado. En cada iteración se extrae el nodo más prometedor (según la función $\hat{f}$) disponible en la Lista Abierta, se lo incluye en la Lista Cerrada, y se le aplican acciones válidas generando nodos sucesores que serán insertados en la Lista Abierta según ciertas condiciones (verificación conocida como *detección de duplicados*). La búsqueda continúa hasta que el nodo extraído de la Lista Abierta sea aquel que representa la solución.

El algoritmo A* (Hart, et al., 1968) es una de las variantes más utilizadas de *BFS* ya que garantiza encontrar soluciones de costo óptimo. Para ello la función $\hat{f}$ incluye información de costo conocida del camino desde el nodo inicial hacia el nodo actual e información heurística para estimar el costo desconocido del camino desde el nodo actual al nodo solución, la cual nunca debe sobreestimar el costo real. De esta manera se guía la búsqueda a que procese primero los caminos más prometedores.

Por otro lado, en los últimos años se ha impulsado el desarrollo de algoritmos paralelos de búsqueda informada ya que el alto requerimiento de memoria y potencia de cómputo causados por el crecimiento exponencial o factorial del grafo dificultan su resolución sobre un procesador *single-core*.

Hoy en día es común poseer máquinas con múltiples cores, por lo que las aplicaciones secuenciales deben adaptarse para aprovechar la potencia de cómputo que brinda esta arquitectura. Asimismo, el agregado de memoria que surge al conectar varias de éstas máquinas a través de una red, formando un *cluster de multicore*, y la potencia de cómputo lograda con esta arquitectura permiten resolver instancias más complejas de un problema.

Hasta el momento distintos autores han propuesto diversas técnicas para paralelizar los algoritmos de búsqueda *BFS*, que difieren en cómo manipulan la Lista Abierta y la Lista Cerrada, y en la estrategia de balance de carga a utilizar entre los procesadores durante la ejecución. La técnica a seleccionar dependerá de la arquitectura y el problema a resolver (Kumar, et al., 1988).

En una arquitectura de memoria compartida, la estrategia más simple se basa en mantener una única Lista Abierta y una única Lista Cerrada compartidas por todos los threads (*estrategia centralizada*), lo que implica que éstos deban sincronizarse para mantener la consistencia de las estructuras, siendo éste un limitante del rendimiento (Kumar, et al., 1988) (Cung & Le Cun, 1994). Si bien la Lista Abierta y la Lista Cerrada pueden ser implementadas a través de estructuras de datos que permitan acceso concurrente por porciones con el objetivo de reducir la contención en el acceso, distintos trabajos han demostrado que esta técnica produce mejoras solamente para problemas con alto tiempo de cómputo de heurística (Kumar, et al., 1988), y en particular estudios actuales han demostrado que no obtiene un rendimiento competitivo sobre máquinas multicore (Burns, et al., 2010).

Para resolver el problema anterior, cada proceso/thread puede tener su propia Lista Abierta y Lista Cerrada local (*estrategia descentralizada o distribuida*) y realizar una búsqueda cuasi independiente. Esta estrategia se adapta bien tanto a arquitecturas de memoria compartida o de memoria distribuida. Sin embargo, surge la necesidad de comunicación entre los procesos/threads por las siguientes razones:

- Dado que al principio sólo un proceso/thread tendrá en su Lista Abierta el nodo inicial y que el grafo se genera durante la ejecución, se debe distribuir la carga de trabajo en forma dinámica.

- Los nodos ubicados en la Lista Abierta de un procesador no necesariamente son los mejores globales, por lo que se debe realizar una equiparación de la calidad de los nodos.

- Los nodos duplicados (que representan un mismo estado) pueden generarse en distintos procesos/threads. Si la *detección de duplicados* sólo se realiza en el proceso/thread que generó el nodo y/o en aquel que recibió el nodo por balance de carga, la detección y poda de duplicados será *parcial* ya que otro puede tener en su Lista Abierta o en su Lista Cerrada un nodo representando el mismo estado. En cambio, si se quiere realizar una detección y poda *absoluta* se necesitan estrategias que asignen cada estado a un procesador particular.

- El criterio de terminación debe modificarse ya que se disponen de múltiples Listas Abiertas inconsistentes y a causa del uso de balance de carga dinámico pueden existir nodos del grafo siendo comunicados entre los procesos/threads.

- Los costos de las soluciones parciales encontradas deben comunicarse para utilizarlos para podar caminos que conducen a soluciones de costo subóptimo.

En este sentido, el algoritmo HDA* (*Hash Distributed A**) (Kishimoto, et al., 2013) paraleliza A* aplicando una *estrategia descentralizada* y utiliza la *función hash de Zobrist* (Zobrist, 1970) para asignar cada estado del problema a un único proceso. De esta manera ante la generación de un nodo ajeno por parte de un proceso se establece quién es su dueño y se lo transfiere al

mismo. Este mecanismo permite balancear la carga de trabajo, equiparar la calidad de los nodos y podar duplicados de forma *absoluta*, dado que los nodos que representan a un mismo estado siempre se enviarán al mismo proceso.

El algoritmo HDA* fue implementado haciendo uso puramente de la biblioteca de paso de mensajes *MPI* y comunicación asincrónica, de modo que es apto para ser ejecutado tanto en arquitecturas de memoria distribuida como arquitecturas de memoria compartida. Para evitar el congestionamiento de la red a consecuencia del envío de mensajes conteniendo un único nodo, el algoritmo incorpora la técnica propuesta por (Romein, et al., 2002) de empaquetar en un mismo mensaje un número determinado de nodos con igual destinatario antes de que éste sea enviado (dicho número es conocido en esta tesis como *LNPT – Limite de Nodos Por Transferencia*).

El trabajo experimental lo realizan ubicando a HDA* como el mecanismo de búsqueda de un planificador independiente del dominio llamado *Fast-Downward* (Helmert , 2006) y también con el *24-Puzzle,* una aplicación específica en la cual el procesamiento de un estado es más rápido por ser dependiente del dominio y para la cual utilizan una heurística basada en *bases de datos de patrones disjuntos* (Culberson & Schaeffer, 1998) (Korf, 2000). Señalan la buena escalabilidad del algoritmo para distintas arquitecturas paralelas actuales tales como una máquina multicore, un *cluster* de multicore *HPC* con interconexión *Infiniband* y un *cluster de multicore convencional* con conexión 1Gb (x2) Ethernet.

Por otra parte, en el trabajo (Burns, et al., 2010) se presenta una adaptación del algoritmo HDA* desarrollado haciendo uso de las herramientas de programación para memoria compartida provistas por la biblioteca *Pthreads*; de este modo se eliminan ciertas ineficiencias que surgen cuando se ejecuta el algoritmo HDA* original sobre una máquina con memoria compartida. Los autores presentan un análisis del rendimiento alcanzado por el algoritmo sobre una máquina multicore. Para evitar la degradación de rendimiento debido a la contención en el acceso a las estructuras manejadas por el gestor de memoria dinámica causada por las frecuentes operaciones alloc/free, utilizan la biblioteca Jemalloc (Evans, 2006). Además, incluyen en el algoritmo una técnica para abstraer el espacio de estados y asignar bloques de estados a los threads en lugar de estados individuales como sucede con la función hash de Zobrist. Luego presentan el algoritmo *PBNF (Parallel Best-NBlock First)* cuyo objetivo es permitir que los threads trabajen en períodos libres de sincronización. El trabajo experimental de los algoritmos fue realizado en parte sobre instancias *poco complejas* del 15-Puzzle, utilizando la heurística clásica Suma de las Distancias de Manhattan y analizando el speedup alcanzado a medida que escala la arquitectura. Si bien *PBNF* obtiene mejor rendimiento, el algoritmo es complejo y no paraleliza A* puramente, por lo que alcanza en algunos casos un speedup muy superior al teóricamente posible. Asimismo, presenta mayor overhead de sincronización que la versión de HDA* presentada en el mismo trabajo.

Por lo expuesto anteriormente, el algoritmo HDA* sigue siendo de interés actual por su simpleza y buena escalabilidad.

Esta tesis se centra en implementar una versión propia del algoritmo HDA* para su ejecución sobre arquitecturas de memoria compartida (*multicore*) y una versión propia del algoritmo HDA* para su ejecución sobre arquitecturas de memoria distribuida (*cluster)*, desarrollar

técnicas para optimizar el rendimiento de dichos algoritmos, y analizar en cada caso el speedup y eficiencia alcanzados a medida que escala la complejidad del problema y la cantidad de procesadores. Asimismo, este trabajo presenta un análisis de cantidad de memoria utilizada por cada algoritmo cuando se ejecutan sobre una máquina multicore.

El problema elegido como caso de estudio es el 15-Puzzle (Ratner & Warmuth, 1990), el cual posee una especificación exacta y simple, ha sido usado por diversos autores en el ámbito del cómputo paralelo para evaluar el rendimiento de sus algoritmos y ha sido estudiado por distintos autores para la generación de heurísticas más potentes. Actualmente ha generado interés debido a que se ajusta a problemas reales tales como el movimiento de objetos por parte de un vehículo en un depósito con alta densidad de almacenamiento (Gue, et al., 2013).

Los resultados aportados por esta tesis sugieren que se obtendría un potencial beneficio al adaptar el algoritmo HDA* para convertirlo en una aplicación híbrida cuando será ejecutado sobre un *cluster de multicore*, de forma que utilice herramientas de comunicación y sincronización por memoria compartida entre los threads que se ejecutan en la misma máquina, y paso de mensajes para comunicar procesos que residen en distintas máquinas del cluster. En consecuencia, se sientan las bases para el algoritmo HDA* híbrido.

1.2 Objetivos y contribuciones

Esta tesis desarrolla una versión propia del algoritmo paralelo HDA* apto para su ejecución sobre arquitecturas de memoria compartida, que permite encontrar soluciones a problemas de optimización combinatoria. El algoritmo está basado en aquel propuesto en (Burns, et al., 2010) pero difiere en:

- La incorporación de un algoritmo para detectar terminación en forma *descentralizada*, que es una adaptación del algoritmo propuesto por *Dijkstra y Safra* (Dijkstra, 1987) (Dijkstra, et al., 1983).
- La incorporación del parámetro *LNPI* (*Límite de Nodos por Iteración*), que indica la cantidad límite de nodos a procesar por iteración del algoritmo. Si bien los autores del algoritmo original proponen el uso de este parámetro, no analizan su influencia sobre el rendimiento.
- La acumulación por parte del thread de una cantidad parametrizable de nodos dirigidos a otro thread antes de intentar el traspaso de los mismos (*LNPT o Límite de Nodos por Transferencia)*, es decir, no se realizan transferencias tras cada generación de nodo como en el algoritmo original. Esto permite disminuir la contención en el acceso a las estructuras utilizadas para la comunicación.
- La utilización de una técnica *Memory Pool* para prevenir la degradación de rendimiento causada por operaciones alloc-free en una relación productor-consumidor entre distintos threads. Dicho de otra manera, la problemática resuelta es la siguiente: durante el algoritmo los threads irán transfiriendo los punteros a nodos cuyo procesamiento

corresponde a otro thread; cuando un thread genera un nodo, éste es alocado en una de las áreas de memoria dinámica asignada al thread por el gestor de memoria (*Jemalloc* llama a ésta zona *arena*) las cuales están protegidas para mantener su consistencia ante el posible acceso por parte de distintos threads; cuando un thread A transfiere el puntero a un nodo generado por él a otro thread B, si éste último realiza una liberación del espacio asignado para el mismo (por ejemplo a causa que el nodo conduce a una solución subóptima), tendría que acceder a la *arena* asignada al thread A, lo cual genera contención en el acceso a la misma. La técnica propuesta consiste en que cada thread almacene los nodos cuyo espacio quiera liberar en un *pool*, para que sean re-utilizados ante futuras generaciones de nodos.

El trabajo experimental utiliza el problema 15-Puzzle como caso de estudio, haciendo uso de una heurística que mejora a la clásica, y demuestra los beneficios de las dos últimas técnicas incorporadas, siendo un aporte original en el área. Asimismo, se analiza la escalabilidad del algoritmo sobre una máquina multicore, evaluando el speedup y la eficiencia alcanzados al aumentar la carga de trabajo (es decir, la complejidad de la instancia) y la cantidad de cores utilizados. En este sentido el trabajo experimental es de particular interés ya que en (Burns, et al., 2010) sólo se utilizan instancias poco complejas del 15-Puzzle, y únicamente evalúan el speedup a medida que escala la arquitectura. Por último, se documenta el efecto de los parámetros *LNPI* y *LNPT* sobre el rendimiento.

Por otra parte, la tesis presenta la implementación de una versión propia del algoritmo paralelo HDA* para arquitecturas de memoria distribuida. El algoritmo está basado en aquel propuesto en (Kishimoto, et al., 2013) pero difiere en:

- La utilización del algoritmo de detección de terminación propuesto por *Dijkstra y Safra (Dijkstra, 1987)* (Dijkstra, et al., 1983), el cual no incrementa el tamaño de los mensajes de trabajo enviados por los procesos para el balance de carga.

- La incorporación de un parámetro *LNPI (Limite de Nodos por Iteración),* que indica la cantidad límite de nodos a procesar por iteración del algoritmo. En este sentido, la versión difiere del algoritmo original el cual expande un único nodo por iteración.

El trabajo experimental fue realizado con distintas instancias del 15-Puzzle, utilizando la heurística antes mencionada, sobre una máquina multicore y sobre un cluster de multicore, variando los valores de los parámetros LNPI y LNPT, y en consecuencia se analiza el efecto de los mismos sobre el rendimiento. Cabe destacar que en (Kishimoto, et al., 2013) no realizan el análisis del parámetro LNPT cuando se ejecuta el algoritmo sobre una máquina multicore. Por último, se evaluó la escalabilidad del algoritmo sobre ambas arquitecturas.

Para finalizar, la tesis presenta una comparación de la memoria consumida por los algoritmos HDA* para memoria compartida y HDA* para memoria distribuida, y del rendimiento alcanzado (speedup y eficiencia) cuando se los ejecuta sobre una máquina multicore con distintas instancias iniciales del problema; estos análisis presentan originalidad en el área. Los resultados arrojados indican que se obtendría un beneficio al convertir HDA* en una aplicación híbrida, cuando la arquitectura subyacente es un cluster de multicore, por lo que se sientan las bases para éste algoritmo híbrido.

1.3 Publicaciones derivadas

Los resultados parciales de esta tesis fueron publicados en los siguientes *congresos con referato internacional*:

- *On the Optimization of HDA* for Multicore Machines. Performance Analysis.*
  Victoria Sanz, Armando De Giusti, Marcelo Naiouf.
  Proceedings of The 2014 International Conference on Parallel and Distributed Processing Techniques and Applications.
  Las Vegas, Nevada, USA.
  Julio de 2014.
  ISBN: 1-60132-282-8.

- *Ajuste de rendimiento del algoritmo HDA* para máquinas multicore.*
  Victoria Sanz, Armando De Giusti, Marcelo Naiouf.
  Proceedings del XX Congreso Argentino de Ciencias de la Computación (CACIC 2014)
  San Justo, Buenos Aires, Argentina.
  Octubre de 2014.
  ISBN: 978-987-3806-05-6

A continuación, se citan publicaciones de los mismos autores referidas a algoritmos de búsqueda paralela sobre grafos, siguiendo la misma temática, que están relacionados con esta tesis:

En Revistas con Referato Internacional

- *4-(N^2-1) Puzzle: Parallelization and performance on clusters.*
  Sanz V., De Giusti A., Naiouf M.
  Journal of Computer Science and Technology.
  Iberoamerican Science & Technology Education Consortium (ISTEC) - ISTEC Executive Offices – University of New Mexico – Albuquerque –USA
  Junio 2010.
  Vol. 10 Nro 2, pp. 86-90.
  ISSN: 1666-6046.

- *Análisis de Performance en el procesamiento paralelo sobre clusters del Problema del Puzzle N^2 -1.*
  Sanz Victoria, Chichizola Franco, De Giusti Armando, Naiouf Marcelo, De Giusti Laura.
  Mecánica Computacional. High Performance Computing in Computational Mechanics (B).
  Asociación Argentina de Mecánica Computacional
  Vol. XXVII. Num. 39. pp. 2967-2978.
  ISSN: 1666-6070.

En Congresos con Referato Internacional

- *Superlinearidad sobre Clusters. Análisis experimental en el problema del Puzzle N^2 -1.*
  Victoria Sanz, Franco Chichizola, Marcelo Naiouf, Laura De Giusti, Armando De Giusti.
  Proceedings del XIII Congreso Argentino de Ciencias de la Computación.
  Facultad de Ciencias Exactas y Naturales y Agrimensura Universidad Nacional del Nordeste, Corrientes, Argentina – Facultad Regional Resistencia, Universidad tecnológica nacional, Resistencia, Argentina.
  Octubre de 2007.
  ISBN: 978-950-656-109-3.

- *Parallel Processing Puzzle N^2 -1 on Cluster Architectures. Performance Analysis.*
  Victoria Sanz, Armando De Giusti, Franco Chichizola, Marcelo Naiouf, Laura De Giusti.
  Proceedings of The 30th International Conference on Information Technology Interfaces.
  Cavtat, Dubrovnik, Croacia.
  Junio de 2008.
  ISBN: 978-953-7138-12-7

- *Análisis de Performance en el procesamiento paralelo sobre Clusters del problema del Puzzle N^2-1.*
  Victoria Sanz, Franco Chichizola, Armando De Giusti, Marcelo Naiouf, Laura De Giusti.
  Proceedings del XVII Congreso sobre Métodos Numéricos y sus Aplicaciones.
  Departamento de Matemáticas, Facultad de Cs Exactas. Universidad Nacional de San Luis – UNSL, San Luis, Argentina.
  Noviembre de 2008.
  ISSN: 1666-6070.

- *4-(N^2-1) Puzzle: parallelization and performance on clusters.*
  Victoria Sanz, Armando De Giusti, Marcelo Naiouf.
  Proceedings del XV Congreso Argentino de Ciencias de la Computación).
  Facultad de Ingeniería, Universidad Nacional de Jujuy, Jujuy, Argentina.
  Octubre de 2009.
  ISBN: 978-897-24068-4-1.

- *Parallel Optimal and Suboptimal Heuristic Search on multicore clusters.*
  Victoria Sanz, Marcelo Naiouf, Armando De Giusti.
  Proceedings of The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications).
  Las Vegas, Nevada, USA.
  Julio de 2011.
  ISBN: 1-60132-193-7

- *Parallel Suboptimal Heuristic Search for finding a w-admissible solution. Performance analysis.*
  Victoria Sanz, Marcelo Naiouf, Armando De Giusti.
  Proceedings of The International Conference on COMPUTERS, DIGITAL COMMUNICATIONS and COMPUTING (ICDCC'11)
  Barcelona, España
  Septiembre de 2011.
  ISBN: 978-1-61804-030-5

CAPÍTULO 2: ALGORITMOS SECUENCIALES DE BÚSQUEDA

La Inteligencia Artificial combina un amplio número de áreas de la ciencia para automatizar actividades que requieren pensamiento humano permitiendo así crear máquinas que realicen tareas, idealmente en forma óptima, en las cuales se requiere inteligencia.

Para alcanzar dicho objetivo las distintas disciplinas de la Inteligencia Artificial trabajan en conjunto para diseñar *agentes* los cuales perciben el ambiente a través de sensores, transforman a partir de un programa lo percibido (entrada) en acciones a realizar (salida), y luego ejecutan dichas acciones a través de actuadores. (Russel & Norvig, 2003)

Un tipo de programa agente de gran importancia es aquel que debe determinar una secuencia de acciones para alcanzar un estado objetivo o meta particular a partir de un estado inicial. Este proceso conocido como *búsqueda* se realizará sobre un *grafo* que incluye todos los *estados* alcanzables desde el estado inicial aplicando *acciones*, para esto previamente se debe realizar una abstracción limitando las acciones que el agente puede realizar y los estados que se tendrán en cuenta.

En este capítulo se describen los pasos para definir el problema a resolver aplicando un proceso de abstracción.

A continuación, se estudian los algoritmos de búsqueda de propósito general más utilizados los cuales se dividen en dos grandes grupos: algoritmos de búsqueda no informada y algoritmos de búsqueda informada. Éstos se diferencian entre sí en que los últimos utilizan información específica del problema para guiarse en la búsqueda con el fin de encontrar soluciones en forma más eficiente. En particular, de los algoritmos de búsqueda informada se analizarán aquellos que encuentran soluciones de costo óptimo.

Para finalizar, se comparan las ventajas y desventajas de los algoritmos expuestos con el objetivo de seleccionar el más apropiado para resolver el problema propuesto como caso de estudio.

2.1 Construcción del espacio de estados del problema

Un problema se puede definir mediante tres componentes básicos (Nilsson, 1998):

- *Estado inicial*[1]*:* es el estado desde el que se comienza la búsqueda. Representa el problema a resolver.

[1] En la literatura se utiliza terminología variada para referirse a estados y acciones: los estados también pueden ser llamados configuraciones y las acciones pueden ser llamadas movimientos u operadores. Se utilizarán estos términos como sinónimos de aquí en adelante.

- *Conjunto finito de acciones*[1]*:* la aplicación de una acción sobre un estado genera un nuevo estado. No todas las acciones se pueden aplicar a un estado en particular, por lo que llamaremos *acciones legales* al conjunto de acciones cuya aplicación está permitida sobre un estado particular.

- *Estado final:* es el estado que se desea alcanzar. Puede ser único o pueden haber varios estados objetivo que tengan propiedades en común las cuales se pueden describir a través de una condición.

El *espacio de estados del problema* es el conjunto de todos los estados alcanzables desde el estado inicial a partir de la aplicación de acciones legales y se representa a partir de un grafo G=(V,E,s,T) donde: V es el conjunto de todos los estados alcanzables a partir del estado inicial, T es el conjunto de los estados finales, s es el estado inicial y E es el conjunto de aristas que conectan dos estados en el grafo (si existe una arista conectando los estados *u* y *v,* debe existir una acción que cuando es aplicada al estado *u* genera el estado *v*). Cada arista del grafo puede ser etiquetada por la acción que representa. El espacio de estados puede estar estructurado como un árbol si cada estado se puede alcanzar a través de un único camino o como grafo general en caso de existir ciclos; el segundo caso incluye a los espacios de estados de problemas donde las acciones son reversibles.

Para resolver el problema se debe encontrar un camino en el grafo que conecte al estado inicial con el estado final, siendo la solución encontrada la secuencia ordenada de acciones aplicadas.

La calidad de la solución se mide teniendo en cuenta una función *w* que asigna un costo a cada acción. El costo de un camino *g* se calcula sumando los costos individuales de las acciones aplicadas a lo largo del mismo. Una solución óptima es aquella cuyo costo es menor al de todas las soluciones posibles del problema.

2.2 Algoritmos de búsqueda

Los algoritmos de búsqueda a tratar trabajan sobre *grafos de búsqueda implícitos* cuyos nodos se van generando dinámicamente a medida que el proceso de búsqueda avanza y en todo momento la única parte accesible del mismo es aquella que fue generada. Estos algoritmos se diferencian de aquellos que trabajan sobre *grafos de búsqueda explícitos,* en los cuales la estructura *grafo* es accesible en forma completa en todo momento del proceso de búsqueda y en general se representa con una matriz de adyacencia o lista de adyacencia, lo cual se torna prohibitivo cuando el espacio de estados del problema es muy grande.

Antes de comenzar la descripción del algoritmo de búsqueda es importante realizar las siguientes aclaraciones sobre la terminología a utilizar:

- Un *estado* es una representación de los elementos que describen el problema en un momento dado.
- Un *nodo* del grafo de búsqueda es una *estructura de datos* que reúne la información que representa al estado e información adicional necesaria para el proceso de búsqueda:

- *Puntero al nodo padre:* ayuda a conocer el camino por el cual se alcanzó el nodo.
- *Profundidad:* representa el número de acciones aplicadas para llegar al nodo.
- *Costo del camino:* indica el costo del camino desde el nodo inicial al nodo en cuestión *n*, y se denota como *g(n).*
- *Acción aplicada:* es la acción que se aplicó al estado que representa el nodo padre para generar el estado que está representando el nodo en cuestión y servirá para evitar aplicar la acción inversa que regenere el estado padre cuando el proceso de búsqueda trabaje sobre este nodo.

- El *espacio de estados del problema* es un grafo donde cada estado posible del problema aparece una única vez.
- El *grafo de búsqueda* – que tendrá forma de árbol - puede tener varios nodos distintos que representen al mismo estado y que fueron alcanzados por distintos caminos, esto ocurre en caso que el espacio de estados este estructurado como un grafo general, y será responsabilidad del algoritmo de búsqueda evitar procesarlos varias veces mediante estrategias conocidas como *detección de duplicados,* ya que en caso contrario el árbol generado podría tornarse infinito.

El proceso de búsqueda comenzará creando un *nodo raíz* que representa al estado inicial. En cada iteración del algoritmo se selecciona un *nodo* entre aquellos que fueron generados y no han sido procesados, el cual se *expande* aplicando acciones legales sobre el estado *generando* nodos sucesores. Así se continúa hasta que el nodo seleccionado sea aquel que represente al estado final o se hayan procesado todos los nodos generados. Una vez encontrado el *nodo final* se recupera la secuencia de acciones que se aplicaron para llegar a él siguiendo los punteros al nodo padre de cada nodo en el camino.

La elección del nodo a procesar en cada iteración la determina la *estrategia de búsqueda*, la cual influye en la calidad de solución a obtener y la cantidad de nodos a procesar. Por ejemplo: se puede seleccionar siempre el nodo no procesado que fue generado más recientemente (estrategia conocida como *Depth First*) o el nodo generado menos recientemente (estrategia *Breadth First*).

El conjunto de nodos que han sido generados a lo largo de la búsqueda se divide en dos grupos y constituyen *estructuras de datos* sobre las cuales operará el algoritmo:

- Los nodos que han sido generados pero no han sido expandidos, es decir *nodos hoja* cuyos sucesores no han sido generados, se almacenan en una estructura llamada *Lista Abierta*[2] *o Frontera*[3]. La implementación de esta estructura dependerá de la estrategia de búsqueda seleccionada.
- Los nodos que han sido generados y ya fueron expandidos se almacenan en una estructura llamada *Lista Cerrada*[2]. Estos nodos se deben almacenar para la mayoría de las estrategias de búsqueda para posibilitar obtener la secuencia de acciones que condujo al nodo que

[2] El término *Lista* no se refiere a la implementación de la estructura de datos sino que es un legado de la literatura de Inteligencia Artificial.
[3] En Inglés se utilizan los términos *Fringe o Frontier*

representa el estado final. Para los problemas cuyo espacio de estados está estructurado como un grafo general, su implementación tradicional es a través de una *Tabla Hash* ya que la *Lista Cerrada* servirá para realizar la *detección de duplicados*, es decir verificar si un estado ya fue procesado, permitiendo búsquedas e inserciones eficientes.

Algoritmo Búsqueda-Árbol
Precondición: *espacio de estados estructurado como árbol*
Entrada: *nodo inicial s*
Salida: *nodo solución t o NULL*

```
t ← NULL
Lista Cerrada ← ϕ                          {Inicializar la Lista Cerrada en vacío}
Lista Abierta ← {s}                        {Insertar s en la Lista Abierta, inicialmente vacía}
while (Lista Abierta ≠ ϕ) and (t = NULL)   {Siempre y cuando haya nodos en Lista Abierta
                                            y no se ha encontrado la solución}
    u ← Eliminar nodo de Lista Abierta     {Seleccionar un nodo de la Lista Abierta y eliminarlo}
    Insertar u en Lista Cerrada            {Almacenar el nodo en la Lista Cerrada}
    if (EsSolucion(u))                     {Verificar si el estado cumple con las propiedades del estado final}
        t:= u                              {Se ha encontrado la solución}
    else
        hijos ← Expandir(u)                {Generar los nodos sucesores aplicando acciones legales}
        for each v in hijos                {Para cada sucesor v de u, insertarlo en la Lista Abierta }
            Insertar v en Lista Abierta
return t                                   {Retornar NULL si la solución no existía.}
                                           {Caso contrario retornar el puntero al nodo final}
```

Figura 2.1. Algoritmo de búsqueda general sobre espacio de estados estructurado como árbol.

Algoritmo Búsqueda-Grafo
Entrada: *nodo inicial s*
Salida: *nodo solución t o NULL*

```
t ← NULL
Lista Cerrada ← ϕ                          {Inicializar la Lista Cerrada en vacío}
Lista Abierta ← {s}                        {Insertar s en la Lista Abierta, inicialmente vacía}
while (Lista Abierta ≠ ϕ) and (t = NULL)   {Siempre y cuando haya nodos en Lista Abierta
                                            y no se ha encontrado la solución}
    u ← Eliminar nodo de Lista Abierta     {Seleccionar un nodo de la Lista Abierta y eliminarlo}
    Insertar u en Lista Cerrada            {Almacenar el nodo en la Lista Cerrada}
    if (EsSolucion(u))                     {Verificar si el estado cumple con las propiedades del estado final}
        t:= u                              {Se ha encontrado la solución}
    else
        hijos ← Expandir(u)                {Generar los nodos sucesores aplicando acciones legales}
        for each v in hijos                {Para cada sucesor v de u}
            DeteccionDuplicados(u,v, Lista Abierta, Lista Cerrada)  {Invocar al algoritmo para detectar ciclos}
return t                                   {Retornar NULL si la solución no existía.}
                                           {Caso contrario retornar el puntero al nodo final}
```

Figura 2.2. Algoritmo de búsqueda general sobre espacio de estados estructurado como grafo.

Algoritmo DetecciónDuplicados
Entrada: *nodo u, nodo v, Lista Abierta, Lista Cerrada*
Efectos laterales: *se actualiza la Lista Abierta y/o Lista Cerrada*
if (v no está en la Lista Cerrada) and (v no está en la Lista Abierta) *{El estado que representa v no ha sido generado}*
    Insertar v en Lista Abierta

Figura 2.3. Algoritmo básico de Detección de Duplicados

La Figura 2.1 muestra un algoritmo general de búsqueda adecuado para espacios de estados estructurados como árbol. Se comienza con la Lista Cerrada vacía, a ser utilizada por aquellas estrategias que requieran conservar todos los nodos expandidos, y con la Lista Abierta almacenando únicamente el nodo inicial. El algoritmo iterará seleccionando un nodo de la Lista Abierta (este paso es dependiente de la *estrategia de búsqueda*), a continuación lo inserta en la Lista Cerrada y verifica si el mismo representa el estado final; en ese caso retorna el puntero al nodo final encontrado, y en caso contrario lo expande generando sus sucesores, insertando cada uno en la Lista Abierta. Cuando no existe una solución para el problema, en algún momento la Lista Abierta quedará vacía y el algoritmo retornará NULL.

La Figura 2.2 adapta el algoritmo anterior para los casos en que el espacio de estados tenga forma de grafo general. La única diferencia reside en la manera en que se procesan los nodos generados recientemente: antes de insertarlos en la Lista Abierta se verifica si el estado que representa el *nodo sucesor* no había sido generado con anterioridad a través del algoritmo de *Detección de Duplicados* de la Figura 2.3. En caso de haber coincidencia el algoritmo detectó dos caminos hacia el mismo estado y se puede descartar uno. Cabe destacar que este algoritmo no encontrará soluciones óptimas ya que siempre descarta caminos encontrados en segundas instancias.

En la Sección 2.2.1 se exponen las propiedades que caracterizarán a los algoritmos de búsqueda y las medidas de complejidad a tener en cuenta. Luego en la Sección 2.2.2 se analizan los algoritmos de búsqueda no informada, y para finalizar en la Sección 2.2.3 se estudian los algoritmos de búsqueda informada.

2.2.1 Propiedades y complejidad algorítmica

Los algoritmos de búsqueda que se estudian en las secciones posteriores poseen diferentes características las cuales serán factores importantes a la hora de elegir el algoritmo adecuado para resolver un problema particular:

- *Completitud*: un algoritmo es *completo* si encuentra la solución al problema en caso de que la misma exista.
- *Admisibilidad*: un algoritmo es *admisible* si encuentra la mejor solución según el criterio de costo que se elija.
- *Complejidad temporal:* generalmente se mide en términos de la cantidad de nodos generados, aunque a veces se expresa en términos de la cantidad de nodos expandidos. Para ambos casos se deben identificar algunos de los siguientes datos: el *factor de*

ramificación b, es decir el máximo número de sucesores de un estado; la *profundidad del nodo solución d;* la *longitud máxima de cualquier camino m* del espacio de estados*; y* el *costo c de la solución*.

- *Complejidad espacial:* se mide teniendo en cuenta algunos de los factores comentados en el inciso anterior.

2.2.2 Algoritmos de búsqueda no informada

Los algoritmos de búsqueda no informada encuentran soluciones sistemáticamente seleccionando un nodo, verificando si el mismo representa el estado final y en caso de no serlo generando sus sucesores, es decir no utilizan información adicional de los estados más que aquella provista por la definición del problema siendo ineficientes cuando el espacio de estados del problema es muy grande.

Las distintas *estrategias de búsqueda* que se exponen a continuación difieren entre sí en el orden en el cual los nodos son seleccionados para su procesamiento.

Para explicar la ejecución de los algoritmos se utilizarán los espacios de estados que se muestran en la Figura 2.4.a y en la Figura 2.4.b; en ambos los estados solución son *G1* y *G2*.

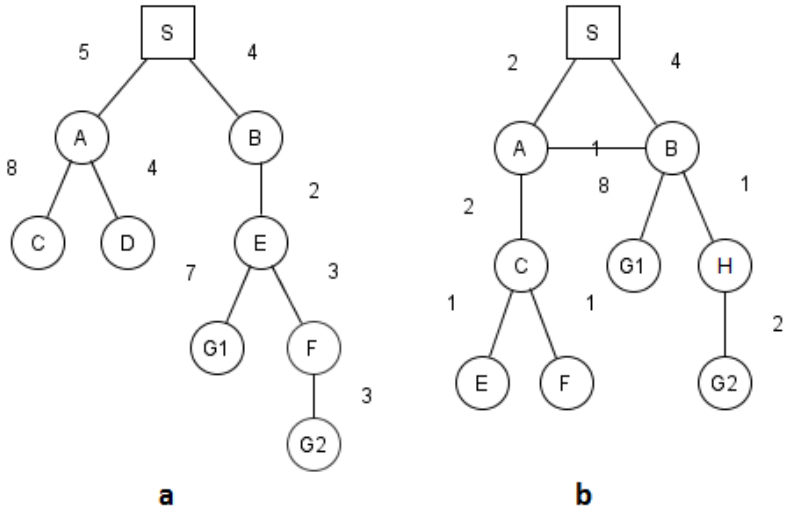


Figura 2.4.a. Espacio de estados estructurado como un árbol.
b. Espacio de estados estructurado como un grafo general.

2.2.2.1 Estrategia Depth-First

En su versión más simple el algoritmo *Depth-First Search (DFS)* (Russel & Norvig, 2003) (Nilsson, 1998) (Edelkamp & Schrödl, 2011) comienza a evolucionar nodos a partir del nodo inicial. En cada iteración selecciona de la Lista Abierta el nodo generado más recientemente para su expansión, es decir uno de los nodos más profundos. De esta manera la búsqueda progresa por una rama hasta seleccionar un nodo sin sucesores, momento en el cual se

selecciona un nuevo nodo de la Lista Abierta que tenga la misma profundidad (*nodo hermano)*, y en caso de no existir retrocede al nivel anterior seleccionando un nodo hermano de su padre.

La implementación más adecuada para la Lista Abierta es una *Pila* (Aho, et al., 1983) ya que permite insertar y eliminar nodos generados recientemente en orden constante. La inserción de un nodo generado es realizada a través de la operación *Push,* mientras que la selección y eliminación es mediante la operación *Pop*.

La Figura 2.5 muestra el árbol de búsqueda generado por el algoritmo *Búsqueda-Árbol* estudiado en la Sección 2.2 para el espacio de estados de la Figura 2.4.a cuando se utiliza la estrategia de selección *Depth-First*; la numeración en los nodos representa el orden en que son seleccionados para su expansión y los nodos de color gris son aquellos que se generaron pero no se expandieron. La Tabla 2.1 muestra los pasos realizados durante la ejecución del algoritmo; entre paréntesis al lado de cada nodo se denota su nodo padre y el costo del camino desde el nodo inicial.

De lo anterior se puede notar que el algoritmo se detiene cuando encuentra por primera vez un nodo que representa el estado final, en este caso la solución encontrada tiene costo 13 cuando existía una solución de costo menor, es decir no encuentra la solución óptima. Cabe destacar que esta estrategia aplicada sobre espacios de estados con forma de árbol no requiere almacenar todos los nodos expandidos, sino que alcanza con conservar aquellos que estén en el camino que actualmente está siendo procesado para poder recuperar la secuencia de acciones una vez encontrado el nodo final. Una optimización sencilla en el algoritmo consiste en expandir parcialmente el nodo seleccionado generando sólo un sucesor por vez; esta modificación requerirá que el nodo expandido se conserve en la Lista Abierta para permitir generar un nuevo sucesor en caso que la búsqueda por el camino anterior no sea exitosa, y de esta manera se puede evitar mantener la Lista Cerrada.

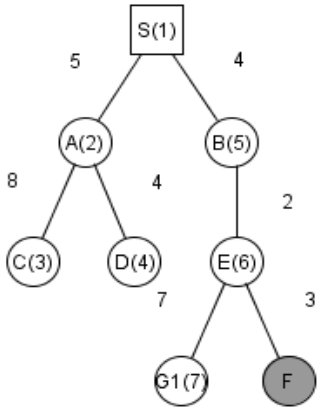


Figura 2.5. Árbol de búsqueda generado para el espacio de estados árbol ejemplo, utilizando Depth-First con el algoritmo *Búsqueda-Árbol.*

Tabla 2.1 Ejecución del algoritmo *Búsqueda-Árbol* con la estrategia *Depth-First* para el espacio de estados árbol ejemplo.

Pasos algoritmo *Búsqueda-Árbol*	Selección	Lista abierta	Lista cerrada	Nota
1		{s(-,0)}	{}	
2	s(-,0)	{A(s,5), B(s,4)}	{ s(-,0)}	
3	A(s,5)	{C(A,13), D(A,9), B(s,4)}	{ s(-,0), A(s,5)}	
4	C(A,13)	{D(A,9), B(s,4)}	{ s(-,0), A(s,5) , C(A,13)}	
5	D(A,9)	{B(s,4)}	{ s(-,0), A(s,5), D(A,9)}	
6	B(s,4)	{E(B,6)}	{ s(-,0), B(s,4)}	
7	E(B,6)	{G1(E,13), F(E,9)}	{ s(-,0), B(s,4), E(B,6)}	
8	G1(E,13)	{F(E,9)}	{s(-,0), B(s,4), E(B,6), G1(E,13)}	*Se encontró la solución. Termina el algoritmo*

La Figura 2.6.a muestra el árbol de búsqueda infinito generado por el algoritmo *Búsqueda-Árbol* utilizando la estrategia de selección *Depth-First* con el espacio de estados de la Figura 2.4.b; resultado de no aplicar detección de duplicados, en este caso el algoritmo se queda atrapado en un ciclo sin encontrar la solución cuando la misma existe. La Figura 2.6.b muestra el árbol de búsqueda generado por el algoritmo *Búsqueda-Grafo* para el mismo espacio de estados, mientras que en la Tabla 2.2 se detallan los pasos realizados durante su ejecución (entre paréntesis al lado de cada nodo generado se indica cuál es su padre y el costo del camino). Es importante aclarar que para este último caso es necesario almacenar todos los nodos expandidos para posibilitar la detección de duplicados. Al igual que en el caso anterior, la solución encontrada no es la de costo óptimo.

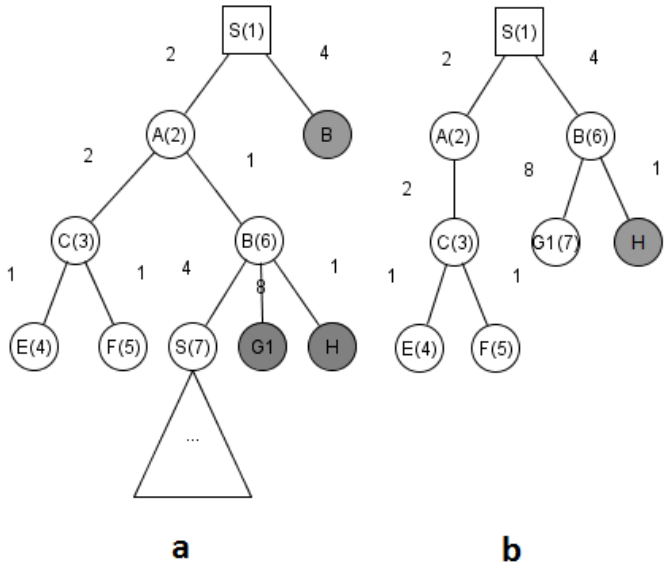


Figura 2.6.a. Árbol de búsqueda infinito generado para el espacio de estados grafo de ejemplo, utilizando *Depth-First* con *Búsqueda-Árbol* **b.** Árbol de búsqueda generado utilizando *Depth-First* con *Búsqueda-Grafo* para dicho espacio de estados.

Tabla 2.2. Ejecución del algoritmo Búsqueda-Grafo con la estrategia Depth-First sobre el espacio de estados grafo de ejemplo.

Pasos Algoritmo *Búsqueda-Grafo*	Selección	Lista Abierta	Lista Cerrada	Nota
-		{s(-,0)}	{}	
1	s(-,0)	{A(s,2), B(s,4)}	{s(-,0)}	
2	A(s,2)	{C(A,4),B(s,4)}	{s(-,0), A(s,2)}	Regenera B pero está en la Lista Abierta (*duplicado*)
3	C(A,4)	{E(C,5),F(C,5),B(s,4)}	{s(-,0), A(s,2), C(A,4)}	
4	E(C,5)	{F(C,5),B(s,4)}	{s(-,0), A(s,2), C(A,4), E(C,5)}	
5	F(C,5)	{B(s,4)}	{s(-,0), A(s,2), C(A,4), E(C,5), F(C,5)}	
6	B(s,4)	{G1(B,12),H(B,5)}	{s(-,0), A(s,2), C(A,4), E(C,5), F(C,5), B(s,4)}	Regenera A pero está en Lista Cerrada (*duplicado*)
7	G1(B,12)	{H(B,5)}	{s(-,0), A(s,2), C(A,4), E(C,5), F(C,5), B(s,4), G1(B,12)}	*Se encontró la solución. Termina el algoritmo*

El algoritmo es *completo* siempre y cuando el espacio de estados sea finito (no existen caminos infinitos) y se utilice el algoritmo *Búsqueda-Grafo* si el espacio de estados está estructurado como grafo general.

Como se explicitó en ejemplos anteriores, esta estrategia no garantiza que el nodo final encontrado represente una solución óptima, es decir *no es admisible*.

Con respecto a la *complejidad temporal*, en el peor caso el algoritmo generará $O(b^m)$ nodos. La desventaja de esta estrategia es que puede tomar una mala decisión al explorar caminos muy largos a pesar que el nodo final esté cercano a la raíz. (Russel & Norvig, 2003)

Respecto a la *complejidad espacial*, el algoritmo *Búsqueda-Árbol* en conjunto con la estrategia *Depth-First* requiere almacenar sólo los nodos expandidos que están en el camino siendo procesado actualmente junto con sus hermanos que fueron generados y no expandidos (en caso de no utilizar expansiones parciales) siendo como máximo $b \times m + 1$ nodos; por otra parte el algoritmo *Búsqueda-Grafo* requiere almacenar todos los nodos generados siendo en total $O(b^m)$. (Russel & Norvig, 2003)

2.2.2.2 Estrategia Breadth-First

La estrategia Breadth-First (Russel & Norvig, 2003) (Nilsson, 1998) (Edelkamp & Schrödl, 2011) permite realizar un recorrido del espacio de estados por niveles: primero se selecciona y expande el *nodo inicial*, luego sus sucesores y luego los sucesores de éstos, y así continua hasta alcanzar el nodo final. Cada vez que se selecciona un nodo de la Lista Abierta, todos los nodos hermanos anteriores fueron expandidos, y también todos los nodos de niveles previos.

La implementación más adecuada para la Lista Abierta es una *Cola* (Aho, et al., 1983): la inserción de un nodo generado se realizará a través de la operación *Enqueue,* mientras que la selección y eliminación se realizará mediante la operación *Dequeue,* siendo ambas de orden constante.

La Figura 2.7 muestra el árbol de búsqueda generado por el algoritmo *Búsqueda-Árbol* utilizando la estrategia de selección *Breadth-First* con el espacio de estados de la Figura 2.4.a; la numeración en los nodos representa el orden en que son seleccionados para su expansión y los nodos de color gris son aquellos que se generaron pero no se expandieron. La Tabla 2.3 detalla los pasos realizados durante la ejecución del algoritmo. La solución encontrada por esta estrategia no posee costo óptimo pero, a diferencia de *Depth-First,* garantiza que su longitud de camino sea óptima.

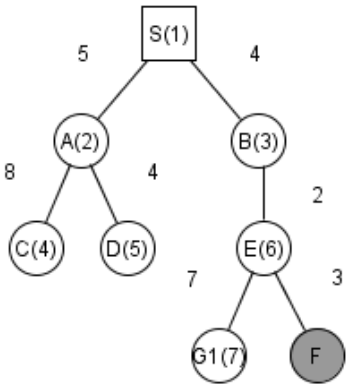


Figura 2.7. Árbol de búsqueda generado cuando se utiliza el algoritmo Búsqueda-Árbol sobre el espacio de estados árbol de ejemplo.

Tabla 2.3. Ejecución del algoritmo Búsqueda-Árbol con la estrategia Breadth-First sobre el espacio de estados árbol de ejemplo.

Paso Algoritmo *Búsqueda-Árbol*	Selección	Lista Abierta	Lista Cerrada	Nota
0		{s(-,0)}	{}	
1	s(-,0)	{A(s,5), B(s,4)}	{s(-,0)}	
2	A(s,5)	{B(s,4), C(A,13), D(A,9)}	{s(-,0), A(s,5)}	
3	B(s,4)	{C(A,13), D(A,9), E(B,6)}	{s(-,0), A(s,5), B(s,4)}	
4	C(A,13)	{D(A,9), E(B,6)}	{s(-,0), A(s,5), B(s,4), C(A,13)}	
5	D(A,9)	{E(B,6)}	{s(-,0), A(s,5), B(s,4), C(A,13), D(A,9)}	
6	E(B,6)	{G1(E,13),F(E,9)}	{s(-,0), A(s,5), B(s,4), C(A,13), D(A,9), E(B,6)}	
7	G1(E,13)	{F(E,9)}	{s(-,0), A(s,5), B(s,4), C(A,13), D(A,9), E(B,6), G1(E,13)}	*Se encontró la solución. Termina el algoritmo*

La Figura 2.8.a muestra el árbol de búsqueda generado por el algoritmo *Búsqueda-Árbol* utilizando la estrategia *Breadth-First* con el espacio de estados de la Figura 2.4.b. Resultado de no aplicar *detección de duplicados*, el algoritmo eventualmente encontrará la solución pero procesará una cantidad de nodos duplicados que puede ser significativa, aumentando también la cantidad de nodos generados. La Figura 2.8.b muestra el árbol de búsqueda generado por el

algoritmo *Búsqueda-Grafo* para el mismo espacio de estados, mientras que en la Tabla 2.4 se detallan los pasos realizados durante su ejecución. Al igual que en el caso anterior, la solución encontrada no es de costo óptimo, pero si es óptima su longitud.

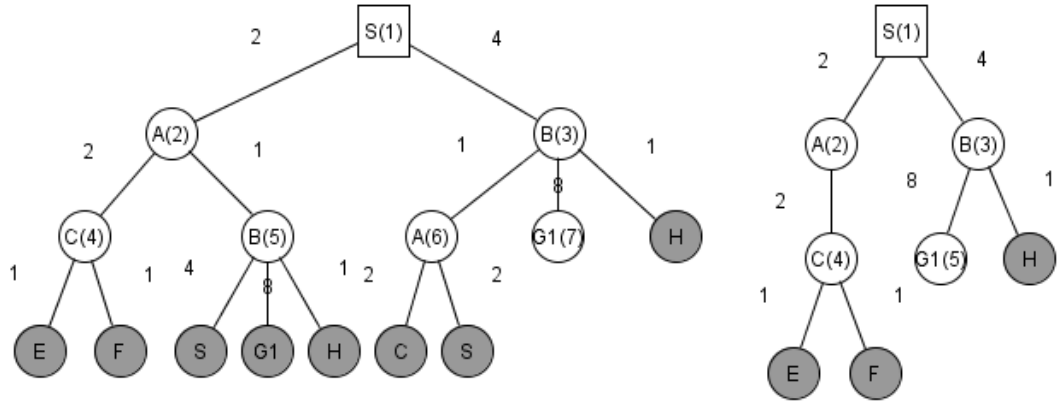


Figura 2.8.a. Árbol de búsqueda generado por *Búsqueda-Árbol* aplicando Breadth-first (eventualmente encuentra la solución) para el espacio de estados *grafo* de ejemplo **b.** Árbol de búsqueda generado por Búsqueda-Grafo para el mismo espacio de estados.

Tabla 2.4. Ejecución del algoritmo Búsqueda-Grafo con la estrategia *Breadth-First* sobre el grafo de ejemplo.

Paso Algoritmo *Búsqueda-Grafo*	Selección	Lista abierta	Lista cerrada	Nota
0		{s(-,0)}	{}	
1	s(-,0)	{A(s,2), B(s,4)}	{s(-,0)}	
2	A(s,2)	{B(s,4),C(A,4)}	{s(-,0),A(s,2)}	Regenera B pero es duplicado
3	B(s,4)	{C(A,4),G1(B,12), H(B,5)}	{s(-,0), A(s,2), B(s,4)}	Regenera A pero es duplicado
4	C(A,4)	{G1(B,12), H(B,5), E(C,5), F(C,5)}	{s(-,0), A(s,2), B(s,4), C(A,4)}	
5	G1(B,12)	{H(B,5), E(C,5), F(C,5)}	{s(-,0), A(s,2), B(s,4), C(A,4), G1(B,12)}	*Se encontró la solución. Termina el algoritmo*

La estrategia *Breadth-First* siempre encuentra la solución en caso de existir, ya sea utilizando el algoritmo *Búsqueda-Árbol* o *Búsqueda-Grafo* sin importar cómo está estructurado el espacio de estados, por lo que siempre es *completo.*

Como se demostró en los ejemplos, el costo de la solución encontrada no es óptimo, por lo que el algoritmo *no es admisible* a menos que todas las acciones tengan costo idéntico, pero si es óptima la longitud del camino de la solución.

Con respecto a la *complejidad temporal*, la cantidad de nodos generados durante la búsqueda es $O(b^{d+1})$, mientras que la *complejidad espacial* es similar ya que se deben almacenar siempre

todos los nodos generados para poder recuperar la secuencia de acciones que condujo a encontrar el nodo final. (Russel & Norvig, 2003)

2.2.2.3 *Iterative Deepening Depth-First*

La estrategia Iterative Deepening Depth-First (Russel & Norvig, 2003) (Korf, 1985) combina los beneficios de *Depth-First* y *Breadth-First* para obtener un algoritmo con la complejidad espacial de la primera estrategia, pero evitando la exploración de caminos muy largos a pesar que el nodo final esté cercano a la raíz, tal y como se comporta la segunda estrategia mencionada.

El algoritmo realiza una serie de iteraciones, en cada una ejecuta una búsqueda siguiendo la estrategia *Depth-First* pero incluye un límite *l* correspondiente a la profundidad admitida para la rama siendo explorada actualmente: cuando se selecciona un nodo será expandido siempre y cuando no esté a profundidad *l*, de lo contrario se pasará a seleccionar el próximo nodo generado más recientemente. En la primera iteración el límite de profundidad es 1, y en cada iteración del algoritmo se incrementa en una unidad. El algoritmo finaliza cuando se encuentra la primera solución que puede no ser la óptima en costo pero, al igual que la estrategia *Breadth-First*, sí lo será en su longitud de camino. La Figura 2.9 ejemplifica el funcionamiento del algoritmo.

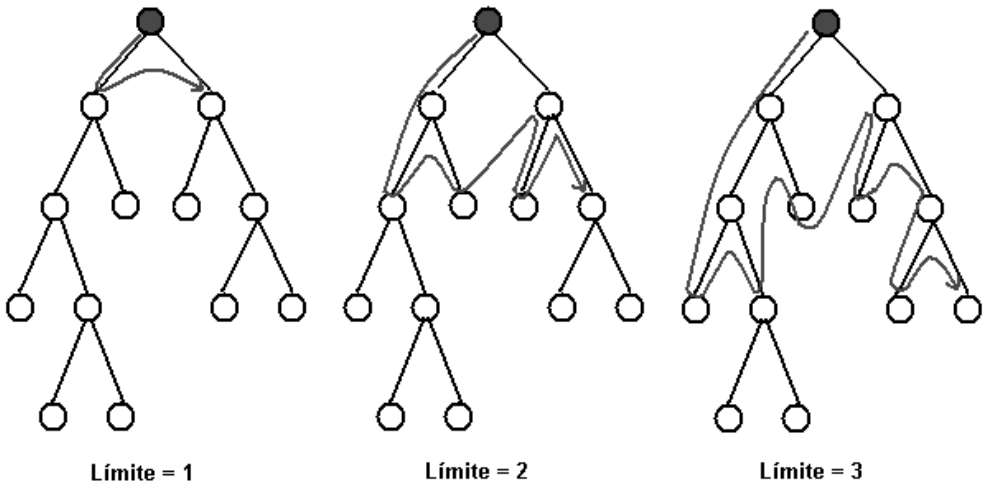


Figura 2.9. Etapas en la búsqueda con la estrategia Iterative Deepening Depth-First

El algoritmo de búsqueda, utilizando *Búsqueda-Árbol* o *Búsqueda-Grafo* en cada etapa, es *completo* ya que eventualmente se encontrará la solución, sin importar la forma del espacio de estados. La solución encontrada será óptima siempre y cuando las acciones tengan igual costo, en caso contrario el algoritmo *no será admisible*.

Para calcular la cantidad total de nodos generados se debe tener en cuenta que si el nodo final se encontró en la iteración con valor de límite *d*, cada nodo de un nivel *i* particular (1<=*i* <= *d*) se generó d-i+1 veces. Por lo anterior, el total de nodos generados en un nivel i particular entre todas las iteraciones del algoritmo es (d-i+1)×(b^i), donde b^i es la cantidad de nodos del nivel i, y por lo tanto el total de nodos generados por el algoritmo es $\sum_{i=1}^{d}(\mathrm{d}-\mathrm{i}+1)\times(b^i)$ causando

que la *complejidad temporal* sea $O(b^d)$. Al igual que *DFS*, la *complejidad espacial* es lineal en caso que no se realice detección de duplicados. (Korf, 1985).

2.2.2.4 Búsqueda de costo uniforme

La búsqueda de costo uniforme (Russel & Norvig, 2003) es una variante del algoritmo *Búsqueda-Grafo* la cual garantiza encontrar un camino de costo óptimo entre el nodo inicial y un nodo final siempre y cuando las acciones tengan costo no menor a una constante *c* positiva, de esta manera cuando se recorre un camino el costo *g* del mismo es creciente (se dice que *g es una función monótona)*. El algoritmo de búsqueda de caminos mínimos propuesto por Dijkstra (Dijkstra, 1959) es una especialización de la búsqueda de costo uniforme.

Para cumplir su objetivo, en cada paso selecciona el nodo abierto con menor costo de camino *g*. Dado que pueden existir múltiples caminos desde el estado inicial a un estado particular con distinto costo, se debe adaptar el algoritmo *Detección-Duplicados* para que en el caso que detecte un duplicado descarte el camino con costo mayor (Figura 2.10). Cabe destacar que cuando un nodo se selecciona para su expansión, se ha encontrado el camino de costo óptimo al mismo (por la condición de *monotonía de g*). Por lo anterior, el algoritmo de detección de duplicados no debe realizar ninguna acción en caso de encontrar que el estado que representa el nodo sucesor siendo evaluado está en la Lista Cerrada, de manera que sólo realizará correcciones de costo en caso de encontrar que éste se encuentra en la Lista Abierta con costo de camino mayor al nuevo camino encontrado debiendo reordenar la misma. (Edelkamp & Schrödl, 2011)

La implementación adecuada para la *Lista Abierta* es una *Minheap* (Aho, et al., 1983) ordenada por costo de camino *g:* la inserción de un nodo generado será a través de la operación *Insert*, la selección y eliminación se realizará mediante la operación *DeleteMin,* la corrección de costo de un nodo que está en la Lista Abierta es a través de la operación *DecreaseKey,* siendo todas de orden logarítmico. La operación de búsqueda por la representación del estado necesaria para la detección de duplicados debería recorrer toda la estructura; una alternativa es asociar a la *MinHeap* una *Tabla Hash,* que permita realizar búsquedas por representación del estado en orden constante.

Algoritmo Detección-Duplicados
Entrada: *nodo u, nodo v, Lista Abierta, Lista Cerrada*
Efectos laterales: *se actualiza la Lista Abierta y/o la Lista Cerrada*

```
if (v no está en la Lista Cerrada)
    if ( v no está en la Lista Abierta) {El estado que representa v no ha sido generado}
        g(v)←g(u)+w(u,v)
        Insert( Lista Abierta,v, g(v))
    Else                          {El estado que representa v está en Lista Abierta}
        If g(u) + w(u,v) < g(v)   {El nuevo camino es de menor costo, actualizo su costo en la Lista Abierta}
            g(v)←g(u)+w(u,v)
            Padre(v) ← u
            Decreasekey(Lista Abierta, v , g(v))
{Si el estado representado por v está en Lista Cerrada ya se había encontrado el camino óptimo hacia él}
```

Figura 2.10. Algoritmo de detección de duplicados para la Búsqueda de costo uniforme

La Figura 2.11 muestra el árbol de búsqueda generado por el algoritmo para el grafo espacio de estados de la Figura 2.4.b; por encima de cada nodo se muestra el costo del camino desde el nodo inicial s hasta el mismo, calculado a partir de sumar los costos de las acciones aplicadas a lo largo del camino. La Tabla 2.5 muestra los pasos realizados por el algoritmo, destacando entre paréntesis para cada nodo generado quién es su padre y el costo de su camino. Notar que el algoritmo expande nodos en orden creciente de costo g, por lo que la primera solución encontrada será la de costo óptimo. (Russel & Norvig, 2003)

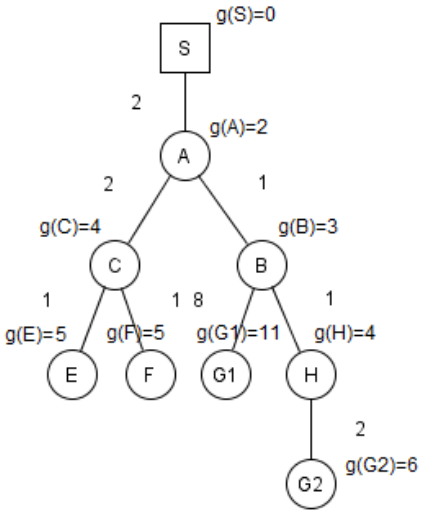


Figura 2.11. Árbol de búsqueda generado por el algoritmo Búsqueda de costo uniforme.

Tabla 2.5. Pasos ejecutados por el algoritmo Búsqueda de costo uniforme

Pasos Algoritmo *Búsqueda de costo uniforme*	**Selección**	**Lista abierta**	**Lista cerrada**	**Nota**
0		{s(-,0)}	{}	
1	s(-,0)	{A(s,2), B(s,4)}	{s(-,0)}	
2	A(s,2)	{ B(A,3), C(A,4)}	{s(-,0), A(s,2)}	Regenera B con costo de camino 3, es duplicado pero mejora el costo.
3	B(A,3)	{C(A,4), H(B,4), G1(B,11)}	{s(0,-), A(s,2), B(A,3)}	Regenera S, pero está en la lista cerrada.
4	C(A,4)	{H(B,4), E(C,5), F(C,5), G1(B,11)}	{s(0,-), A(s,2), B(A,3), C(A,4) }	
5	H(B,4)	{ E(C,5), F(C,5), G2(H,6), G1(B,11)}	{s(0,-), A(s,2), B(A,3), C(A,4), H(B,4) }	
6	E(C,5)	{ F(C,5), G2(H,6), G1(B,11)}	{s(0,-), A(s,2), B(A,3), C(A,4), H(B,4), E(C,5) }	
7	F(C,5)	{ G2(H,6), G1(B,11)}	{s(0,-), A(s, 2), B(A,3), C(A,4), H(B,4), E(C,5), F(C,5)}	
8	G2(H,6)	{G1(B,11)}	{s(0,-), A(s,2), B(A,3), C(A,4), H(B,4), E(C,5), F(C,5), G2(H,6)}	*Se encontró la solución. Termina el algoritmo*

El algoritmo es *completo* y *admisible* bajo las condiciones impuestas al grafo mencionadas al principio. Con respecto a la *complejidad temporal*, si el costo de la solución óptima es C*, la longitud de su camino no es mayor a C*/c, por lo que la cantidad de nodos generados por el algoritmo en el peor caso es $O(b^{1+\lfloor C^*/c \rfloor})$, siendo la *complejidad espacial* del mismo orden por tener que almacenar todos los nodos generados. (Russel & Norvig, 2003)

2.2.3 Algoritmos de búsqueda informada

Los algoritmos de búsqueda informada recurren a información específica del problema para reducir la cantidad de nodos a considerar, es decir estiman el costo del mejor camino a recorrer para transformar el estado representado por un nodo particular en el estado final y utilizan dicho valor para seleccionar el nodo que parece estar en el camino menos costoso. De esta manera, el algoritmo de búsqueda incorpora conocimiento dependiente del problema a resolver causando una pérdida de generalidad pero logrando una mejora en la complejidad temporal, lo que posibilita encontrar soluciones óptimas generando árboles de búsqueda más pequeños en comparación a la *búsqueda de costo uniforme* estudiada en la Sección 2.2.2.4.

La estrategia general es conocida como *Best-First Search (BFS),* instancia de las búsquedas generales estudiadas en la Sección 2.2 en la cual se asocia a cada nodo un valor dado por la función de costo $\hat{f}$ y en cada iteración selecciona el nodo con menor valor. Las variantes de *Best-First Search* se diferencian entre sí por la función de costo $\hat{f}$ que utilizan, causando que encuentren o no soluciones de costo óptimo.

En particular esta sección se concentrará en aquellas variantes de *BFS* que encuentran soluciones óptimas sobre espacios de estados estructurados como grafos generales, en las cuales la función de costo $\hat{f}$ está constituida por:

- *Función* $\hat{g}(n)$: calcula el costo del mejor camino conocido al momento desde el nodo inicial hasta *n*. Este costo puede ser reducido si el algoritmo encuentra un camino mejor hacia n, por lo que $\hat{g}(n) \geq g(n)$ donde g(n) es el costo real del camino mínimo entre dichos nodos.
- *Función heurística* $\hat{h}(n)$: a partir del estado que representa el nodo *n* particular, estima el costo del mejor camino desde *n* hacia un nodo final y debe cumplir la propiedad $\hat{h}(n) = 0$ si n es un nodo que representa un estado final y $\hat{h}(n) > 0$ en caso contrario. En general esta estimación es optimista, es decir $\hat{h}(n) \leq h(n)$ donde $h(n)$ es el costo real del camino mínimo desde *n* al nodo final, por lo que se dice es *admisible*. Una heurística *admisible* nunca sobreestima el costo real del mejor camino desde n hacia el nodo objetivo.

En su forma general la función $\hat{f}(n)$ estima el costo del camino desde el nodo inicial al nodo final pasando por n (Figura 2.12).

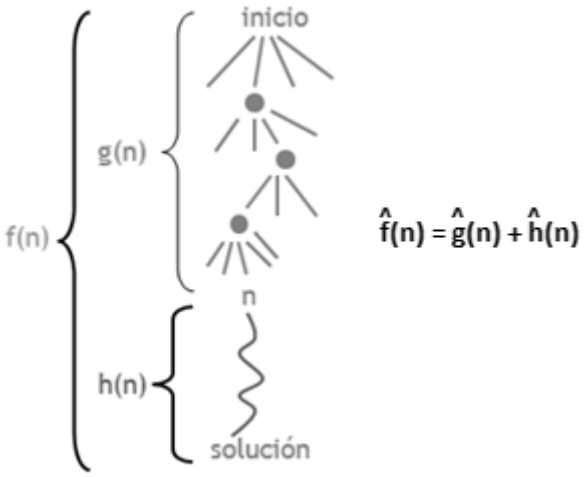


Figura 2.12. Forma general de la función de costo *f* utilizada por BFS.

La implementación adecuada para la *Lista Abierta* es una *MinHeap* ordenada por $\hat{f}$(n)*:* la inserción de un nodo generado será a través de la operación *Insert*, la selección y eliminación se realizará mediante la operación *DeleteMin,* la corrección de costo de un nodo que está en la Lista Abierta es a través de la operación *DecreaseKey,* siendo todas de orden logarítmico. La operación de búsqueda por la representación del estado necesaria para la detección de duplicados debería recorrer toda la estructura, una alternativa es asociar a la *MinHeap* una *Tabla Hash,* que permita realizar búsquedas por representación en orden constante.

*2.2.3.1 A**

A* (Hart, et al., 1968) es una variante de *BFS* que permite encontrar soluciones de costo mínimo siempre y cuando la función heurística $\hat{h}$ sea admisible, el costo de las acciones sea no menor a una constante c positiva y cada nodo tenga un número finito de sucesores. Además si la función $\hat{f}$ es monótona el algoritmo explora menor cantidad de nodos que cualquier otro algoritmo de búsqueda.

La Figura 2.13 muestra el algoritmo A*, que comienza con la Lista Cerrada vacía y con la Lista Abierta almacenando el nodo inicial s, con costo $\hat{f}(s) = \hat{h}(s)$ dado que $\hat{g}(s) = g(s) = 0$. Siempre y cuando la Lista Abierta no esté vacía y no se haya encontrado la solución: selecciona el nodo u con costo mínimo de la Lista Abierta, lo almacena en la Lista Cerrada y verifica si representaba al estado final, caso en el cual $\hat{h}(u) = h(u) = 0$ (por propiedad de la función heurística $\hat{h}$), y en caso contrario expandirá el nodo u generando nodos sucesores v_i, y para cada uno se ejecuta el algoritmo de detección de duplicados de la Figura 2.14, el cual tiene en cuenta 3 casos:

- Si el estado que representa el nodo v_i no había sido generado hasta el momento, es decir no estaba en la Lista Abierta ni en la Lista Cerrada, se inserta v_i en la Lista Abierta con valor $\hat{f}(v_i) \leftarrow \hat{g}(\mathrm{u}) + \mathrm{w}(\mathrm{u}, v_i) + \hat{h}(v_i)$
- Si el estado que representa el nodo v_i no estaba en la Lista Cerrada pero si en la Lista Abierta y además el costo del nuevo camino encontrado hacia v_i es mejor que aquel

hallado con anterioridad, se conserva el nuevo camino encontrado por ser de mejor costo, teniendo que reordenar la Lista Abierta *promoviendo* el nodo v_i por tener ahora menor valor $\hat{f}$.

- Si el estado que representa el nodo v_i estaba en la Lista Cerrada y además el costo del nuevo camino encontrado hacia v_i es mejor que aquel hallado con anterioridad, se conserva el nuevo camino encontrado por ser de mejor costo y se *reabre* el nodo v_i , es decir se elimina de la Lista Cerrada y se inserta en la Lista Abierta.

El tercer caso indica que se cerró un nodo *n*, es decir se insertó en la Lista Cerrada, cuando el camino encontrado desde *s* a *n* no era óptimo. Esto puede suceder únicamente cuando la función $\hat{f}$ no es monótona creciente a causa de que $\hat{h}$ no es consistente[4,5], es decir no cumple $\hat{h}(n) \leq w(n, n') + \hat{h}(n')$ para cada nodo *n* y cada sucesor *n'* de *n* (Figura 2.15).

Algoritmo A*
Entrada: *nodo inicial s*
Salida: *nodo solución t o NULL*

```
t ← NULL
Lista Cerrada ← ϕ                          {Inicializar la Lista Cerrada en vacío}
Insert(Lista Abierta , s, ĥ(s))            {Insertar s en la Lista Abierta, inicialmente vacía, con costo ĥ(s)}
while (Lista Abierta ≠ ϕ) and (t = NULL)   {Siempre y cuando haya nodos en Lista Abierta
                                           y no se ha encontrado la solución}
    u ← DeleteMin(Lista Abierta)           {Eliminar nodo con costo f mínimo}
    Insertar u en Lista Cerrada            {Almacenar el nodo en la Lista Cerrada}
    if (ĥ(u)=0)                            {Verificar si es un nodo final}
        t:= u                              {Se ha encontrado la solución}
    else
        hijos ← Expandir(u)                {Generar los nodos sucesores aplicando acciones legales}
        for each v in hijos                {Para cada sucesor v de u}
            DeteccionDuplicados(u,v Lista Abierta, Lista Cerrada)   {Invocar al algoritmo para detectar ciclos}
return t                                   {Retornar NULL si la solución no existía.}
                                           {Caso contrario retornar el puntero al nodo final}
```

Figura 2.13 Algoritmo de búsqueda A*.

[4] Cuando $\hat{h}$ no es consistente no satisface la regla *Desigualdad Triangular* que indica que cada lado de un triángulo no puede ser mayor a la suma de los dos lados restantes.
[5] Una heurística consistente es admisible pero no viceversa. Cuando se utiliza una heurística consistente la función $\hat{f}$ será monótona (Russel & Norvig, 2003)

Algoritmo DetecciónDuplicados
Entrada: *nodo u, nodo v, Lista Abierta, Lista Cerrada*
Efectos laterales: *se actualiza la Lista Abierta y/o la Lista Cerrada*

```
if (v no está en la Lista Cerrada)
    if ( v no está en la Lista Abierta)          {El estado que representa v no había sido generado}
        f̂(v) ←ĝ(u)+w(u,v) + ĥ(v)                 {Se inserta en la Lista Abierta}
        Insert (Lista Abierta, v, f̂(v))
    Else
        If ĝ(u) + w(u,v) < ĝ(v)                  {El estado que representa v está en la Lista Abierta}
            f̂(v) ←ĝ(u)+w(u,v) + ĥ(v)             {y además el nuevo camino hacia v tiene mejor costo }
            Padre(v) ← u
            Decreasekey(Lista Abierta, v , f̂(v))  {Promueve v en la Lista Abierta}
Else
    If ĝ(u) + w(u,v) < ĝ(v)                      {El estado que representa v está en la Lista Cerrada}
        f̂(v) ←ĝ(u)+w(u,v) + ĥ(v)                 { y además el nuevo camino hacia v tiene mejor costo }
        Padre(v) ← u
        Remover v de la Lista Cerrada            {Reabre v}
        Insert (Lista abierta, v, f̂(v))
```

Figura 2.14. Algoritmo de detección de duplicados para A*

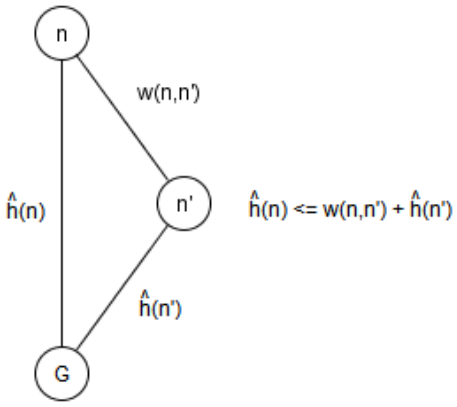


Figura 2.15 Regla de consistencia

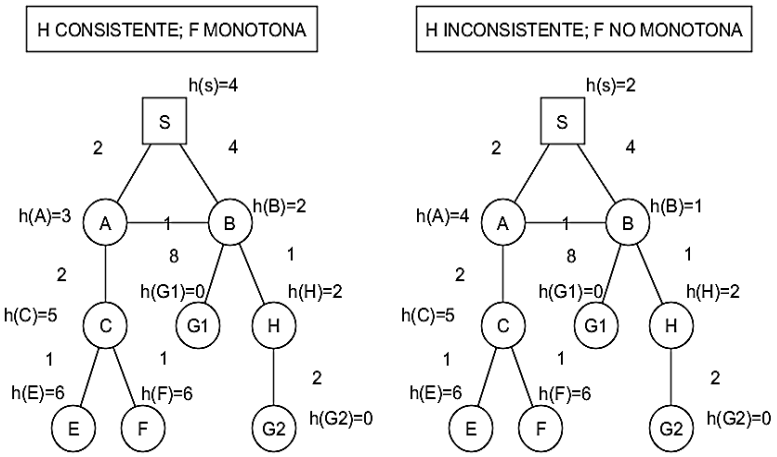


Figura 2.16.a. Espacio de estados con $\hat{h}$ consistente y $\hat{f}$ monótona.
b. Espacio de estados con $\hat{h}$ inconsistente y $\hat{f}$ no monótona.

La Figura 2.16.a muestra un espacio de estados con forma de grafo general, con $\hat{h}$ consistente provocando que $\hat{f}$ sea monótona, mientras que el grafo general de la Figura 2.16.b muestra una función $\hat{h}$ inconsistente y $\hat{f}$ no monótona.

La Tabla 2.6 muestra la ejecución del algoritmo A* para el grafo de la Figura 2.16.a; al lado de cada nodo n se denota cuál es su padre, el valor $\hat{g}(n)$ y el valor $\hat{h}(n)$. En este caso, al utilizar una heurística consistente A* nunca *reabre* nodos, sin embargo puede ocurrir que se *promuevan* nodos en la Lista Abierta. Dado que $\hat{f}$ es monótona, el valor $\hat{f}$ de los nodos a lo largo de un camino no decrece, lo que implica que los nodos seleccionados por A* también posean valor $\hat{f}$ creciente. Intuitivamente se pueden dibujar contornos en el espacio de búsqueda etiquetados por costo $\hat{f}$, es decir el contorno etiquetado con valor 6 posee todos los nodos n cuyo $\hat{f}(n) \leq 6$; la búsqueda comenzará con el contorno que contiene el nodo inicial e irá añadiendo contornos con valor de etiqueta creciente, hasta que en algún momento se alcance aquel que contiene el nodo final. Todas las soluciones en un contorno con valor de etiqueta mayor no serán óptimas. (Figura 2.17)

Tabla 2.6. Ejecución del algoritmo A* para el grafo de la Figura 2.16.a

Pasos del Algoritmo A*	Selección	Lista Abierta	Lista Cerrada	Nota
0		{S(-,0,4)}	{}	
1	S(-,0,4)	{A(S,2,3), B(S,4,2)}	{S(-,0,4)}	
2	A(S,2,3)	{B(A,3,2), C(A,4,5)}	{S(-,0,4), A(S,2,3)}	Regenera B(A,3,2), lo promueve en Lista Abierta
3	B(A,3,2)	{H(B,4,2), C(A,4,5), G1(B,11,0)}	{S(-,0,4), A(S,2,3), B(A,3,2)}	Regenera S(B,7,4) pero el primer camino generado era mejor
4	H(B,4,2)	{G2(H,6,0), C(A,4,5), G1(B,11,0)}	{S(-,0,4), A(S,2,3), B(A,3,2), H(B,4,2)}	
5	G2(H,6,0)	{C(A,4,5), G1(B,11,0)}	{S(-,0,4), A(S,2,3), B(A,3,2), H(B,4,2), G2(H,6,0)}	*Termina el algoritmo*

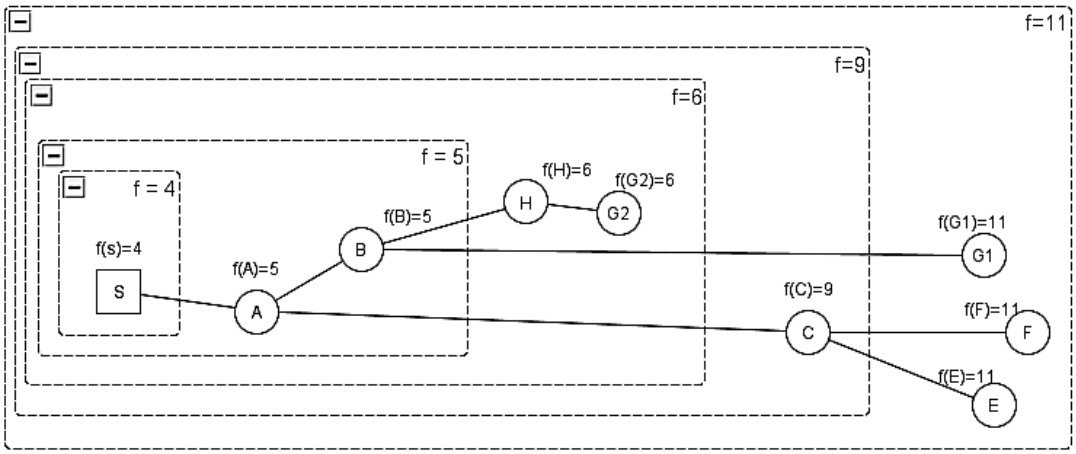


Figura 2.17. Espacio de búsqueda organizado en contornos.

La Tabla 2.7 muestra la ejecución del algoritmo A* para el grafo de la Figura 2.16.b, donde se puede notar que la utilización de una heurística inconsistente causa *reapertura* de nodos. En este caso el valor de $\hat{f}$ de los nodos puede decrecer a lo largo de un camino; aún así el algoritmo termina encontrando una solución óptima.

Para garantizar la propiedad de *admisibilidad,* el algoritmo A* procesará todos los nodos con $\hat{f}(n) < C^*$, siendo C* el costo de la solución optima, y algunos nodos con $\hat{f}(n) = C^*$, lo que puede variar según la política de desempate utilizada, sin embargo nunca procesará nodos con $\hat{f}(n) > C^*$.

Tabla 2.7. Ejecución de A* para el grafo de la Figura 2.16.b

Pasos del Algoritmo A*	Selección	Lista abierta	Lista cerrada	Nota
0		{S(-,0,2)}	{}	
1	S(-,0,2)	{B(S,4,1), A(S,2,4) }	{ S(-,0,2)}	
2	B(S,4,1)	{A(S,2,4), H(B,5,2), G1(B,12,0)}	{S(-,0,2), B(S,4,1)}	Regenera A(B,5,4), pero el primer camino generado era mejor
3	A(S,2,4)	{B(A,3,1),H(B,5,2), C(A,4,5) G1(B,12,0)}	{S(-,0,2), A(S,2,4)}	Regenera B(A,3,1) con mejor costo y lo reabre.
4	B(A,3,1)	{H(B,4,2), C(A,4,5), G1(B,11,0)}	{S(-,0,2), A(S,2,4), B(A,3,1)}	Regenera H(B,4,2) y G1(B,11,0), los promueve en la Lista Abierta
5	H(B,4,2)	{G2(H,6,0),C(A,4,5), G1(B,11,0)}	{S(-,0,2), A(S,2,4), B(A,3,1), H(B,4,2) }	
6	G2(H,6,0)	{C(A,4,5), G1(B,11,0)}	{S(-,0,2), A(S,2,4), B(A,3,1), H(B,4,2), G2(H,6,0) }	*Termina el algoritmo*

Las demostraciones formales de las propiedades del algoritmo A* se encuentran con amplio grado de detalle en el trabajo (Hart, et al., 1968):

- A* es *completo* ya que en algún momento se alcanzará el contorno conteniendo un nodo final.
- Si $\hat{h}$ es admisible, A* es *admisible* por lo que el algoritmo termina encontrando la solución óptima.
- Si $\hat{h}$ es consistente, A* es *óptimo*, es decir no hay otro algoritmo que examine menos nodos que A* y que garantice encontrar la solución óptima, a menos que evalúe en forma distinta los empates entre nodos cuyo valor $\hat{f}$ coincide. Además A* no reabre nodos cerrados, es decir cuando se selecciona un nodo para ser expandido se ha encontrado el camino óptimo desde el nodo inicial hacia él.

Con respecto a la *complejidad temporal,* en el peor caso $\hat{h}(n) = 0$ para todo n y A* se comportará como la *búsqueda de costo uniforme* siendo de orden $O(b^{1+\lfloor C^*/c \rfloor})$. En el mejor caso la heurística es exacta, es decir $\hat{h}(n) = h(n)$, todos los nodos en el camino óptimo tendrán $\hat{f}(n) = f(n) = f(s)$ y todos los demás tendrán $\hat{f}(n) > f(s)$, causando que se examinen sólo los nodos en el camino óptimo siendo de orden lineal.

Con respecto a la *complejidad espacial,* el orden coincide con la complejidad temporal ya que el algoritmo debe almacenar todos los nodos generados.

2.2.3.2 Iterative Deepening A* (IDA*)

El algoritmo IDA* (Korf, 1985) busca reducir la complejidad espacial de A*, a expensas de expandir mayor cantidad de nodos, y al mismo tiempo garantizar que la solución encontrada tenga costo óptimo cuando se utiliza una heurística admisible.

IDA* está basado en la idea de *Iterative Deepening Depth First* (Sección 2.2.2.3): realiza una serie de búsquedas siguiendo la estrategia *Depth-First* y utiliza un límite *l*, que en este caso representa un costo, para descartar aquellos nodos *n* generados que cumplan $\hat{f}(n) > l$, donde $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$ y $\hat{h}$ es una heurística *admisible*. En la primera búsqueda el límite se inicializa con valor $\hat{f}(s) = \hat{h}(s)$, puesto que si $\hat{h}$ es admisible el costo de la solución óptima $h(s)$ será mayor o igual a dicho límite. Si la primera búsqueda *DFS* termina encontrando un nodo final *t,* éste tendrá costo $f(t) = h(s)$. En caso que la búsqueda termine sin éxito, el costo de la solución es mayor a *l* por lo que se incrementa el límite y se comienza una nueva búsqueda *DFS*. El nuevo valor límite no tiene que sobrepasar $h(s)$, por lo que toma el valor $\hat{f}$ del nodo descartado en la iteración anterior más cercano al límite previo.

Al igual que *Iterative Deepening Depth First* el algoritmo es *completo*. La demostración formal de la propiedad de *admisibilidad,* siempre y cuando la heurística utilizada sea admisible, se puede encontrar en el trabajo (Korf, 1985).

Con respecto a la *complejidad temporal*, el autor demuestra teóricamente que cuando se trabaja sobre espacios de estados estructurados como árboles el algoritmo IDA* expande asintóticamente la misma cantidad de nodos que A*, basándose en que la cantidad de nodos examinados en la última iteración de IDA* será equivalente a A* cuando éste último utiliza la política de desempate de procesar primero el nodo generado más recientemente y teniendo en cuenta que los nodos generados en iteraciones previas no afectan el orden asintótico (Korf, 1985). Sin embargo, en la práctica el algoritmo expandirá mayor cantidad de nodos respecto a A*, ya que cada búsqueda olvida los nodos generados en iteraciones anteriores. Estudios posteriores demuestran que si A* expande N nodos *esenciales* (nodos cuyo costo es menor a C*, donde C* es el costo de la solución óptima), IDA* expandirá $Nx(N+1)/2$ (Patrick, et al., 1992).

Por último, la *complejidad espacial* será lineal al igual que *DFS* siempre y cuando no se realicen verificaciones por ciclos (Korf, 1985).

2.2.3.3 Frontier Search A*

La principal limitación de *A** es la cantidad de memoria utilizada. *Frontier Search A** (Korf, et al., 2005) reduce la *complejidad espacial* manteniendo en memoria sólo los nodos generados y no expandidos, es decir la *Lista Abierta,* y a diferencia de los algoritmos *BFS* no conserva la *Lista Cerrada* teniendo que aplicar un esfuerzo adicional para regenerar el camino a la solución una vez encontrado el nodo final. El algoritmo utiliza la misma función de costo que A* y el autor demuestra que cuando se utiliza una heurística consistente nunca se regenera un nodo ya expandido.

Para cumplir con su objetivo, junto con cada nodo requiere almacenar un vector de bits, teniendo un bit por cada operador definido en el problema; las posiciones del vector marcadas en 0 representan operadores posibles de aplicar, mientras que las posiciones marcadas en 1 son operadores que regenerarán algún estado previamente procesado.

Cuando se trabaja con grafos no dirigidos, el algoritmo comienza añadiendo a la Lista Abierta el nodo inicial, cuyo vector de bits tiene un 0 en cada posición. En cada iteración selecciona el nodo de menor costo de la Lista Abierta y verifica si representa al estado final, caso en el cual el algoritmo termina; en caso contrario expande el nodo a partir de aplicar los operadores legales cuyas posiciones en el vector de bits están marcadas en 0. Para cada nodo sucesor generado se debe marcar en 1 la posición del vector de bits correspondiente al operador que regenerará al estado padre. En este punto el nodo seleccionado se borra de la memoria. Luego para cada sucesor verifica que no exista en la Lista Abierta otro nodo representando el mismo estado, caso en el cual se lo inserta; en caso contrario se detectó un duplicado y el algoritmo sólo conservará aquel que fue alcanzado por el camino menos costoso realizando la unión de su vector de bits con el vector del nodo copia que se destruirá.

Para grafos dirigidos, al expandir un nodo no sólo se generan sus sucesores a través de los operadores legales que el vector de bits indica como no usados, sino que también se generan predecesores vía operadores ilegales no usados al momento, los cuales se inicializan con costo infinito. Este mecanismo evitará regenerar el nodo seleccionado a través de un nodo que todavía no ha sido generado o que ha sido generado pero no expandido.

Respecto a la *complejidad temporal* el algoritmo expande los mismos nodos que A*. Sin embargo, el inconveniente de esta técnica yace en que se debe reconstruir el camino desde el nodo inicial hasta el nodo final una vez terminada la búsqueda para poder obtener la secuencia de acciones que representa la solución, lo cual implica expandir nodos adicionales.

2.3 Discusión y conclusiones

El algoritmo A* fue seleccionado como base para resolver el problema elegido como caso de estudio ya que garantiza encontrar la solución de menor costo siempre y cuando la heurística a utilizar sea *admisible*. Además, cuando se utiliza una heurística consistente, A* es óptimo ya que expande menor cantidad de nodos en comparación con cualquier otro algoritmo.

IDA* fue descartado ya que en cada iteración vuelve a realizar trabajo de la iteración anterior. No obstante, debido al uso eficiente que hace de la memoria, es utilizado frecuentemente para resolver instancias difíciles de problemas NP-Complejo para las cuales una búsqueda BFS agota la memoria disponible en cuestión de minutos. Sin embargo, dado que uno de los objetivos de esta tesis es utilizar un *cluster* que provea un agregado de memoria importante a medida que aumenta la cantidad de máquinas conectadas, se dio prioridad a la simplicidad de A* y a su eficiencia temporal frente al requerimiento de memoria.

Por otra parte, Frontier A* no fue elegido debido a la complejidad que implica realizar una búsqueda adicional para reconstruir el camino de la solución, lo cual requiere expandir nodos adicionales.

Han sido descartados de antemano los algoritmos DFS e Iterative Deepening DFS: el primero no encuentra la solución óptima requerida por el caso de estudio; si bien el segundo encontraría una solución óptima para problemas cuyas acciones tengan igual costo, el algoritmo posee las mismas desventajas que IDA*.

La técnica Breadth-First Search podría aplicarse para la resolución del problema propuesto. Al procesar el espacio de estados por niveles, si se encuentra una solución será la que requirió menor número de movimientos para resolver el tablero inicial. Sin embargo, esta estrategia puede tomar un tiempo considerable para encontrar una solución, ya que suelen residir en zonas alejadas de la raíz y para alcanzar dicho nivel debe procesar todos los nodos de los niveles anteriores.

El algoritmo *búsqueda de costo uniforme* es un caso especial del algoritmo A* que se obtiene al utilizar $\hat{h}(n) = 0$ para todo nodo n del grafo, su complejidad es equivalente a la de A* pero el tiempo de ejecución promedio de este último es mejor. En el caso del Puzzle, cada movimiento tiene costo 1, por lo que la ejecución de una *búsqueda de costo uniforme* sería similar a la de Breadth-First Search. Además, supongamos otro problema donde las acciones tengan distinto costo, y que la heurística utilizada por el algoritmo A* es exacta (caso ideal). En este caso todos los nodos en el camino hacia el nodo solución tendrían igual costo, lo que conduciría al algoritmo a procesar en cada iteración un nodo que está en el camino de la solución óptima. Dado que la *búsqueda de costo uniforme* visita los nodos en orden creciente de costo desde el vértice inicial, el algoritmo perdería tiempo explorando otros nodos en dirección opuesta al nodo solución.

CAPÍTULO 3: CARACTERIZACIÓN DEL CASO DE ESTUDIO

El N^2*-1 Puzzle* propuesto como caso de estudio es un problema de optimización combinatoria NP-Complejo (Ratner & Warrnuth, 1986) el cual puede resolverse a través de aplicar una búsqueda informada sobre el espacio de estados del problema ya que requiere encontrar el camino menos costoso entre dos vértices, que representan la configuración inicial y configuración final respectivamente. Este problema es ampliamente utilizado para evaluar el rendimiento de los algoritmos de búsqueda informada por poseer una descripción exacta y simple; además diversos autores han recurrido a él para ejemplificar técnicas para el desarrollo de funciones heurísticas. (Pearl, 1984) (Russel & Norvig, 2003)

El *Puzzle* es un problema para el cual el tiempo de resolución de una instancia no sólo depende de la dimensión del tablero (*N*) sino que también depende de los datos en sí; está caracterizado por generar gran cantidad de nodos con igual costo durante la búsqueda y a medida que ésta avanza y se incrementa el costo de los nodos siendo procesados, la cantidad de nodos para dicho costo aumenta exponencialmente; estas características son similares a las poseídas por el problema de Cobertura de Vértices (VCP), razón por la cual ambos generan un particular interés en el ámbito del cómputo paralelo ya que abren la posibilidad a obtener un rendimiento por encima del teóricamente posible (*Speedup Superlineal*) y por ello diversos autores han investigado las causas (Grama, et al., 2003).

La Sección 3.1 describe el problema N^2*-1 Puzzle*; en la Sección 3.2 se especifica cómo evaluar la solubilidad de una instancia del problema teniendo una configuración inicial y una configuración final; la Sección 3.3 estudia heurísticas desarrolladas por distintos autores para el problema; por último, la Sección 3.4 presenta una discusión en la que se nombran problemas del mundo real los cuales pueden resolverse con los algoritmos presentados en esta tesis.

3.1 Definición del problema

El problema del N^2*-1 Puzzle* es una generalización del *15-Puzzle* ideado por Sam Lloyd (Ratner & Warmuth, 1990). Consiste en N^2-1 piezas numeradas de 1 a N^2-1 colocadas en un tablero de tamaño N^2. N^2-1 casilleros del tablero contienen exactamente una pieza, quedando sólo una casilla vacía la cual se denomina *hueco*.

Un movimiento legal en este juego implica mover el hueco a una posición adyacente a él, en sentido horizontal o vertical, trasladando la ficha que estaba en ese lugar a la posición anterior del hueco.

El objetivo del *Puzzle* es aplicar movimientos legales repetidamente hasta convertir el tablero inicial en el tablero final elegido.

La Figura 3.1.a muestra un tablero instancia del 15-*Puzzle* (N= 4). La Figura 3.1.b representa el tablero solución clásico.

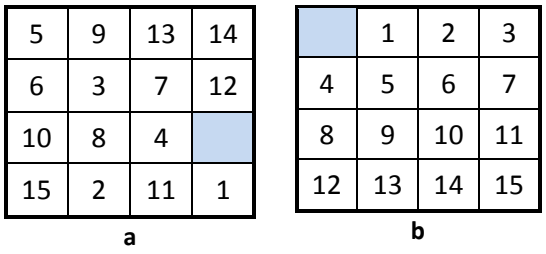


Figura 3.1: Tableros de 15-Puzzle (N=4)
a. Tablero inicial. **b.** Tablero final clásico.

La solución al problema planteado tendrá que ser aquella que minimice la cantidad de movimientos que deben realizarse para alcanzar la configuración final desde la configuración inicial dada.

3.2 Solubilidad

El problema original del 15-*Puzzle*, planteado por Sam Lloyd, consistía en encontrar la secuencia de movimientos que transforme el tablero de la Figura 3.2.a en el tablero solución de la Figura 3.2.b. No obstante, nadie pudo resolver dicho problema y esto se debe a que no existe solución para el mismo (Johnson & Storey, 1879).

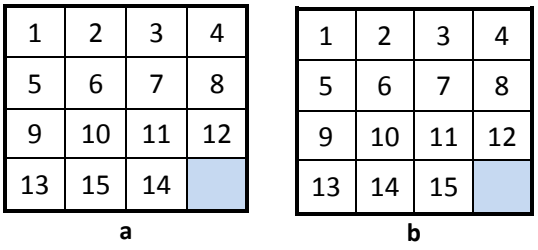


Figura 3.2. Problema original planteado por Sam Lloyd.

El espacio de estados del *N^2-1 Puzzle* consta de $N^2!$ estados. Este número surge ya que un tablero puede representarse como un vector de N^2 posiciones, y en cada posición debe colocarse una ficha o el hueco, por lo tanto cualquier ordenamiento de los mismos es admitido como un tablero.

Sin embargo, no todo estado puede alcanzarse desde otro aplicando movimientos legales. Esto se debe a que el grafo que representa el espacio de estados del *Puzzle* tiene dos componentes conexas de igual tamaño, por lo tanto si el estado inicial y el final no están en la misma componente no habrá solución para el problema.

En la Figura 3.3.a se ilustra la situación en la que el tablero inicial y final están en la misma componente conexa, por lo que existe al menos una solución para el problema. En la Figura 3.3.b se muestra un caso donde el estado final es inalcanzable desde el estado inicial (no existe solución alguna).

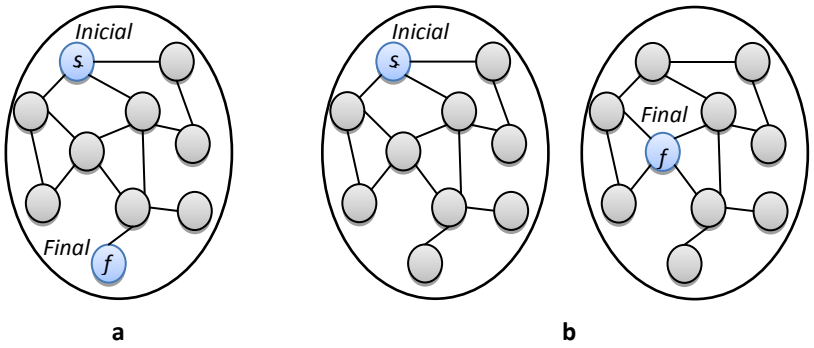


Figura 3.3. Espacio de estados para el problema del Puzzle
a. Estado inicial soluble para el estado final **b.** Estado inicial no soluble para el estado final

A partir de lo comentado anteriormente, se puede afirmar lo siguiente: sólo la mitad de los estados es alcanzable desde cualquier otro, esto implica una reducción en el tamaño del espacio de estados a $N^2!/2$. Aplicando un simple paso de detección previo a la búsqueda se puede determinar si la configuración final es alcanzable desde la configuración inicial.

3.2.1 Algoritmo para la detección de solubilidad

El procedimiento para verificar si un tablero inicial tiene solución para un tablero final es el siguiente (Edelkamp & Schrödl, 2011):

- Para cada ficha i (i = 2..N^2-1) de los tableros inicial y final, se cuenta la cantidad de piezas con menor número que aparecen después de ella, ya sea en la misma fila a su derecha, o en cualquier fila inferior. Llamaremos a este número "inversión de i", y se denotará n_i.
- A continuación se calcula para el tablero inicial y final NT = $n_2 + n_3 + \ldots n_{(N^2-1)}$ (la suma de las inversiones de todas sus fichas). Si N es par, se le suma a NT el número de la fila donde se encuentra el hueco. [6]
- Si la paridad de ambos resultados es la misma, entonces el tablero final es alcanzable desde el tablero inicial. Esto se debe a que (NT mod 2) se mantiene invariante con cualquier movimiento legal.

Para el tablero de la Figura 3.1.a *NT = 60*, mientras que el tablero de la Figura 3.1.b tiene *NT*=0, por lo tanto se puede alcanzar el tablero de la Figura 3.1.b a partir del tablero de la Figura 3.1.a.

El estudio informal de la correctitud de este procedimiento se encuentra en el Apéndice A.

[6] Las filas se numeran desde arriba hacia abajo, comenzando en 1.

3.3 Funciones heurísticas para el problema del Puzzle

Las funciones heurísticas se utilizan durante la búsqueda para valuar los nodos, el valor heurístico estimará el costo del camino mínimo desde el nodo en cuestión hasta el nodo objetivo. Para un problema en particular pueden existir distintas funciones heurísticas, algunas más precisas que otras. A medida que la heurística se vuelve más afinada, la estimación del costo de los nodos es más próxima al costo real, por lo que los algoritmos que la utilizan tenderán a procesar menor cantidad de nodos.

Las heurísticas son dependientes del dominio del problema, por lo que ha sido de gran interés descubrir métodos generales para su generación. En especial, para el problema del *Puzzle* se han desarrollado diferentes heurísticas admisibles que mejoran a la heurística clásica.

Una metodología para generar nuevas heurísticas se basa en crear un problema simplificado a partir del original mediante la eliminación de alguna o varias reglas del mismo. De este modo, la función heurística para el problema original será el costo de la solución óptima para el problema simplificado (Pearl, 1984). Dado que la solución al problema original es también solución del problema simplificado y es al menos tan costosa como la solución óptima para este último, la heurística que surge a partir de esta metodología es admisible y puede demostrarse que es consistente. (Russel & Norvig, 2003)

También puede obtenerse una heurística admisible para un problema a partir del costo de solución de un subproblema, por ejemplo: para el problema del Puzzle teniendo el subproblema que corresponde a ubicar las fichas 1, 2, 3 y 4 en sus posiciones finales, si se conoce el costo de la solución óptima del subproblema, éste será una cota inferior para el costo del problema original.

Teniendo en cuenta lo anterior, diversos autores han investigado acerca de la generación de bases de datos de patrones, cuyo objetivo es almacenar el costo óptimo de la solución para cada posible instancia de un subproblema, es decir siguiendo el ejemplo para cada posible ubicación de las fichas 1, 2, 3 y 4 (patrón) se almacena el costo óptimo de ubicarlas en la posición final. Durante la búsqueda, para valuar un nodo sólo se tendrá en cuenta las posiciones de las fichas que son parte del patrón, de modo de realizar una búsqueda en la base de datos y encontrar la estimación de costo requerida. A medida que se incluyen más fichas en el patrón más afinada será la heurística, pero el cómputo de la base de datos tomará mayor cantidad de tiempo y su almacenamiento requerirá mayor espacio. (Culberson & Schaeffer, 1998)(Korf, 2000)

Para el caso de estudio, la estimación heurística para un tablero indicará el número mínimo de movimientos requeridos para transformar dicho tablero en el tablero solución. Se presentan en las siguientes secciones distintas funciones heurísticas que deben ser computadas sobre la representación del estado. No se trabajará en esta tesis con heurísticas pre-calculadas y almacenadas en base de datos de patrones ya que el objetivo es estudiar la paralelización de un algoritmo de búsqueda sobre grafos y no el desarrollo de funciones heurísticas más potentes.

3.3.1 Suma de las Distancias de Manhattan (SDM)

La heurística clásica utilizada para el problema del N^2-1 Puzzle es la Suma de las Distancias de Manhattan *(SDM)* de todas las fichas del tablero a estimar *t* respecto al tablero final *s* (Pearl, 1984).

La Distancia de Manhattan (*DM)* para una ficha representa la mínima cantidad de movimientos que se debe hacer para trasladarla desde su posición actual a la posición en la que debe aparecer en el tablero final, y se calcula de la siguiente manera: supongamos que cada posición del tablero se representa como un par ordenado; sea una ficha cualquiera *r* del tablero *t*, si la misma se encuentra en la posición (i,j) y en el tablero final *s* debe localizarse en el casillero (k,l), entonces DM(r)= |i-k|+|j-l|.

El tablero *t* de la Figura 3.1.a tiene *SDM(t)=43* respecto al tablero final de la Figura 3.1.b, siendo las Distancias de Manhattan de cada ficha: DM(7,8)=1, DM(5,6,9,10,11)=2, DM(3,4,15)=3, DM(2,13,14)=4, DM(1,12)=5.

La *SDM* es una heurística admisible, ya que asume que las fichas pueden moverse independientemente pasando incluso por encima de otras quitando la restricción de movimiento legal planteado en la Sección 3.1, y es consistente dado que la diferencia en los valores heurísticos de un nodo u y su sucesor v es a lo sumo 1 ($\left|\hat{h}(n) - \hat{h}(n')\right| \leq 1$, para todo n y su sucesor n'), y que $w(n,n') = 1$ por lo tanto se cumple $\hat{h}(n') - \hat{h}(n) + w(n,n') \geq 0$. (Edelkamp & Schrödl, 2011)

3.3.2 Conflictos Lineales

Un *"conflicto lineal"* (Hansson, et al., 1985) (Korf & Taylor, 1996) (Bauer, 1994) entre dos fichas *x* e *y* ocurre cuando las mismas están posicionadas en su fila/columna correcta pero invertidas en orden. En este caso, una de ellas deberá cambiar de fila/columna para que la otra pueda trasladarse hasta su posición final y, luego de desplazarse en forma horizontal/vertical tantas veces como lo indica su *DM*, la ficha debe retornar a su fila/columna adecuada. Así, por cada conflicto lineal se podría agregar 2 movimientos adicionales a la *SDM* del tablero, excepto casos particulares.

La Figura 3.4 presenta un tablero con N=4 y un conflicto lineal entre las fichas 8 y 10 (respecto al tablero final de la Figura 3.1.b). En este caso DM(8)=1 y DM(10)=2, la *SDM* parcial para estas fichas sería 3 pero no es un costo real ya que la ficha 8 debería pasar por encima de la ficha 10 y viceversa. Se muestra además la secuencia de movimientos que se debería realizar ante el conflicto lineal, lo cual implica dos movimientos más.

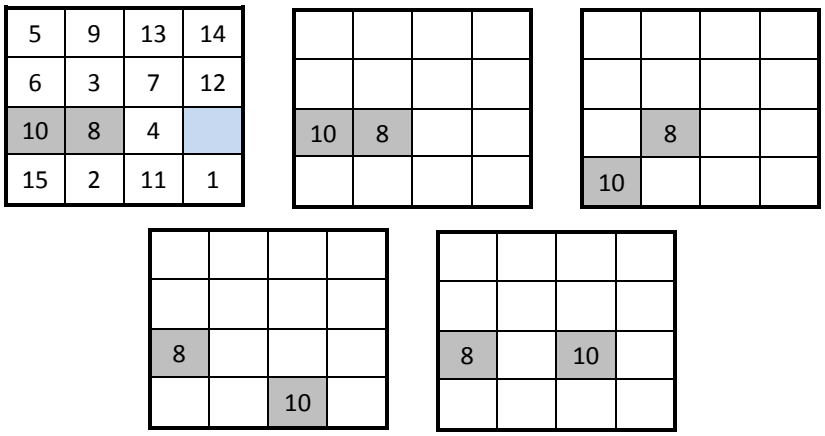


Figura 3.4. Secuencia de movimientos ante un conflicto lineal en una fila.

Un caso en el cual no se puede contabilizar 2 movimientos por cada conflicto lineal entre fichas es aquel donde las fichas X, Y, Z están en su fila correcta en el siguiente orden Z, X, Y. En dicho caso sólo la ficha Z cambiará de fila, dejando transitar a X e Y hacia sus respectivos lugares, por lo que se harían sólo dos movimientos adicionales. El mismo concepto puede aplicarse a las columnas.

La heurística *SDMCL* combina la Suma de las Distancias de Manhattan y la detección de conflictos lineales de las fichas, definiéndose *SDMCL (t) = SDM(t) + CL(t),* donde t representa el tablero, SDM es la función que calcula la Suma de las Distancias de Manhattan para t, y CL(t) = (cantidad de conflictos lineales en t) * 2, preservando la admisibilidad y consistencia de la heurística. (Hansson, et al., 1985).

Continuando con el ejemplo planteado, para el tablero de la Figura 3.4 SDM(t)=43 *y* CL(t)=2, por lo tanto SDMCL(t)= 45.

3.3.3 Últimos Movimientos ("Last Moves")

El último movimiento (Korf & Taylor, 1996) que se realiza para resolver un *Puzzle* siendo N=4 y con el tablero final mostrado en la Figura 3.1.b, es trasladar la ficha 1 desde la posición (1,1) a la posición (1,2), o mover la ficha 4 desde la posición (1,1) a la posición (2,1). De lo anterior se puede decir que antes del movimiento final la ficha 1 o la 4 deben estar en la esquina superior izquierda del tablero.

Dado que la *DM* para una ficha se calcula desde su posición actual a la posición final, si la ficha 1 no está en la primera columna o la ficha 4 no está en la primera fila, la *SDM* no es real ya que el camino para dejar alguna de estas fichas en su posición final no haría que la misma pase por la esquina superior izquierda. Por consiguiente, si la ficha 1 no se encuentra en la primera columna y la ficha 4 no se encuentra en la fila superior, se puede sumar dos movimientos adicionales a la SDM, preservando la admisibilidad de la heurística.

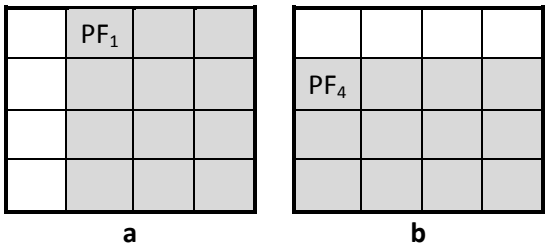


Figura 3.5. Tablero en el que la DM de las fichas 1 y 4 no acomoda el paso de la ficha por la esquina

La Figura 3.5 muestra la zona en la que debería estar la ficha 1 (a) y la ficha 4 (b) junto con un indicador de su posición final (*PF*). La *DM* calcula la mínima cantidad de pasos para mover una ficha a su posición final, por lo que si la ficha 1 está en la zona indicada en la Figura 3.5.a su *DM* acomodará el paso de la ficha por esa zona. Algo parecido ocurre con la ficha 4, Figura 3.5.b. Al sumarse dos movimientos se obliga a alguna de estas fichas a pasar por la esquina superior izquierda.

En caso de combinar la *SDMCL* con la mejora planteada deben tenerse en cuenta ciertas interacciones entre ambas para mantener la *admisibilidad*. Supongamos que la ficha 1 y la 4 no están en la primera columna y fila respectivamente. Si la ficha 1 está en su columna respectiva y se encuentra en un conflicto lineal con otra ficha, se suman dos movimientos por conflicto lineal. Pero esto podría provocar que la ficha 1 se mueva a la primera columna, y así podría pasar por la esquina superior izquierda del tablero. Un caso similar ocurre con la ficha 4. Por lo tanto, si la ficha 1 está en conflicto lineal en la columna o la ficha 4 está en conflicto lineal en la fila entonces no se puede sumar dos movimientos adicionales por "últimos movimientos".

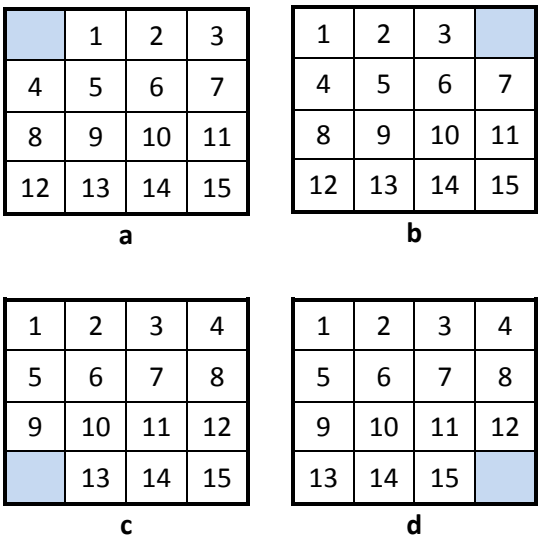


Figura 3.6. Ejemplos de tableros finales con posibilidad de aplicar la heurística "Últimos movimientos"

El mismo concepto puede ser aplicado a cualquier tablero final que tenga el hueco en alguna de las esquinas, como se muestra en la Figura 3.6, y para cualquier tamaño del tablero.

3.3.4 Fichas de las Esquinas del Puzzle ("Corner Tiles")

Esta mejora de la heurística se centra en las esquinas del tablero. (Hansson, et al., 1985)(Korf & Taylor, 1996)

Por ejemplo, tomando en cuenta el tablero final de la Figura 3.6.a, si la ficha 2 está en su posición correcta, y la posición de la ficha 3 está ocupada por otra ficha distinta, la ficha 2 tendrá que moverse temporalmente para dejar pasar a la ficha 3 a su posición final. Esto requiere dos movimientos adicionales. Si se quiere combinar con las heurísticas anteriores, el 2 no debería estar en conflicto en la fila[7], ya que se habrían sumado dos movimientos. La misma regla se puede aplicar al 7, a menos que este en conflicto en la columna. Si ocurren ambos casos, es decir el 2 y el 7 están en su posición y el 3 no, entonces se podrían sumar a la estimación heurística base 4 movimientos. Esta situación se ilustra en la Figura 3.7.

Las mismas reglas se aplican a las demás esquinas, no siendo así para la esquina que contiene el hueco en el tablero final[8].

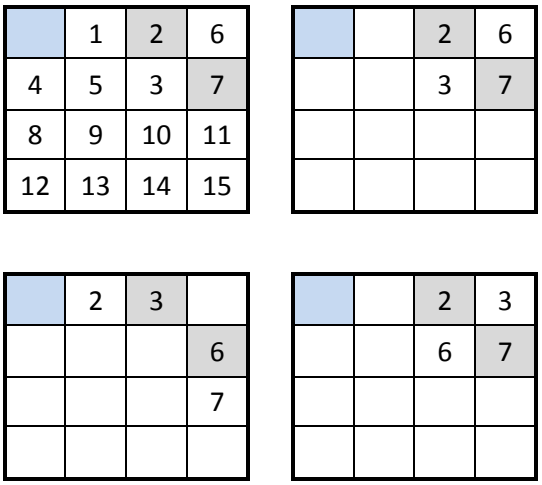


Figura 3.7. Fichas de las esquinas del Puzzle.

[7] Nunca se dará el caso en que la ficha 2 estando en su posición final se encuentre en un conflicto en la columna.

[8] Esto se debe a la difícil interacción de esta heurística con la de "últimos movimientos".

3.4 Discusión y conclusiones

Las búsquedas en espacios de estados son aplicables a diversas áreas ya que muchos problemas pueden definirse en términos de estados y transiciones, como por ejemplo: planificación de rutas óptimas, navegación automática de un robot, alineamiento óptimo de secuencias, entre otras. (Russel & Norvig, 2003) (Edelkamp & Schrödl, 2011) (Gudaitis, 1994) (Fitch, et al., 2005) (Kobayashi, et al., 2011)

Dado que estos problemas tienen una especificación compleja, se ha recurrido a un problema denominado *juguete,* como es el caso del N^2-1 Puzzle*,* ya que posee una especificación exacta y simple, ha sido usado por diversos autores en el ámbito del cómputo paralelo para evaluar el rendimiento de sus algoritmos y ha sido estudiado por diversos autores para la generación de heurísticas más potentes. Asimismo, en la actualidad el problema genera interés debido a que se ajusta a problemas reales tales como el movimiento de objetos por parte de un vehículo en un depósito con alta densidad de almacenamiento (Gue, et al., 2013)

Todas las heurísticas presentadas para el caso de estudio son admisibles, ya que el algoritmo A* a paralelizar impone dicha restricción, pero no todas son consistentes. Históricamente las heurísticas admisibles e inconsistentes han sido consideradas en forma negativa a causa de la reapertura de nodos que provoca su uso para los algoritmos de búsqueda *BFS*, siendo teóricamente exponencial en el peor caso para el algoritmo A*. Sin embargo hoy en día son motivo de estudio ya que los peores casos difícilmente se encuentran en la práctica e incluso se investigan los beneficios de su aplicación. (Zhang, et al., 2009) (Felner, et al., 2011)

CAPÍTULO 4: ARQUITECTURAS PARALELAS Y HERRAMIENTAS DE PROGRAMACIÓN

Un usuario que plantea el desarrollo de una aplicación paralela tiene como objetivo obtener ciertas mejoras en el rendimiento en relación a la aplicación secuencial respectiva. Dichas mejoras significarán por ejemplo reducir su tiempo de ejecución, resolver instancias más complejas del problema en igual cantidad de tiempo, resolver un conjunto de instancias del problema en igual cantidad de tiempo, entre otras.

La arquitectura del sistema es un aspecto importante a tener en cuenta durante el desarrollo de una aplicación paralela porque define el modelo de comunicación a utilizar para la interacción de los procesos/threads, es decir en arquitecturas de memoria compartida la comunicación será a través de variables compartidas y en arquitecturas de memoria distribuida será a través de paso de mensajes. Asimismo existen arquitecturas que permiten ambos mecanismos de comunicación.

En este capítulo se estudian las arquitecturas paralelas más utilizadas actualmente (*multicore*, *cluster* y *cluster de multicore*). La Sección 4.1 realiza un repaso de las clasificaciones de las arquitecturas paralelas; la Sección 4.2 estudia el surgimiento y utilización de clusters; la Sección 4.3 realiza una breve reseña del surgimiento de los procesadores multicore y examina las diferencias básicas entre los modelos más conocidos; la Sección 4.4 analiza las herramientas de programación paralela más utilizadas hoy en día; por último, la Sección 4.5 presenta una discusión acerca de los retos que debe afrontar el programador de aplicaciones paralelas para sacarle provecho a la arquitectura.

4.1 Clasificación de arquitecturas paralelas

Una computadora paralela está conformada por un conjunto de elementos de procesamiento que se comunican y cooperan entre sí para resolver problemas más rápidamente. (Grama, et al., 2003)

La amplia variedad de arquitecturas paralelas puede clasificarse según distintos criterios. Uno de ellos fue propuesto por Flynn y se basa en identificar el flujo de control (instrucciones) y el flujo de datos del sistema, estableciendo así tres categorías[9] de las cuales sólo dos predominan hoy en día (Stallings, 2006) (Hennessy & Patterson, 2011):

- SIMD (*Single Instruction, Multiple Data*): los elementos de procesamiento acceden a datos privados en memoria, pudiendo compartir o no el espacio de direcciones; asimismo, existe una única memoria de programa desde la cual se leen las instrucciones y se decodifican. En cada etapa, cada elemento de cómputo obtiene la misma instrucción decodificada y carga sus propios datos sobre los cuales se ejecutará la instrucción. De este modo, la misma instrucción se ejecuta sincrónicamente sobre datos distintos. Un ejemplo de arquitectura

[9] La categoría SISD corresponde a una computadora uniprocesador convencional.

paralela que sigue este modelo es aquella que poseen las *GPUs* (Graphics Processing Units).

- MIMD (*Multiple Instruction, Multiple Data*): los elementos de procesamiento acceden a su propia memoria de programa y a sus propios datos, que pueden residir en un espacio de direcciones compartido o distribuido. En cada paso, cada elemento de cómputo carga su instrucción, la decodifica y carga sus datos para luego ejecutar la instrucción sobre dichos datos. De esta manera, los elementos de cómputo ejecutan las instrucciones en forma asíncrona. Los *multicore* y los *clusters* son ejemplos que siguen este modelo.

Otra forma de clasificar las arquitecturas paralelas es según la organización de la memoria. Así podemos distinguir dos grupos: las arquitecturas de memoria compartida y las arquitecturas de memoria distribuida; las computadoras paralelas del primer tipo reciben el nombre de *multiprocesador* y las correspondientes al segundo tipo reciben el nombre de *multicomputador* (Grama, et al., 2003) (Stallings, 2006) (Rauber & Rünger, 2010) (Hennessy & Patterson, 2011):

- Arquitecturas de memoria compartida: los elementos de cómputo comparten una memoria global, que puede estar dividida en varios módulos, a la cual se conectan a través de un medio de interconexión. Los procesos se comunican a través de variables compartidas en la memoria global. Dado que los elementos de cómputo accederán a datos compartidos, puede ocurrir que alguna de las copias de un mismo bloque de datos residente en caché de distintos elementos de cómputo sea modificada. Para garantizar la consistencia de los datos entre las caché y la memoria global se utilizan *protocolos de coherencia de caché*. Los procesadores *multicore* y los *SMP* (*Symetric Multiprocessor*) poseen este tipo de arquitectura.
- Arquitecturas de memoria distribuida: los elementos de cómputo (*nodos*) poseen una memoria privada y local, y se conectan a través de una red de interconexión. La comunicación entre procesos residentes en nodos distintos se realiza a través del paso de mensajes vía red. Este tipo de arquitectura se volvió habitual ya que se puede lograr una máquina paralela con memoria distribuida a partir de conectar múltiples PC a través de una red. Los *clusters* poseen este tipo de arquitectura.

Por otro lado, las computadoras con memoria compartida suelen dividirse en dos categorías según el tipo de acceso a la memoria global por parte de los elementos de cómputo:

- UMA (*Uniform Memory Access*): el tiempo de acceso a cualquier bloque de datos en memoria global es idéntico para cualquier elemento de cómputo.
- ccNUMA (*On cache-coherent Nonuniform Memory Access*): la memoria está dividida en módulos pero es vista como una única memoria global; cuando un proceso quiere acceder a un bloque de memoria el tiempo de acceso depende del lugar donde residen dichos datos y del elemento de cómputo que los quiere acceder.

4.2 Arquitectura tipo Cluster

Un *cluster* es un multicomputador con memoria distribuida formado por un conjunto de computadoras (*nodos*) conectadas a través de una red dedicada que se comportan como si fuesen un único recurso de cómputo. Su objetivo principal consiste en mejorar el rendimiento y/o la disponibilidad de un sistema, siendo más económico que una computadora de velocidad y disponibilidad comparables. Esta arquitectura surgió como alternativa a los grandes sistemas multiprocesadores de memoria compartida, y se volvieron plataformas comunes en el cómputo paralelo de problemas complejos debido a sus ventajas en cuanto a la relación costo/rendimiento (Hwang, et al., 2012).

Por otro lado, un cluster puede ser homogéneo, si la arquitectura y sistema operativo de todas las máquinas son idénticos, o heterogéneo en caso contrario, siendo éste un factor importante para el análisis del rendimiento que puede obtenerse. Asimismo, cada nodo del cluster puede ser un multiprocesador de memoria compartida.

Los clusters pueden ser utilizados para: el cómputo de problemas complejos en paralelo, haciendo que sus nodos trabajen en conjunto resolviendo partes más pequeñas del problema original; servidores web, diseñando el *cluster* para brindar alta disponibilidad, detectando y solucionando automáticamente fallos que puedan surgir; bases de datos de alto rendimiento, teniendo un *cluster* de servidores que trabajen al unísono balanceando la carga y proveyendo alta disponibilidad; entre otras. (Lucke, 2004) (Baker & Buyya, 1999)

4.3 Evolución hacia la arquitectura multicore

Durante muchos años el incremento en la frecuencia del reloj, la incorporación de cachés junto con pequeñas modificaciones en el código de la aplicación para utilizarlas en forma sensata, y otros cambios introducidos en la arquitectura para explotar el paralelismo a nivel de instrucción ó *IPL* (*pipelining*, ejecución fuera de orden, predicción de saltos, ejecución especulativa, etc.) provocaron una mejora en el rendimiento de las aplicaciones. (Hennessy & Patterson, 2011)

Sin embargo se ha alcanzado un límite: problemas térmicos y energéticos imposibilitaron continuar aumentando la frecuencia del reloj; la velocidad de acceso a memoria no mejoró al ritmo que lo hizo la velocidad de los microprocesadores; y las técnicas para explotar el *IPL* pueden no ajustarse bien para aplicaciones cuyo código es difícil de predecir. Lo anterior ha motivado la búsqueda de mejoras en rendimiento a partir de explotar el paralelismo inherente de algunas aplicaciones mediante arquitecturas multithreading y multiprocesadores (Akhter & Roberts, 2006).

La arquitectura multicore, también conocida como *CMP* (*Chip Multiprocessor*), es un multiprocesador que incluye en un único chip 2 o más procesadores independientes llamados *core* o *núcleo*, teniendo un único socket para la conexión del chip en el *motherboard*. El Sistema Operativo percibe cada *core* como un procesador lógico independiente, pudiendo asignar a cada uno una aplicación distinta para su ejecución en paralelo. Otra alternativa que

surge al programar las aplicaciones utilizando técnicas de programación paralela es la de ejecutar los distintos procesos/threads de la aplicación sobre distintos *cores* reduciendo así el tiempo de ejecución comparado a la aplicación secuencial.

Hoy en día existe una variedad de procesadores multicore en su mayoría homogéneos, es decir todos sus *cores* son idénticos, que difieren entre sí en: el número de *cores*, la estructura y tamaño de las caché, y en la forma de acceso a la jerarquía de memoria. A grandes rasgos, los dos tipos de arquitecturas *multicore* más utilizadas son aquellas que cuentan con un diseño jerárquico y aquellas que poseen un diseño basado en red (Rauber & Rünger, 2010):

- Diseño jerárquico: los *cores* se agrupan de forma de compartir uno o varios niveles de caché; las caché se organizan en forma jerárquica, teniendo mayor capacidad aquellas que están más alejadas de los *cores*. La Figura 4.1 muestra un Quad-core Xeon E5405 conformado por 4 *cores*, cada *core* posee dos caché L1 de 32KB exclusivas, una para instrucciones y otra para datos, entre pares comparten una caché L2 de 6MB, y todos los *cores* tienen acceso a una memoria global. Los *cores* se comunican con el *controlador de memoria* a través de un bus de sistema llamado *Front Side Bus*, que causa un cuello de botella en el acceso a memoria. (Intel, 2007) La Figura 4.2 muestra una máquina con dos procesadores Quad-core de este modelo, resultando un multiprocesador cuyo tipo de acceso a memoria es *UMA.*

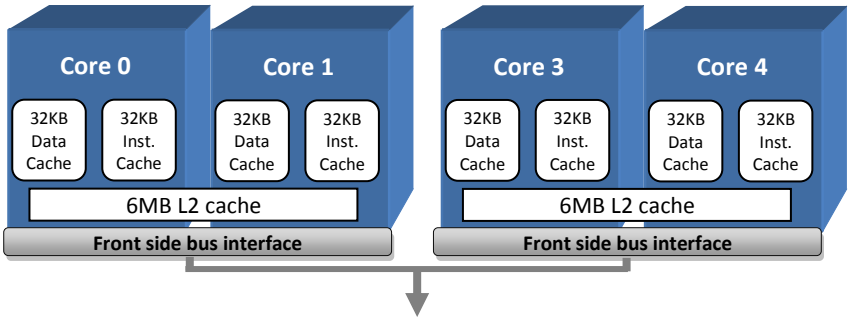


Figura 4.1 Quad-core Xeon 5405.

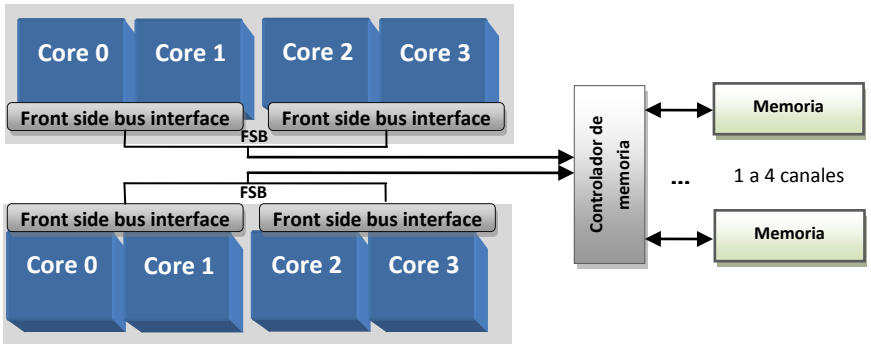


Figura 4.2 Dos Quad-core Xeon 5405 conectados a un mismo motherboard, tipo de acceso a memoria UMA

- Diseño basado en red: enlaces punto a punto de alta velocidad conectan los *cores* del chip, sus caché y las memorias. La transferencia de datos entre *cores* se realiza a través de estos

enlaces. La Figura 4.3 muestra un Quad-core Xeon E5620 que posee 4 cores físicos con tecnología *Hyperthreading*, cada core tiene dos cachés L1 de 64KB exclusivas para datos e instrucciones, una caché L2 propia de 256KB, y todos los cores comparten una caché L3 de 12MB. La red de interconexión *Quick Path Interconnect* reemplaza al *FSB* y asume que el procesador tiene un controlador de memoria integrado (Intel, 2010). Si se conectasen dos procesadores Quad-core de este modelo sobre el mismo *motherboard* el acceso a memoria del multiprocesador resultante es de tipo ccNUMA. Dicho esquema se muestra en la Figura 4.4.

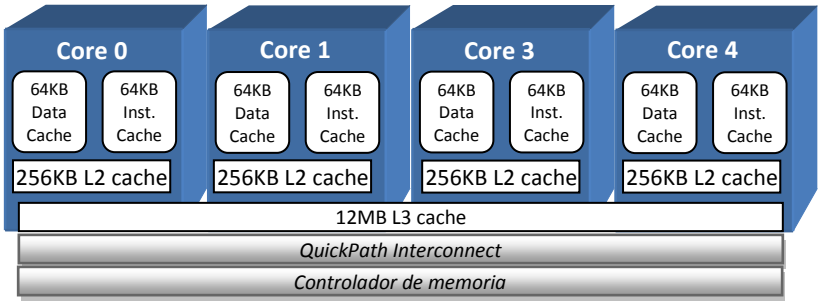


Figura 4.3 Quad-core Xeon E5620.

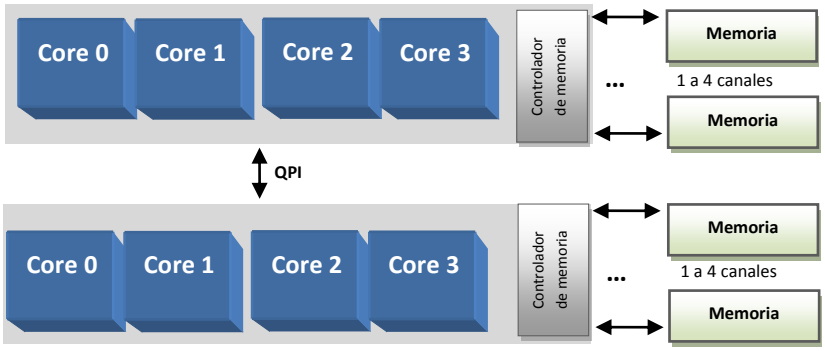


Figura 4.4 Dos Quad-core Xeon E5620 conectados a un motherboard, tipo de acceso ccNUMA.

4.4 Bibliotecas para el desarrollo de aplicaciones paralelas

El uso generalizado de *clusters* como plataforma de cómputo paralelo ha impulsado el surgimiento de diversas especificaciones para bibliotecas de paso de mensajes explícito, posibilitando así la implementación de aplicaciones portables. Estas bibliotecas son utilizadas en conjunto con lenguajes de programación estándar tales como C, C++, Fortran, entre otros. MPI (MPIForum, 2014) es un ejemplo de herramienta estándar para paso de mensajes.

Por otra parte, *Pthreads* (Engelschall, 2006) y *OpenMP* (OpenMP-ARB, 2014) se han convertido en las bibliotecas más utilizadas para la programación de multiprocesadores.

En las siguientes secciones se estudian las bibliotecas MPI y Pthreads, ya que son las que se utilizarán en esta tesis.

4.4.1 MPI

MPI (*Message Passing Interface*) es un estándar que define una API para la comunicación por paso de mensajes. Existen diversas implementaciones para distintas arquitecturas, de modo que internamente aprovechan las características físicas del sistema, y distintas distribuciones de MPI siendo OpenMPI y MPICH2 las más utilizadas. Dichas implementaciones respetan el estándar y en consecuencia el mismo código puede ser compilado en cualquiera de ellas. Se debe tener en cuenta que un programa compilado en una distribución no puede ser ejecutado en otra.

Entre las funciones básicas que provee MPI se encuentran aquellas que posibilitan:

- Inicializar y finalizar el entorno MPI.
- Comunicación punto a punto.
- Comunicación entre grupos de procesos (comunicación colectiva).
- Crear tipos derivados para permitir la comunicación de datos estructurados.

Por otro lado, un proceso MPI puede crear varios threads, los cuales podrían invocar a funciones MPI. El programador debe tener en cuenta que las funciones de comunicación punto a punto definidas por el estándar identifican a los procesos origen y destino en sus parámetros y no threads, y debe considerar también que un mensaje enviado por un thread hacia un proceso destino podría ser recibido por cualquier thread del mismo, siendo su responsabilidad la prevención de carreras entre éstos.

En caso de ser necesaria la comunicación entre threads de distintos procesos, el programador deberá implementar el direccionamiento explícitamente haciendo uso de *tags* especiales conformados por el identificador de thread destino (id) y tipo de mensaje, es decir los threads de un proceso diferenciarán a cuál de ellos está dirigido un mensaje recibido según el campo *tag* (Intel, 2008).

Asimismo, el estándar establece que las funciones MPI son *thread-safe*, es decir dos threads pueden invocar a funciones MPI y el resultado de la ejecución será como si dichas llamadas se hubiesen ejecutado en algún orden (aun cuando el procesamiento de las mismas no sea atómico). Adicionalmente determina que una función bloqueante sólo demora al *thread* que la invoca.

MPI define cuatro niveles diferentes de interoperabilidad entre los threads y MPI:

- MPI_THREAD_SINGLE: cada proceso tiene un único thread.
- MPI_THREAD_FUNNELED: el proceso tiene múltiples threads, pero sólo el thread maestro realizará llamadas a funciones MPI.
- MPI_THREAD_SERIALIZED: el proceso tiene múltiples threads, cualquiera puede realizar llamadas a funciones MPI pero en forma exclusiva, es decir de a uno por vez.
- MPI_THREAD_MULTIPLE: el proceso tiene múltiples threads, los cuales pueden invocar a funciones MPI en cualquier momento sin restricción alguna.

4.4.2 Pthreads

POSIX Threads (*Pthreads*) es una API para la creación y manipulación de threads. Existen diversas implementaciones para los distintos Sistemas Operativos como por ejemplo: GNU/Linux, FreeBSD, NetBSD, OpenBSD, entre otras.

Pthreads provee:

- Funciones para administrar los threads: creación y destrucción, espera entre threads (*join*).
- Mecanismos para garantizar exclusión mutua mediante espera pasiva (*mutex lock)*.
- Mecanismos para garantizar la exclusión mutua mediante espera activa (*spinlock)*.
- Sincronización por barrera.
- Variables condición.

A su vez existe una biblioteca llamada *Semaphore* que brinda la funcionalidad requerida para la sincronización a través de semáforos.

4.5 Discusión y conclusiones

Las arquitecturas cluster de *singlecore* se han convertido en plataformas comunes en el cómputo paralelo en décadas pasadas. Por otro lado, hoy en día existe una tendencia instalada y creciente al uso de procesadores *multicore* y *cluster de multicore*.

En general, gran parte de las aplicaciones paralelas existentes que se ejecutan sobre un *cluster* utilizan el estándar MPI. Si bien una aplicación MPI puede ejecutarse sobre un multiprocesador de memoria compartida, para adaptarse a las nuevas arquitecturas las implementaciones principales del estándar, tales como OpenMPI, han sido optimizadas para tomar ventaja de aquellas comunicaciones que puedan realizarse mediante memoria compartida siendo transparente al programador.

Sin embargo, existen ciertas ineficiencias al utilizar un modelo de paso de mensajes sobre una arquitectura de memoria compartida: los procesos acceden a su espacio de direcciones exclusivo, por lo que si éstos necesitan utilizar estructuras de datos comunes de sólo lectura las mismas estarán replicadas en el espacio de direcciones de cada proceso. Asimismo, en algunas aplicaciones sería necesario serializar los datos que deben comunicarse entre procesos para hacer efectiva la operación de envío, ya que en general ésta requiere que los datos residan en una zona contigua en la memoria.

A partir de lo anterior, los programadores han volcado sus esfuerzos para sacarle provecho a la arquitectura *cluster de multiprocesador* a través de la *programación híbrida* de las aplicaciones paralelas, es decir se crea un proceso por cada máquina del cluster, cada proceso crea threads para su ejecución dentro de la máquina multiprocesador y se utiliza: comunicación a través de variables compartidas entre los threads de la máquina y comunicación por paso de mensajes entre los procesos que residen en distintas máquinas del cluster.

CAPÍTULO 5: ALGORITMOS PARALELOS BEST-FIRST SEARCH

Los algoritmos de búsqueda informada *BFS,* utilizados para resolver problemas de optimización combinatoria, requieren una alta capacidad de cómputo y memoria. En el Capítulo 2 se han estudiado estrategias algorítmicas alternativas para reducir la cantidad de memoria utilizada por *BFS*, requiriendo para ello un esfuerzo adicional durante la búsqueda ya que llevan a expandir una mayor cantidad de nodos. Por otra parte, en el Capítulo 3 se presentó el desarrollo de heurísticas más afinadas como un recurso para disminuir la cantidad de nodos a tratar y, en consecuencia, reducir el tiempo de ejecución. Aun así, a medida que la complejidad del problema aumenta, el tiempo de resolución y cantidad de memoria a utilizar se incrementa exponencialmente volviéndose imposible su procesamiento mediante algoritmos secuenciales.

Por otro lado, hoy en día es común para las instituciones poseer una máquina con uno o varios procesadores *multicore*, que proveen gran potencia de cómputo, y clusters de procesadores *single-core* o *multicore*, que además de ampliar la capacidad de cómputo proveen un agregado de memoria importante. Por lo anterior, el diseño de algoritmos de búsqueda ha pasado de centrarse en el desarrollo de técnicas para optimizar al máximo un algoritmo secuencial o utilizar memoria externa para solventar la carencia de memoria RAM a desarrollar algoritmos que puedan aprovechar la potencia de estas arquitecturas y en consecuencia surgen nuevos retos tales como: la elección adecuada de las estructuras de datos a ser compartidas por los threads y la reducción de la contención y sincronización en el caso de las máquinas *multicore*; la búsqueda de la relación adecuada entre comunicación y procesamiento en el caso de los clusters; el desarrollo de técnicas para balancear la carga de trabajo y su calidad entre los procesos/threads, algoritmos para detectar la terminación del cómputo distribuido y métodos para detectar duplicados en forma absoluta; entre otras.

Es preciso aclarar que en Inteligencia Artificial el término *búsqueda paralela* se refiere a encontrar una secuencia de acciones para una instancia del problema distribuyendo la exploración entre diferentes procesadores, en contraste a una *búsqueda distribuida* que se refiere tanto a aspectos abarcados por la *búsqueda paralela* como también a resolver distintas instancias del problema en distintos procesadores.

Este capítulo realiza una síntesis del estado del arte de los algoritmos paralelos *Best First Search* con énfasis en la búsqueda de soluciones óptimas mediante A*, difíciles de paralelizar ya que requieren mantener la Lista Cerrada para podar duplicados en forma parcial o absoluta y además el uso de la información heurística debe ser idealmente a escala global permitiendo procesar la menor cantidad de nodos posible con costo mayor o igual al de la solución óptima. Las distintas técnicas analizadas se diferencian entre sí en la forma de administración de la Lista Abierta y de la Lista Cerrada, pudiendo ser globales a todos los threads (*estrategia centralizada*) o locales a cada proceso/thread (*estrategia distribuida o descentralizada*), lo que ocasiona que algunos algoritmos se ajusten mejor a arquitecturas de memoria compartida y otros a arquitecturas de memoria distribuida (Kumar, et al., 1988) (Dutt & Mahapatra, 1993)

(Cung & Le Cun, 1994) (Grama, et al., 2003) (Edelkamp & Schrödl, 2011). Por otro lado, las características del problema en particular y la heurística utilizada pueden influir en el rendimiento obtenido según la estrategia de paralelización seleccionada. (Kumar, et al., 1988) (Grama & Kumar, 1999)

En la Sección 5.1 se presentan las métricas para analizar el rendimiento de los algoritmos paralelos de búsqueda y las causas comunes de *overhead*. Luego, la Sección 5.2 estudia la estrategia centralizada, mientras que la Sección 5.3 estudia la estrategia distribuida, para ambos casos se analizan los retos a afrontar para obtener mayor rendimiento, las ventajas y desventajas de cada estrategia y se hace referencia a implementaciones conocidas al momento. A continuación, la Sección 5.4 presenta algoritmos simples para la detección de terminación en aplicaciones con cómputo distribuido y la Sección 5.5 trata la importancia de la elección de la biblioteca de gestión de memoria dinámica para los algoritmos de búsqueda paralela cuando se ejecutan sobre un multiprocesador. Por último, la Sección 5.6 presenta una discusión de posibles mejoras a los algoritmos paralelos de búsqueda existentes que constituyen los desafíos a afrontar en esta tesis.

5.1 Análisis de rendimiento. Causas de degradación de rendimiento de un sistema paralelo.

En general el rendimiento de un algoritmo secuencial se evalúa en términos de su tiempo de ejecución, que depende del tamaño de la entrada y/o de los datos de entrada en sí, siendo su comportamiento asintótico igual en cualquier plataforma.

Sin embargo, el rendimiento de un algoritmo paralelo no sólo depende la instancia de entrada, además existen otros factores con influencia tales como: la arquitectura (cantidad de procesadores, características del medio de comunicación, jerarquías de memoria), la distribución de los procesos en procesadores, la técnica utilizada para balancear la carga, etc.

Es importante que al evaluar el desempeño del sistema paralelo luego pueda ser comparado contra el rendimiento obtenido por otro. En este sentido, entre las métricas más conocidas se encuentran el *Speedup,* la *Eficiencia*, y el concepto de *Escalabilidad*.

Asimismo, se deben tener en cuenta las causas de *overhead* que provocan la degradación del rendimiento del sistema paralelo, en particular para los algoritmos paralelos *Best First Search.*

5.1.1 Métricas: Speedup y Eficiencia

El factor de *Speedup absoluto* S_p mide la ganancia en rendimiento obtenida al resolver un problema en paralelo con respecto a su implementación secuencial, y se define como la relación entre el tiempo de ejecución del mejor algoritmo secuencial (T_s) y el tiempo de ejecución del algoritmo paralelo sobre una máquina con P procesadores (T_p) (Grama, et al., 2003) (Quiin, 1994). La Figura 5.1 muestra la fórmula para calcular el *Speedup*. En este caso se

asume que la arquitectura de la máquina sobre la que se ejecutó el algoritmo secuencial es idéntica a la arquitectura de la máquina paralela usada para ejecutar el algoritmo paralelo.

$$S_p = \frac{T_s}{T_p}$$

Figura 5.1. Speedup

Teóricamente, si la arquitectura paralela está conformada por *P* procesadores se esperaría obtener un *Speedup* entre 1 y *P.* En algunos casos el *Speedup* puede ser mayor a *P* dando lugar al concepto de *Superlinealidad* que se presenta usualmente en algunos algoritmos no determinísticos (por ejemplo los algoritmos de búsqueda en grafos) a causa de anomalías que ocurren cuando el trabajo realizado por el algoritmo paralelo es menor al realizado por el algoritmo secuencial (Rao & Kumar, 1988), o cuando debido a características de hardware el algoritmo secuencial se encuentra en desventaja respecto al algoritmo paralelo ejecutado sobre la máquina paralela (Gustafson, 1990).

Normalmente el *Speedup* estará limitado por diversos factores (Grama, et al., 2003), que deberán tenerse en cuenta con el objetivo de alcanzar el máximo rendimiento posible:

- El máximo grado de concurrencia que puede obtenerse de la aplicación paralela.
- El componente secuencial del algoritmo que no puede ser resuelto en paralelo.
- Exceso de cómputo en el algoritmo paralelo: el algoritmo secuencial más rápido puede ser difícil o imposible de paralelizar, teniendo que implementar un algoritmo paralelo en base a una técnica deficiente. Asimismo, un algoritmo paralelo basado en técnicas óptimas puede presentar exceso de cálculo debido a que algunos resultados, que en el algoritmo secuencial eran reusados, no pueden reutilizarse durante el cómputo paralelo ya que están siendo generados por distintos procesadores y por eso deben calcularse múltiples veces.
- *Overhead* de sincronización y comunicación entre procesos.
- Desbalance de carga.

La *Eficiencia* mide el uso efectivo de los recursos de cómputo y resulta una métrica de calidad y de costo del algoritmo paralelo. La misma se define como la relación entre el *Speedup* y el *Speedup óptimo* (Grama, et al., 2003) (Quiin, 1994) . En el caso de una arquitectura homogénea, el Speedup óptimo es *P* (el número de procesadores utilizados). La Figura 5.2 muestra la fórmula para calcular la Eficiencia.

$$E = \frac{S_p}{P}$$

Figura 5.2. Eficiencia

La Eficiencia varía entre 0 y 1, alcanzar valores cercanos a 1 significa que se logra un S_p cercano al óptimo *P*, y no siempre puede mantenerse al incrementar el tamaño de los problemas, al incrementar el número de procesadores o al portar el algoritmo sobre otra arquitectura.

5.1.2 Escalabilidad

La *Escalabilidad* de un sistema paralelo es una medida de su capacidad para utilizar eficientemente un número creciente de procesadores.

Diversos autores han propuesto métodos para evaluar la escalabilidad de los sistemas paralelos. Un método representativo para arquitecturas homogéneas es el análisis de *isoeficiencia*, el cual plantea que un sistema es escalable si es posible mantener la eficiencia del sistema en un valor fijo al aumentar el número de procesadores (escalar la arquitectura) y el tamaño del problema (escalar el problema). La *función de isoeficiencia* indica cuánto tiene que aumentar el tamaño del problema para poder incluir más procesadores sin que la eficiencia del sistema se vea afectada (Grama, et al., 1993).

El modelo de *isoeficiencia* se basa en las siguientes consideraciones:

- Si un sistema paralelo es usado para resolver un problema de tamaño fijo, el *Speedup* no continúa creciendo a medida que se incrementa el número de procesadores, debido a los factores mencionados en la sección anterior, provocando la caída de la *Eficiencia*. Esto sucede porque el tiempo de overhead (T_o) se incrementa a medida que P crece.

- En algunos sistemas paralelos, la eficiencia aumenta al escalar el tamaño del problema manteniendo el número de procesadores constante. En estos casos el aumento en la eficiencia se debe a que T_o crece en menor medida que el tiempo de resolución del problema.

5.1.3 Métricas para Algoritmos de Búsqueda Paralela *Best-First.*

En general, el análisis de rendimiento de los algoritmos de búsqueda paralela sobre grafos no es trivial ya que el *Speedup* obtenido para una misma instancia de entrada puede variar de una ejecución a otra. Esto es causado por la determinación dinámica de la porción del grafo que examina cada proceso/thread, ya que se utilizan técnicas de balance de carga dinámicas, y por el arribo en distinto momento de los nodos del grafo comunicados entre procesos/threads en las diferentes ejecuciones. Además, a colación de lo anterior, la solución encontrada para una misma instancia de entrada puede variar entre las ejecuciones, ya que pueden existir múltiples soluciones óptimas.

Asimismo, esta clase de algoritmos es propensa a manifestar anomalías en el *Speedup*, es decir se puede obtener un *Speedup* mayor que P (siendo P el número de procesadores).

Antes de analizar las causas de la aparición de *Superlinealidad* concretamente para el algoritmo A*, vale la pena recordar que el algoritmo secuencial expande siempre todos los nodos con costo menor a C* (costo de la solución óptima), llamados *nodos esenciales*, algunos nodos con costo igual a C* y ningún nodo con costo mayor a *C*,* los nodos de las dos últimas categorías son llamados *no esenciales*. Por otra parte, el algoritmo paralelo debe procesar

todos los nodos *esenciales*[10], algunos nodos con costo igual a C*, e incluso podría trabajar sobre nodos con costo mayor a C* (por ejemplo esta situación se da cuando quedan por procesar nodos con costo C* y la cantidad de nodos abiertos con dicho costo es menor que *P*, provocando que algunos procesos realicen *trabajo especulativo* sobre nodos con costo mayor al de la solución óptima). En consecuencia, el procesamiento adicional de nodos por parte del algoritmo paralelo es una causa de *overhead*, que empeora cuando se trabaja con una *estrategia distribuida* donde cada proceso/thread tiene su propia Lista Abierta, ya que éstos tendrán una visión *local* de la guía heurística, es decir los nodos expandidos por algunos procesadores son localmente prometedores pero pueden no serlo globalmente.

Para relacionar la cantidad de nodos que expanden el algoritmo secuencial y el algoritmo paralelo se utiliza una fórmula conocida como *Overhead de Búsqueda* (Grama, et al., 2003), que se muestra en la Figura 5.3, la cual calcula el porcentaje de nodos que el algoritmo paralelo expande de más respecto al algoritmo secuencial. El valor resultante puede ser: cero, lo cual indicaría que ambos algoritmos están expandiendo la misma cantidad de nodos; mayor a cero, lo cual indicaría que el algoritmo paralelo está expandiendo mayor cantidad de nodos que el algoritmo secuencial y su valor indica el porcentaje del aumento; menor a cero, lo que indica que el algoritmo paralelo procesó menor cantidad de nodos que el algoritmo secuencial.

$$OB = 100x(\frac{nro\ estados\ expandidos\ por\ alg.paralelo}{nro\ estados\ expandidos\ por\ alg.secuencial} - 1)$$

Figura 5.3. *Overhead de Búsqueda.*

Un *Overhead de Búsqueda* menor a cero puede darse cuando el algoritmo paralelo procesa una cantidad menor de nodos con costo C* respecto al algoritmo secuencial, lo que puede conducir a obtener un *Speedup Superlineal*. Esto ocurre generalmente en problemas con gran cantidad de nodos de igual costo, donde el algoritmo paralelo distribuye estos nodos entre los procesadores causando que se encuentre la solución más rápido en comparación al algoritmo secuencial. (Grama, et al., 2003)

Otro aspecto importante a tener en cuenta al momento de analizar el rendimiento obtenido por esta clase de algoritmos que utilizan balance de carga dinámico es la bondad de la distribución de nodos entre procesadores. Una mala distribución de la carga causará una degradación considerable en el rendimiento obtenido ya que algunos procesadores, en vez de realizar trabajo útil, pueden quedar ociosos por tiempo prolongado. Para analizar dicho aspecto se utiliza una fórmula que mide el *Desbalance de Carga,* la cual se muestra en la Figura 5.4. El valor obtenido por la fórmula es 0 si la carga de trabajo estuvo balanceada en forma perfecta, y mayor a 0 en caso contrario.

$$DC = \frac{max\ nro\ de\ estados\ expandidos\ por\ un\ proceso - min\ \ nro\ de\ estados\ expandidos\ por\ un\ proceso}{promedio\ de\ estados\ expandidos\ por\ todos\ los\ procesos}$$

Figura 5.4. *Desbalance de carga*

[10] La cantidad de nodos esenciales que expande el algoritmo paralelo puede ser mayor a aquella del algoritmo secuencial a causa de la reapertura de nodos, aun cuando se trabaja con heurísticas consistentes.

5.2 Estrategia Centralizada: A* Paralelo con Estructuras de Datos Globales.

La *estrategia centralizada*, también conocida como *Parallel Global A* (PGA*),* mantiene una única Lista Abierta y una única Lista Cerrada globalmente accesibles por todos los procesos/threads y permite que cada uno trabaje sobre uno de los *p* nodos más prometedores disponibles en la Lista Abierta al momento, siendo *p* la cantidad de procesos/threads. Dado que las listas serán accedidas frecuentemente, se deben mantener en una memoria cercana a todos los procesadores, adaptándose bien a arquitecturas de memoria compartida ya que se puede almacenar las mismas en la memoria global. Puesto que todos los threads estarán en cada momento eliminando el nodo más prometedor de la Lista Abierta e insertando en ésta los nodos sucesores generados, insertando nodos procesados en la Lista Cerrada y en algunos casos promoviendo o reabriendo nodos, se deberán usar *locks* para garantizar el acceso exclusivo y así mantener la consistencia de estas estructuras.

Por otra parte, el criterio de terminación del algoritmo paralelo debe modificarse respecto al utilizado en el algoritmo secuencial, ya que si se finaliza la búsqueda paralela cuando se encuentra una solución no se garantizaría que la misma sea óptima. Esto puede ocurrir porque cada procesador trabaja sobre uno de los *p* mejores nodos de la Lista Abierta; uno de ellos puede detectar un nodo final mientras que otro puede estar todavía trabajando sobre un nodo con menor costo que la solución encontrada y que conduce a la solución óptima (Kumar, et al., 1988) . La terminación debe ocurrir cuando se encuentra un nodo que representa el estado final y ningún otro thread está trabajando sobre un nodo con costo menor a la solución parcial encontrada, siendo este criterio suficiente ya que los nodos que estaban siendo expandidos eran los *p* mejores globalmente (Cung & Le Cun, 1994).

Por último, todos los threads deben conocer el costo de la mejor solución parcial encontrada con el propósito de utilizarlo para podar nodos que no conducen a una solución de costo menor.

Las ventajas de poseer listas globalmente accesibles son: la simplicidad de programación, ya que no se requiere ninguna técnica adicional para balancear la carga de trabajo entre los procesadores ni tampoco para equiparar la calidad de los nodos sobre los cuales se trabaja; la detección de duplicados la sigue realizando cada thread sobre los nodos que genera, siendo la misma absoluta; y por último la técnica genera un bajo *Overhead de Búsqueda* gracias a que usa la información heurística en forma completa dando lugar a que todos los threads estén igualmente informados.

La principal desventaja reside en que el rendimiento estará limitado por la contención en el acceso a las listas globales. Si bien para la Lista Cerrada, por estar implementada como una *Tabla Hash*, se pueden manejar múltiples *locks* permitiendo bloquear sólo la parte sobre la cual trabajará el thread, para mantener la consistencia en la *MinHeap* tradicional utilizada para implementar la Lista Abierta se debe utilizar un único lock. Por ello, cuando el acceso a la Lista Abierta es frecuente para el problema en particular a causa de que el cálculo de la heurística

no es costoso computacionalmente, el rendimiento empeora aún más ya que los threads estarán reiteradamente bloqueados esperando por el acceso a la misma.

Si bien la estrategia centralizada se adecua mejor a problemas de granularidad gruesa (por el alto tiempo en el cómputo de la heurística), la técnica no escala bien cuando crece la cantidad de procesadores, siendo en algunos casos peor su rendimiento que aquel obtenido por el algoritmo A* secuencial. (Kumar, et al., 1988)(Dutt & Mahapatra, 1993)(Burns, et al., 2010)(Cung & Le Cun, 1994) (Kishimoto, et al., 2013)

Diversos autores han investigado acerca de la concurrencia en las operaciones realizadas sobre colas de prioridades, las cuales podrían aplicarse para reducir la contención en el acceso a la Lista Abierta (Rao & Kumar, 1988) (Jones, 1989) (Mans & Roucairol, 1990) (Hunt, et al., 1996) (Sundell & Tsigas, 2005) (Herlihy & Shavit, 2008) (Edelkamp & Schrödl, 2011). En el trabajo (Kumar, et al., 1988) notan mejoras en el rendimiento sólo para instancias y problemas cuyo tiempo de computo de heurística es alto. Estas conclusiones también son compartidas por (Cung & Le Cun, 1994) pero, aunque alegan que en algunos casos su algoritmo obtiene mayor eficiencia en comparación al implementado por Kumar, el algoritmo en realidad está paralelizando la estrategia de búsqueda AnytimeWA* (Hansen & Zhou, 2007) ya que modifican la función de costo para conducir la búsqueda rápidamente hacia una solución y continúan la misma hasta obtener la solución óptima. Asimismo, los autores del trabajo (Burns, et al., 2010) utilizan estructuras de acceso concurrente sofisticadas para la implementación de la Lista Abierta y realizan pruebas sobre procesadores multicore actuales, pero no obtienen un rendimiento competitivo.

5.3 Estrategia Distribuida: A* Paralelo con Estructuras de Datos Locales.

La *estrategia distribuida*, también conocida como *Parallel Local A** (PLA*), mantiene una Lista Abierta y una Lista Cerrada local para cada proceso/thread, por lo que cada uno realizará una búsqueda cuasi independiente y así desaparece el problema de contención en las listas. Esta estrategia se adapta bien a arquitecturas de memoria compartida y a arquitecturas de memoria distribuida, pero surge la necesidad de comunicación entre los procesos/threads por las siguientes razones:

- Dado que al principio sólo uno tendrá en su Lista Abierta el nodo inicial y que el grafo se genera durante la ejecución, se debe distribuir la carga de trabajo en forma dinámica evitando así que un proceso/thread sin nodos abiertos quede ocioso hasta el final del cómputo.
- Los nodos ubicados en la Lista Abierta de un procesador no necesariamente son los mejores globales, por lo que se debe realizar una equiparación de la calidad de los nodos para que todos los procesadores realicen su trabajo sobre *nodos esenciales.*
- Los nodos duplicados (que representan un mismo estado) pueden generarse en distintos procesos/threads. Si la detección de duplicados sólo se realiza en el proceso/thread que generó el nodo y/o en aquel que recibió el nodo por balance de carga, la detección y poda de duplicados será *parcial* ya que otro puede tener en su Lista Abierta o en su Lista Cerrada un nodo representando el mismo estado. En cambio, si se quiere realizar una

detección y poda *absoluta* se requieren estrategias que asignen cada estado a un procesador particular.

- El criterio de terminación debe modificarse ya que se dispone de múltiples Listas Abiertas inconsistentes y, a causa del uso de balance de carga dinámico, pueden existir nodos del grafo siendo comunicados entre los procesos/threads. La terminación debe detectarse cuando se ha encontrado una solución, ningún proceso/thread tiene nodos con mejor costo en su Lista Abierta y no existen nodos siendo comunicados entre los procesos/threads que no fueron recibidos.
- Los costos de las soluciones parciales encontradas deben comunicarse y serán utilizados para podar caminos que conducen a soluciones de costo subóptimo.

Los primeros algoritmos paralelos *BFS* propuestos realizaban una *distribución estática* inicial de los nodos del grafo, y luego durante la ejecución utilizaban una técnica de *balance de carga dinámica* que también se encargaba de realizar la equiparación de calidad de los nodos. Algunas de estas técnicas transfieren trabajo periódicamente desde un procesador donador a otro procesador seleccionado en forma *random (Estrategia de Comunicación Random)* o al procesador sucesor en una organización en anillo (*Estrategia de Comunicación en Anillo*). Otras se basan en el requerimiento de trabajo por parte del procesador ocioso a un procesador donador seleccionado ya sea en forma random *(Random Polling*), round-robin (*Asynchronous Round-Robin*) o consultando un valor globalmente compartido (*Global Round-Robin*). Los nodos enviados por el donador pueden ser sucesores del nodo que estaba siendo expandido o algunos de los mejores nodos de la Lista Abierta. Las alternativas anteriores son distribuidas y se adaptan a arquitecturas de memoria compartida o distribuida. Por otro lado, se han desarrollado estrategias de balance de carga centralizadas sólo aptas para arquitecturas de memoria compartida. Una de ellas, conocida como estrategia *Blackboard,* mantiene en memoria un repositorio de nodos a través del cual los threads realizan intercambios: cuando el thread selecciona de su Lista Abierta el mejor nodo local, lo expande sólo si su valor no sobrepasa un límite de tolerancia impuesto en relación al mejor nodo del repositorio, en ese caso envía algunos de sus mejores nodos al repositorio, y en caso contrario toma algunos nodos desde el repositorio (Kumar, et al., 1988) (Grama, et al., 2003). Cualquiera de estas técnicas provocan que un estado pueda aparecer en cualquier procesador por lo que sólo lograban una *detección de duplicados parcial.*

En este contexto, el trabajo (Kumar, et al., 1988) realiza un estudio de las estrategias de balance de carga dinámica *Blackboard, Anillo* y *Random* y relaciona el rendimiento obtenido con las características propias del problema a resolver (*Puzzle-15, VCP y TSP*). Si bien el trabajo experimental es realizado sobre una máquina de memoria compartida, sus conclusiones se generalizan para arquitecturas de memoria distribuida.

Con este objetivo, generan un histograma de costos para distintas instancias de los problemas antes mencionados, en el cual para cada costo de nodos procesados durante ejecución c_i (eje x) se indica la cantidad de nodos con dicho costo V_i (eje y). Para los problemas *Puzzle-15 y VCP*, V_i crece en forma abrupta a medida que aumenta c_i, y el mayor porcentaje de nodos procesados tiene costo igual al de la solución óptima C*; para el problema *TSP* ocurre lo contrario, V_i crece lentamente siendo baja la cantidad de nodos para cada costo c_i.

Para todos los problemas la estrategia *Blackboard* resulta ser la más efectiva, ya que se hace uso completo de la información heurística por ser una estrategia centralizada, pero requiere un intercambio con el repositorio muy frecuente, caso contrario el rendimiento disminuye, por lo que si se implementase sobre memoria distribuida sería poco efectiva.

Por otro lado, para los problemas *VCP* y *15-Puzzle* las estrategias *Anillo* y *Random* son más efectivas que para *TSP*, esto ocurre ya que al ser técnicas distribuidas se pierde parte de la guía heurística, lo cual no afecta tanto a problemas que tienen gran cantidad de nodos con costo idéntico.

Por último, analizan las anomalías en el *speedup* obtenido para *VCP* y *15-Puzzle* y concluyen que la razón se encuentra en que dichos problemas poseen gran cantidad de nodos con costo igual a C* que, al distribuirlos entre los procesos, provocan que el algoritmo paralelo encuentre la solución procesando menor cantidad de nodos que el algoritmo secuencial.

Asimismo, el artículo (Dutt & Mahapatra, 1993) realiza la paralelización del algoritmo SEQ_A*[11] para el problema *TSP* sobre un multicomputador Hipercubo y utiliza diversas técnicas de balance de carga, a saber: *Estrategia de Comunicación Random, Round-Robin* y una estrategia propia llamada *Quality Equalizing* que mejora en rendimiento a las anteriores.

Concluyen que si bien los algoritmos propuestos muestran un beneficio significativo en rendimiento cuando se los compara con algoritmos paralelos similares que no realizan poda de duplicados, la ventaja se estrecha al incrementar la cantidad de procesadores ya que el espacio de estados se fragmenta más y se genera una cantidad considerable de duplicados entre procesadores, incrementando la cantidad de trabajo replicado y la cantidad de memoria utilizada, lo que provoca que el beneficio de la *poda parcial* desaparezca. Por lo anterior, manifiestan que es importante eliminar los nodos duplicados en forma *absoluta*.

La única técnica conocida para realizar una detección de duplicados y poda *absoluta* es asignar cada estado a un procesador basándose en una clave calculada a partir de la representación del estado y haciendo un módulo por la cantidad de procesadores. De esta manera, todos los nodos que representen a un mismo estado arribarán al mismo procesador el cual podrá realizar la verificación por duplicados sobre sus estructuras locales. La función que realiza la asignación es llamada *función hash global*, ya que los nodos generados por un procesador pueden pertenecer a cualquier otro procesador del sistema, requiriendo una comunicación global (Manzini & Somalvico, 1990) (Edelkamp & Schrödl, 2011) (Grama, et al., 2003) (Mahapatra & Dutt, 1993). Cuando un procesador expande un nodo generando los sucesores del mismo, calcula el valor hash de cada sucesor y los comunica a su procesador asignado. Luego se abren dos posibilidades:

- El procesador asignado realiza la verificación por duplicados sobre sus estructuras y le envía el resultado al procesador que generó el nodo para que éste lo procese. Esta estrategia genera sincronismo entre ambos procesadores, ya que aquel que generó el nodo tiene que esperar la respuesta, requiere la utilización de una técnica para balancear la carga entre procesadores y en arquitecturas de memoria distribuida produce mayor

[11] SEQ_A* se diferencia de A* en que la expansión de un nodo se realiza en forma parcial.

tráfico en la red por los mensajes de respuesta. Esta técnica es conocida como *Estrategia Request-Response Distribuida.*

- El procesador asignado realiza la verificación de duplicados sobre sus estructuras y si el nodo es aceptable lo inserta en su Lista Abierta para procesarlo. De esta manera, la comunicación puede ser asíncrona, no se requiere una técnica adicional para balancear la carga ya que implícitamente lo logra la *función hash,* y en arquitecturas de memoria distribuida se evita incrementar la cantidad de mensajes en la red por respuestas enviadas al emisor del nodo. Esta técnica es conocida como *Transposition-Driven Scheduling (TDS)* (Romein, et al., 1999). El algoritmo A* paralelo que utiliza esta técnica se denomina *PLA* GOHA (Parallel Local A* GlObal HAshing of Nodes).* Además, para reducir la cantidad de comunicaciones por migración de nodos, cada procesador puede armar un paquete con varios nodos dirigidos al mismo procesador destino antes de transferirlos.[12]

La estrategia *TDS* es la que se utiliza comúnmente. La Figura 5.5 muestra la comunicación de nodos en las técnicas antes mencionadas.

El algoritmo *Parallel Retracting A* (PRA*)* (Evett, et al., 1991)(Evett, et al., 1995) paraleliza *Retracting A*,* una variante de A* apta para su ejecución sobre máquinas con memoria limitada, sobre una arquitectura *SIMD* y utiliza una *función hash global* para lograr la detección y poda de duplicados absoluta y el balance de carga en forma implícita. La diferencia con A* reside en que no se almacenan todos los nodos generados; cuando la memoria de alguno de los procesadores se agota se libera parte de la misma eliminando algunos nodos de su Lista Abierta, pero se debe almacenar el costo de éstos en el nodo padre el cual volverá a estar disponible para expansión y re-generación de los sucesores eliminados teniendo valor de costo igual al mínimo costo entre sus hijos. El algoritmo utiliza un mecanismo síncrono de comunicación para distribuir nodos lo cual es fuente de *overhead*: todo proceso que recibe un nodo debe informar al emisor si efectivamente fue recibido y almacenado, o si fue eliminado por el mecanismo de liberación de memoria antes comentado, ya que en ese caso el emisor tendrá que actualizar el nodo padre.

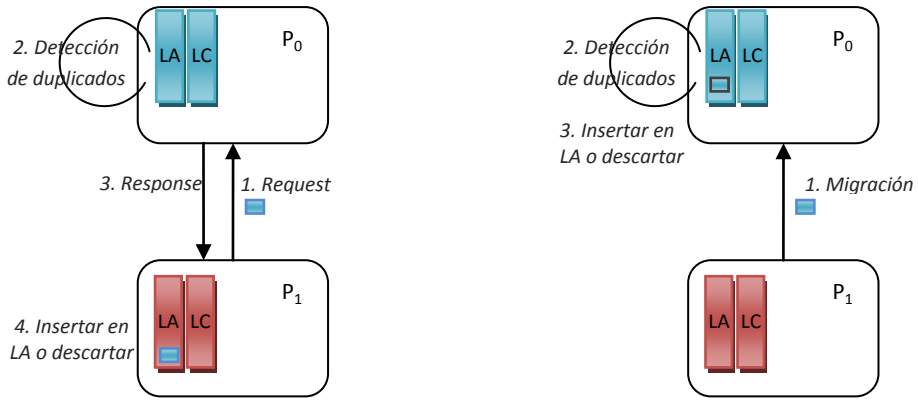


Figura 5.5. A la izquierda, Estrategia Request-Response Distribuida.
A la derecha, Transposition-Driven Scheduling (TDS).

[12] Ésta técnica fue propuesta en el trabajo (Romein, et al., 2002) donde se evalúa el comportamiento de IDA* sobre una máquina con memoria distribuida y analizan la influencia del tamaño del paquete armado sobre el rendimiento del algoritmo paralelo.

Por otra parte, el trabajo (Mahapatra & Dutt, 1993) estudia las desventajas de utilizar una *función hash global* para paralelizar SEQ_A* sobre un multicomputador Hipercubo, ya que: (a) el balance de carga implícito realizado por la *función hash* podría ser insatisfactorio para el problema particular (b) la transmisión de nodos por poda de duplicados se realiza a nivel global y puede generar gran cantidad de conflictos en la red.

Para solucionar el primer problema proponen desacoplar el balance de carga de la detección de duplicados. Para ello utilizan una *función hash* para realizar la detección de duplicados absoluta y la técnica de balance de carga independiente *Quality Equalizing.* De esta manera, el procesador que genera un nodo lo envía al procesador dueño quien verifica si es o no un duplicado y en caso de ser aceptable lo inserta en su Lista Abierta para expandirlo potencialmente o enviarlo a otro procesador por medio del algoritmo de balance de carga.

Para resolver el segundo problema realizan un particionado del grafo de búsqueda por niveles y asignan en forma estática y circular un conjunto de niveles contiguos (llamado *cluster*) a subconjuntos de procesadores vecinos en la arquitectura. De esta forma, la comunicación de nodos para la detección de duplicados queda confinada a subconjuntos de procesadores vecinos reduciendo los conflictos en la red lo cual aumenta la escalabilidad.

El algoritmo generado que utiliza ambas técnicas es llamado *PLA* LOHA & QE (Parallel Local A* LOcal HAshing of Nodes & Quality Equalizing)*. El problema que utilizan como caso de estudio es el *TSP* en donde los nodos duplicados sólo aparecen dentro de un nivel dado, es decir cualquier duplicado pertenece al mismo nivel que su copia y por ello será enviado al mismo subconjunto y al mismo procesador. Por lo anterior, la comunicación por detección de duplicados es interna al subconjunto de procesadores, en caso que un procesador genere un nodo perteneciente a alguno de los *clusters* asignados a su subconjunto, o localizada al subconjunto de procesadores vecino, en caso que un procesador genere un nodo que pertenece a otro nivel fuera de sus *clusters*.

Por último, comparan el speedup obtenido por los algoritmos paralelos *PLA* GOHA* y una versión de éste llamada *PLA* GOHA & QE,* que utiliza la técnica de balance de carga independiente *Quality Equalizing (QE)*, contra el algoritmo *PLA* LOHA & QE* sobre un multicomputador Hipercubo. Los resultados muestran una mayor eficiencia de los algoritmos *PLA* GOHA & QE* y *PLA* LOHA & QE*, los cuales alcanzan resultados similares, siendo seguidos por *PLA* GOHA* y luego por *PLA** con detección de duplicados parcial y técnica de balance de carga *QE.*

Sin embargo, la técnica *Local Hashing* expuesta anteriormente no puede aplicarse a cualquier clase de problemas ya que es dependiente del dominio y sensible a la arquitectura paralela. Los problemas que se ajustan a esta técnica son aquellos donde los nodos que representan un mismo estado aparecen en el mismo nivel, es decir tienen igual longitud de camino partiendo desde el nodo inicial, que no es el caso de los problemas de planificación ni tampoco del *15-Puzzle*. (Edelkamp & Schrödl, 2011) (Kishimoto, et al., 2013)

Basándose en lo mencionado anteriormente, el algoritmo paralelo HDA* (*Hash Distributed A**) (Kishimoto, et al., 2009)(Kishimoto, et al., 2013) paraleliza A* sobre arquitecturas actuales *cluster de multicore*, utilizando una *función hash global* para distribuir la carga y alcanzar una

poda de duplicados absoluta, en este sentido es similar al algoritmo PLA* GOHA propuesto en (Mahapatra & Dutt, 1993) y a PRA* pero difieren en que HDA* paraleliza A*. El algoritmo fue implementado haciendo uso puramente de la biblioteca de paso de mensajes MPI, y a diferencia de PRA* el mecanismo de comunicación es asincrónico. Al principio, el proceso correspondiente inserta el nodo inicial en su Lista Abierta. Luego cada procesador realiza iteraciones donde:

- Comprueba si ha recibido uno o más nodos a través de mensajes. En ese caso, para cada estado recibido *s* verifica si está en la Lista Cerrada y/o en la Lista Abierta, para determinar si debe ser insertado en la Lista Abierta o podado. [13]
- Si no había recibido mensaje con nodos, el proceso selecciona un nodo de su Lista Abierta y lo expande generando sucesores, luego calcula para cada uno su *valor hash* para conocer a qué proceso pertenece y lo envía a su dueño a través de un mensaje en forma asincrónica y no bloqueante.

Las funciones MPI utilizadas son MPI_Bsend y MPI_Iprobe. Para reducir el *overhead* de comunicación ocasionado por el envío de nodos, utilizan la idea propuesta por (Romein, et al., 2002) de empaquetar en un mismo mensaje un número determinado de nodos antes de que éste sea enviado. La cantidad de nodos a empaquetar depende de factores como: la configuración de la red, el número de procesadores, su velocidad, entre otros.

Para alcanzar una distribución de nodos uniforme, necesario para lograr un balance de carga efectivo, utilizan la *función hash* de *Zobrist* (Apéndice B) (Zobrist, 1970)(Millington & Funge, 2009). Por otro lado, para detectar terminación utilizan el *algoritmo de tiempo de Mattern* (Mattern, 1987).

Los mismos autores realizan una implementación del algoritmo *PLA* GOHA* sincrónico para arquitecturas de memoria compartida, utilizando la herramienta de programación Pthreads y *locks* altamente optimizados en lenguaje *ASM*. Cada thread t_i del sistema paralelo posee sus listas locales. Cuando el thread t_i genera un nodo perteneciente a otro thread t_j debe depositarlo en un espacio de memoria compartido para que t_j pueda tomarlo, por ello cada thread posee una *cola de entrada* globalmente conocida por todos los threads del sistema. Dado que varios threads pueden querer depositar al mismo tiempo nodos en la cola de entrada de un thread particular, la misma debe ser protegida por un lock para garantizar la consistencia. Este algoritmo sigue la misma línea que el algoritmo PRA* original ya que existirá sincronismo en la comunicación, a excepción que se está paralelizando el algoritmo A*.

La Figura 5.6 muestra el esquema de comunicación del algoritmo PLA* GOHA sincrónico para arquitecturas de memoria compartida: los threads T_0 y T_3 han generado un nodo cuyo dueño es T_2, por lo que deben depositarlo en la cola de entrada de dicho thread para lo cual deben obtener el acceso a la misma. En este ejemplo T_0 no logra obtener el lock por lo que se bloquea; en cambio T_3 obtiene el acceso, copia el puntero al nodo en la cola de entrada de T_2 y libera el lock.

[13] A diferencia del algoritmo secuencial A*, aunque se utilice una heurística consistente puede darse lugar a reapertura de nodos, ya que un proceso puede recibir distintos nodos representando al mismo estado vía mensaje de distintas fuentes, los cuales pueden arribar en cualquier orden.

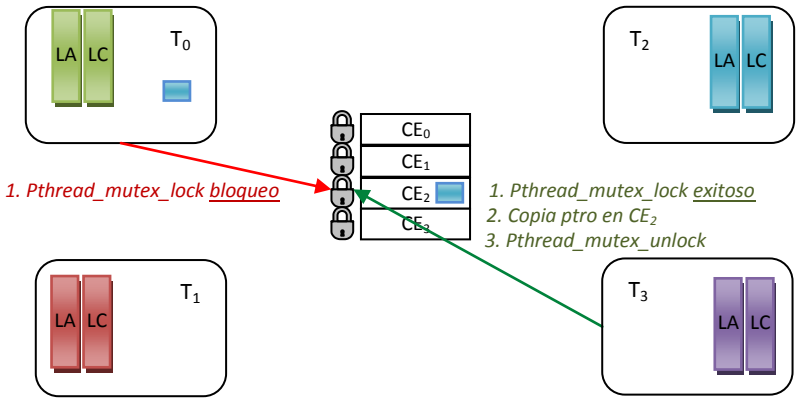


Figura 5.6. Esquema de comunicación del algoritmo PLA* GOHA sincrónico para memoria compartida.

El trabajo experimental lo realizan ubicando a HDA* como el mecanismo de búsqueda de un planificador independiente del dominio llamado *Fast-Downward* (Helmert , 2006) y por otro lado utilizando una aplicación específica, en este caso el *24-Puzzle,* en la cual el procesamiento de un estado es más rápido por ser dependiente del dominio.

Con el fin de investigar el impacto de las comunicaciones sincrónicas y asincrónicas, comparan el rendimiento de HDA* contra el algoritmo *PLA* GOHA* sincrónico para memoria compartida sobre una máquina multicore y demuestran el beneficio del asincronismo por lograr mejor rendimiento con HDA*.

Por otro lado, señalan la buena escalabilidad del algoritmo HDA* para distintas arquitecturas paralelas actuales tales como una máquina multicore, un *cluster* de multicore *HPC* con interconexión *Infiniband* y un *cluster* de multicore convencional con conexión 1Gb (x2) Ethernet.

Los experimentos realizados sobre el *cluster HPC* muestran que las comunicaciones no son un factor crítico para el rendimiento; sin embargo si lo es la contención generada por el acceso a memoria cuando se ubican varios procesos en una misma máquina multicore, es decir puede obtenerse un mejor rendimiento manteniendo ociosos algunos cores de la máquina. Para el problema *24-Puzzle* notan que el rendimiento se degrada con mayor rapidez, ya que al ser bajo tiempo de procesamiento de un estado[14], el *overhead* generado por el paralelismo tiene mayor impacto sobre los tiempos de ejecución. Asimismo, realizan una comparación de una versión de PLA* utilizando como técnica de balance de carga la *Estrategia de Comunicación Random* contra HDA* sobre esta arquitectura, siendo la primera más lenta que HDA* e incluso que el algoritmo A* secuencial.

Los experimentos realizados sobre el cluster convencional muestran un comportamiento similar al obtenido sobre el *cluster HPC*. Asimismo, realizan pruebas variando la cantidad de nodos enviados en cada mensaje y concluyen que al empaquetar pocos nodos por mensaje se reduce el *overhead* de búsqueda, y al empaquetar muchos nodos por mensaje se reduce el *overhead* de comunicación.

[14] Utilizan una heurística basada en base de datos de patrones disjuntos, pero excluyen de los tiempos de ejecución el acceso a la misma.

Por último, destacan que la buena escalabilidad del algoritmo permite resolver instancias complejas de problemas que requieren gran cantidad de memoria, dado que al utilizar un cluster se realiza un agregado de memoria importante a medida que se incrementa la cantidad de máquinas.

Para finalizar, el estudio llevado a cabo en (Burns, et al., 2009) (Burns, et al., 2010) realiza un análisis de rendimiento de algoritmos paralelos *Best-First Search* adaptados para ser ejecutados sobre máquinas multicore, utilizando *threads* y *locks* para mantener la consistencia de las estructuras[15], y presentan un algoritmo propio llamado *Parallel Best-NBlock First (PBNF)*.

En este marco, comparan el rendimiento obtenido por el algoritmo *PLA* GOHA* para memoria compartida presentado en (Kishimoto, et al., 2009), que hace uso de comunicación sincrónica, contra el alcanzado por una adaptación del mismo que utiliza comunicación asincrónica, siendo por lo tanto una variante de *HDA** para memoria compartida. La implementación de éste último se basa en lo siguiente:

- Cada thread posee una *cola de entrada*[16] globalmente conocida donde los demás threads del sistema depositarán nodos que éste debe procesar, la cual estará protegida por un *lock* para mantener su consistencia.
- Cada thread posee una *cola de salida*[7] local por cada thread par, las cuales no requieren locks ya que serán de uso propio y serán utilizadas para evitar bloqueos.
- Cuando un thread t_i genera un nodo que pertenece a otro thread t_j, debe comunicarlo depositándolo en algún momento en la cola de entrada de t_j. Para ello, intenta tomar el lock[17] de la cola de entrada de t_j, si lo obtiene en forma inmediata la transferencia del nodo se realiza copiando el puntero y luego suelta el lock permitiendo acceso posterior a dicha cola por parte de otro thread, en caso contrario ubica el puntero en la cola de salida local para t_j (operación que no requiere espera).
- Luego de que el thread t_i realiza un cierto número de expansiones de nodos de su Lista Abierta:
  - Para cada *cola de salida* local no vacía, intenta comunicar los nodos almacenados en ella a su thread dueño. Para esto, intenta tomar el lock de la cola de entrada del thread correspondiente, si lo obtiene inmediatamente transfiere todos los punteros a nodos quedando la cola de salida local vacía, y en caso contrario no se lo fuerza a esperar.
  - Intenta consumir los nodos que dejaron los demás threads en la cola de entrada propia para lo cual debe tomar el lock, pero sólo se lo fuerza a esperar si la Lista Abierta del thread está vacía (caso en el cual no tiene nodos para seguir trabajando).

La Figura 5.7 muestra el esquema de comunicación del algoritmo HDA* para memoria compartida, en el cual se observa que T_0 y T_3 generaron un nodo correspondiente a T_2, por lo que intentan ambos tomar el lock de la cola de entrada de T_2; en este caso T_3 obtiene el acceso inmediatamente por lo que realiza la transferencia del puntero, sin embargo T_0 falla en su

[15] Utilizan POSIX threads y locks provistos la biblioteca *Pthreads*.
[16] Implementada como arreglo dinámico conteniendo punteros a nodo.
[17] Utilizan la función pthread_mutex_trylock

intento de adquirir el lock, por lo que copia el puntero en su cola de salida local para T_2 y continua trabajando.

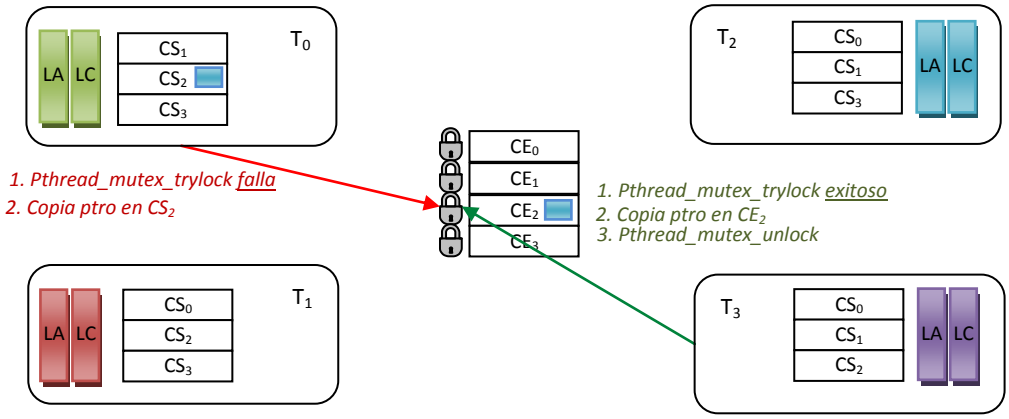


Figura 5.7. PLA* GOHA asincrónico (HDA*) para memoria compartida.

El trabajo experimental es realizado tomando como casos de estudio los problemas *Grid Pathfinding y 15-Puzzle,* en particular para este último toman en cuenta 250 instancias generadas aleatoriamente cuya resolución mediante A* secuencial no sobrepasa 3 millones de nodos procesados (es decir su complejidad es baja), y muestran el tiempo de ejecución en segundos al aumentar la cantidad de cores utilizados. Las pruebas confirman lo observado por (Kishimoto, et al., 2009) ya que HDA* para memoria compartida mejora a PLA* GOHA sincrónico para memoria compartida; en particular demuestran que el no bloqueo en el envío de nodos tiene mayor beneficio que el no bloqueo en la recepción.

Por otro lado, observan que cuando se utiliza una *función hash global* para asignar cada estado a un thread, los sucesores de un nodo son asignados potencialmente a threads remotos cuando podría asignarse la mayor parte de estos al propio thread que los generó o a un subconjunto de threads remotos disminuyendo la comunicación y contención en el acceso a las colas de entrada.

Para lograr dicho objetivo, dividen el espacio de estados original en *bloques disjuntos* generando así un *espacio de estados abstracto,* donde cada nodo será un *estado abstracto* que contiene varios estados del espacio de estados original. Este mapeo *estado-estado abstracto* se realiza a través de una *función de abstracción* que debe poder evaluarse en forma eficiente ya que se calculará durante ejecución: cuando se genera un nodo se calcula el valor de la *función de abstracción* y mediante el operador módulo se determina a que thread le corresponde su procesamiento. De esta manera, la *función de abstracción* se utiliza como técnica para balancear la carga.

El método utilizado para dividir el espacio de estados depende del dominio y generalmente se basa en ignorar ciertas variables de la representación del estado. Por ejemplo: para el problema del *15-Puzzle* la abstracción puede basarse en las posiciones de las fichas 1, 2 y 3 del tablero; de esta manera, se puede asignar a un thread el *estado abstracto* que posee todos los estados cuyas fichas 1, 2 y 3 están en las posiciones (1,1) (1,2) y (1,3) respectivamente, entonces cuando procese un estado y realice un movimiento generando un estado sucesor, si dichas fichas se mantienen inmóviles, el sucesor será asignado al mismo thread.

Dentro de este orden de ideas, introducen dos nuevos algoritmos para arquitecturas de memoria compartida: PLA* GOHA sincrónico con abstracciones y HDA* con abstracciones, llamadas por los autores APRA* y AHDA*. El trabajo experimental realizado sobre los casos de estudio antes mencionados demuestra que los algoritmos que utilizan abstracciones y asincronismo mejoran a las demás versiones para el problema *Grid Pathfinding,* no siendo tan pronunciado este beneficio para el *15-Puzzle.*

Con el objetivo de lograr que los threads trabajen en periodos libres de sincronización, implementan un algoritmo propio llamado *Parallel Best-NBlock First (PBNF)*, basándose en el algoritmo diseñado para ejecutar A* utilizando memoria externa *Structured Duplicate Detection* (*SDD*) (Zhou & Hansen, 2004) y en el algoritmo *Parallel Structured Duplicate Detection* (*PSDD*) (Zhou & Hansen, 2007)

El concepto *Structured Duplicate Detection* se desarrolla para poder realizar una búsqueda secuencial sobre un grafo utilizando memoria externa en forma eficiente, para lo cual la detección de duplicados es un factor crítico ya que cada nodo generado debe compararse contra nodos almacenados lo que implica realizar operaciones sobre disco.

En este contexto, para reducir la cantidad de nodos contra los que se debe comparar un sucesor generado proponen utilizar el concepto de *espacio de estados abstracto* para estructurar el grafo de búsqueda. Por ejemplo para problema del 8-Puzzle si se ignoran todas las fichas a excepción del hueco se pueden generar 9 estados abstractos (*bloques)*, a cada uno le pertenecen todos los estados con el hueco en la misma posición. De esta manera, cuando se expande un nodo generando sucesores, los estados representados por éstos difieren del estado padre únicamente en la posición del hueco, que varía en una fila ó una columna, por lo que al realizar la detección de duplicados no hace falta comparar contra aquellos estados cuyo hueco difiera en más de una fila o columna del estado padre. El conjunto de los *bloques* sobre los que se debe realizar la detección de duplicados se denomina *ámbito de detección de duplicados* (*Duplicate-Detection Scope* - *DDS*). Notar que en este caso la *función de abstracción* se utiliza con el propósito de limitar el *DSS*. La Figura 5.8 muestra gráficamente un estado del 8-Puzzle, el grafo abstracto obtenido al realizar la abstracción antes mencionada, y los *bloques* que se deben cargar en RAM al procesar nodos del bloque B_0.

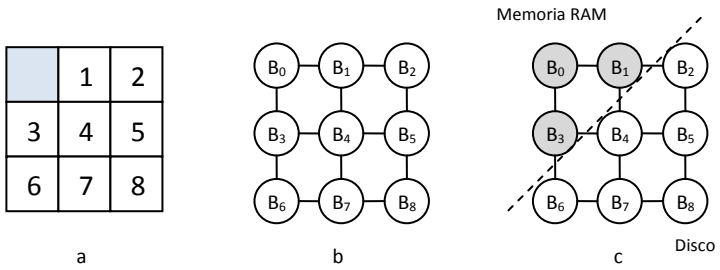


Figura 5.8. a) Estado del 8-Puzzle **b)** Grafo abstracto generado teniendo en cuenta la posición del hueco, cada nodo es un bloque **c)** Bloques en memoria interna *(DDS de B_0)* y externa (demás bloques)

La técnica *SDD* puede aplicarse a algoritmos de búsqueda que permitan expandir en cada etapa varios nodos pertenecientes al mismo *bloque.* Por ejemplo con *Breadth-First-Search* en

cada etapa se pueden expandir los nodos que estén en el nivel siendo procesado actualmente, mientras que con *Best-First Search* en cada etapa se pueden expandir todos los nodos que tengan costo igual al que se procesa actualmente (siempre y cuando se utilice una función de costo monótona). En general, se asocia a cada *bloque* las estructuras necesarias (Lista Abierta y/o Lista Cerrada) según el algoritmo de búsqueda base. El grafo abstracto se utiliza para seleccionar el próximo *bloque* a procesar, intentando minimizar la cantidad de operaciones sobre disco. Cuando se procesa un bloque, también se deben cargar en memoria RAM los bloques pertenecientes a su *DDS.* Cuando se expanden los nodos de la Lista Abierta de un bloque, los sucesores generados que no sean duplicados se colocan en la Lista Abierta del bloque correspondiente.

Por otro lado, *Parallel Structured Duplicate Detection* utiliza el concepto de *SDD* para reducir la sincronización en algoritmos paralelos de búsqueda sobre grafo, ya que los procesos pueden expandir en paralelo los *bloques* libres con *DDS* disjunto que no estén siendo utilizados por otro. De esta manera, el grafo abstracto se utiliza para encontrar *bloques libres* a expandir por lo que requiere protección a través de un lock para mantener su consistencia. La ventaja de *PSDD* es que los threads sólo sincronizan cuando tienen que tomar un nuevo *bloque* y no necesitan sincronizarse cuando están expandiendo nodos. La Figura 5.9 muestra un espacio de estados y la abstracción aplicada, generando un grafo de estados abstracto, junto con los bloques libres con DDS disjunto (marcados en color) que pueden trabajarse en paralelo.

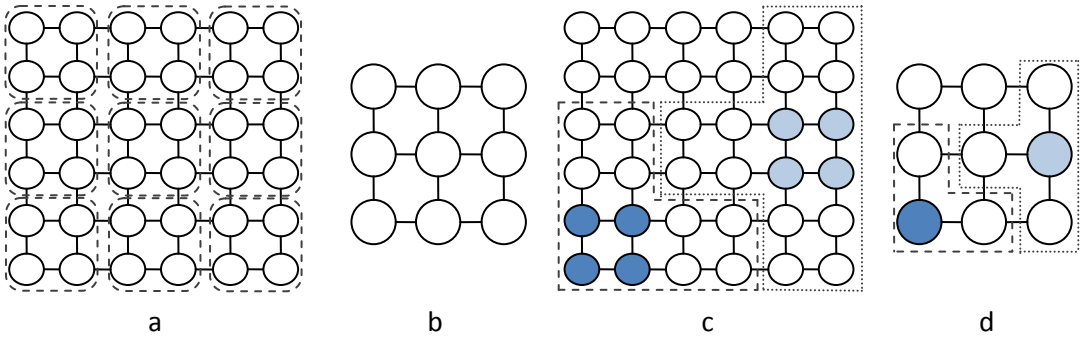


Figura 5.9. a) Espacio de estados original **b)** grafo abstracto generado **c)** Espacio de estados original con los nodos pertenecientes a dos bloques con DDS disjunto que pueden trabajarse en paralelo **d)** Grafo abstracto con los dos bloques cuyo DDS es disjunto.

Parallel Best-NBlock First (PBNF) utiliza el concepto *PSDD* para paralelizar el algoritmo *Best-First Search* sobre máquinas multicore. El algoritmo mantiene una heap compartida y protegida que representa el grafo abstracto, ordenada por el valor $\hat{f}$ del mejor nodo de cada *bloque*, la que será utilizada por los threads para seleccionar el próximo *bloque libre* a expandir.

Un thread toma un *bloque libre* de la heap y lo trabaja mientras contenga nodos cuyo valor $\hat{f}$ sea mejor al de aquel *bloque* que está en el tope de la heap. En ese momento intentará dejar su *bloque* y tomar otro libre, para lo cual debe obtener el *lock* usado para proteger la heap, pero en caso de no conseguirlo inmediatamente continuará trabajando sobre el *bloque* actual lo que provoca una *expansión especulativa* de nodos, es decir se procesan nodos que

potencialmente no serían expandidos en el algoritmo secuencial. Los threads no necesitan esperar a menos que no posean un *bloque* y no encuentren uno libre.

El orden de búsqueda se aproxima a *BFS*. Es posible que un *bloque* tenga pocos nodos mejores que los contenidos por aquel *bloque* libre en el tope de la heap. Con el objetivo de evitar la degradación del rendimiento causado por la sincronización excesiva por cambio de bloque, obligan al thread a procesar al menos *m* nodos del *bloque* que posee antes de cambiarlo (esto también introduce *expansión especulativa* de nodos). Por lo tanto, habrá una relación entre la calidad de los nodos a expandir y la contención en el acceso a la heap.

La primera solución encontrada puede ser subóptima (por ejemplo: la solución óptima puede estar en un bloque que no está libre) por lo que la búsqueda debe continuar hasta que todos los nodos abiertos tengan valor $\hat{f}$ superior al poseído por la mejor solución encontrada.

Los factores que tienen influencia sobre el rendimiento del algoritmo PBNF son: el número mínimo de expansiones a realizar sobre un bloque, ya que tiene relación con la cantidad de trabajo especulativo y con la contención en el acceso a la heap por cambio de bloque; la cantidad de bloques generados por la abstracción realizada, que está vinculado con el tiempo ocioso por espera de bloque libre junto con la sincronización por cambio de bloque y la cantidad de trabajo especulativo; y el numero de threads, que influye en el *overhead* por sincronización en *heap* y en la cantidad de trabajo especulativo realizado.

El rendimiento (*speedup*) obtenido por PBNF es más estable y en la mayoría de las pruebas mejora al obtenido por AHDA*. Sin embargo, al evaluar las causas notan que en realidad su algoritmo exhibe mayor tiempo de sincronización que AHDA*, por lo que las mejoras se deben a que PBNF utiliza una Lista Abierta por cada bloque en comparación a AHDA* que sólo utiliza una por thread, lo que causa que su tamaño sea más chico acelerando las operaciones. Además PBNF expande menor cantidad de nodos con costo mayor al que posee la solución óptima ya que los threads tienen una visión global de la guía heurística por tener la heap compartida que representa el grafo abstracto; en cambio en AHDA* un thread sólo expande los nodos que se le asignan y no tiene una visión global de los valores heurísticos.

5.4 Algoritmos para la Detección de Terminación de Aplicaciones con Cómputo Distribuido

Un aspecto imprescindible y no trivial cuando se trabaja con algoritmos de búsqueda en grafos siguiendo una estrategia distribuida y haciendo uso de un modelo de programación distribuida, donde no se tiene conocimiento global del estado del sistema ni los procesadores se rigen bajo un tiempo global, es detectar el momento en el cual los procesadores han terminado su tarea localmente (es decir están *ociosos*) y no existen mensajes con trabajo siendo comunicados entre estos, instante en el cual los resultados producidos por el sistema pueden ser utilizados. Para detectar dicho estado de terminación se utiliza un *algoritmo de detección de terminación* que involucra a todos los procesos y se ejecuta conjuntamente con el cómputo mismo, el cual no debe interferir ni demorar a este último, es decir un proceso puede enviar y/o recibir mensajes relacionados con el algoritmo de terminación sólo cuando está ocioso. (Kshemkalyani & Singhal, 2008) (Grama, et al., 2003) (Garg, 2004)

Antes de analizar los algoritmos propuestos para este fin, se debe detallar el modelo del sistema que realiza un cómputo distribuido. Luego se describen algunos de los algoritmos de terminación más simples y eficientes que son utilizados por los algoritmos de búsqueda paralela antes mencionados.

5.4.1 Modelo del sistema

El cómputo en el sistema es realizado por *N* procesos que se comunican mediante paso de mensajes y a través de *comunicación asincrónica*. Un mensaje es recibido luego de un tiempo finito de haber sido enviado. Los mensajes relacionados al algoritmo de terminación se conocen como *mensajes de control,* mientras que aquellos que transportan tareas a realizar por los procesadores a causa del balance de carga dinámico se conocen como *mensajes de trabajo.*

- En todo momento un proceso puede encontrarse en estado *activo*, es decir tiene trabajo localmente, o en estado *ocioso*, determinando que el proceso ha terminado su cómputo local por el momento.
- Un proceso en estado *activo* en algún momento se vuelve *ocioso* al detectar que no tiene más trabajo localmente.
- Un proceso en estado *ocioso* puede volver a estado *activo* luego de recibir un *mensaje de trabajo.*
- Sólo los procesos en estado *activo* pueden enviar *mensajes de trabajo.*
- Un proceso puede recibir *mensajes de trabajo* estando *activo* u *ocioso*.
- Las operaciones de *envío* y *recepción* se consideran *operaciones atómicas*.

El algoritmo de terminación debe obtener de alguna manera la cantidad de mensajes en tránsito, que son aquellos mensajes que fueron enviados y no fueron recibidos, entonces cuando dicha cantidad sea cero y todos los procesos estén ociosos se detectará terminación. La única forma de obtener ese valor es realizando una suma distribuida.

5.4.2 Algoritmo de terminación de Dijkstra

Uno de los primeros algoritmos planteados por *Dijkstra* para la detección de terminación asumía comunicación sincrónica (Dijkstra, et al., 1983). Luego, *Safra* realizó una variación del algoritmo anterior que elimina la condición de sincronismo, adaptándolo para el uso de comunicación asincrónica (Dijkstra, 1987) (Feijen & Van Gasteren, 1999).

El algoritmo propuesto por *Safra* permite que un proceso, por ejemplo aquel con identificador 0, detecte el estado de terminación; la detección involucra a todos los procesos que se consideran conectados en un anillo lógico y para ello se propaga un token (*mensaje de control*) por el anillo, que al arribar al proceso 0 será evaluado para saber si se alcanzó el estado de terminación.

Cada proceso mantiene un contador *c* propio, inicializado en cero al principio del algoritmo, el cual incrementará en una unidad al enviar un *mensaje de trabajo* y disminuirá en una unidad al recibir un *mensaje de trabajo*. La suma de todos los contadores de los procesos es el número

de mensajes en tránsito en la red (enviados y no recibidos) y se deberá obtener realizando una suma distribuida. Además cada proceso mantiene un *color*, cuyo valor puede ser *blanco* o *negro,* inicialmente su valor es *blanco*. El color de un proceso se torna *negro* cuando recibe un *mensaje de trabajo*.

Para obtener la suma distribuida de los contadores, cuando el proceso 0 se vuelve ocioso envía al siguiente proceso en el anillo el *token* que posee un campo *contador de mensajes* y un campo *color,* inicializados en *cero y blanco* respectivamente. Un proceso mantiene el *token* hasta que se vuelve ocioso, en ese momento incrementa el *contador de mensajes* del *token* en *c*, si su color es negro entonces asigna dicho color al *token,* y por último envía el token al sucesor en el anillo. Cuando un proceso envía el *token* su color vuelve a blanco.

Cuando el proceso 0 recibe el *token* nuevamente, lo examina estando ocioso y detecta terminación si:

- Su *color* es *blanco*.
- El color del *token* es *blanco*.
- La suma entre el contador de mensajes del token y *c* es cero.

Caso contrario, el proceso 0 inicia una nueva vuelta de detección de terminación.

Los *colores* tanto del token como de los procesos se utilizan para detectar inconsistencias, ya que si sólo se utilizaran contadores podría cometerse un error detectando falsa terminación como se especifica el siguiente caso: supongamos que no se han enviado *mensajes de trabajo* hasta el momento, por lo tanto el contador *c* de cada proceso está en cero, y que el contador del token es cero; además asumamos que el token ya pasó por el proceso j el cual está ocioso, pero todavía no llegó al proceso k quien se encuentra activo (k>j); el proceso k envía al proceso j un *mensaje de trabajo*, por lo que su contador c queda con valor 1; el proceso j recibe el mensaje quedando su contador c con valor -1; luego el proceso j envía un mensaje de trabajo al proceso k quien, al recibirlo, disminuye su contador en una unidad quedando en 0; cuando el token llega al proceso *k* el primer *mensaje de trabajo* enviado aparenta haberse recibido antes que el token abandonara el proceso j, lo cual no es cierto; si el token terminara su vuelta llegaría al proceso 0 detectándose terminación cuando hay procesos activos.

La Figura 5.10 muestra un ejemplo donde se detecta terminación erróneamente cuando sólo se utilizan contadores, el esquema muestra los procesos activos en aquellos en color blanco y los procesos pasivos en color gris.

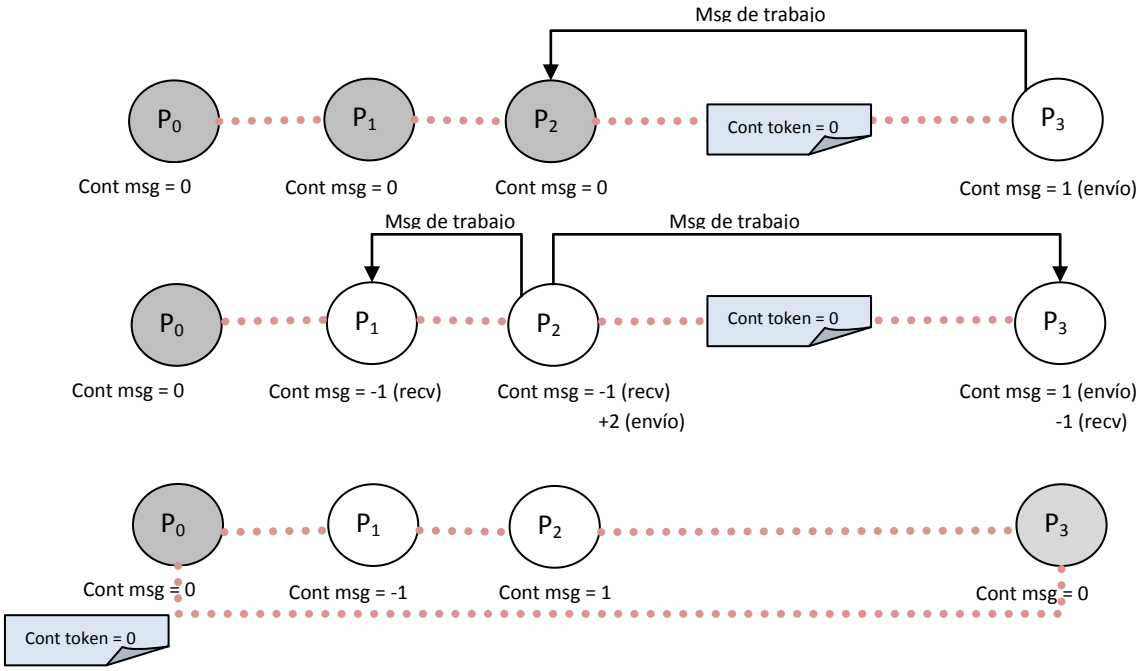


Figura 5.10. Inconsistencia durante la detección de terminación cuando sólo se utilizan contadores.

5.4.3 Algoritmo de terminación de Mattern de los 4 contadores

Un método simple propuesto en el trabajo (Mattern, 1987) consiste en que cada proceso cuente los mensajes enviados y los mensajes recibidos sobre variables independientes *s* y *r* respectivamente. Los procesos, que estarán ordenados en anillo, se pasan un *mensaje de control* sobre el cual se acumulará la suma de los contadores locales de mensajes enviados y mensajes recibidos de cada proceso, siendo *S** y *R** las variables totalizadoras. Si en un mismo instante de tiempo *t* todos los procesos pudiesen acumular los valores de sus variables locales sobre los contadores del mensaje de control, sería trivial detectar terminación ya que se cumpliría que S*=S(t) y R*=R(t), siendo S(t) y R(t) la suma de los contadores locales de mensajes enviados y recibidos de los procesos al instante t, y para la finalización se debería cumplir que R*= S*.

Sin embargo, en la realidad no es posible para los procesos acumular simultáneamente sus valores locales sobre el mensaje de control, porque éste pasa de un proceso a otro y cada uno acumulará sus contadores en momentos distintos, por lo que prueban que si los contadores de la primera vuelta no son inconsistentes se puede realizar la detección de terminación en dos pasadas. Para detectar inconsistencias cada proceso tendrá un *flag*, que se inicializará en la primera vuelta del mensaje de control y luego su valor será cambiado por el proceso en caso de enviar o recibir mensajes. Luego de la primera vuelta, el proceso 0 tendrá disponibles los acumulados S* y R*, y luego de la segunda vuelta S*' y R*', junto con la información correspondiente a los *flags*. Se detectará terminación si los cuatro contadores tienen igual valor y los *flags* de los procesos no cambiaron luego de la primera vuelta del mensaje de control. Este algoritmo es similar al propuesto por *Safra*.

En el mismo trabajo Mattern propone algoritmos alternativos que permiten detectar terminación en una única vuelta del mensaje de control, pero se incrementa la información contenida en cada *mensaje de trabajo* enviado, y por lo tanto su tamaño.

5.5 Importancia de la Biblioteca de Gestión de Memoria Dinámica

Un aspecto importante cuando se trabaja con aplicaciones con múltiples hilos[18] que realizan constantemente reserva y liberación de memoria es la correcta elección de la biblioteca de gestión de memoria dinámica, ya que si estas operaciones no escalan bien con un número creciente de hilos el rendimiento obtenido por la aplicación paralela será pobre.

La sección de memoria donde se almacenan las variables dinámicas es conocida como *heap o memoria dinámica*. Cualquier gestor de memoria mantiene estructuras que permiten conocer cuáles son los espacios libres y alocados, y persigue dos metas: minimizar el tiempo de las operaciones de manejo de memoria (*malloc*, *free*, *realloc,* entre otras) y minimizar el espacio de memoria dinámica utilizada.

Uno de los gestores de memoria dinámica ampliamente utilizados por las aplicaciones es *ptmalloc* (Gloger, 2014), que forma parte de la biblioteca Glibc, el cual permite a un proceso poseer varias *heaps*, cada *heap* se relaciona con un área (*arena)* donde se almacenan los datos realmente y debe ser protegida para mantener su consistencia. Cuando un thread realiza una reserva de memoria, se busca si existe alguna arena sin bloquear y de ser así se realiza la reserva de espacio, en caso contrario puede esperar o crear una nueva heap y luego se almacenan los datos en la *arena*. Cada thread mantiene información específica de la última arena que satisfizo una llamada a malloc realizado por él. Cuando un thread realiza un free, el espacio es retornado a la arena que pertenecía para lo cual debe adquirir el lock que la protege (Michael, 2004). En ocasiones la operación free retorna la memoria al Sistema Operativo, pero usualmente el espacio se mantiene en la lista de libres utilizada por el gestor y se podrá reusar más tarde tras un llamado a malloc. No tiene sentido liberar bloques al final del programa, ya que todo el espacio utilizado por este será devuelto al Sistema Operativo cuando se finalice la ejecución. (FSF Free Software Foundation, 2014)

Por otra parte, Jemalloc (Evans, 2006) reduce la contención utilizando múltiples arenas pero se basa en un mecanismo para asignar a cada thread un conjunto de arenas disminuyendo la probabilidad que una arena sea utilizada por múltiples threads. La primera vez que un thread realiza una operación sobre memoria dinámica es asignado a una arena la cual se elige siguiendo la estrategia *round-robin,* garantizando que todas las arenas tengan aproximadamente un mismo número de threads asignados. En aplicaciones con múltiples hilos se debe utilizar al menos 4 veces más arenas que la cantidad de procesadores, así se reduce la probabilidad que una arena esté siendo utilizada por múltiples threads. Los resultados experimentales muestran que Jemalloc escala bien para aplicaciones multihilo y exhibe rendimiento similar a glibc para programas con un único hilo.

[18] Valen las mismas aclaraciones para aplicaciones con múltiples procesos ejecutados sobre una misma máquina

Un enfoque diferente es el que realiza TCMalloc (*Thread-Cache Malloc)* (Ghemawat & Menage, 2014) que asigna a cada thread una cache local, desde la cual se satisfacen las reservas de objetos chicos[19], y mantiene una heap central compartida.

Cuando un thread aloca un objeto chico, el gestor reconoce la clase del objeto según su tamaño, busca en la cache local del thread y si había un objeto libre lo remueve y retorna (sin sincronización ni espera), caso contrario toma un conjunto de objetos de esta clase desde la heap central compartida y se ubican en la cache local retornando luego uno de los objetos para satisfacer la operación. En el caso que la heap central esté vacía, se aloca un conjunto de páginas desde *alocador de páginas central*, se dividen en fragmentos según la clase de objeto, se ubican en la heap central y luego se mueven algunos de estos objetos a la cache local del thread.

Por otro lado, las reservas de objetos grandes (>32KB) son manejadas por la heap central la cual consta de 256 entradas, cada entrada posee una lista de paquetes de páginas libres, donde el paquete consiste de i páginas de 4KB (siendo i el número de la entrada). Cuando se realiza la reserva, el gestor divide el tamaño del objeto en páginas de 4KB, lo cual resulta en el tamaño del paquete que se buscará, accede a la entrada correspondiente de la heap central removiendo un paquete de la lista si la misma no estaba vacía, y en caso contrario busca en la siguiente entrada de la heap central.

La operación de liberación es similar: si el objeto liberado es chico, devuelve el espacio a la cache local del thread y si la misma excede cierto límite se devuelven algunos objetos a la heap central; si el objeto liberado es grande, devuelve el espacio a la lista de paquetes libres correspondiente de la heap central.

Otro gestor de memoria dinámica conocido es *Lockless Memory Allocator (Llalloc)* (LocklessInc, 2014) el cual utiliza diferentes algoritmos para optimizar la velocidad de las operaciones y minimizar la fragmentación, siendo su rendimiento similar al alcanzado por *Jemalloc*. Las reservas de memoria de tamaño chico y mediano son llevadas a cabo por los alocadores *slab* y *btree* respectivamente, existiendo un par para cada thread, mientras que las reservas de objetos grandes las lleva a cabo el Sistema Operativo de forma exclusiva.

Un problema frecuente que causa la degradación de rendimiento aparece cuando existe una relación productor-consumidor entre distintos threads por operaciones alloc-free, lo cual abre la necesidad de sincronización para mantener la consistencia de las estructuras asignadas a cada thread. *Llalloc* evita dicha sincronización manteniendo una cola de liberación para cada thread: cuando el thread A libera un bloque alocado por el thread B, ubica el bloque en la cola de liberación del thread B y retorna rápidamente. Un thread vaciará su cola de bloques liberados eventualmente cuando su memoria se esté por agotar.

[19] Existen distintas clases de objetos chicos caracterizadas por tamaño de objeto

5.6 Discusión y conclusiones

Hasta el momento distintos autores han propuesto diversas técnicas para paralelizar los algoritmos de búsqueda *Best-First Search,* que difieren en cómo manipulan la Lista Abierta y la Lista Cerrada (centralizada o distribuida), y en la estrategia de balance de carga a utilizar entre los procesadores durante la ejecución. La técnica a seleccionar dependerá de la arquitectura y el problema a resolver. (Kumar, et al., 1988) (Grama, et al., 2003)

En una arquitectura de memoria compartida, la estrategia más simple es mantener una única Lista Abierta y Lista Cerrada compartidas por todos los threads, lo que implica que estos deban sincronizarse para mantener la coherencia de las mismas, provocando *overhead por sincronización* y *contención* que será limitante en el rendimiento. (Kumar, et al., 1988) (Grama, et al., 2003) Para reducir dichos *overheads* se pueden implementar estructuras de datos que permitan acceso concurrente por porciones (Cung & Le Cun, 1994), pero esta técnica no alcanza buen rendimiento al escalar la cantidad de threads /cores (Burns, et al., 2010)

Para solucionar el problema anterior, cada proceso puede tener su propia Lista Abierta y Lista Cerrada local. Esta estrategia se adapta bien a máquinas con memoria distribuida o con memoria compartida. Al principio sólo un proceso/thread tiene el nodo inicial y generará un conjunto de nodos que serán distribuidos por una *estrategia de balance de carga dinámica*. Cuando un proceso/thread tiene trabajo en su Lista Abierta, realiza una serie de iteraciones de A*, y en caso de encontrar una solución comunica su costo a los demás procesos/threads de modo de determinar globalmente si dicha solución es óptima ó si se debe continuar la búsqueda. Debido a la distribución dinámica del grafo, un nodo particular puede residir en la Lista Abierta y/o Lista Cerrada de cualquier proceso/thread, provocando que la detección de duplicados sea *parcial* y aumentando el *overhead de búsqueda* (cantidad de nodos procesados de más por el paralelo). (Kumar, et al., 1988) (Sanz, et al., 2011)

Otra estrategia de distribución de carga es la que propone PRA* (Evett, et al., 1995) y su sucesor HDA* (Kishimoto, et al., 2013): cada proceso tiene su propia Lista Abierta y Lista Cerrada y se utiliza una *función hash* para asignar cada estado a un único procesador. Apenas se genera un nodo, se establece quién es su dueño y se envía al mismo, quién será responsable de su procesamiento. Este mecanismo realiza simultáneamente un balance de carga dinámico, así como también permite una detección de duplicados *absoluta*. La diferencia entre los dos algoritmos es que el primero utiliza comunicación síncrona, y el segundo comunicación asíncrona. HDA* fue implementado puramente utilizando la librería MPI. Los autores realizan un análisis de escalabilidad sobre una máquina multicore, cluster de multicore convencional y un cluster HPC obteniendo buenos resultados.

Por otra parte, el trabajo publicado por (Burns, et al., 2010) presenta una estrategia de paralelización de HDA* sobre memoria compartida, utilizando la herramienta de programación *Pthreads*; si bien analizan su rendimiento al incrementar la cantidad de threads sobre una máquina multicore, no realizan una comparación de rendimiento entre ésta y una implementación que utilice MPI puramente, ni tampoco estudian la escalabilidad del algoritmo al incrementar la carga de trabajo.

Asimismo, añaden a estos algoritmos una estrategia para *abstraer* el espacio de estados de forma de asignar bloques de nodos a los threads en vez de nodos individuales, como ocurre si se calcula simplemente una función hash sobre el estado; de esta forma, se evita que todos los nodos generados tras una expansión migren a otros threads logrando una reducción de comunicación. Esta estrategia produce un beneficio significativo en el problema *Grid Pathfinding* y moderado en el problema 15-*Puzzle.*

También presentan un nuevo algoritmo *PBNF*, que se basa en PRA* e incorpora la estrategia de abstracción del espacio de estados, cuyo objetivo es permitir que los threads trabajen sobre nodos en periodos libres de sincronización. Si bien obtienen buen rendimiento, el algoritmo es complejo y no paraleliza A* puramente, razón por la cual obtienen en algunos casos un *speedup* muy superior al teóricamente alcanzable. Asimismo, este algoritmo presenta mayor *overhead de sincronización* que la versión de HDA* presentada en el mismo trabajo.

Por lo expuesto anteriormente, resulta de interés en el área:

- Analizar el rendimiento obtenido (speedup, eficiencia) por el algoritmo HDA* para memoria compartida cuando escala la cantidad de threads/cores y la carga de trabajo.
- Comparar la cantidad de memoria consumida por el algoritmo HDA* para memoria compartida y por HDA* para memoria distribuida y el rendimiento alcanzado, de forma de evaluar los beneficios que se obtendrían al adaptar HDA* en una aplicación hibrida.
- Sentar las bases para una versión de HDA* híbrida, utilizando herramientas de programación para memoria compartida y memoria distribuida, de modo de explotar toda la potencia brindada por la arquitectura *cluster de multicore* ya que se podrían eliminar ciertas ineficiencias de la versión MPI como son: las replicaciones de datos entre procesos que residen en la misma máquina, datos que podrían compartirse entre threads si se utilizase una versión híbrida; la serialización de datos (*nodos)* para la comunicación entre procesos que ejecutan sobre la misma máquina, innecesaria cuando se utilizan threads ya que podrían comunicarse los punteros a nodo directamente; y la reserva de múltiples buffers realizada por MPI para sustentar la comunicación entre los procesos que ejecutan sobre una misma máquina.

Cabe considerar que, si bien en el trabajo (Vidal, et al., 2011) se presenta una estrategia híbrida para encontrar soluciones a problemas de planificación, la paralelización se basa en el algoritmo WA* que busca soluciones subóptimas.

CAPÍTULO 6: IMPLEMENTACIONES

Este capítulo se concentra en presentar las implementaciones de los algoritmos propuestos con anterioridad para resolver el problema caso de estudio. Todas las implementaciones se realizaron en Lenguaje C, la herramienta utilizada para la comunicación sobre memoria distribuida es MPI (MPIForum, 2014), mientras que las herramientas utilizadas para la creación de threads y sincronización sobre memoria compartida son aquellas que provee la biblioteca Pthreads (Engelschall, 2006). Por último se utilizó la biblioteca de gestión de memoria dinámica Jemalloc (Evans, 2006).

La Sección 6.1 analiza la implementación del algoritmo secuencial A*, visto con anterioridad en el Capítulo 2. Luego la Sección 6.2.1 muestra la implementación del algoritmo paralelo PLA*GOHA asincrónico (HDA*) sobre memoria distribuida; la Sección 6.2.2 analiza la implementación de PLA*GOHA asincrónico (HDA*) sobre memoria compartida. Para finalizar, la Sección 6.3 plantea una discusión sobre los factores a analizar de cada algoritmo.

6.1 Algoritmo secuencial A*

El algoritmo secuencial A* implementado espera recibir como entrada las representaciones del estado inicial y estado final del problema a resolver y retorna como salida la secuencia de acciones que se deben aplicar para transformar el estado inicial en el estado final seleccionado.

Como se indicó en el Capítulo 2, cada nodo generado durante la ejecución almacena la siguiente información:

- Valor heurístico ($\hat{h}$) calculado sobre la representación del estado.
- Nivel ($\hat{g}$) donde fue encontrado el estado
- Estado que representa el nodo (dependiente del problema). Por ejemplo: para el caso de estudio será un tablero que represente la configuración junto con la ubicación del hueco.
- Operador utilizado para generar este estado a partir del estado padre.
- Puntero al nodo padre.

La estructura seleccionada para implementar la Lista Abierta es una *MinHeap* (Aho, et al., 1983) asociada a una *Tabla Hash Extensible* (Ramakrishnan & Gehrke, 1999), a la cual se hará referencia con el nombre *HeapNHash* de aquí en adelante. Por otra parte, se utilizó una *Tabla Hash Extensible* (Ramakrishnan & Gehrke, 1999)para implementar la Lista Cerrada.

Las claves asociadas a los elementos (*nodos*) almacenados en las *Tablas Hash* se obtienen a partir de calcular la *Función de Zobrist* (Zobrist, 1970) (Millington & Funge, 2009)sobre la representación del estado (Apéndice B). La *Tabla Zobrist* utilizada se carga desde archivo y será la misma para todas las ejecuciones e instancias del problema seleccionado como caso de

estudio. La clave se representará con un entero de 64 bits, dando lugar a que la función asigne la misma clave a dos estados distintos del espacio de búsqueda (esto ocurre ya que la cantidad de estados de posibles del problema puede ser mucho mayor a 2^{64}), siendo infrecuente este caso durante la ejecución del algoritmo de búsqueda.

La estructura *HeapNHash* utilizada para implementar la Lista Abierta permite ordenar los elementos (nodos) por prioridad (función de costo $\hat{f} = \hat{h} + \hat{g}$) de modo que la operación de extracción del nodo con menor costo, la inserción de un nodo, y la modificación del costo de un nodo existente son de orden logarítmico. Además permite realizar búsquedas en orden cuasi constante por clave (*Función de Zobrist* calculada sobre la representación del estado) lo cual optimiza la detección de duplicados. Por otra parte, se incorporó un mecanismo de desempate entre elementos con igual prioridad de forma que los nodos que representen la solución (aquellos que tienen $\hat{h} = 0$) tienen preferencia para ser seleccionados.

La *Tabla Hash Extensible* utilizada para implementar la Lista Cerrada permite realizar búsquedas por clave, inserciones y eliminaciones de elementos en orden constante o cuasi constante.

La operación sobre estas estructuras se realiza de la siguiente manera: cuando se generan los sucesores de un nodo, para cada uno se calcula la clave asociada al estado que representa utilizando la *Función de Zobrist* y con dicho valor se accede al *bucket* correspondiente de la *Tabla Hash* que representa la Lista Cerrada y de la *Tabla Hash* asociada a la Lista Abierta, de forma de verificar la existencia de un nodo cerrado o abierto que esté representando al mismo estado que el nodo sucesor generado recientemente (*detección de duplicados*). Un *bucket* de la *Tabla Hash* puede almacenar varios nodos, siendo dicha cantidad configurable. Al acceder al bucket, se debe realizar una comparación de la clave buscada contra aquellas pertenecientes a los nodos almacenados en el *bucket* y en caso de haber coincidencia se debe proceder a comparar el estado en sí (ya que la *Función de Zobrist* puede asignar a dos estados distintos la misma clave).

La clase de funciones heurísticas utilizada es aquella que calcula el valor heurístico directamente tomando como entrada la representación del estado, ya que no es el propósito de esta tesis ahondar en la optimización de la búsqueda por medio de afinar la calidad del valor heurístico de los nodos a través de bases de datos de patrones disjuntos. La función heurística será dependiente del problema a resolver y puede seleccionarse previamente a la etapa de compilación; esto permite experimentar con distintas heurísticas y analizar el rendimiento obtenido.

6.2 Algoritmos Paralelos

En esta sección se estudia la implementación del algoritmo HDA* sobre arquitecturas de memoria distribuida y sobre arquitecturas de memoria compartida.

6.2.1 HDA* para memoria distribuida

A partir de la escueta información provista por el trabajo (Kishimoto, et al., 2013) acerca de la implementación de HDA*, se realizó una versión propia que se explicará en esta sección. Para ello se utilizó la biblioteca de paso de mensajes MPI, en particular la implementación OpenMPI, mensajes asincrónicos (MPI_Isend[20]), comprobaciones por llegada de mensajes no bloqueantes (MPI_IProbe) cuando el proceso no esté ocioso y en caso contrario comprobaciones bloqueantes (MPI_Probe).

El algoritmo de detección de terminación utilizado es aquel propuesto por *Safra* (Dijkstra, 1987), a diferencia de (Kishimoto, et al., 2013) que utiliza el *algoritmo de tiempo de Mattern* (Mattern, 1987), ya que se prefirió no incrementar el tamaño de cada *mensaje de trabajo* enviado con información adicional impuesta por el algoritmo de terminación.

Al igual que (Kishimoto, et al., 2013), la asignación de nodos entre los procesos se realizó a haciendo uso de la *Función de Zobrist*[21] y se utilizó la técnica propuesta por (Romein, et al., 2002) para empaquetar una cantidad de nodos determinada antes de realizar el envío del *mensaje de trabajo* a un proceso destino (la cantidad se puede configurar antes de realizar la compilación).

6.2.1.1 Síntesis

Cada proceso realizará una búsqueda A* en el ámbito local, manteniendo su Lista Abierta y su Lista Cerrada locales, comunicándose con sus pares por alguno de los siguientes motivos: envío / recepción de *mensajes de trabajo* conteniendo nodos del grafo, envío / recepción de costo de mejor solución encontrada al momento, envío / recepción de *token* de terminación, envío / recepción de mensaje de finalización de cómputo.

Al finalizar el cómputo se procederá a recuperar la solución en forma distribuida obteniendo la secuencia de operadores cuya aplicación permite transformar el estado inicial en el estado final. Dado que cada nodo del grafo se asigna a un proceso particular, se incorporó en la representación del nodo el identificador del proceso que lo generó ya que es quién almacena el nodo padre en su memoria, permitiendo así rastrear en qué espacio de direcciones reside cada nodo perteneciente al camino de la solución encontrada.

---

[20] Se dio preferencia a la función MPI_Isend para el envío de mensajes por sobre MPI_Bsend, utilizado por (Kishimoto, et al., 2013), ya que el *modo buffereado* de MPI requiere que el programador reserve al comienzo un espacio de memoria, que será utilizado por MPI_Bsend, y en caso de realizar un envío cuando el espacio libre es insuficiente se producirá un error que puede causar que el programa termine anormalmente. Para evitar el error se debe incorporar sincronización adicional en el programa y manejar el espacio libre de modo que, cuando se sepa que no hay espacio, se espere a que todas las comunicaciones pendientes en el buffer finalicen (MPI_buffer_dettach). No sería seguro ni es posible esperar sólo a que una operación *buffereada* finalice ya que MPI utiliza una política circular cuando busca espacio contiguo en el buffer (MPIForum, 2014). Además a causa del no determinismo de la aplicación, el espacio a reservar inicialmente es desconocido.

[21] Cabe destacar que cada proceso tendrá su *Tabla Zobrist,* pero los valores que almacenan son idénticos para todos los procesos e incluso son los mismos que aquellos utilizados en el algoritmo secuencial A* ya que se cargan desde archivo.

6.2.1.2 Variables

Cada proceso posee localmente su Lista Abierta y Lista Cerrada[22] que al principio estarán vacías, el costo de la mejor solución conocida al momento (*costo_mejor_solucion*), la mejor solución encontrada por el proceso (*mejor_solucion)*, los datos necesarios para el algoritmo de detección de terminación (*estado* que representa un color, inicializado en blanco, y *contador* que representa la cantidad de mensajes enviados y recibidos, inicializado en cero), y una variable que cambia de valor cuando se alcanza el fin del cómputo (*fin).*

A fin de empaquetar varios *nodos* en un mensaje antes de enviarlos a su dueño, se equipa a los procesos con un buffer para cada proceso par (*buffer_envio_trabajo[])*, indexado por el identificador del proceso destino; cada buffer contendrá registros *nodo*, su dimensión física es configurable previamente a la compilación y será conocida como *LNPT (Limite de Nodos Por Transferencia)*, y se debe mantener actualizada su dimensión lógica para conocer la cantidad de nodos acumulados para cada proceso par determinado.

Asimismo, con el propósito de implementar comunicación asincrónica, el proceso tendrá una serie de buffers adicionales que contienen registros *nodo* (*buffer_adicional[])* utilizados para el envío de mensajes, ya que la operación MPI_Isend requiere que la zona de memoria correspondiente al mensaje siendo enviado no se modifique hasta tener certeza de que ha sido recibido. La cantidad de buffers adicionales es configurable. Cuando el proceso debe enviar un mensaje controla si hay un buffer adicional libre, caso en el cual realiza el envío y toma el buffer adicional libre como nuevo buffer para acumular los próximos nodos a enviar para el destino y en caso negativo esperará a que termine alguna de las comunicaciones pendientes[23].

6.2.1.3 Procesamiento

El código de los procesos es idéntico, sólo al proceso 0 se le encargará la tarea adicional de: cargar la configuración inicial generando el nodo inicial, el cual lo insertará en su Lista Abierta si le pertenece[24] o enviará a su proceso dueño[25], y realizar la detección de terminación enviando el respectivo aviso de finalización a los demás procesos.

Cada proceso realizará una serie de iteraciones hasta tener conocimiento de que se alcanzó el fin del cómputo. En cada iteración ejecuta las siguientes etapas:

- *Etapa de recepción de mensajes de trabajo:* el proceso verifica en forma no bloqueante (MPI_Iprobe) si han arribado *mensajes de trabajo* con nodos. En caso afirmativo, recibe

[22] La implementación de la Lista Abierta y de la Lista Cerrada utiliza las mismas estructuras de datos que en la implementación secuencial del algoritmo A*.
[23] Esta es otra razón por la que se eligió MPI_Isend en vez de MPI_Bsend. Con la primera se puede controlar el caso de la ausencia buffers ya que los maneja el programador, de esta manera si no hay uno libre se espera a que sólo una comunicación termine, a diferencia de la técnica de MPI_Bsend donde la única opción para evitar fallas durante ejecución es esperar a que todas las comunicaciones pendientes finalicen.
[24] Esto se determina a partir del valor retornado por la *Función de Zobrist* aplicada a la representación del estado.
[25] Esta es la única vez que se envía un mensaje de trabajo con un único nodo.

cada mensaje y para cada registro *nodo* contenido cuyo costo mejore al valor *costo_mejor_solucion* realiza las siguientes acciones: reserva en memoria dinámica un espacio; le asigna los valores recibidos; realiza la detección de duplicados insertándolo en la Lista Abierta según sea adecuado (en caso de existencia de un duplicado en la Lista Abierta/Lista Cerrada con costo superior al que posee el nodo recibido, se realiza una promoción/reapertura respectivamente).

- *Etapa de recepción de mensajes de costo de solución parcial encontrada:* el proceso comprueba en forma no bloqueante (MPI_Iprobe) si han llegado mensajes conteniendo costos de mejores soluciones encontradas por otros procesos. En caso afirmativo, recibe los mensajes actualizando si es adecuado la variable local *costo_mejor_solucion.* Tener en cuenta que si el nodo abierto de menor costo tiene valor $\hat{f}$ mayor o igual a *costo_mejor_solucion*, se debe proceder a vaciar la Lista Abierta ya que los nodos abiertos conducirían a soluciones con costo subóptimo.
- *Etapa de procesamiento:* en esta etapa el proceso expande a lo sumo *LNPI* nodos *(Límite de Nodos por Iteración)* de su Lista Abierta[26]. Para cada nodo extraído verifica si su costo supera o iguala a *costo_mejor_solucion*, en cuyo caso vacía la Lista Abierta. En caso contrario, comprueba si el nodo representa la solución. En caso de serlo, vacía la Lista Abierta y guarda una copia del puntero en *mejor_solucion,* actualizando también la variable *costo_mejor_solucion*. En caso contrario, inserta el nodo en la Lista Cerrada, genera sus sucesores, para cada uno calcula el valor de la *Función de Zobrist* y determina a qué proceso pertenece. Cuando el nodo le corresponde al mismo proceso, realiza la detección de duplicados y lo inserta en la Lista Abierta según sea adecuado; en caso contrario, ubica el registro nodo en el *buffer_envio_trabajo* asignado para el proceso destino y si se completó dicho buffer, es decir posee *LNPT* nodos, envía el mensaje en forma asincrónica (MPI_Isend) siguiendo los pasos que se explicaron en la Sección 6.2.1.2.
- *Etapa proceso ocioso:* cuando el proceso no tiene nodos en la Lista Abierta, si había encontrado una nueva solución en la etapa de trabajo previa, envía el costo de la misma a los demás procesos. Luego se envían *mensajes de trabajo* para aquellos procesos destino cuyo *buffer_envio_trabajo* contenga nodos que no fueron comunicados. Por último, se queda esperando (MPI_Probe) cualquier tipo de mensaje: (1) mensaje de trabajo, (2) mensaje comunicando una mejor solución encontrada, (3) mensaje con *token* de terminación, (4) mensaje comunicando el fin del cómputo. El proceso termina esta etapa cuando posee nodos en su Lista Abierta resultado de haber recibido un *mensaje de trabajo* o cuando se le ha comunicado el fin del cómputo. El procesamiento de los mensajes tipo (1) y (2) es similar a lo comentado anteriormente; al recibir un mensaje tipo (3) se procede a la actuación según el algoritmo de terminación (sabiendo que el proceso está ocioso y debe pasar el token al siguiente proceso o evaluar la condición de terminación en caso de ser el proceso 0); ante el arribo de un mensaje tipo (4) se procede cambiar el valor de la variable *fin*, con lo cual el proceso terminará el procesamiento.

La cantidad de *nodos* recibidos en un *mensaje de trabajo* puede variar, pero no superará un valor máximo determinado *LNPT* (*Limite de Nodos por Transferencia*); para obtener la cantidad de elementos recibidos en un mensaje se utiliza MPI_Get_count.

[26] Esta es otra diferencia respecto a la versión implementada por Kishimoto, en la cual por cada iteración procesa un único nodo.

En cada envío / recepción de un *mensaje de trabajo* se debe proceder según el algoritmo de terminación de *Safra*. Dentro del token se añaden tres campos adicionales que permitirán realizar la recuperación de la solución: costo de la mejor solución global encontrada al momento, el identificador del proceso que encontró dicha solución, y dirección en memoria de dicho nodo solución (zona de memoria accesible únicamente por el proceso que la encontró). Antes de pasar el token, el proceso debe actualizar dichos campos en caso de poseer la mejor solución del momento.

La Figura 6.1 muestra el esquema del algoritmo PLA* GOHA asincrónico (HDA*) para memoria distribuida, en el cual se puede observar: el orden en que los procesos se pasan el token de terminación, P_0 comunicando un costo de mejor solución encontrada, P_2 enviando un mensaje de trabajo tras completar el buffer de envío de trabajo para el proceso P_3.

La Figura 6.2 muestra el algoritmo PLA* GOHA asincrónico para memoria distribuida implementado.

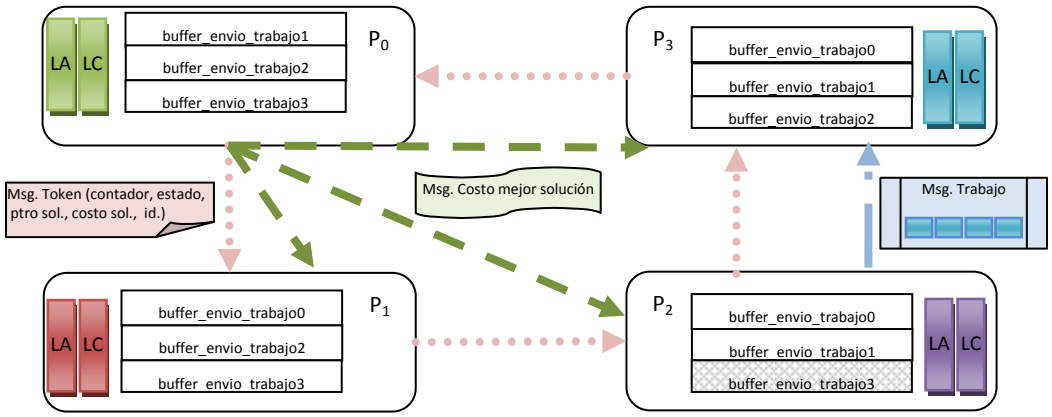


Figura 6.1 Esquema del algoritmo PLA* GOHA asincrónico (HDA*) implementado con MPI

```
Algoritmo PLA*GOHA-asincrónico-MPI
Proceso i::
Entrada: cantidad de procesos (N)

1) Inicializar LA y LC (Lista Abierta y Lista Cerrada)
2) Inicializar estado en blanco, contador en 0, fin en falso {alg.
terminación}
3) Inicializar costo_mejor_solucion en valor alto y mejor_solucion en NULL
4) Inicializar buffer_envio_trabajo[] con buffers vacíos
5) Inicializar buffer_adicional[] con buffers vacíos
6) si i==0
7)   Cargar el estado inicial y crear el nodo
8)   Inicializar token nulo y enviarlo a sí mismo en forma asíncrona
9)   destino = Zobrist (nodo) % N
10)  si destino == i
11)    Insertar nodo en LA
12)  sino
13)    Insertar registro nodo en buffer_envio_trabajo[destino]
14)    Destruir "nodo"
15)    Send_async buffer_envio_trabajo[destino]
16)    Buscar en buffer_adicional[] un buffer libre para reemplazar
       buffer_envio_trabajo[destino]
17)    Actualizar contador y estado por envío de mensaje según el algoritmo de
       terminación
```

```
18)mientras (not fin)
     {Etapa: recepción de mensajes con trabajo}
19)  Probe_async de llegada de msgs con TAG "trabajo"
20)  mientras (se detectó llegada de msg de trabajo)
21)    Recv msg de trabajo
22)    Actualizar contador y estado por recepción de mensaje según el
       algoritmo de terminación
23)    Crear un nodo para todo aquel recibido cuyo costo sea menor a
       costo_mejor_solucion, copiar los datos del nodo desde el buffer
       recibido, realizar la detección de duplicados e insertar en LA en caso
       adecuado.
24)    Probe_async de llegada de msgs con TAG "trabajo"
     {Etapa: recepción de mensajes con costo de mejor solución}
25)  Probe_async de llegada de msgs con TAG "solución"
26)  mientras (se detectó llegada de msg con costo de mejor solución)
27)    Recv msg con costo de mejor solución
28)    si el costo recibido mejora a costo_mejor_solucion local
29)      actualizar costo_mejor_solucion con el costo recibido
30)      si el mejor nodo de LA tiene costo >= costo_mejor_solucion
31)        Vaciar LA
32)    Probe_async de llegada de msgs con TAG "solución"
     {Etapa: procesamiento de nodos}
33)  mientras no esté vacía LA y no procesé LNPI nodos
34)    nodo = Eliminar nodo con costo mínimo de LA
35)    si f(nodo) >= costo_mejor_solucion
36)      Vaciar LA
37)    sino
38)      si h(nodo) == 0
39)        costo_mejor_solucion = g(nodo)
40)        mejor_solucion = nodo
41)        Vaciar LA
42)      sino
43)        Insertar nodo en LC
44)        Expandir nodo generando los sucesores
45)        Para cada sucesor "s" de nodo
46)          si costo(s) >= costo_mejor_solucion
47)            Destruir "s"
48)          sino
49)            Marcar en "s" que éste proceso lo generó
50)            destino = Zobrist (s) % N
51)            si destino == i
52)              Realizar la detección de duplicados e insertar en LA
                 en caso adecuado.
53)            sino
54)              Poner "s" en buffer_envio_trabajo[destino]
55)              Destruir "s"
56)              si buffer_envio_trabajo[destino] tiene LNPT nodos
57)                Send_async buffer_envio_trabajo[destino]
58)                Buscar en buffer_adicional[] un buffer libre para
                   reemplazar buffer_envio_trabajo[destino]
59)                Actualizar contador y estado por envío de mensaje según el
                   algoritmo de terminación
     {Etapa: espera ocioso}
60)  si LA está vacía
61)    si en la etapa de procesamiento encontré una mejor solución
62)      Send_async de costo_mejor_solucion a todos los procesos
63)    Para cada buffer_envio_trabajo correspondiente al proceso "destino"
64)      si buffer_envio_trabajo[destino] no está vacío
65)        Send_async buffer_envio_trabajo[destino]
66)        Buscar en buffer_adicional[] un buffer libre para reemplazar
           buffer_envio_trabajo[destino]
67)        Actualizar contador y estado por envío de mensaje según el
           algoritmo de terminación
68)    mientras  (not fin) y LA está vacía
69)      Probe_sync de llegada de cualquier tipo de mensaje
70)      si llego msg de trabajo
71)        Recv msg de trabajo
72)        Actualizar contador y estado por recepción de mensaje según el
```

```
            algoritmo de terminación
73)         Crear un nodo para todo aquel recibido cuyo costo sea menor a
            costo_mejor_solucion, copiar los datos del nodo desde el buffer
            recibido, realizar la detección de duplicados e insertar en LA
            en caso adecuado.
74)       sino si llego msg token
75)         si costo_mejor_solucion del token < costo_mejor_solucion local
76)           Actualizar costo_mejor_solucion local con el costo_mejor_solucion
              contenido en el token
77)         sino
78)           Actualizar los campos del token mejor_solucion,
              costo_mejor_solucion y proc_mejor_solucion si este proceso había
              encontrado la mejor solución
79)         si i==0
              si están dadas las condiciones para detectar terminación
80)             fin = true
81)             Send_async de msg de fin a todos los procesos
82)           sino
83)             Inicializar el estado del token en blanco y su contador en 0
84)             Send_async de msg de token al proceso 1
85)         sino
86)           Actualizar los campos contador y estado del token
87)           Send_async de msg de token al proceso siguiente (i+1)%N
88)         estado = blanco
89)       sino si llego msg con costo de mejor solución
90)         Actualizar costo_mejor_solucion local en caso adecuado
91)       sino {llego msg de fin de procesamiento}
92)         fin=true
```

Figura 6.2 Algoritmo PLA* GOHA asincrónico (HDA*) implementado con MPI

6.2.1.4 *Ineficiencias para arquitecturas de memoria compartida*

Los programas que utilizan la biblioteca MPI para la comunicación entre procesos pueden ser ejecutados sobre máquinas con memoria distribuida, multiprocesadores de memoria compartida, o una combinación de ambos. Sin embargo, existen una serie de ineficiencias cuando se ejecuta un programa MPI sobre un multiprocesador de memoria compartida, en particular se hará referencia a aquellas relacionadas con el algoritmo estudiado en la sección anterior:

- *Serialización de datos para envío:* cuando se realiza una operación de envío de mensaje conteniendo múltiples elementos, éstos deben residir en posiciones contiguas de la memoria (en general a esta zona se le conoce como *buffer*). El algoritmo planteado trabaja con punteros a registros nodo (ya que los nodos deben residir en la memoria dinámica) los cuales no se asegura que estén en memoria contigua, por lo que el envío de un mensaje de trabajo conteniendo nodos del grafo implica serializar de alguna manera los datos. Cuando un proceso detecta que un *nodo* generado por él, para el cual se reservó espacio en memoria dinámica (*malloc*), pertenece a otro proceso debe copiar el dato (registro nodo) en el *buffer_envio_trabajo* para el proceso destino, quedando disponible el espacio dinámico que el nodo ocupaba (*free*). Cuando el *mensaje de trabajo* sea recibido por el proceso destino, si el nodo tiene costo menor a su variable local *costo_mejor_solucion,* debe alocar espacio en su memoria dinámica (*malloc)* y realizar una copia de los datos del nodo desde el buffer del mensaje.
- *Estructuras replicadas en cada proceso:* el paradigma de paso de mensajes implica que cada proceso accede únicamente a su espacio de direcciones exclusivo; por lo anterior, si

todos los procesos necesitan datos de sólo lectura para el cálculo, deberán replicarse los mismos en la memoria de cada proceso lo que implica un incremento en la memoria utilizada. En particular el algoritmo propuesto maneja datos de sólo lectura, que corresponden a la *Tabla Zobrist* y a una estructura utilizada para conocer las ubicaciones finales de las fichas en el tablero seleccionado como configuración final la cual posibilita el cálculo del valor heurístico correspondiente a la Suma de las Distancias de Manhattan de las fichas, debiendo existir copias en la memoria de cada proceso.

- *Memoria reservada por MPI:* MPI realiza una reserva importante de memoria al comienzo de la aplicación para gestionar comunicaciones la cual se incrementa con el número de procesos en el sistema.

Los threads que se ejecutan sobre un multiprocesador de memoria compartida podrían beneficiarse de la reducción de memoria utilizada al compartir estructuras de datos de sólo lectura y eliminar la reserva de memoria de la biblioteca MPI. Además éstos podrían evitar la serialización de datos durante la comunicación, ya que se abriría la posibilidad de intercambiar punteros a nodo – dado que todos los threads tienen acceso al mismo espacio de direcciones.

6.2.2 HDA* para memoria compartida

Se implementó una versión propia del algoritmo PLA* GOHA asincrónico (HDA*) para memoria compartida similar al propuesto por (Burns, et al., 2010) utilizando POSIX threads y mecanismos de sincronización provistas en la biblioteca *Pthread*.

La distribución de nodos entre los threads se realizó a través de la *Función de Zobrist.* Los threads depositarán nodos generados cuyo procesamiento no le corresponde en la cola de entrada del thread adecuado, habiendo una para cada thread, siendo globales y accesibles por todos ellos razón por la cual deberán ser protegidas.

Para evitar encargar a un thread la tarea de detectar terminación verificando los estados de sus pares y el estado de las colas de entrada, se realizó una adaptación del algoritmo de terminación de *Safra* permitiendo que todos los threads colaboren en dicha tarea. Para ello cada thread mantiene un estado (*color*) y un contador el cual lleva la cantidad de nodos enviados y recibidos – en vez de la cantidad de "comunicaciones" realizadas[27]; el token de terminación se representará con una variable compartida, llevando un contador, un estado (color) y el identificador del thread que posee el token al momento.

6.2.2.1 Síntesis

Cada *thread* realizará una búsqueda A* en el ámbito local, manteniendo su Lista Abierta y Cerrada locales, comunicándose con los demás *threads* cuando deba enviar nodos que fueron generados por él y cuyo procesamiento no le corresponde. La comunicación de estos nodos pertenecientes a un *thread* particular se realiza a partir del depósito de los mismos sobre una cola de entrada poseída por éste globalmente visible y protegida por un lock. El traslado del

[27] Las colas de entrada no contabilizan la cantidad de veces que se depositó sobre ellas, sino la cantidad total de nodos que almacenan (dimensión lógica), por ello se calcula la cantidad de nodos en tránsito en vez de la cantidad de "mensajes" o "depósitos" que no fueron recibidos aún.

nodo significa realizar una copia de puntero. En ningún momento se obliga al thread a esperar, por esto cuando no pueda efectivizar la comunicación inmediatamente depositará temporalmente el nodo en un espacio propio y local de almacenamiento para dicho thread destino conocido como cola de salida – existiendo una cola de salida por cada thread par. El esquema de comunicación de nodos del algoritmo se muestra en la Figura 6.3.

Al finalizar el cómputo, se procederá a recuperar la solución. Dado que todos los threads comparten el espacio de direcciones, un único thread puede acceder a la secuencia de operadores cuya aplicación permite transformar el estado inicial en el estado final a partir del puntero al nodo solución.

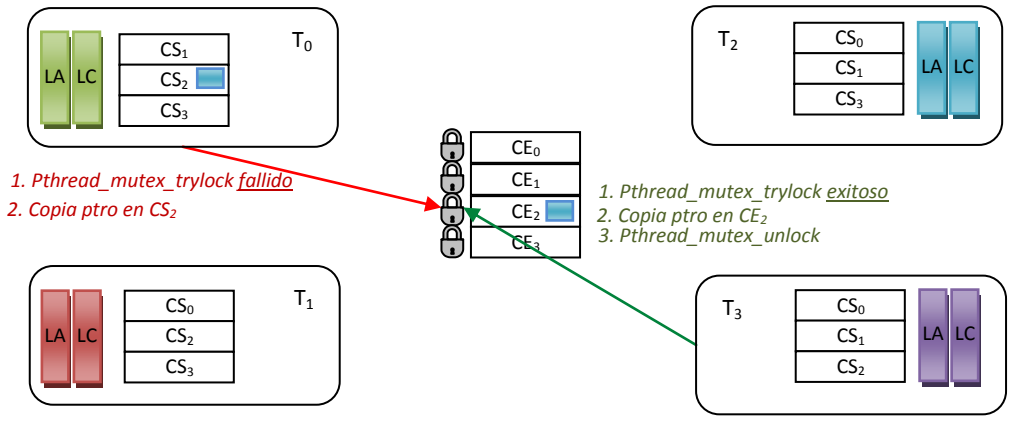


Figura 6.3. Esquema de comunicación de nodos para el algoritmo PLA* GOHA asincrónico (HDA*) para memoria compartida

6.2.2.2 *Variables compartidas por los threads.*

Los threads comparten los siguientes datos:

- Una variable *fin* cuyo valor cambia al detectar terminación.
- Un arreglo *cola_entrada[]*, habiendo una cola de entrada para cada thread, protegidas para mantener su consistencia. La cola fue implementada como un arreglo dinámico conteniendo punteros a registro nodo. Para cada cola se debe llevar la dimensión lógica.
- El *token* utilizado por el algoritmo de terminación que posee: *contador* de nodos residentes en todas las colas de entrada (nodos en tránsito que fueron comunicados por los threads que los generaron pero no fueron "recibidos" por su dueño), *estado* y un *identificador* del thread que posee el token al momento. Estos datos no son protegidos ya que sólo un thread podrá modificar el token en un momento dado.
- Un puntero a la mejor solución encontrada al momento por todos los threads (*mejor_solucion*) y su costo (*costo_mejor_solucion*), ambas deben ser protegidas ya que dos threads pueden encontrar dos soluciones distintas y querer actualizar estos valores al mismo tiempo.

Las variables compartidas son inicializadas por el thread 0 antes de la creación de threads.

6.2.2.3 Variables Locales al thread

- Un arreglo *cola_salida*[], habiendo una cola de salida para cada thread par.
- Lista Abierta y Lista Cerrada.
- *Contador* y *Estado* correspondientes al algoritmo de terminación de *Safra*.

6.2.2.4 Procesamiento

El código a ejecutar por todos los threads es idéntico, sólo al thread 0 se le encargan las tareas adicionales de: generar el nodo inicial ubicándolo en la cola de entrada de su thread dueño, inicializar las estructuras comunes, detectar el estado de terminación y recuperar desde la memoria compartida la secuencia de pasos que representa la solución al problema una vez finalizado el cómputo.

Cada thread realizará una serie de iteraciones hasta que detecte que se ha alcanzado el fin del cómputo (a través de un cambio en el valor de la variable *fin)*. En cada iteración realiza las siguientes etapas:

- *Etapa de consumo de nodos de la cola de entrada:* comprueba si la cola de entrada propia no está vacía, en ese caso intenta tomar el lock asociado a la misma; si obtiene el acceso en forma inmediata toma todos los punteros a nodo depositados en ella vaciando la cola de entrada y suelta el lock, luego para cada nodo cuyo costo es menor a *costo_mejor_solucion* realiza la detección de duplicados insertándolo en la Lista Abierta en caso que sea adecuado.
- *Etapa de procesamiento:* el thread procesa a lo sumo *LNPI* nodos *(Límite de Nodos por Iteración)* de su Lista Abierta. Cuando extrae un nodo verifica si su costo es al menos *costo_mejor_solucion,* en cuyo caso vacía la Lista Abierta ya que los nodos en ella conducirían a soluciones subóptimas. En caso contrario, comprueba si el nodo representa la solución y en tal situación vacía su Lista Abierta y actualiza *mejor_solucion* y *costo_mejor_solucion* luego de haber tomado el lock que protege dichos datos y consultado nuevamente si el costo de la solución hallada no supera *costo_mejor_solucion*[28]. Cuando el nodo extraído no era la solución, se inserta en la Lista Cerrada, se expande generando los sucesores y para cada uno se calcula la *Función de Zobrist* con el propósito de conocer a qué thread le corresponde su procesamiento. En caso que el nodo le pertenezca al mismo thread que lo generó, realiza la detección de duplicados insertándolo si fuera adecuado en su Lista Abierta; en caso contrario, ubica el nodo en la cola de salida local para el thread destino y si la misma supera una cierta cantidad de nodos almacenados *LNPT* (*Límite de Nodos Por Transferencia*) intenta tomar el lock de la cola de entrada del thread destino depositando, en caso de obtenerlo en forma

[28] Esto es necesario porque dos threads pueden encontrar dos soluciones con costo distinto en el mismo instante. Si todavía no se había encontrado una solución o ambas mejoraban el costo de la mejor solución parcial encontrada (*costo_mejor_solución*), el thread con la solución de menor costo podría actualizar los datos primero por haber obtenido el lock, y a continuación el segundo thread podría obtener el acceso para realizar su actualización. Si no se verifica nuevamente la condición acerca del costo de la solución, el segundo thread podría efectivizar la actualización de los datos compartidos quedando almacenada una solución subóptima.

inmediata, los nodos almacenados y vaciando la cola de salida propia.[29] La comunicación de nodos implica simplemente una copia de punteros. Cuando un thread es el primero en depositar nodos sobre la cola de entrada de otro thread, es decir que en ese momento la cola de entrada estaba vacía, debe realizar un aviso por si éste último estaba ocioso esperando el depósito.

- *Etapa thread ocioso:* luego de la etapa de procesamiento, si el thread quedó ocioso por haberse vaciado su Lista Abierta, realizará el envío de los nodos residentes en las colas de salida no vacías (en forma bloqueante ya que no tiene trabajo) y esperará un aviso correspondiente a alguno de los siguientes eventos:
  - *Fin de cómputo*: el thread 0 detectó terminación y cambió el valor de la variable fin para que todos se enteren de este estado.
  - *Llegada del token del algoritmo de detección de terminación:* el thread debe actualizar las variables compartidas correspondientes al token, según corresponda siguiendo el algoritmo de terminación de *Safra*, y pasarlo al thread siguiente lo que implica cambiarle el valor al campo dueño del token avisándole al thread sucesor que es el nuevo dueño. Por otra parte, el thread 0 realiza la tarea diferenciada de verificar si se dan las condiciones de terminación, caso en el cual cambia el valor de la variable *fin,* y en caso contrario inicia una nueva vuelta para la detección de terminación.
  - *Depósito de trabajo en la cola de entrada propia:* el thread debe obtener el lock de la cola de entrada propia, tomar todos los punteros a nodo depositados en ella vaciando la cola de entrada y soltar el lock. Para cada nodo cuyo costo es menor a *costo_mejor_solucion,* se realiza la detección de duplicados insertándolo en la Lista Abierta en caso que sea adecuado.

  El thread finaliza esta etapa de espera cuando posee nodos en su Lista Abierta, resultado de haber consumido *nodos* de su cola de entrada, o cuando se le ha comunicado el *fin* del cómputo.

Notar que a diferencia de la versión MPI del algoritmo, los threads siempre tienen actualizado el valor *costo_mejor_solucion* ya que es compartido.

El algoritmo de terminación implica actualizar la variable *estado y contador* propias del thread cada vez que deposita trabajo en la cola de entrada de otro thread o recibe trabajo desde su cola de entrada, incrementando/disminuyendo el contador en la cantidad de nodos depositados/ recibidos respectivamente.

La Figura 6.4 muestra el algoritmo PLA* GOHA asincrónico (HDA*) para memoria compartida. Cabe destacar que terminado el algoritmo las Listas Abiertas pueden vaciarse pero no las Listas Cerradas, ya que los nodos cerrados deben quedar accesibles por el thread 0 para recuperar la solución.

[29] Esta es una diferencia respecto a la versión de Burns, que tras cada generación de nodo correspondiente a otro thread intenta tomar el lock de su cola de entrada. Además cuando se obtiene el lock se aprovecha a comunicar todos los nodos almacenados en la cola de salida para el mismo thread destino.

```
Algoritmo PLA*GOHA-asincrónico-Pthreads
Thread i::
Entrada: cantidad de threads (N)

1) Inicializar LA y LC (Lista Abierta y Lista Cerrada)
2) Inicializar estado en blanco, contador en 0, fin en falso
   {alg. terminación}
3) Crear colas de salida vacías
4) Si i == 0
5)   Cargar el estado inicial y crear el nodo
6)   destino = Zobrist (nodo) % N
7)   si destino == i
8)     Insertar nodo en LA
9)   sino
10)    Tomar el lock de la cola de entrada del thread "destino"
11)    Depositar nodo
12)    Actualizar dimensión lógica
13)    Soltar el lock
       {* solo espera pasiva}
14)    Aviso para despertar al thread "destino" por depósito de trabajo
       {* fin solo espera pasiva}
15)    Actualizar contador y estado por envío siguiendo el algoritmo de Safra
16)mientras (not fin)
     {Etapa: recibir trabajo de la Cola de Entrada propia}
17)  si la cola de entrada i no está vacía
18)     Intentar tomar el lock de la cola de entrada i
19)     si se adquirió el lock inmediatamente
20)       Tomar los punteros a nodo de la cola
21)       Actualizar la dimensión lógica de la cola en 0
22)       Soltar el lock
23)       Realizar la detección de duplicados para los nodos con f(nodo) <
          costo_mejor_solucion e insertar en LA en caso adecuado.
          {* solo espera pasiva}
24)       Recibir aviso para despertar por depósito de trabajo
          {* fin solo espera pasiva}
25)       Actualizar contador y estado por recepción siguiendo el algoritmo de
          Safra
     {Etapa: procesamiento de nodos}
26)  mientras no esté vacía LA y no procesé LNPI nodos
27)    nodo = Eliminar nodo con costo mínimo de LA
28)    si f(nodo) >= costo_mejor_solucion
29)      Vaciar LA
30)    sino
31)      si h(nodo) == 0
32)        Tomar lock asociado a mejor_solucion
32)        si f(nodo) < costo_mejor_solucion
33)          costo_mejor_solucion = g(nodo)
34)          mejor_solucion = nodo
35)        Soltar lock
36)        Vaciar LA
37)      sino
38)        Insertar nodo en LC
39)        Expandir nodo generando los sucesores
40)        Para cada sucesor "s" de nodo
41)          si costo(s) >= costo_mejor_solucion
42)            Destruir "s"
43)          sino
44)            destino = Zobrist (s) % N
46)            si destino == i
```

```
47)              Realizar la detección de duplicados e insertar en LA
                 en caso adecuado.
48)            sino
49)              Poner "s" en la cola de salida propia para el thread
                 "destino"
50)              si cola de salida del thread "destino" tiene LNPT nodos
51)                Intentar tomar el lock asociado a la cola de entrada
                   "destino"
52)                si obtuve el lock en forma inmediata
53)                  Transferir todos los punteros nodo de la cola de salida
                     "destino" a la cola de entrada "destino"
54)                  Actualizar la dimensión lógica de la cola de entrada
                     "destino"
55)                  Soltar el lock
                     {* sólo espera pasiva}
56)                  si originalmente la cola de entrada "destino" estaba
                     vacía
57)                    Aviso para despertar al thread "destino"
                       por depósito de trabajo
                     {* fin solo espera pasiva}
58)                  Actualizar cola de salida "destino" dejándola vacía.
59)                  Actualizar contador y estado por envío siguiendo el
                     algoritmo de Safra
   {Etapa: espera ocioso}
60) si LA está vacía
61)   Para cada cola de salida correspondiente a un thread par "destino"
62)     si la cola de salida "destino" no está vacía
63)       Tomar lock de cola de entrada "destino"
64)       Transferir todos los punteros nodo de la cola de salida "destino"
          a la cola de entrada "destino"
65)       Actualizar la dimensión lógica de la cola de entrada "destino"
66)       Soltar el lock
          {* sólo espera pasiva}
67)       si originalmente la cola de entrada "destino" estaba vacía
68)         Aviso para despertar al thread "destino" por depósito de trabajo
          {* fin solo espera pasiva}
69)       Actualizar cola de salida "destino" dejándola vacía.
70)       Actualizar contador y estado por envío siguiendo el algoritmo de
          Safra
71)   mientras  (not fin) y LA está vacía
72)       Esperar aviso  {** en forma activa o pasiva}
73)       si not fin
74)         si soy el dueño del token
75)           si id == 0
76)             si están dadas las condiciones para detectar terminación
77)               fin = true
                  {* sólo espera pasiva}
78)               Avisar a todos el cambio de estado de la variable fin
                  {* fin sólo espera pasiva}
79)             sino
80)               Inicializar el estado del token en blanco y su contador en 0
81)               Establecer que el thread 1 es el dueño del token
                  {*sólo espera pasiva)
82)               Avisar al thread 1 que es el dueño del token
                  {*fin sólo espera pasiva)
83)           sino
84)             Seguir los pasos del algoritmo de Safra actualizando
                estado y color del token
85)             Establecer que el siguiente thread es el dueño del token
```

```
                {* solo espera pasiva}
86)             Avisar al thread siguiente que es el dueño del token
                {* fin solo espera pasiva}
87)           Estado= blanco
88)         sino
                {recibió trabajo}
90)             Tomar el lock de la cola de entrada i
91)             Tomar los punteros a nodo de la cola
92)             Actualizar la dimensión lógica de la cola en 0
93)             Soltar el lock
94)             Realizar la detección de duplicados para los nodos con
                f(nodo) < costo_mejor_solucion e insertar en LA en caso
                adecuado.
95)             Actualizar contador y estado por recepción siguiendo
                el algoritmo de Safra
```

Figura 6.4. Algoritmo PLA GOHA* (HDA*) para memoria compartida

En particular se implementaron dos versiones de este algoritmo. La primera versión utiliza *espera pasiva* mediante semáforos cuando el thread está ocioso y debe esperar por un evento, y *mutex locks* para implementar los locks que protegen las estructuras compartidas. Los semáforos son variables declaradas en el ámbito global habiendo uno para cada thread y son visibles por todos los threads. La segunda versión utiliza *espera activa* mediante *busy waiting* o *spinning* cuando el thread está ocioso esperando por algún evento, de esta forma verifica si hay cambios en el valor de las variables de interés, y *spin locks* para implementar los locks que protegen las estructuras compartidas. En el código de la Figura 6.4 se encuentran diferenciadas ambas técnicas. La siguiente sección explica las diferencias entre los mecanismos de sincronización.

6.2.2.5 Consideraciones sobre Mutex lock, Spin lock y Semáforos

Las primitivas de sincronización son necesarias para coordinar la actividad de múltiples threads, por ejemplo son requeridas cuando: se quiere asegurar que sólo un thread modifique una estructura compartida, es decir se deben excluir mutuamente en la ejecución del código que modifica el recurso compartido (conocido como *sección crítica*); se debe asegurar que todos los threads alcanzaron un cierto punto del código antes de continuar su ejecución, es decir los threads deben llegar a una *barrera* antes de continuar; se tiene una cantidad limitada de un recurso y se requiere un mecanismo para imponer dicho límite; un thread necesita señalizar a otro que estaba esperando que el primero complete alguna tarea. (Andrews, 1999) (Butenhof, 1997)

El primer ejemplo es un caso típico donde se utilizaría un *lock* o *cerrojo:* los threads siguen un protocolo de entrada (adquirir el *lock*) antes de ingresar a la sección crítica y un protocolo de salida al dejarla (soltar el *lock*). Existen diversas formas de implementar dichos protocolos, una de ellas es conocida como *mutex lock* y otra es conocida como *spin lock*.

Un *mutex lock* es un mecanismo de sincronización que puede ser adquirido por un único thread; cuando un segundo thread intenta adquirir un *mutex lock* que está ocupado se bloqueará; cuando el primer thread libere el *mutex lock,* el thread bloqueado deberá ser despertado permitiendo que intente nuevamente la adquisición del lock. Las operaciones para bloquear o despertar un thread requieren intervención del Sistema Operativo ya que éste es

responsable de elegir un nuevo proceso para ocupar la CPU que se está liberando. Por otro lado un *spin lock* se diferencia del *mutex lock* en que cuando un segundo thread intenta adquirir un *spin lock* ocupado, el thread seguirá realizando intentos para adquirirlo sin nunca bloquearse. (Grove, 2011)

La utilización de *spin lock* sobre arquitecturas single-core no tiene sentido, ya que cuando el thread no adquiere el *lock* podría liberar la CPU inmediatamente para que otro thread listo pueda continuar su ejecución. En arquitecturas multiprocesador el uso de *mutex lock* cuando la sección crítica posee pocas instrucciones y cuya ejecución dura un tiempo menor que aquel consumido por un *cambio de contexto,* podría degradar el rendimiento a causa del tiempo que conlleva bloquear al thread y despertarlo. (Butenhof, 1997)

El estándar POSIX Threads provee las siguientes primitivas: *pthread_mutex_lock()* permite a un thread adquirir un *mutex lock*, una vez finalizada la sección crítica debe soltar el lock invocando a la función *pthread_mutex_unlock().* Además el estándar provee la función *pthread_mutex_trylock()* que permite al thread que invoca realizar un intento de adquisición del *mutex lock*, quedando libre para continuar su ejecución en caso de no obtenerlo inmediatamente. Adicionalmente provee la función *pthread_spin_lock()* permitiendo al thread realizar continuos intentos de adquisición del *spin lock* hasta efectivamente conseguirlo, la función *pthread_spin_unlock()* para soltar el *spin lock* una vez ejecutada la sección crítica, y *pthread_spin_trylock()* que permite al thread que invoca realizar el intento de adquirir el *spin lock* pero en caso de no lograrlo inmediatamente queda libre para continuar su ejecución. (Engelschall, 2006)

Los últimos dos ejemplos planteados donde se requiere sincronización son casos donde se utilizaría un mecanismo conocido como semáforo, el cual se representa a través de una variable contador que puede ser incrementado o disminuido a través de las operaciones P y V respectivamente. Un thread que ejecuta un P disminuye en 1 el contador asociado al semáforo de manera atómica cuando el valor del mismo era mayor que 0, caso contrario el thread se bloqueará hasta que el contador asociado tenga un valor mayor a cero y pueda efectivizar el decremento. Cuando un thread ejecuta un V incrementa en 1 el contador asociado al semáforo atómicamente despertando a los threads bloqueados sobre el mismo. El semáforo también puede ser utilizado para proteger secciones críticas o para implementar barreras (Grove, 2011).

La biblioteca *Semaphore.h* posibilita al usuario declarar semáforos y provee las siguientes funciones para trabajar con ellos: *sem_wait()* que implementa la operación P, *sem_post()* que implementa la operación V, y *sem_getvalue()* que retorna el valor del contador asociado al semáforo. (Engelschall, 2006)

6.2.2.6 Gestión de memoria dinámica con Jemalloc

Con el objetivo de obtener el mayor rendimiento posible al trabajar con múltiples procesos/threads sobre un multiprocesador se debe tener en cuenta la relación existente entre la aplicación, que constantemente realiza operaciones malloc() y free(), y la biblioteca de gestión de memoria dinámica elegida.

En el Capítulo 5 se estudiaron diversas bibliotecas para gestión de memoria dinámica, y se seleccionó a Jemalloc (Evans, 2006) para ser utilizada en conjunto con las implementaciones presentadas, por lo que resulta de interés analizar la interacción éstas.

En el algoritmo PLA* GOHA para memoria distribuida (HDA* MPI) cada proceso accede únicamente a su espacio de direcciones exclusivo y se comunican vía mensajes, por esto el proceso realizará operaciones *malloc*() y *free*() sobre las *arenas* asignadas al mismo. A medida que se incrementa el número de *arenas*, disminuye la probabilidad que los procesos compartan las mismas.

Por otro lado, en el algoritmo PLA* GOHA para memoria compartida (HDA* Pthreads) los threads comparten el espacio de direcciones y se comunican los nodos enviándose punteros; por esto cuando un thread A genera un nodo mediante un *malloc* sobre una de sus arenas asignadas y pasa el puntero a un thread B, si este último decidiera realizar una operación *free* –por ejemplo al vaciar su Lista Abierta- estaría accediendo a la *arena* del thread A causando contención en el acceso a la misma lo cual degradará el rendimiento.

Para solucionar la situación antes expuesta, se incorporó un *pool* de punteros a nodo para cada thread (*Memory Pool*) en el cual se almacenarán los punteros a nodo que el thread quiera "liberar" para su posterior reúso ante generaciones de nodos futuras, evitando así que al realizar un free se esté accediendo a la *arena* de otro thread. Cuando se genera un nodo, se reservará un nuevo espacio para el mismo (malloc) cuando el *pool* está vacío, en caso contrario se tomará un puntero del *pool* para ser reusado.

6.3 Discusión y conclusiones

Se implementó una versión propia del algoritmo HDA* para memoria distribuida en Lenguaje C y utilizando la biblioteca de paso de mensajes MPI. Este algoritmo se basa en aquel propuesto en (Kishimoto, et al., 2013) pero difiere en:

- La utilización del algoritmo para detección de terminación propuesto por *Dijkstra* y *Safra* (Dijkstra, 1987) (Dijkstra, et al., 1983) el cual no incrementa el tamaño de los mensajes de trabajo enviados por los procesos para el balance de carga, a diferencia del algoritmo HDA* original que utiliza el *algoritmo de tiempo de Mattern* el cual incorpora datos adicionales en cada mensaje enviado.
- La incorporación de un parámetro *LNPI* (Límite de Nodos Por Iteración) que indica la cantidad de nodos a procesar por iteración del algoritmo, a diferencia del algoritmo

original el cual procesa 1 nodo por iteración. De esta manera, el parámetro LNPI establece el intervalo de verificación de llegada de mensajes con trabajo y mensajes que transportan costos de mejores soluciones encontradas. Al procesar más de un nodo por iteración se reducirán las constantes comprobaciones por llegada de mensaje, dando lugar a una expansión de nodos especulativa.

Además, en el capítulo se estudió el desarrollo de una versión propia de HDA* para memoria compartida en Lenguaje C y utilizando las herramientas de programación paralela sobre memoria compartida de la biblioteca *Pthreads*. El algoritmo está basado en aquel propuesto en (Burns, et al., 2010) pero difiere en:

- La incorporación de un algoritmo para detectar terminación en forma *descentralizada*, que es una adaptación de aquel propuesto por *Dijkstra y Safra* (Dijkstra, 1987) (Dijkstra, et al., 1983).
- La incorporación del parámetro *LNPI (Limite de Nodos por Iteración),* que indica la cantidad límite de nodos a procesar por iteración del algoritmo. Si bien los autores del algoritmo original proponen el uso de este parámetro, no analizan su influencia sobre el rendimiento.
- La acumulación por parte del thread de una cantidad parametrizable de nodos dirigidos a otro thread antes de intentar el traspaso de los mismos (*LNPT o Límite de Nodos por Transferencia)*, es decir, no se realizan transferencias tras cada generación de nodo como en el algoritmo original. Esto permite disminuir la contención en el acceso a las estructuras utilizadas para la comunicación.
- La utilización de una técnica *Memory Pool* para prevenir la degradación de rendimiento causada por operaciones alloc-free en una relación productor-consumidor entre distintos threads. Dicho de otra manera, la problemática resuelta es la siguiente: durante el algoritmo los threads irán transfiriendo los punteros a nodos cuyo procesamiento corresponde a otro thread; cuando un thread genera un nodo, el mismo es alocado en una de las áreas de memoria dinámica asignada al mismo por el gestor de memoria (*Jemalloc* llama a ésta zona *arena*) las cuales están protegidas para mantener su consistencia ante el posible acceso por parte de distintos threads; cuando un thread A transfiere el puntero a un nodo generado por él a otro thread B, si éste último realiza una liberación del espacio asignado para el mismo (por ejemplo a causa que el nodo conduce a una solución subóptima), tendría que acceder a la *arena* asignada al thread A, lo cual genera contención en el acceso a la misma. La técnica propuesta consiste en que cada thread almacene los nodos cuyo espacio quiera liberar en un *pool*, para que sean re-utilizados ante futuras generaciones de nodos.

En el Capítulo 7 se evaluará el rendimiento de los algoritmos desarrollados.

CAPÍTULO 7: RESULTADOS

En este capítulo se muestran los resultados experimentales obtenidos a partir de la ejecución de los algoritmos propuestos en el Capítulo 6.

Para todos los casos se utilizó una plataforma de cómputo paralelo compuesta por 7 máquinas conectadas por una red Gigabit Ethernet. Cada máquina posee dos procesadores Intel ® Xeon ® E5620 (Intel, 2010), cada uno cuenta con 4 *cores* físicos de 2.4 Ghz con tecnología *Hyperthreading*, es decir en total son 8 *cores* virtuales, y *Turbo Boost* que permite que la velocidad del *core* se incremente a un límite máximo de 2.66 Ghz; cada *core* físico posee dos caché L1 de 64KB para datos e instrucciones respectivamente y una caché L2 de 256KB; todos los *cores* del procesador comparten una caché L3 de 12MB. Cada procesador posee un controlador de memoria, por lo que la máquina cuenta con arquitectura NUMA, y utiliza una interconexión *QuickPath interconnect (QPI)* de 5.86 GT/s. Cada máquina posee 32 GB de RAM, DDR3 1066 Mhz. Por último, el Sistema Operativo es Fedora 14.

Los algoritmos fueron compilados haciendo uso de la biblioteca de gestión de memoria dinámica *ptmalloc* (Gloger, 2014) o con la biblioteca *Jemalloc* (Evans, 2006).

Las pruebas se realizaron tomando en cuenta las 100 configuraciones iniciales del 15-Puzzle utilizadas por (Korf, 1985), numeradas del 1 a 100, y también se tuvieron en cuenta 6 de las configuraciones que tienen mayor cantidad de pasos para su solución (Brüngger, 1998) para el análisis de escalabilidad de los algoritmos paralelos ya que el tiempo de resolución de alguna de éstas es considerable, numeradas desde 101 a 106.

La Sección 7.1 analiza los resultados obtenidos de la ejecución del algoritmo A* secuencial utilizando distintos gestores de memoria dinámica y diferentes heurísticas.

La Sección 7.2 muestra los resultados experimentales para el algoritmo HDA* para memoria compartida (HDA* Pthreads), utilizando espera pasiva o activa, incorporando o no la técnica *Memory Pool,* y realiza un análisis del impacto de los parámetros configurables del algoritmo sobre el rendimiento.

La Sección 7.3 analiza el comportamiento del algoritmo HDA* para memoria distribuida (HDA* MPI) cuando se ejecuta sobre una máquina multicore y cuando se ejecuta sobre un cluster de multicore. Asimismo, se estudia el impacto de los parámetros del algoritmo sobre el rendimiento obtenido.

La Sección 7.4 realiza mediciones de consumo de memoria para las aplicaciones HDA* Pthreads y HDA* MPI sobre una máquina multicore.

Por otro lado, la Sección 7.5 realiza una comparación del rendimiento obtenido por los algoritmos HDA* Pthreads y HDA* MPI cuando se los ejecuta sobre multicore.

Por último, la Sección 7.6 presenta las conclusiones del capítulo.

7.1 A* secuencial

La ejecución del algoritmo secuencial es determinista, es decir cuando se realizan varias pruebas para la misma entrada (configuración inicial y final) el resultado arrojado por el algoritmo es idéntico siempre que se mantenga la función heurística y la configuración de las estructuras de datos que utiliza.

7.1.1 Efecto de la heurística

A los efectos de comprobar la posible mejora de los tiempos secuenciales a medida que se afina la función heurística utilizada para estimar el costo de los nodos durante la búsqueda, se ejecutó el algoritmo secuencial con las 100 configuraciones iniciales y la configuración final propuestas por (Korf, 1985), y las distintas heurísticas planteadas para el Puzzle en el Capítulo 3: Suma de las Distancias de Manhattan de las Fichas (SDM), SDM a la cual se adiciona la detección de Conflictos Lineales entre las Fichas (SDM+CL), SDM+CL agregando el análisis de Últimos Movimientos (SDM+CL+UM), SDM+CL+UM incorporando la detección de conflictos con las Fichas de las Esquinas (SDM+CL+UM+FE), las cuales para simplificar la descripción de los resultados experimentales se citarán de ahora en adelante como H1, H2, H3 y H4 respectivamente.

Para cada configuración inicial y cada heurística se realizaron 10 ejecuciones sobre una de las 7 máquinas de la computadora paralela descripta con anterioridad, calculando para cada muestra: tiempo de ejecución en segundos, cantidad de nodos procesados o expandidos, cantidad de nodos generados, cantidad de nodos podados[30], cantidad de promociones de nodos en la Lista Abierta, cantidad de reapertura de nodos cerrados y cantidad total de duplicados encontrados durante la búsqueda[31]. Todas las pruebas finalizaron exitosamente a excepción de aquellas para la configuración número 88 y la heurística H1 las cuales fueron abortadas manualmente por haber extinguido la memoria RAM disponible.

En general los resultados experimentales muestran que al afinar la heurística disminuye el tiempo de ejecución promedio de cada configuración, siendo el porcentaje de aceleración obtenido dependiente de la configuración inicial seleccionada. En resumen, la heurística H2 muestra aceleraciones respecto a H1 en un rango de 40.32% a 94.67%, las aceleraciones obtenidas para H3 respecto a H2 varían entre un 0.54% a 83.06%[32], y las obtenidas para H4 respecto a H3 se encuentran entre 2.95% al 80.6%[33].

[30] Los nodos podados son *nodos duplicados* que tienen costo superior a un nodo que representa el mismo estado generado con anterioridad.

[31] Los nodos duplicados son todos aquellos para los cuales, al momento de realizar la detección de duplicados, se encontró un nodo generado con anterioridad que representa el mismo estado, independientemente si su costo mejoraba o no a este último.

[32] Solamente las pruebas para la configuración número 93 no mejoraron los tiempos; el tiempo de procesamiento en segundos fue 1.68, 0.44 y 0.55 para H1, H2 y H3 respectivamente; el aumento en el tiempo fue a causa de un incremento en la cantidad de nodos procesados.

[33] Únicamente las pruebas para las configuraciones número 6, 20 y 21 no mejoraron los tiempos obtenidos con la heurística H3, siendo casos similares al planteado para la configuración 93.

Al afinar la función heurística el costo de cada nodo se estima de manera más exacta. De lo anterior se podría pensar que es imposible que el algoritmo que utiliza una heurística más afinada pueda procesar mayor cantidad de nodos o demorar más tiempo para resolver una configuración inicial. Sin embargo, esta situación puede ocurrir debido a las siguientes causas:

- En general, a medida que se afina una función heurística, su tiempo de procesamiento se incrementa. En consecuencia, puede suceder que para una configuración inicial y dos heurísticas H1 y H2, siendo H2 una mejora de H1, el algoritmo procese similar cantidad de nodos o incluso que con H2 procese menor cantidad de nodos y sin embargo obtener un tiempo de ejecución mayor.
- Al mejorar la heurística podría ocurrir que se procese mayor cantidad de nodos debido a un incremento en los nodos de costo igual que la solución. (Russel & Norvig, 2003)

Como conclusión se puede decir que al mejorar la heurística es probable que se procesen menor cantidad de nodos y que el tiempo de ejecución disminuya.

De ahora en adelante, se utilizará la función heurística H4 debido a los buenos resultados obtenidos en el tiempo de ejecución para la mayoría de las configuraciones. Cabe destacar que las heurísticas H3 y H4 no son consistentes, por lo que puede darse que un nodo cerrado sea reabierto por haber encontrado un camino mejor hacia el mismo.

7.1.2 Efecto de la biblioteca de gestión de memoria dinámica

En el Capítulo 5 se planteó la importancia de la correcta elección de la biblioteca de gestión de memoria dinámica para reducir la contención en el acceso a las estructuras manejadas por el gestor a raíz de constantes operaciones de reserva (malloc) y liberación (free) de memoria realizadas por distintos hilos/procesos de una aplicación paralela.

El gestor disponible en el Sistema Operativo *Linux* es *ptmalloc*. Por otro lado, *Jemalloc* (Evans, 2006) es una alternativa usada en el ámbito científico y comercial con la que se logra buen rendimiento a medida que aumenta el número de procesos o hilos.

Se ejecutó el algoritmo secuencial con las 100 configuraciones iniciales y la configuración final propuestas por (Korf, 1985), utilizando la función heurística H4 y variando el gestor de memoria dinámica entre *ptmalloc* y *Jemalloc*. Se configuró Jemalloc para trabajar con 256 *arenas* en total ya que esta configuración se utilizará cuando se realicen las pruebas para los algoritmos paralelos.

Para cada configuración inicial se realizaron 10 ejecuciones sobre una de las 7 máquinas de la computadora paralela descripta con anterioridad, calculando para cada prueba el tiempo de ejecución en segundos. De los resultados se observa que el tiempo promedio de ejecución para cada configuración arrojado por las muestras que utilizan *Jemalloc* presenta una

reducción que va en el rango 0.91% y 15.8%[34] respecto al tiempo promedio de ejecución para las mismas configuraciones utilizando *ptmalloc*.

Si bien la contención en el acceso a las estructuras manejadas por el gestor de memoria ocurre cuando la aplicación tiene múltiples procesos o hilos y no en el caso de una aplicación secuencial, el autor de *Jemalloc* considera que el rendimiento alcanzado por las aplicaciones secuenciales dependerá del patrón de reservas y liberaciones que realice.

Dado que se ha comprobado que Jemalloc utilizando 256 *arenas* mejora el rendimiento en general, se utilizará dicho gestor en las pruebas a realizar de aquí en adelante.

7.1.3 Efecto de la técnica "*Memory pool*"

En el Capítulo 6 se propuso el uso de un *pool* de punteros a nodo (*Memory Pool)*, que permite almacenar aquellos nodos que un thread quiera "liberar", cuando se utiliza *Jemalloc* junto con el algoritmo HDA* para memoria compartida. Esta estructura permite evitar la contención en el acceso a las *arenas* asignadas a un thread por el gestor de memoria, que aparece cuando un thread A realiza la liberación del espacio de memoria de un nodo (*free*) cuya reserva (*malloc*) fue realizada por otro thread B. El *pool* es utilizado de forma tal que cuando se quiere destruir un nodo, el puntero se almacena dentro del *pool* en vez de ser liberado con *free*. Estos punteros almacenados serán reutilizados ante operaciones posteriores de generación de nodo; únicamente cuando el *pool* está vacío se realizará un *malloc*.

Con el objetivo de demostrar que esta técnica no afecta el rendimiento del algoritmo secuencial, se ejecutaron 10 pruebas para cada una de las 100 configuraciones iniciales y la configuración final propuestas por (Korf, 1985), utilizando la función heurística H4, Jemalloc configurado para usar 256 *arenas* y el pool de punteros *"Memory Pool".*

Se contrastaron los tiempos promedio de ejecución obtenidos para cada una de las configuraciones iniciales contra aquellos que surgen de las pruebas de la sección anterior cuando no se utiliza *"Memory Pool".* De la comparación se observa que el uso del *pool* no tiene ningún beneficio para la aplicación secuencial: 17 configuraciones sufrieron un incremento en el tiempo promedio de ejecución de entre un 2% y 10%; 15 configuraciones aumentaron su tiempo entre un 1% y 2%; 43 configuraciones tuvieron un leve incremento en su tiempo que varía entre 0% y 1%; por último, 25 configuraciones tuvieron una reducción en su tiempo que está entre 0% y 6.45%.

En general no se observan variaciones relevantes en el rendimiento para las configuraciones cuyo tiempo de ejecución es más significativo, por lo que de aquí en adelante se utilizarán los resultados secuenciales obtenidos sin utilizar la técnica *Memory Pool* para el análisis del rendimiento de los algoritmos paralelos.

[34] La única excepción se produjo para la configuración número 71 cuyo tiempo de ejecución promedio obtenido de las pruebas utilizando ptmalloc fue 0.038 segundos y 0.04 segundos para las pruebas utilizando Jemalloc, alcanzando un aumento de un 3% en el tiempo promedio de ejecución. Estas pruebas fueron descartadas por no ser significativas.

7.2 HDA* para memoria compartida (HDA* Pthreads)

El algoritmo paralelo HDA* para memoria compartida (HDA* Pthreads) es no determinista, es decir cuando se realizan múltiples ejecuciones con los mismos datos de entrada (estado inicial y final) el resultado arrojado por el algoritmo puede ser distinto, aun manteniendo la misma función heurística y la configuración inicial de las estructuras de datos que utiliza. Esto es posible ya que pueden existir múltiples soluciones óptimas para el estado inicial y, dado que los threads se reparten el espacio de estados en forma dinámica, los nodos procesados por un thread varían según cómo se dan los eventos asincrónicos en el sistema.

Las pruebas experimentales se llevaron a cabo en una de las 7 máquinas de la computadora paralela mencionada con anterioridad, por lo tanto se dispone de 8 *cores* físicos[35]. En las distintas pruebas se utilizó afinidad para asociar cada thread a un *core* exclusivo; para ello se empleó la función *sched_setaffinity()* (Grove, 2011). En aquellas pruebas con 4 threads se ubicó un par de threads en cada procesador de la máquina, y en aquellas pruebas con 8 threads se ubicó un thread por cada *core* físico de la máquina multiprocesador.

Las variables a evaluar en cada ejecución son: tiempo de ejecución en segundos, cantidad de nodos procesados o expandidos entre todos los threads, cantidad de nodos generados entre todos los threads, cantidad de nodos podados entre todos los threads, cantidad de promociones de nodos en la Lista Abierta entre todos los threads, cantidad de reaperturas de nodos cerrados entre todos los threads, cantidad total de duplicados encontrados durante la búsqueda entre todos los threads, y la cantidad mínima y máxima de nodos procesados por un thread.

Las configuraciones iniciales seleccionadas son aquellas utilizadas en la sección anterior cuya ejecución secuencial es al menos 5 segundos[36]. Para el análisis de escalabilidad también se tuvieron en cuenta 6 configuraciones que tienen mayor cantidad de pasos para su solución (numeradas del 101 a 106) (Brüngger, 1998).

Se utilizó el gestor de memoria dinámica *Jemalloc* configurado con 256 *arenas* y la heurística H4. Para cada configuración inicial y cada grupo de parámetros se obtuvieron 100 muestras. Los parámetros utilizados son: la cantidad de cores/threads, cuyo valor varía entre 4 y 8; el límite de nodos a procesar por iteración del algoritmo (*LNPI - Límite de Nodos Por Iteración*) cuyos valores serán 1, 5, 50 o 500; y la cantidad de nodos ajenos que un thread almacena en las colas de salida antes de intentar el traspaso de los mismos a su thread dueño *(LNPT - Límite de Nodos Por Transferencia)*, cuyos valores se fijaron en 26 nodos (1KB datos), 210 nodos (8KB datos) o 1680 nodos (64KB datos). Luego se promediaron los datos arrojados de las 100 ejecuciones realizadas para cada configuración y conjunto de parámetros, a lo que se llamará *muestra promedio*.

[35] Se desactivaron las características *Hyperthreading* y *Turbo Boost* para evitar variaciones entre las ejecuciones causadas por la arquitectura y no por eventos asincrónicos del algoritmo.
[36] Las configuraciones son las siguientes: 3, 15, 17, 21, 26, 32, 33, 49, 53, 56, 59, 60, 66, 82, 88, 100

En las siguientes subsecciones se analizará el efecto producido en el rendimiento al utilizar espera activa o pasiva, al incorporar la técnica *Memory Pool,* y al variar el valor de los parámetros *LNPI* y *LNPT* entre los definidos. Por último, se evaluará el rendimiento obtenido por el algoritmo cuando utiliza las técnicas y valores de parámetros que optimizaron los tiempos de ejecución.

7.2.1 Efecto en el rendimiento de la espera pasiva y espera activa

Se ejecutaron dos baterías de prueba haciendo uso de *espera activa* o *espera pasiva* respectivamente, limitando *LNPT* a 26 nodos (1KB datos), variando *LNPI* entre los valores ya mencionados, y utilizando el pool de punteros *Memory Pool.* Las configuraciones iniciales utilizadas son aquellas propuestas por Korf cuyo tiempo de ejecución secuencial es al menos 5 segundos.

Los tiempos de ejecución de las *muestras promedio* arrojados por las baterías de prueba no muestran un beneficio evidente para ninguna técnica de espera particular. Esto puede deberse al asincronismo del algoritmo, ya que la mayoría de las veces los threads realizan intentos para tomar locks y en caso de no obtenerlo continúan la ejecución normalmente; asimismo cada thread se ejecuta sobre un core exclusivo, por lo tanto el hecho de esperar activamente o acudir al Sistema Operativo para realizar una espera pasiva no produce cambios drásticos en el rendimiento.

7.2.2 Efecto en el rendimiento de la técnica *Memory Pool*

Se ejecutaron dos baterías de prueba incorporando o no la técnica *Memory Pool*, haciendo uso de *espera activa*, limitando *LNPT* a 26 (1KB de datos) y variando *LNPI* entre los valores ya mencionados. Las configuraciones iniciales utilizadas son aquellas propuestas por Korf cuyo tiempo de ejecución secuencial es al menos 5 segundos.

Se compararon los tiempos de ejecución de las dos *muestras promedio* obtenidas para cada configuración y grupo de parámetros (una de ellas incorpora la técnica *Memory Pool)*. Los tiempos de ejecución de las *muestras promedio* que usan la técnica *Memory Pool* reducen entre un 4.5% y un 12.82% a aquellos tiempos de ejecución de las *muestras promedio* que no utilizan dicha técnica. En general el porcentaje de reducción para las pruebas con 4 threads se encuentra entre 4.5% y 8.64%, mientras que para las pruebas con 8 threads se encuentran entre 6.44% y 12.82%.

Se demuestra así el beneficio que produce la técnica *Memory Pool* para reducir la contención en el acceso a las *arenas* asignadas por el gestor de memoria a los distintos threads, en casos donde existe una relación productor-consumidor entre estos (es decir cuando un thread A realiza una reserva de espacio para una variable dinámica, cuyo puntero transmite a un thread B y este último por alguna razón libera el espacio).

7.2.3 Efecto en el rendimiento de los parámetros LNPI y LNPT

En esta sección se analiza el impacto de los parámetros *LNPI* y *LNPT* sobre el rendimiento del algoritmo HDA* para memoria compartida. Las configuraciones iniciales son aquellas utilizadas en el análisis de escalabilidad del algoritmo, presentado más adelante, siendo en total 22.

7.2.3.1 Límite de Nodos Procesados Por Iteración (LNPI)

El parámetro *LNPI* determina cuántos nodos debe procesar un thread en cada iteración del algoritmo, es decir indica la cantidad de nodos que el thread debe expandir antes de que realice otro intento de consumo de nodos de su cola de entrada.

Con el objetivo de analizar el impacto del parámetro sobre el tiempo de ejecución, se tomaron *todas* las *muestras promedio* que surgen de las ejecuciones realizadas con LNPT=26, espera activa y la técnica *Memory Pool*, y se agruparon aquellas con igual configuración y cantidad de cores, es decir cada grupo contiene las *muestras promedio* que difieren únicamente en el parámetro *LNPI*.

Para cada *grupo* se calculó la Desviación Estándar (*DE*) y el Coeficiente de Variación (*CV*) del tiempo de ejecución, lo que proporciona una medida de cuánto difiere el tiempo de ejecución de una *muestra promedio* del grupo respecto al *tiempo promedio grupal*.

En general, los resultados arrojados revelan que la *DE* de los grupos se encuentra entre 0.0017 y 6.51, esto indica que variar el parámetro *LNPI* entre los valores definidos provoca que los tiempos de ejecución de las *muestras promedio* se desvíen en a lo sumo 6.51 segundos de su *promedio grupal*. Además vale destacar que el 93% de los grupos tiene una *DE* menor a 1 y el 86% por debajo de 0.5.

Por otro lado, los valores generales de *CV* obtenidos para los grupos oscilan entre 0.001 y 0.06, es decir el tiempo de ejecución de una *muestra promedio* del grupo se desvía entre un 0.1% y un 6% de la *media grupal* al variar el parámetro *LNPI* entre los valores fijados. Asimismo se destaca que el 95% de los grupos tienen CV por debajo de un 0.03.

Los *grupos* asociados a ejecuciones con 4 cores poseen *DE* entre 0.0017 y 6.5. Por su parte, el *CV* para estos grupos varía entre 0.001 y 0.03, teniendo el 86% de los mismos un CV por debajo de 0.02. Por otra parte, los grupos asociados a ejecuciones con 8 cores poseen *DE* entre 0.0023 y 0.93; mientras que el CV para ellos oscila entre 0.0013 y 0.06, teniendo el 91% de los grupos un CV por debajo de 0.02.

Por último, no se encontró relación entre la complejidad de la instancia inicial y el DE ó el CV.

Por los bajos valores de *CV* obtenidos se concluye que existe poca variación en los tiempos de ejecución al cambiar el valor del parámetro *LNPI* entre los definidos (1, 5, 50 o 500). Esto se

debe a que el parámetro sólo influye en la frecuencia en la que un thread intenta tomar nodos de su cola de entrada[37]:

- Cuanto *más frecuentes* son los intentos de acceso exitoso a la cola de entrada, la calidad de los nodos almacenados en la Lista Abierta del thread estará más actualizada. Los intentos fallidos de acceso a la cola de entrada no influyen en el *overhead* ya que, gracias al asincronismo presente en el algoritmo, no se fuerza a esperar al thread haciendo que éste continúe su trabajo.

- Ante intentos *menos frecuentes* de acceso a la cola de entrada (valores muy altos de LNPI) se estará obligando al thread a trabajar mayor cantidad de nodos de su Lista Abierta en forma *especulativa*, es decir el thread estaría siguiendo solamente su guía heurística local sin incorporar nuevos nodos, por lo que en general aumentará el tiempo de ejecución a causa de un incremento en el *Overhead de Búsqueda*. Esta hipótesis fue comprobada realizando ejecuciones para las 22 configuraciones, utilizando *Memory Pool,* espera activa, limitando *LNPT* a 26, y *LNPI* en 10000.

Asimismo, cuando el parámetro *LNPT* (*Límite de Nodos por Transferencia*) varía entre los valores {26, 210, 1680} se pueden mantener las mismas conclusiones. La Tabla 7.1 resume los rangos de CV y DE que toman los grupos, clasificados por LNPT.

Tabla 7.1. Rangos de CV y DE clasificados por LNPT

LNPT	Rango DE en segundos	Rango CV
26	0.0017-6.5	0.001-0.06
210	0.0036 - 0.78	0.0009-0.036
1680	0.004-0.72	0.001-0.062

7.2.3.2 Límite de Nodos Por Transferencia (LNPT)

El algoritmo HDA* Pthreads incorpora el parámetro *Limite de Nodos Por Transferencia (LNPT*) que indica la cantidad de nodos que el thread debe acumular en la cola de salida local para un thread destino antes de intentar la transferencia de los mismos. Además, cuando se obtiene el lock de la cola de entrada del thread destino para realizar la transferencia, se aprovecha a comunicar *todos* los nodos almacenados en la cola de salida. Estas son diferencias respecto a la versión de HDA* para memoria compartida propuesta por Burns, que tras cada generación de nodo correspondiente a otro thread intenta transferir ese único nodo, y después de trabajar sobre una cierta cantidad de nodos de la Lista Abierta[38] intenta transferir los nodos residentes en las colas de salida no vacías.

Con el objetivo de comprobar si el uso del parámetro *LNPT* trae ventajas en el rendimiento, se ejecutó el algoritmo HDA* Pthreads limitando *LNPT* a 1, haciendo uso de *espera activa* y de la técnica *Memory Pool*. Las pruebas utilizan las 22 configuraciones iniciales presentadas al

[37] Para este algoritmo, LNPI no influye en el conocimiento del costo de soluciones parciales encontradas ya que *costo_mejor_solucion* es global y sus actualizaciones son conocidas apenas cambia su valor.
[38] En el artículo no analizan el impacto de dicho parámetro sobre el rendimiento.

principio de la sección, *LNPI* toma alguno de los valores del conjunto {1,5,50,500} y la cantidad de cores varía entre 4 y 8. Para cada configuración y grupo de parámetros se ejecutaron 100 pruebas y luego se promediaron los tiempos de ejecución obteniendo así la denominada *muestra promedio.*

Se compararon los tiempos de ejecución de las dos *muestras promedio* que utilizan LNPT=1 y LNPT=26 respectivamente, para cada configuración inicial, *LNPI* y cantidad de cores. El 97.7% de las *muestras promedio* que utilizan LNPT=26 presentan una mejora en los tiempos de ejecución que varía entre 0.24% y 10.61%. Las *muestras* restantes (2.3%) presentaron una desmejora en el tiempo de ejecución que va entre 0.39% y 6.13% respecto a las *muestras promedio* que utilizan LNPT=1 (todas estas surgen de ejecuciones con 8 cores).

A partir de los resultados anteriores queda demostrado el beneficio que trae la utilización del parámetro *LNPT*.

Luego se realizaron ejecuciones siguiendo la misma estrategia para LNPT=210 (8KB datos) y LNPT=1680 (64KB datos).

Con el objetivo de analizar el impacto del parámetro *LNPT* sobre el tiempo de ejecución, se tomaron *todas* las *muestras promedio* que surgen de las ejecuciones para LNPT={26,210,1680}, y se agruparon aquellas con igual configuración, cantidad de cores y *LNPI*, es decir cada grupo contiene las *muestras promedio* que difieren únicamente en el parámetro *LNPT*.

Para cada *grupo* se calculó la Desviación Estándar (*DE*) y el Coeficiente de Variación (*CV*) del tiempo de ejecución, lo que proporciona una medida de cuánto difiere el tiempo de ejecución de una *muestra promedio* del grupo respecto al *tiempo promedio grupal*.

En general, los resultados arrojados revelan que la *DE* de los grupos se encuentra entre 0.0077 y 8.51, esto indica que variar el parámetro *LNPT* entre los valores definidos provoca que los tiempos de ejecución de las *muestras promedio* se desvíen en a lo sumo 8.51 segundos de su *promedio grupal*. Además vale destacar que el 84.7% de los grupos tiene una *DE* menor a 1 y el 71.6% por debajo de 0.5.

Por otro lado, los valores generales de *CV* obtenidos para los grupos oscilan entre 0.0035 y 0.26, es decir el tiempo de ejecución de una *muestra promedio* del grupo se desvía entre un 0.35% y un 26% de la *media grupal* al variar el parámetro *LNPT* entre los valores fijados. Asimismo, sólo el 70.45% de los grupos tienen CV por debajo de un 0.1, el 27.8% exhibe un CV por debajo de 0.05 y el 10.2% posee CV por debajo de 0.03.

Por los valores moderadamente altos de CV se concluye que el parámetro LNPT tiene influencia sobre el tiempo de ejecución. Esto se debe a que LNPT impacta en la frecuencia de intento de adquisición del lock de la cola de entrada del thread destino para el traspaso de datos:

- Una frecuencia alta (LNPT bajo) significa una mayor contención en el acceso a los locks asociados a las colas de entrada. Esto lleva a incrementar los intentos fallidos de acceso a la cola de entrada propia por parte de un thread en la etapa de consumo de nodos, por lo

que se notó en general un incremento considerable en el *Overhead de Búsqueda* a causa de que el thread realizará mayor trabajo especulativo sobre nodos locales.

- Una baja frecuencia (LNPT alto) demora la comunicación de nodos entre los threads lo cual provoca mayor ociosidad de threads, desbalance de calidad de nodos poseídos por los threads y aumenta la cantidad de trabajo especulativo sobre nodos locales incrementando en consecuencia el *Overhead de Búsqueda*.

En la mayoría de los casos el valor de LNPT que optimiza los tiempos de ejecución para cada configuración y cantidad de cores es 210 (excepto para la configuración 102 y 4 cores cuyo valor de LNPT óptimo fue 1680), siendo este valor óptimo para la arquitectura utilizada.

Las Figuras 7.1 y 7.2 muestran el Speedup según la configuración utilizando 4 cores y 8 cores respectivamente y para cada valor de LNPT, teniendo en cuenta que la *muestra promedio seleccionada* por configuración y cantidad de cores es aquella cuyo LNPI optimiza el rendimiento; las configuraciones se muestran ordenadas según la carga de trabajo secuencial. De manera similar, las Figuras 7.3 y 7.4 muestran la Eficiencia según la configuración para 4 y 8 cores respectivamente, para cada valor de LNPT.

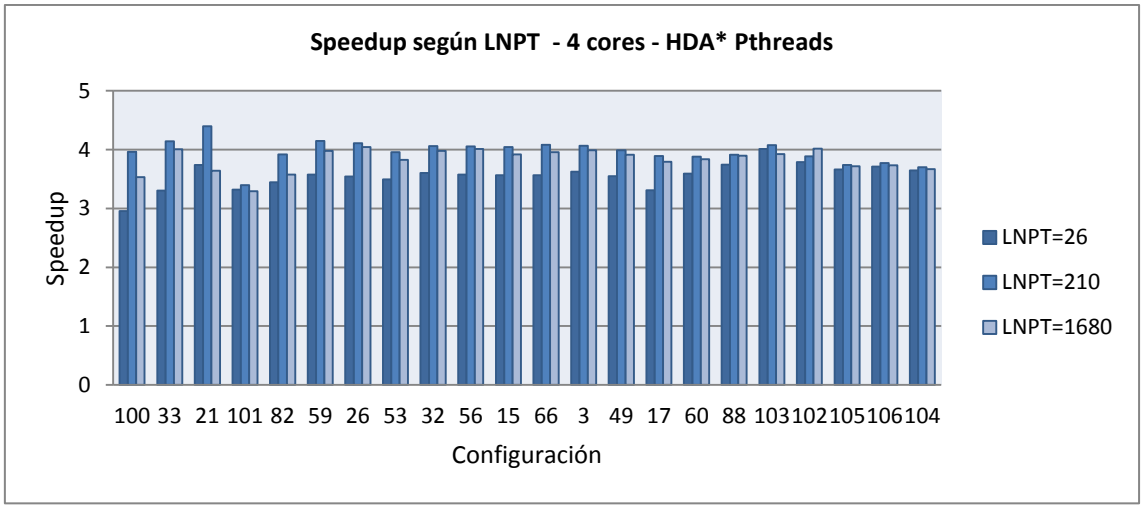


Figura 7.1. Speedup obtenido por HDA* Pthreads según LNPT, para cada configuración y 4 cores.

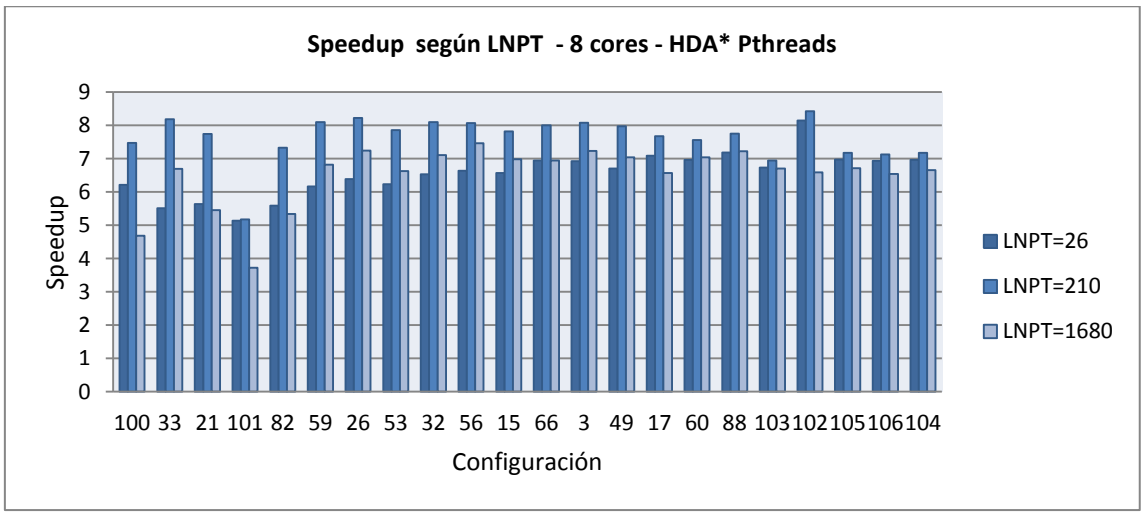


Figura 7.2. Speedup obtenido por HDA* Pthreads según LNPT, para cada configuración y 8 cores.

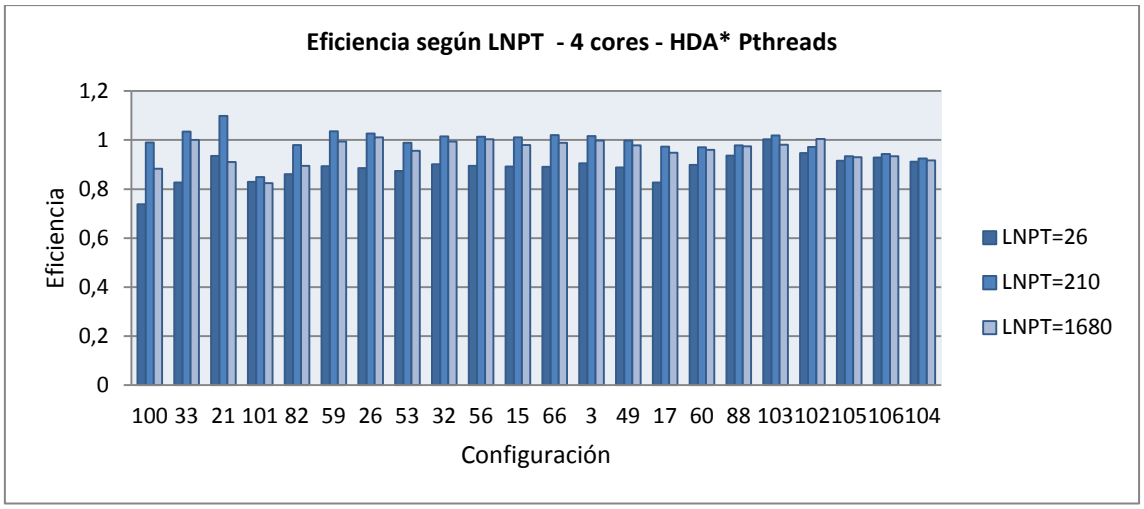


Figura 7.3. Eficiencia obtenida por HDA* Pthreads según LNPT, para cada configuración y 4 cores.

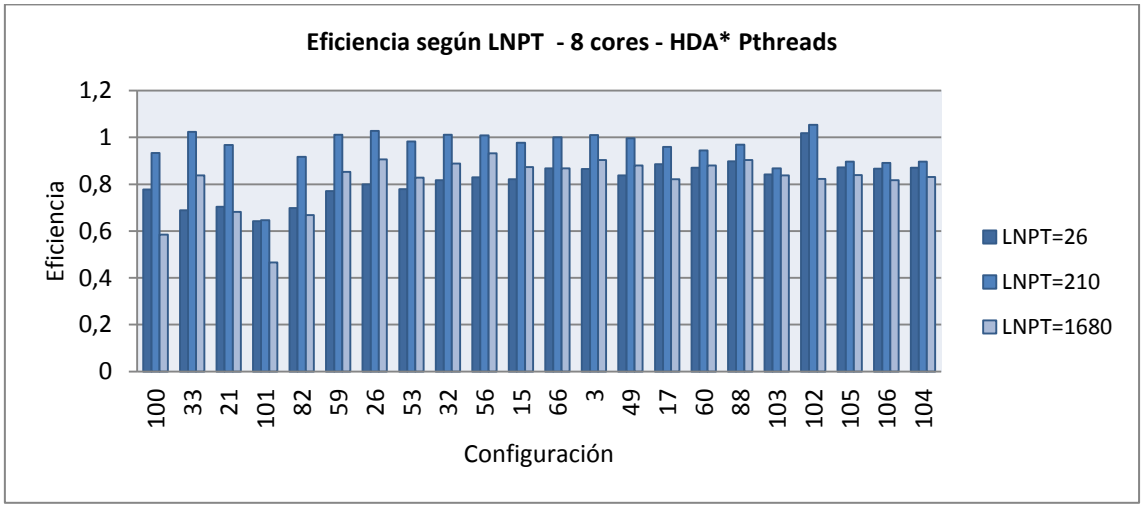


Figura 7.4. Eficiencia obtenida por HDA* Pthreads según LNPT, para cada configuración y 8 cores.

La Tabla 7.2 resume los rangos en los que se encuentran el Speedup y la Eficiencia, según el valor LNPT y la cantidad de cores, tomando las *muestras promedio* con LNPI óptimo para cada LNPT.

Tabla 7.2. Speedup y Eficiencia por LNPT y Cores

LNPT	Cores	Speedup	Eficiencia
26	**4**	2.95 - 4.01	0.74 - 1.003
210	**4**	3.39 - 4.39	0.85 - 1.099
1680	**4**	3.29 - 4.04	0.82 - 1.01
26	**8**	5.14 - 8.15	0.64 - 1.018
210	**8**	5.17 - 8.43	0.65 - 1.053
1680	**8**	3.72 - 7.46	0.46 - 0.933

7.2.4 Desvío Estándar del Tiempo de Ejecución

Con el objetivo de conocer la variación de los tiempos de ejecución de una muestra con respecto a la *muestra promedio* teniendo en cuenta una configuración inicial, cantidad de procesadores y parámetros fijos, se calculó la Desviación Estándar *DE* y el Coeficiente de Variación *CV,* que permitirá interpretar mejor la *DE*.

Tomando *todas* las pruebas ejecutadas con LNPT=26, *espera activa* y la técnica *Memory Pool*, se comprobó que el *CV* se encuentra entre 0.0096 y 0.1325, es decir el tiempo de ejecución de una muestra difiere de su *muestra promedio* en un máximo de 13%. Asimismo, el 92.6% de las pruebas realizadas tienen un CV por debajo del 10% por lo que se puede decir que el no determinismo y asincronismo presente en el algoritmo paralelo no provocan una variación significativa en el tiempo de ejecución.

La Tabla 7.3 muestra el rango en el que varía la *DE* y el *CV* para las pruebas ejecutadas con distintos valores de LNPT, *espera activa* y la técnica *Memory Pool*. De lo observado se puede concluir que el CV disminuye al incrementar LNPT, por lo que el tiempo de resolución de una *muestra* varía en menor medida respecto a la *muestra promedio*.

Tabla 7.3. Desvío Estándar (DE) y Coeficiente de Variación (CV) según el valor del parámetro LNPT

LNPT	Rango DE en segundos	Rango CV
26	0.0156 - 4.37	0.0096 - 0.1325
210	0.0058 - 3.13	0.0048 - 0.1278
1680	0.0214 - 2.78	0.0053 - 0.0636

7.2.5 Análisis de rendimiento

Para evaluar el rendimiento del algoritmo se tomaron en cuenta aquellas pruebas experimentales de las secciones anteriores que optimizaron los resultados. Las pruebas de interés son aquellas que hacen uso de *espera activa*, incorporan la técnica *Memory Pool y* limitan *LNPT* a 210. Se seleccionó para cada configuración y cantidad de hilos la *muestra promedio* que minimiza el tiempo de resolución, es decir aquella cuyo valor de parámetro *LNPI* optimiza el rendimiento.

Para evaluar la escalabilidad del algoritmo se ordenaron las distintas *muestras promedio seleccionadas* para cada configuración según la carga secuencial de trabajo de ésta (tiempo secuencial). En este sentido escalar el problema significa aumentar la cantidad de nodos procesados o generados, por otro lado la arquitectura escala aumentando la cantidad de cores usados para resolver el problema.

La Figura 7.5 muestra el Speedup obtenido por la *muestra promedio seleccionada* para cada configuración utilizando 4 cores y 8 cores, mientras que la Figura 7.6 muestra la Eficiencia obtenida. Para las pruebas con 4 cores el Speedup obtenido varía entre 3.39 y 4.39, mientras que la Eficiencia está en el rango 0.85 y 1.099. Las pruebas con 8 cores exhiben un Speedup entre 5.17 y 8.43, y Eficiencia entre 0.65 y 1.05. Algunos casos de superlinealidad se deben a la

obtención de *Overhead de Búsqueda* negativo (las muestras corresponden a las configuraciones 21 y 103 con 4 cores, y a la configuración 102 y 8 cores). En los demás casos la superlinealidad fue ocasionada por la disminución de la cantidad de elementos contenidos por las estructuras que manejan los threads (Lista Abierta y Lista Cerrada) respecto al algoritmo secuencial, que contribuye a la aceleración en las operaciones realizadas sobre éstas.

Analizando los resultados de las Figuras 7.5 y 7.6 se concluye que si se mantiene la carga de trabajo (configuración inicial) y se aumenta el número de cores, el Speedup obtenido es mejor lo que indica que el problema se resuelve en menor tiempo. Sin embargo, la Eficiencia puede disminuir debido a factores tales como: partes secuenciales en especial al inicio y fin del cómputo, sincronización, tiempo ocioso, desbalance de carga, aumento del overhead de búsqueda, entre otros.

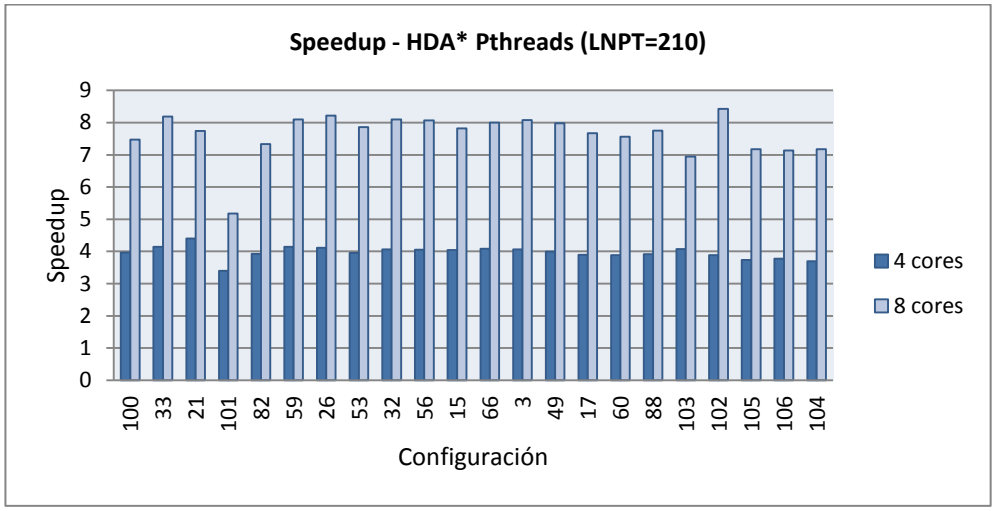


Figura 7.5. Speedup por configuración para el algoritmo HDA* Pthreads, con parámetros que optimizan el rendimiento

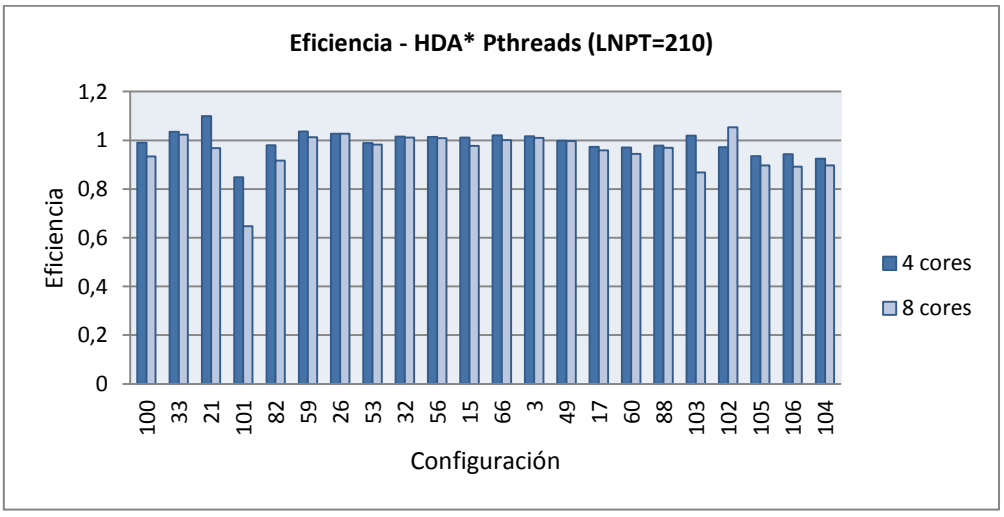


Figura 7.6. Eficiencia por configuración para el algoritmo HDA* Pthreads, con parámetros que optimizan el rendimiento

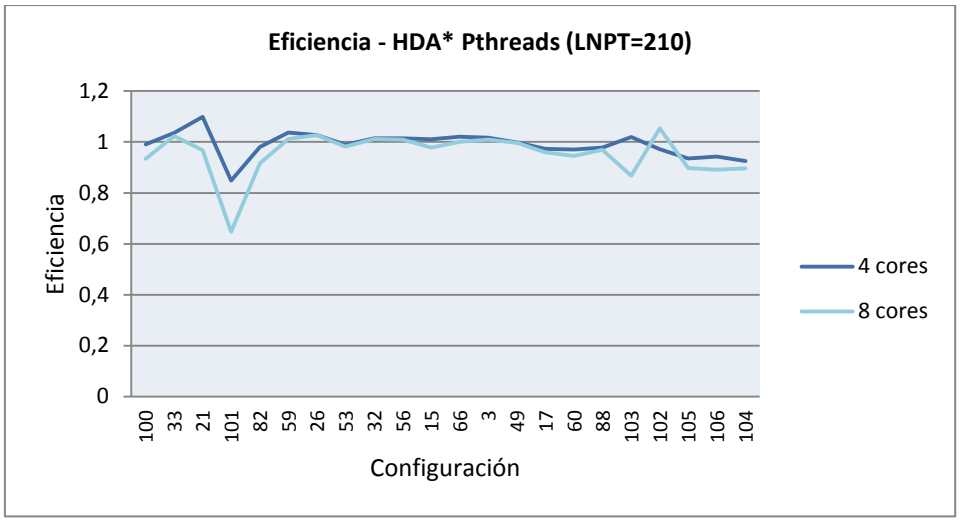


Figura 7.7 Eficiencia del algoritmo HDA* Pthreads al escalar el trabajo y los cores, con parámetros que optimizan el rendimiento

La Figura 7.7 grafica la Eficiencia a medida que aumenta la carga de trabajo para 4 cores y 8 cores. En la misma se observa que cuando escala el problema manteniendo fijo el número de *cores* usados en general la Eficiencia mejora o se mantiene constante dado que el *overhead* es menos significativo en el tiempo total de procesamiento.

De los resultados presentados se concluye que el comportamiento expuesto por el algoritmo es propio de un sistema paralelo escalable, donde la eficiencia puede mantenerse en un valor constante cuando se escala la carga de trabajo y la arquitectura.

7.2.5.1 Factores de overhead

En la Sección 7.2.5 se nombraron algunos factores que influyen en el rendimiento obtenido por un sistema paralelo, en especial para las aplicaciones de búsqueda paralela sobre espacios de estados. En esta sección se analizará el Overhead de Búsqueda y el Desbalance de Carga de las pruebas mostradas con anterioridad.

El *Overhead de Búsqueda (OB)* representa el incremento/reducción porcentual en la cantidad de nodos procesados por el algoritmo paralelo respecto al algoritmo secuencial, y se calcula con la Fórmula 5.3 (Capítulo 5). Cuando el *Overhead de Búsqueda* es positivo significa que el algoritmo paralelo procesó mayor cantidad de nodos que el algoritmo secuencial, en cambio un valor negativo revela que el algoritmo paralelo procesó menor cantidad de nodos.

El Overhead de Búsqueda positivo puede estar relacionado a distintas causas:

- *Incremento en la cantidad de nodos procesados con* $\hat{f} > C^*$ [39] debido a la perdida de la guía heurística. Por utilizar una *estrategia distribuida* de Listas Abiertas, un thread no tiene conocimiento de cuál es el costo del mejor nodo globalmente, por lo que expandirá sus mejores nodos locales en forma *especulativa* en un principio y recuperarán parte del conocimiento global cuando que se encuentren soluciones parciales, que permitirán

[39] C* representa el costo de la solución óptima

realizar poda de nodos con costo superior a ella; mientras tanto el balance de la calidad de los nodos estará dado por la técnica de balance de carga la cual se sustenta sobre la base de una comunicación asincrónica, es decir el tiempo de arribo de los nodos comunicados es no determinístico.

- *Incremento en el procesamiento de nodos con* $\hat{f} < C^*$, que puede ocurrir debido a una mayor reapertura de nodos y su subsecuente procesamiento. Aún utilizando una heurística consistente puede ocurrir en el algoritmo paralelo que se reabran nodos, lo cual no sucede en el algoritmo secuencial.
- *Incremento en el procesamiento de nodos con* $\hat{f} = C^*$.

Por otro lado, un Overhead de Búsqueda negativo es causado por la reducción de la cantidad de nodos procesados con $\hat{f} = C^*$ respecto al algoritmo secuencial.

La Figura 7.8 grafica el Overhead de Búsqueda por configuración para 4 y 8 cores que, en general, se mantiene cercano al 5% lo cual contribuye a alcanzar un rendimiento casi óptimo. Las muestras correspondientes a las configuraciones 21 y 103 con 4 cores, y a la configuración 102 y 8 cores registran un Overhead de Búsqueda negativo, estableciendo una causa por la cual se obtiene un rendimiento por encima del teóricamente posible. También se puede concluir que el bajo rendimiento obtenido por la configuración 101 se debe en parte al alto Overhead de Búsqueda. Lo mismo sucede con las configuraciones 100 y 82 con 8 cores, que alcanzaron menor eficiencia por obtener un Overhead de Búsqueda moderadamente elevado que ronda el 10%.

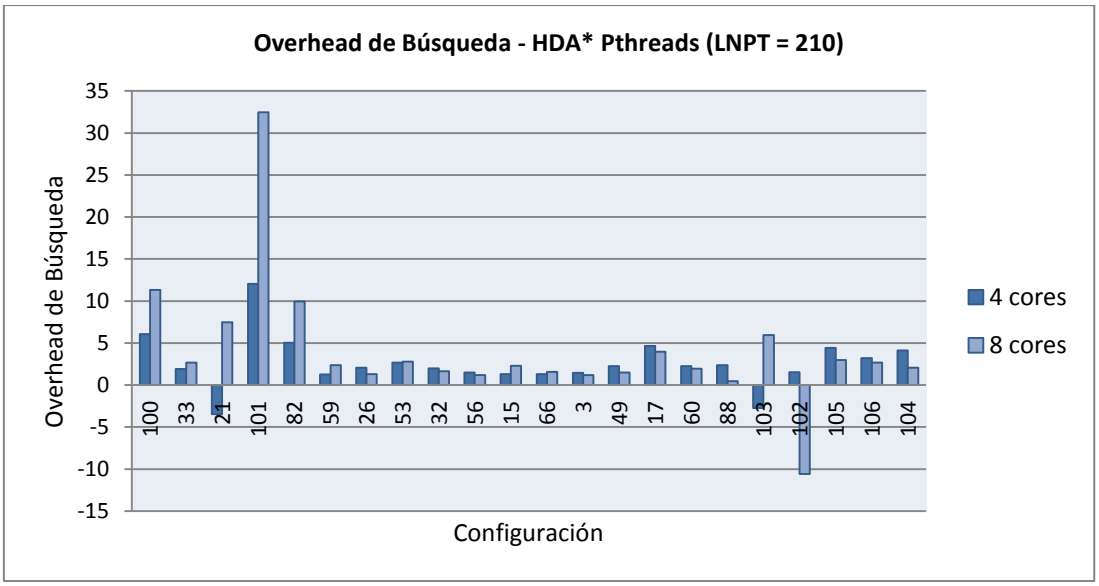


Figura 7.8. Overhead de búsqueda por configuración para el algoritmo HDA* para memoria compartida, LNPT =210

Continuando con el análisis, el Desbalance de Carga se calculó con la Fórmula 5.4 (Capítulo 5). La Figura 7.9 grafica el desbalance obtenido por la *muestra promedio seleccionada* de cada configuración, para 4 y 8 cores. De la gráfica se observa que la mayoría de las muestras obtienen un grado de desbalance que ronda el 0.05 (cercano al balance perfecto). Además, el alto grado de desbalance obtenido para la configuración 101 es otra causante del bajo rendimiento alcanzado. Las configuraciones más complejas obtuvieron un desbalance que va

entre 0.05 y 0.1, lo cual causa la obtención de menor eficiencia en comparación al resto de las muestras promedio.

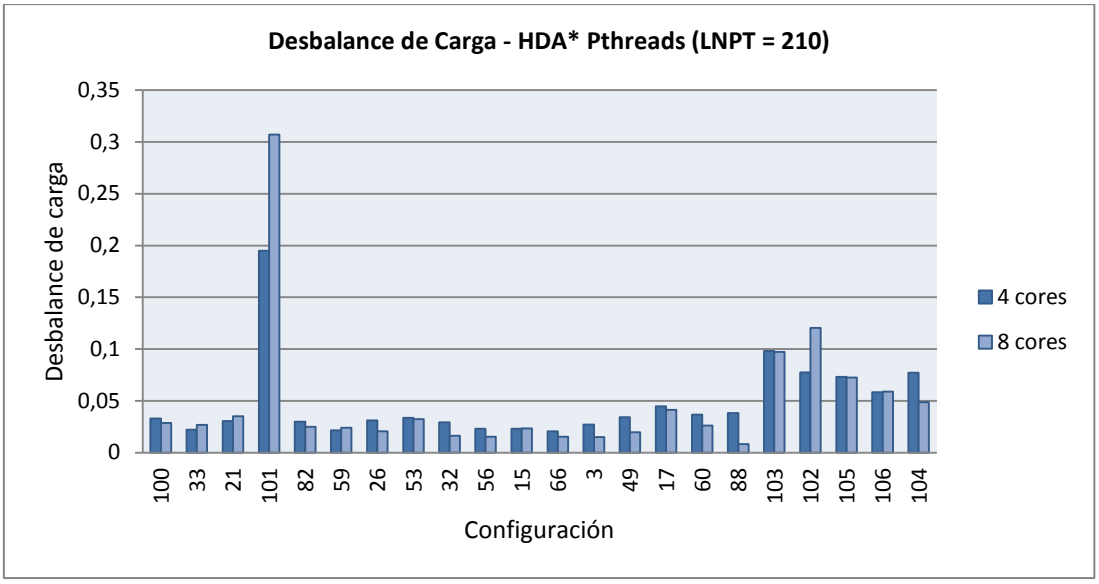


Figura 7.9. Desbalance de carga por configuración para el algoritmo HDA* para memoria compartida, LNPT =210.

7.3 HDA* para memoria distribuida (HDA* MPI)

Al igual que el algoritmo analizado en la Sección 7.2, el algoritmo paralelo HDA* para memoria distribuida (HDA* MPI) es no determinista.

Las pruebas experimentales se llevaron a cabo utilizando las 7 máquinas de la computadora paralela mencionada al principio del capítulo, disponiendo de 8 *cores* físicos[40] por máquina y 56 cores físicos en total entre todas las máquinas. Asimismo, se ejecutaron pruebas sobre una única máquina del cluster lo que permitirá evaluar el beneficio de adaptar la aplicación hacia un modelo híbrido.

En las distintas ejecuciones se utilizó afinidad para asociar cada proceso a un *core* exclusivo, empleando las opciones de *binding* que provee OpenMPI.

Las variables a evaluar en cada ejecución son: tiempo de ejecución en segundos, cantidad de nodos procesados o expandidos entre todos los procesos, cantidad de nodos generados entre todos los procesos, cantidad de nodos podados entre todos los procesos, cantidad de promociones de nodos en la Lista Abierta entre todos los procesos, cantidad de reapertura de nodos cerrados entre todos los procesos, cantidad total de duplicados encontrados durante la búsqueda entre todos los procesos, y la cantidad mínima y máxima de nodos procesados por un proceso.

[40] Se desactivaron las características *Hyperthreading* y *Turbo Boost* para evitar variaciones entre las ejecuciones causadas por la arquitectura y no por eventos asincrónicos del algoritmo.

Las configuraciones iniciales seleccionadas son aquellas utilizadas en la Sección 7.1 cuya ejecución secuencial es al menos 5 segundos[41]. También se tuvieron en cuenta 6 configuraciones que tienen mayor cantidad de pasos para su solución (numeradas del 101 a 106) (Brüngger, 1998).

Se utilizó el gestor de memoria dinámica Jemalloc configurado con 256 *arenas* y la heurística H4; este algoritmo *no* requiere utilizar la técnica *Memory Pool* ya que cada proceso accede únicamente a sus *arenas* asignadas, a causa de que nunca se transmiten entre sí punteros a nodo (esto es porque cada proceso puede acceder exclusivamente a su espacio de direcciones, restricción impuesta por el paradigma de paso de mensajes utilizado).

Para cada configuración inicial y cada grupo de parámetros se obtuvieron 10/100[42] muestras. Los parámetros utilizados son: la cantidad de procesos/procesadores; el límite de nodos a procesar por cada iteración del algoritmo *LNPI* (*Límite de Nodos Por Iteración*) cuyos valores serán 1, 5, 50 o 500; y la cantidad de nodos a empaquetar por mensaje *LNPT (Límite de Nodos Por Transferencia)*, cuyos valores se fijaron en 26 nodos (1KB datos), 210 nodos (8KB datos) o 1680 nodos (64KB datos). Luego se promediaron los datos arrojados de las ejecuciones realizadas para cada configuración y conjunto de parámetros, a lo que se llamará *muestra promedio*.

En las siguientes secciones se analizará el rendimiento obtenido por el algoritmo cuando se ejecuta sobre una única máquina multicore y el rendimiento obtenido por el algoritmo cuando se ejecuta sobre un cluster de multicore. Para ambas arquitecturas se evaluará el efecto de los parámetros LNPI y LNPT sobre el rendimiento.

7.3.1 Comportamiento sobre multicore

Esta sección analiza el comportamiento del algoritmo HDA* MPI sobre una máquina multicore del cluster, disponiendo en total de 8 cores físicos. En las pruebas con 4 procesos, se ubicó un par de procesos en cada procesador de la máquina. En las pruebas con 8 procesos, se ubicó un proceso por cada *core* de la máquina.

7.3.1.1 Impacto de los parámetros LNPI y LNPT sobre el rendimiento

En esta sección se analiza el impacto de los parámetros *LNPI* y *LNPT* sobre el rendimiento del algoritmo HDA* MPI cuando se ejecuta sobre una máquina multicore.

[41] Las configuraciones son las siguientes: 3, 15, 17, 21, 26, 32, 33, 49, 53, 56, 59, 60, 66, 82, 88, 100

[42] Para las pruebas de HDA* MPI sobre una máquina multicore, la cantidad de muestras por configuración y parámetros es 100. Para la evaluación de HDA* MPI sobre el cluster la cantidad de muestras tomadas por configuración y parámetros es 10, ya que el cluster es un recurso compartido entre varios usuarios, se requería exclusividad en el uso para la ejecución de las pruebas y se disponía de un tiempo limitado.

7.3.1.1.1 Límite de Nodos Procesados Por Iteración (LNPI)

El parámetro *LNPI* determina cuántos nodos debe expandir un proceso en cada iteración del algoritmo, es decir establece el intervalo de verificación de llegada de mensajes con trabajo y mensajes que transportan costos de mejores soluciones encontradas.

Con el objetivo de analizar el impacto del parámetro sobre el tiempo de ejecución, se tomaron *todas* las *muestras promedio* que surgen de las ejecuciones realizadas para LNPT=26, y se agruparon aquellas con igual configuración y cantidad de cores, es decir cada grupo contiene las *muestras promedio* que difieren únicamente en el parámetro *LNPI*.

Para cada *grupo* se calculó la Desviación Estándar (*DE*) y el Coeficiente de Variación (*CV*) del tiempo de ejecución, lo que proporciona una medida de cuánto difiere el tiempo de ejecución de una *muestra promedio* del grupo respecto al *tiempo promedio grupal*.

En general, los resultados arrojados revelan que la *DE* de los grupos se encuentra entre 0.19 y 9.92; esto indica que variar el parámetro *LNPI* entre los valores definidos provoca que los tiempos de ejecución de las *muestras promedio* se desvíen en a lo sumo 9.92 segundos de su *promedio grupal*. Sólo 61.36% de los grupos tiene una *DE* menor a 1 y el 25% por debajo de 0.5.

Por otro lado, los valores generales de *CV* obtenidos para los grupos oscilan entre 0.017 y 0.66, es decir el tiempo de ejecución de una *muestra promedio* del grupo se desvía entre un 1.7% y un 66% de la *media grupal* al variar el parámetro *LNPI*. Sólo el 50% de los grupos tienen CV por debajo de un 0.1, y el 27.27% exhibe valores de CV por debajo de 0.05.

Los *grupos* asociados a ejecuciones con 4 cores poseen *DE* entre 0.19 y 9.92. Por su parte, el *CV* para estos grupos varía entre 0.019 y 0.54, teniendo el 32% de los mismos un CV por debajo de 0.05. Por otra parte, los grupos asociados a ejecuciones con 8 cores poseen *DE* entre 0.22 y 2.66; mientras que el CV para ellos oscila entre 0.017 y 0.66, teniendo el 23% de los grupos un CV por debajo de 0.05.

En general, la variación de los tiempos al modificar el parámetro LNPI es menos notoria para las configuraciones de mayor complejidad.

De los resultados obtenidos se concluye que existe una variación significativa en los tiempos de ejecución al variar el parámetro *LNPI* entre los valores definidos (1, 5, 50 o 500).

La Tabla 7.4 muestra los rangos de DE y CV de los grupos para distintos valores de LNPT. De los mismos se observa que a valores de LNPT más chicos, los tiempos de las *muestras promedio* difieren más al variar el parámetro LNPI. Dicho en otras palabras, cuando los mensajes de trabajo enviados contienen pocos nodos, el valor del parámetro LNPI influye más sobre el tiempo de ejecución.

Tabla 7.4. Rangos de DE y CV clasificados por LNPT

LNPT	Rango DE en segundos	Rango CV
26	0.19-9.92	0.017-0.66
210	0.13-6.2	0.03-0.5
1680	0.06-7.7	0.02-0.065

El valor de LNPI que optimiza el rendimiento depende de la configuración inicial, la cantidad de procesos/cores y el valor del parámetro LNPT (es decir, el tamaño del mensaje de trabajo).

Los resultados indican que para LNPT=26 y 4 cores los valores de LNPI óptimos pueden ser 5, 50 o 500; en particular 7 configuraciones obtuvieron rendimiento óptimo con LNPI=5, 11 configuraciones con LNPI = 50 y 4 configuraciones con LNPI=500 (entre las que se encuentran tres de las más complejas). En cambio para LNPT=26 y 8 cores, en general el valor óptimo de LNPI es 5 (sólo la configuración 104 prefirió LNPI=500, siendo ésta la configuración la más compleja).

En general, para LNPT=210 el valor óptimo de LNPI es 50, tanto para 4 y 8 cores (sólo la configuración 88 ejecutada con 4 cores obtuvo rendimiento óptimo con LNPI=500).

Por último, para LNPT=1680 y 4 cores el valor óptimo de LNPI es 500 (exceptuando dos configuraciones de las más simples que alcanzaron el óptimo con LNPI=50). En cambio para LNPT = 1680 y 8 cores, 13 configuraciones alcanzaron el tiempo óptimo con LNPI=50 y 9 configuraciones con LNPI=500 (estas últimas están en el grupo de las 10 configuraciones más complejas utilizadas).

Es importante destacar que en ningún caso LNPI = 1 optimizó el rendimiento para una configuración inicial, cantidad de procesos y LNPT particulares. Esto indica que el parámetro que se incorporó en la versión propia de HDA* para memoria distribuida es de importancia, siendo un aporte original en el área.

De lo anterior se concluye lo siguiente:

- Cuando LNPT es chico, los procesos enviarán muchos mensajes de trabajo conteniendo pocos nodos, por ello debe aumentarse la frecuencia de recepción de mensajes (valor bajo de LNPI) para permitir incorporar nuevos nodos que están entre los mejores globales. En caso de elegir un valor de LNPI muy alto, los procesos estarían realizando excesivo trabajo especulativo sobre nodos locales en cada etapa de procesamiento, sin incorporar nuevos nodos, lo cual aumenta el Overhead de Búsqueda[43], empeorando aún más al incrementar la cantidad de procesos.
- Cuando LNPT es grande, los mensajes contendrán muchos nodos y serán más bien escasos, con lo cual al disminuir la frecuencia de recepción (valor de LNPI alto) no se hacen continuas verificaciones por llegada de mensajes que resulten fallidas. Cuando se elige en este caso un valor de LNPI bajo, no afecta tanto al rendimiento por utilizar

[43] Las excepciones ocurren con algunas configuraciones complejas, las cuales tienen gran cantidad de nodos con igual costo. Esta característica provoca que no influya el hecho de que el proceso no incorpore nodos frecuentemente y realice en cambio trabajo especulativo.

comprobaciones asincrónicas de llegada de mensajes. No se notan variaciones significativas del Overhead de Búsqueda al variar LNPI ni tampoco en los tiempos de ejecución.

En comparación con HDA* Pthreads, el parámetro LNPI toma mayor importancia ya que varía la frecuencia en la que el proceso verifica arribos de mensajes con costo de mejor solución.

7.3.1.1.2 Límite de Nodos Por Transferencia (LNPT)

El algoritmo HDA* MPI presentado incorpora el parámetro *Limite de Nodos Por Transferencia (LNPT*) que indica la cantidad de nodos que contendrá como máximo cada mensaje de trabajo.

Con el objetivo de analizar el efecto del parámetro *LNPT* sobre el tiempo de ejecución, se tomaron todas las *muestras promedio* que surgen de las ejecuciones limitando *LNPT* a 26 nodos (mensajes de 1KB), 210 nodos (mensajes de 8KB) y 1680 nodos (mensajes de 64KB). A continuación, se agruparon las *muestras promedio* con igual configuración, cantidad de cores y *LNPI*, es decir cada grupo contiene las *muestras promedio* que difieren únicamente en el parámetro *LNPT*. Para cada *grupo* se calculó la Desviación Estándar (*DE*) y el Coeficiente de Variación (*CV*) del tiempo de ejecución, lo que proporciona una medida de cuánto difiere el tiempo de ejecución de una *muestra promedio* del grupo respecto al *tiempo promedio grupal*.

En general, los resultados arrojados revelan que la *DE* de los grupos se encuentra entre 0.018 y 11.10, esto indica que variar el parámetro *LNPT* entre los valores definidos provoca que los tiempos de ejecución de las *muestras promedio* se desvíen en a lo sumo 11.10 segundos de su *promedio grupal*. Vale destacar que el 70.45% de los grupos tiene una *DE* menor a 1 y el 50.57% por debajo de 0.5.

Por otro lado, los valores generales de *CV* obtenidos para los grupos oscilan entre 0.01 y 0.76, es decir el tiempo de ejecución de una *muestra promedio* del grupo se desvía entre un 1% y un 76% de la *media grupal* al variar el parámetro *LNPT* entre los valores fijados. Asimismo, el 70.45% de los grupos tienen CV por debajo de un 0.1, el 47.16 % exhibe un CV por debajo de 0.05 y el 20.45% posee CV por debajo de 0.03.

Por los valores moderadamente altos de CV se concluye que el parámetro LNPT tiene influencia sobre el tiempo de ejecución. Esto se debe a que LNPT impacta en el congestionamiento del medio de comunicación, pero también tendrá influencia en la actividad de los procesos y el balance de calidad de los nodos:

- Un valor bajo de LNPT provocará que los procesos formen mensajes de tamaño chico, esto incrementa la cantidad de mensajes, los cuales serán enviados de un proceso a otro por MPI simulando el paso de mensajes por memoria compartida, y genera un *overhead* por la manipulación de buffers asociados a los mensajes.
- Valores muy altos de LNPT también pueden degradar el rendimiento dado que si un proceso empaqueta demasiados nodos para un proceso destino podría influir en la ociosidad de éste último cuando el mismo posee pocos nodos para procesar o cuando al momento se encuentra inactivo por falta de trabajo. También podría estar demorando la

transmisión de nodos de mejor calidad en comparación a aquellos que el destinatario está procesando actualmente, contribuyendo a un incremento del *Overhead de Búsqueda.*

En la mayoría de los casos el valor de LNPT que optimiza los tiempos de ejecución para cada configuración y cantidad de cores es 210, es decir mensajes de 8KB. Solamente para las configuraciones 56 y 60 con 4 cores se obtuvo rendimiento óptimo con LNPT=1680; y para las configuraciones 21 y 100 con 8 cores se alcanzó un tiempo de ejecución óptimo con LNPT = 26.

Las Figuras 7.10 y 7.11 muestran el Speedup según la configuración utilizando 4 cores y 8 cores respectivamente, para cada valor de LNPT, teniendo en cuenta que la *muestra promedio seleccionada* por configuración y cantidad de cores es aquella cuyo LNPI optimiza el rendimiento. De manera similar, las Figuras 7.12 y 7.13 muestran la Eficiencia según la configuración para 4 y 8 cores respectivamente, para cada valor de LNPT.

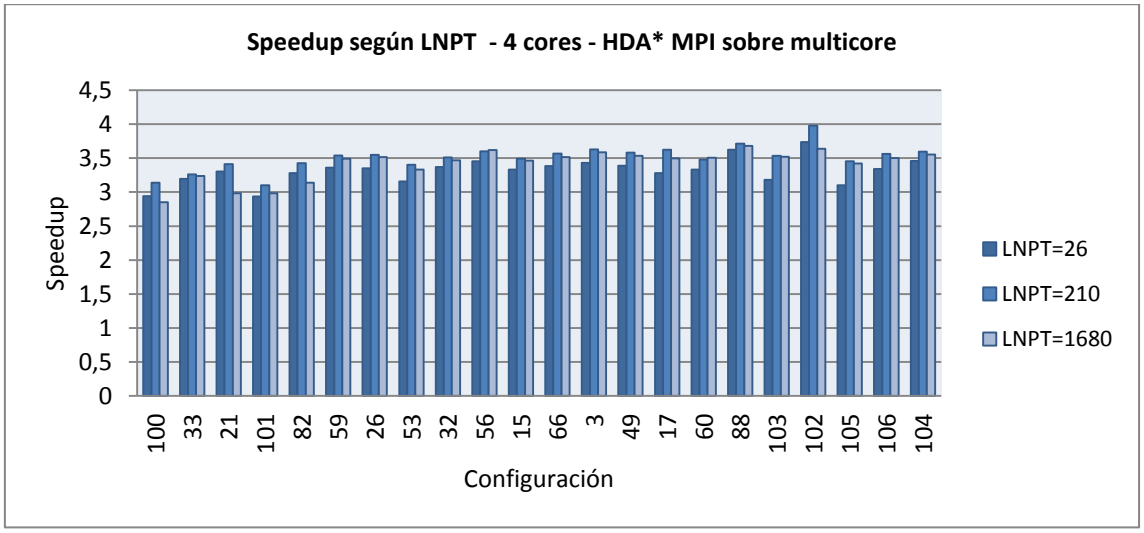


Figura 7.10. Speedup obtenido por HDA* MPI sobre Multicore según LNPT, con 4 cores.

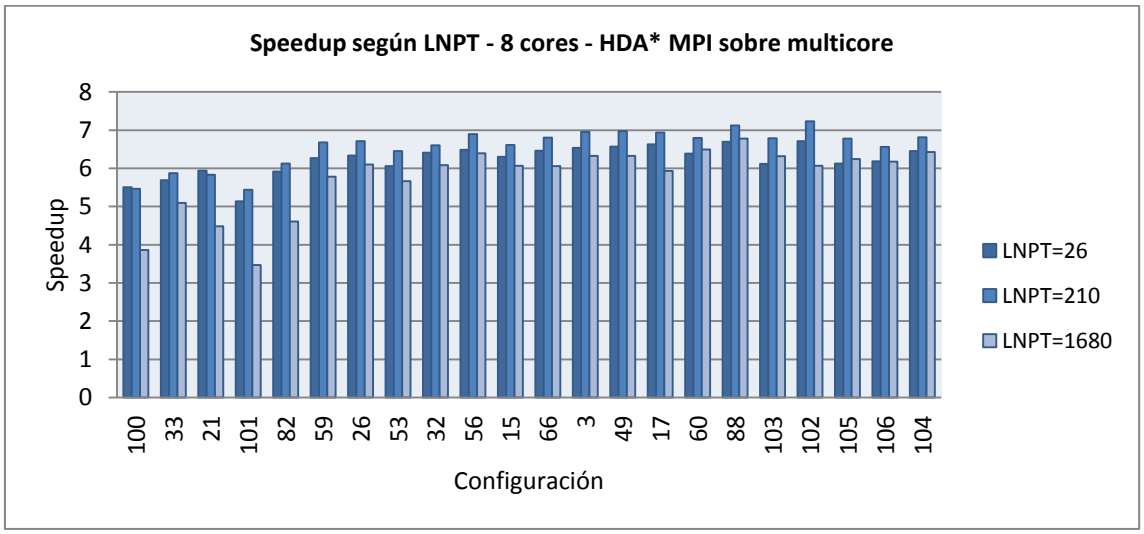


Figura 7.11. Speedup obtenido por HDA* MPI sobre Multicore según LNPT, con 8 cores.

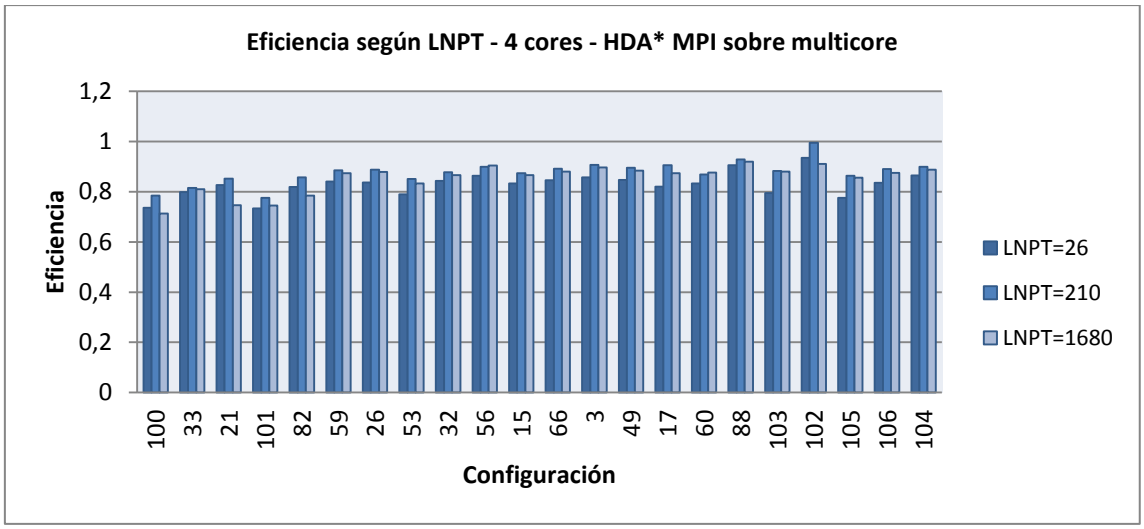


Figura 7.12. Eficiencia obtenida por HDA* MPI sobre Multicore según LNPT, con 4 cores.

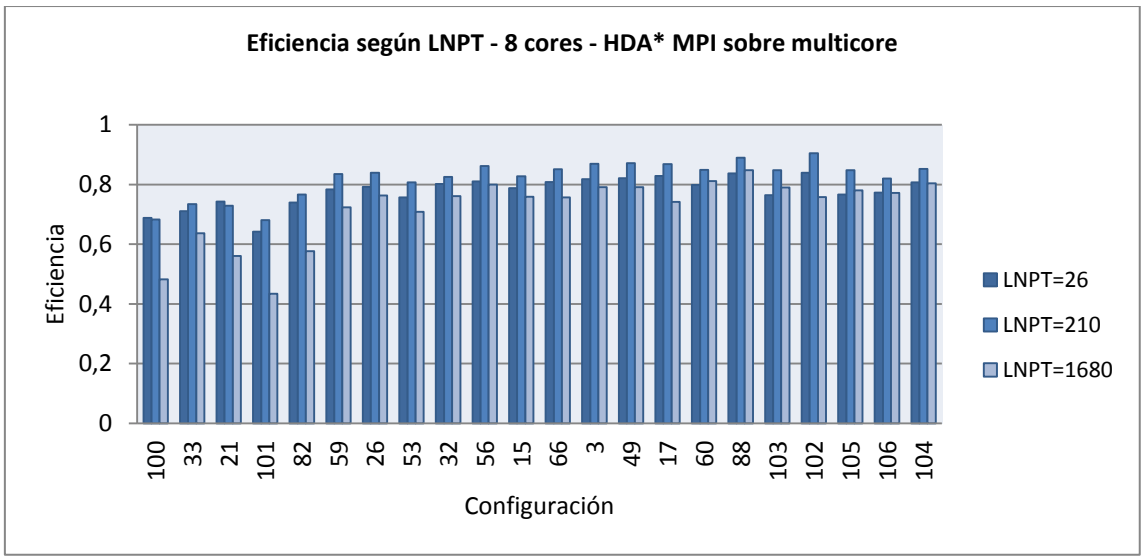


Figura 7.13. Eficiencia obtenida por HDA* MPI sobre Multicore según LNPT, con 8 cores.

La Tabla 7.5 resume los rangos en los que se encuentran el Speedup y la Eficiencia, según el valor LNPT y la cantidad de cores, tomando las *muestras promedio* con LNPI óptimo para cada LNPT.

Tabla 7.5. Speedup y Eficiencia por LNPT y Cores, para HDA* MPI sobre Multicore

LNPT	Cores	Speedup	Eficiencia
26	**4**	2.94 - 3.74	0.73 - 0.94
210	**4**	3.10 - 3.98	0.78 - 0.99
1680	**4**	2.85 - 3.68	0.71 - 0.92
26	**8**	5.14 - 6.71	0.64 - 0.84
210	**8**	5.44 - 7.23	0.68 - 0.90
1680	**8**	3.47 - 6.78	0.43 - 0.85

7.3.1.2 Desvío Estándar del Tiempo de Ejecución

Con el objetivo de conocer la variación de los tiempos de ejecución de una muestra con respecto a la *muestra promedio* teniendo en cuenta una configuración inicial, cantidad de procesadores y parámetros fijos, se calculó la Desviación Estándar *DE* y el Coeficiente de Variación *CV,* que permitirá interpretar mejor la *DE*.

Considerando *todas* las pruebas ejecutadas con LNPT = 26, se comprobó que CV se encuentra entre 0.0088 y 0.17, es decir el tiempo de ejecución de una muestra difiere de su *muestra promedio* en un máximo de 17%. Asimismo, el 93.18% de las pruebas realizadas tienen un CV por debajo del 10% por lo que se puede decir que el no determinismo y asincronismo presente en el algoritmo paralelo no provocan una variación significativa en el tiempo de ejecución.

La Tabla 7.6 muestra el rango en el que varía la *DE* y el *CV* para las pruebas ejecutadas con distintos valores de LNPT. De lo observado se puede concluir que el CV disminuye al incrementar LNPT, por lo que el tiempo de resolución de una *muestra* varía en menor medida respecto a la *muestra promedio*.

Tabla 7.6. Desvío estándar (DE) y Coeficiente de Variación (CV) según el valor del parámetro LNPT

LNPT	Rango DE en segundos	Rango CV
26	0.034 - 7.71	0.0088 - 0.17
210	0.039 - 6.26	0.015 - 0.14
1680	0.039 - 5.69	0.0084 - 0.12

7.3.1.3 Análisis de rendimiento

Para evaluar el rendimiento del algoritmo se tomaron en cuenta aquellas pruebas experimentales de las secciones anteriores que optimizaron los resultados. Las pruebas de interés son aquellas limitan *LNPT* a 210. Se seleccionó para cada configuración y cantidad de procesos la *muestra promedio* que minimiza el tiempo de resolución, es decir aquella cuyo valor de parámetro *LNPI* optimiza el rendimiento.

Para evaluar la escalabilidad del algoritmo se ordenaron las distintas *muestras promedio seleccionadas* para cada configuración según la carga secuencial de trabajo de ésta (tiempo secuencial). En este sentido escalar el problema significa aumentar la cantidad de nodos procesados o generados. Por otro lado, la arquitectura escala aumentando la cantidad de cores/procesos usados para resolver el problema.

La Figura 7.14 muestra el Speedup obtenido por la *muestra promedio seleccionada* para cada configuración utilizando 4 cores y 8 cores, mientras que la Figura 7.15 muestra la Eficiencia obtenida. Para las pruebas con 4 cores el Speedup obtenido varía entre 3.10 y 3.98, mientras que la Eficiencia está en el rango 0.78 y 0.99. Las pruebas con 8 cores exhiben un Speedup entre 5.44 y 7.23, y Eficiencia entre 0.68 y 0.90.

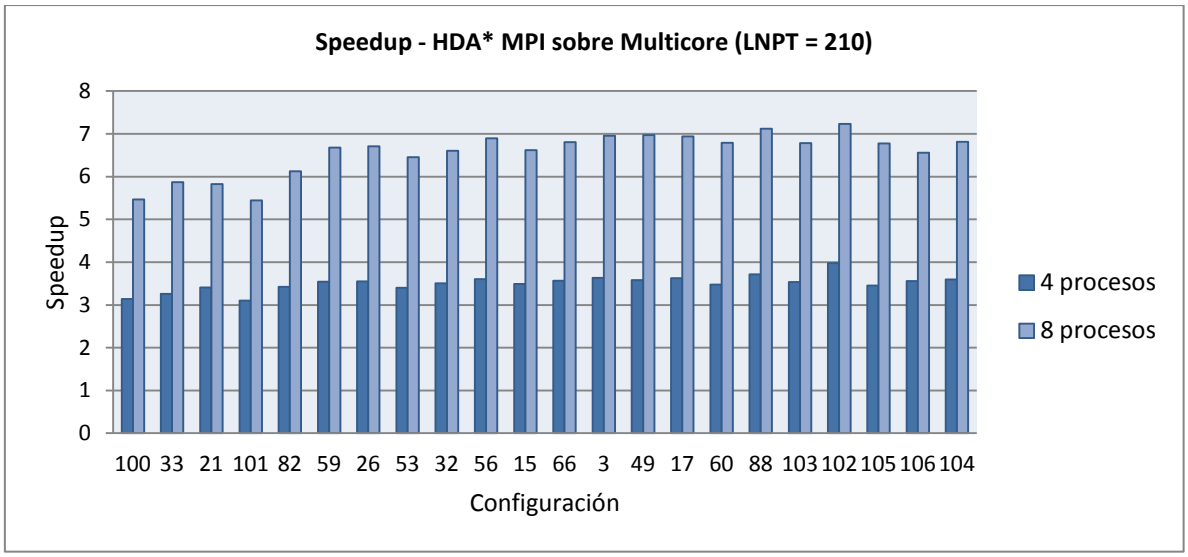


Figura 7.14. Speedup por configuración para el algoritmo HDA* MPI sobre Multicore, LNPT = 210

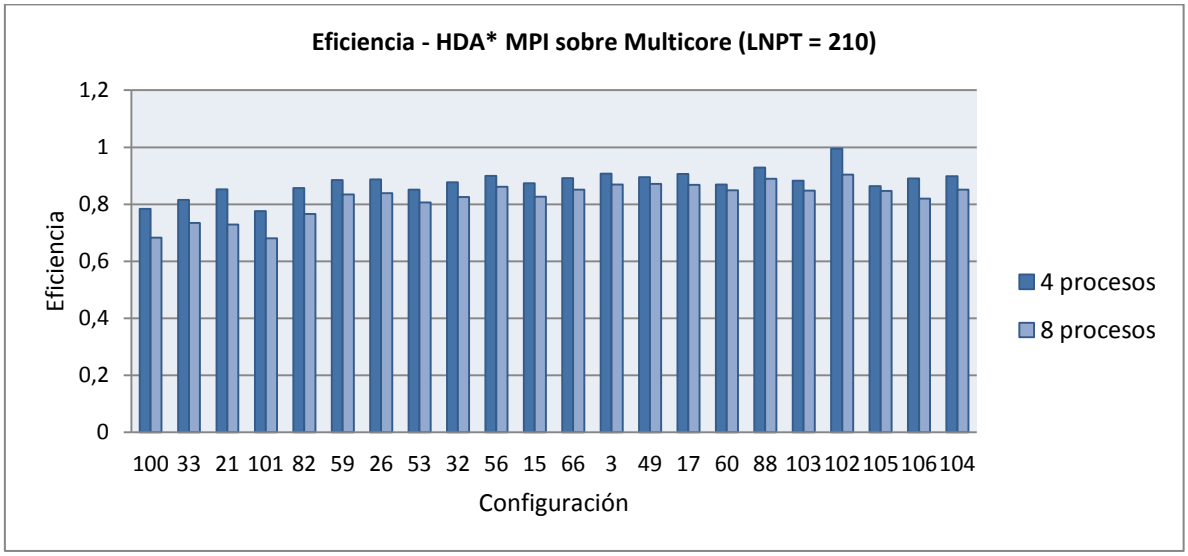


Figura 7.15. Eficiencia por configuración para el algoritmo HDA* MPI sobre Multicore, LNPT = 210

Analizando los resultados de las Figuras 7.14 y 7.15 se concluye que si se mantiene la carga de trabajo (configuración inicial) y se aumenta el número de cores, el Speedup obtenido es mejor lo que indica que el problema se resuelve en menor tiempo. Sin embargo, esta mejora no mantiene la Eficiencia en un valor constante. La desmejora en la eficiencia se debe a factores tales como: partes secuenciales en especial al inicio y fin del cómputo, sincronización, comunicación, tiempo ocioso, desbalance de carga, aumento del overhead de búsqueda, entre otros.

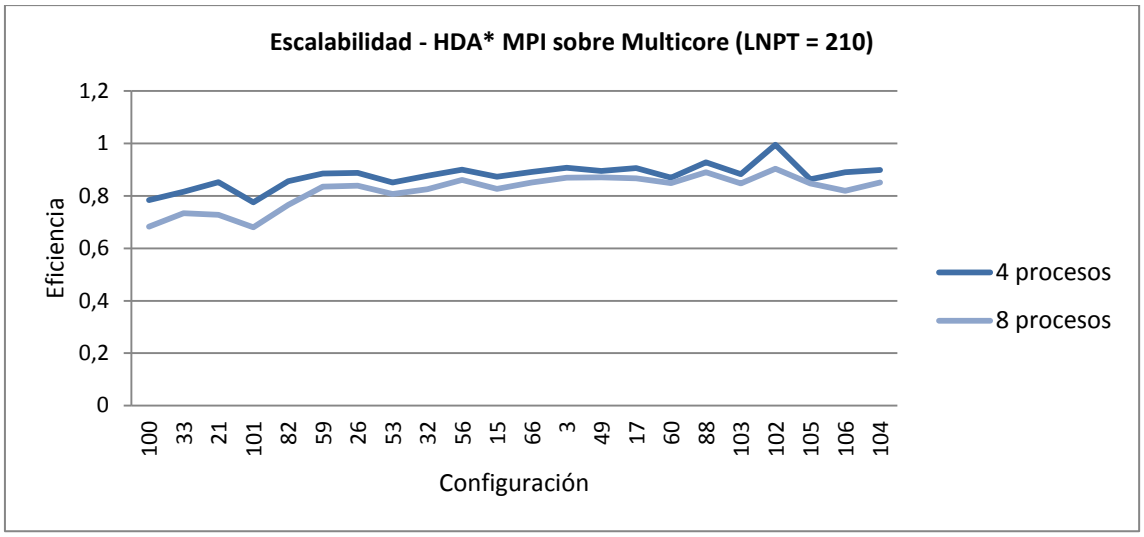


Figura 7.16. Eficiencia por configuración para el algoritmo HDA* MPI sobre Multicore, LNPT = 210

La Figura 7.16 grafica la Eficiencia a medida que aumenta la carga de trabajo para 4 cores y 8 cores. En la misma se observa que cuando escala el problema manteniendo fijo el número de *cores* usados en general la Eficiencia mejora o se mantiene constante dado que el *overhead* es menos significativo en el tiempo total de procesamiento.

De los resultados presentados se concluye que el comportamiento expuesto por el algoritmo es propio de un sistema paralelo escalable, donde la eficiencia puede mantenerse en un valor constante cuando se escala la carga de trabajo y la arquitectura.

7.3.1.3.1 Factores de Overhead

Se calculó el Overhead de Búsqueda y el Desbalance de Carga de las ejecuciones presentadas en el análisis de rendimiento, cuyas graficas se observan en las Figuras 7.17 y 7.18 respectivamente.

El Overhead de Búsqueda se encuentra en general cercano al 5%. Las excepciones ocurren en las configuraciones 100 y 101 con 4 cores; y las configuraciones 100, 21, 101 y 82 con 8 cores; situación similar a la observada en el algoritmo HDA* Pthreads.

El Desbalance de Carga en general se ubica cercano al 0.05; las configuraciones que superaron dicho valor obtienen un grado de desbalance que está por debajo de 0.09.

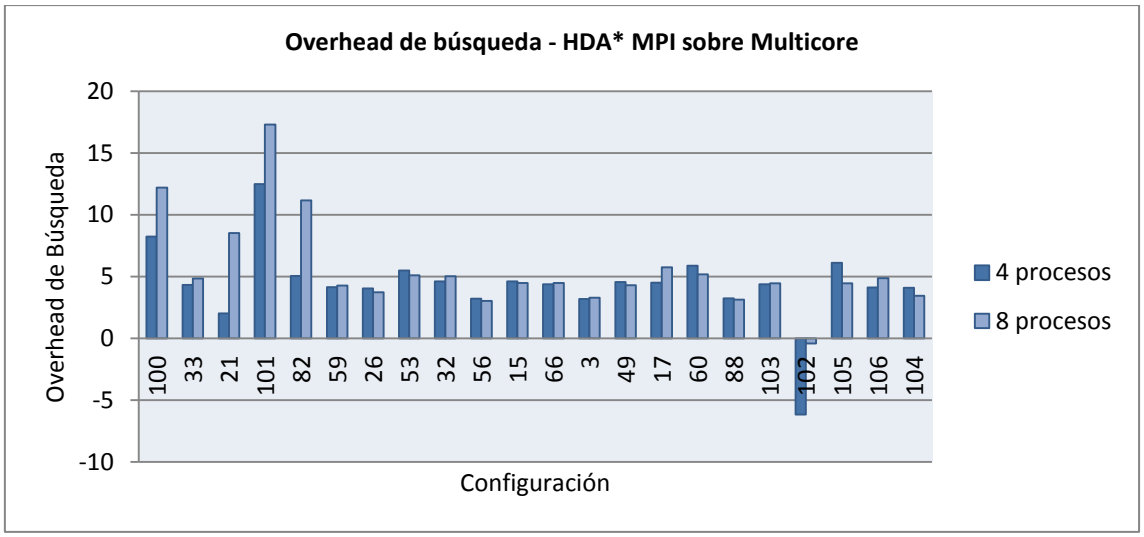


Figura 7.17. Overhead de Busqueda para HDA* MPI sobre multicore (LNPT=210)

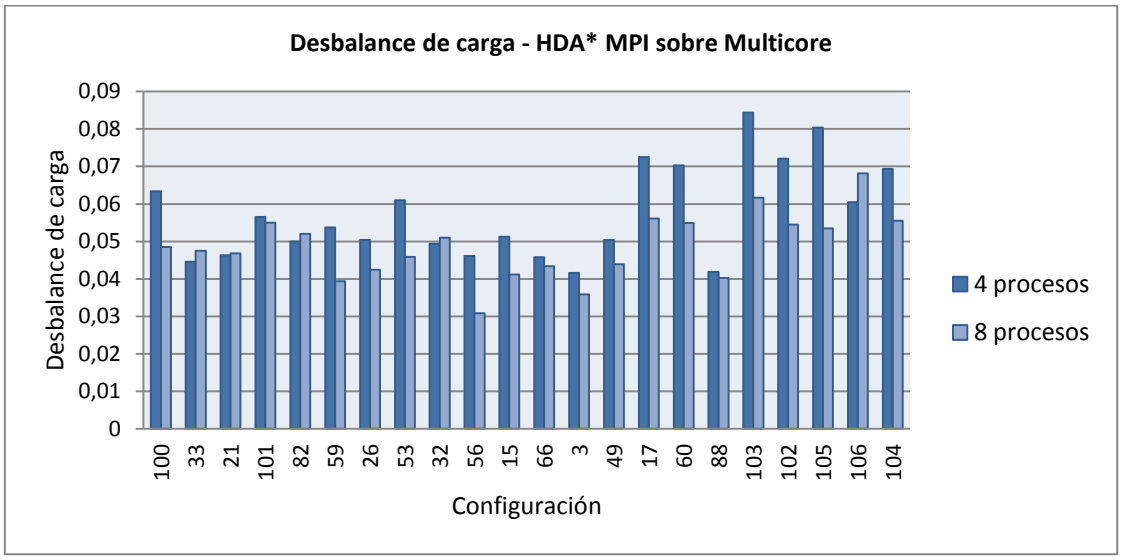


Figura 7.18. Desbalance de carga para HDA* MPI sobre multicore (LNPT=210)

7.3.2 Comportamiento sobre cluster de multicore.

Esta sección estudia el comportamiento del algoritmo HDA* MPI sobre el cluster de multicore compuesto por 7 máquinas, cada una de las cuales cuenta con 8 cores físicos. Las pruebas fueron ejecutadas utilizando distintas configuraciones de cluster (variando la cantidad de procesos a ejecutar por máquina entre 4 y 8, y la cantidad de máquinas entre 2 y 7), las configuraciones iniciales y parámetros mencionados con anterioridad (LNPI entre 1, 5, 50 y 500; LNPT entre 26, 210 y 1680). En las pruebas que ubican 4 procesos por máquina, se colocó un par de procesos en cada procesador de la misma, mientras que para las pruebas con 8 procesos por máquina se colocó un proceso por cada *core* de la misma.

La nomenclatura utilizada para referirse a las configuraciones de cluster indicará primero la cantidad de procesos por máquina y luego la cantidad de máquinas. Por ejemplo: 4px7m se refiere a que se ubican 4 procesos en cada una de las 7 máquinas, teniendo en total 28 procesos, mientras que 8px4m se refiere a que se ubican 8 procesos en cada una de las 4 máquinas, contando en este caso con 32 procesos.

Para cada configuración de cluster, configuración inicial y conjunto de parámetros se ejecutaron 10 pruebas, a partir de las cuales se obtuvo la denominada *muestra promedio.*

Como se verá en la Sección 7.3.2.3 las pruebas que colocan 8 procesos por máquina obtienen un rendimiento pobre.

En las siguientes secciones se analiza el impacto de los parámetros LNPT y LNPI sobre el rendimiento para las pruebas que ubican 4 procesos por máquina, el desvío de los tiempos de resolución entre distintas ejecuciones cuando se utilizan los mismos datos de entrada y configuración de cluster, y el rendimiento obtenido con los parámetros que optimizaron el rendimiento. Estas dos últimas secciones analizan las pruebas que ubican 4 y 8 procesos por máquina.

7.3.2.1 Impacto de los parámetros LNPI y LNPT sobre el rendimiento

En esta sección se analiza el impacto de los parámetros *LNPI* y *LNPT* sobre el rendimiento del algoritmo HDA* MPI cuando se ejecuta sobre un cluster de multicore, ubicando 4 procesos por máquina.

7.3.2.1.1 Límite de Nodos Procesados Por Iteración (LNPI)

El parámetro *LNPI* determina cuántos nodos debe expandir un proceso en cada iteración del algoritmo, es decir establece el intervalo de verificación de llegada de mensajes con trabajo y mensajes que transportan costos de mejores soluciones encontradas.

Con el objetivo de analizar el impacto del parámetro sobre el tiempo de ejecución, se tomaron *todas* las *muestras promedio* que surgen de las ejecuciones realizadas para LNPT=26, y se agruparon aquellas con igual configuración inicial y cantidad de cores, es decir cada grupo contiene las *muestras promedio* que difieren únicamente en el parámetro *LNPI*. Este procedimiento se realizó para las configuraciones de cluster que colocan 4 procesos por máquina.

Para cada *grupo* se calculó la Desviación Estándar (*DE*) y el Coeficiente de Variación (*CV*) del tiempo de ejecución, lo que proporciona una medida de cuánto difiere el tiempo de ejecución de una *muestra promedio* del grupo respecto al *tiempo promedio grupal*.

En general, los resultados arrojados revelan que la *DE* de los grupos se encuentra entre 0.063 y 18.94; esto indica que variar el parámetro *LNPI* entre los valores definidos provoca que los tiempos de ejecución de las *muestras promedio* se desvíen en a lo sumo 18.94 segundos de su *promedio grupal*. Sólo 60% de los grupos tiene una *DE* menor a 1 y el 39% por debajo de 0.5.

Por otro lado, los valores generales de *CV* obtenidos para los grupos oscilan entre 0.054 y 1.28, es decir el tiempo de ejecución de una *muestra promedio* del grupo se desvía entre un 5.4% y un 128% de la *media grupal* al variar el parámetro *LNPI*. Sólo el 11% de los grupos tienen CV por debajo de un 0.1.

En general, las configuraciones más simples son las que exhiben mayor variación de los tiempos al modificar el parámetro LNPI.

De los resultados obtenidos se concluye que existe una variación significativa en los tiempos de ejecución al cambiar el valor del parámetro *LNPI* entre los valores definidos (1, 5, 50 o 500).

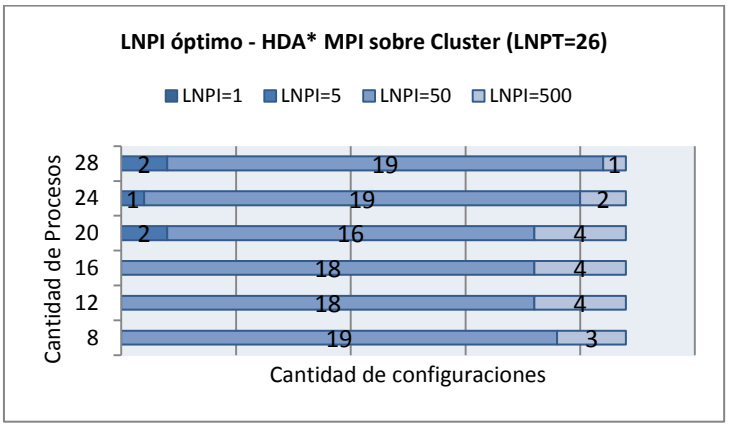
La Tabla 7.7 muestra los rangos de DE y CV de los grupos para distintos valores de LNPT y configuraciones de cluster que colocan 4 procesos por máquina. De los mismos se observa que a valores de LNPT más chicos, los tiempos de las *muestras promedio* de un grupo difieren más entre sí. Dicho en otras palabras, cuando los mensajes de trabajo enviados contienen pocos nodos, el valor del parámetro LNPI influye más sobre el tiempo de ejecución.

Tabla 7.7. Rangos de DE y CV clasificados por LNPT, para configuraciones de cluster que colocan 4 procesos por máquina

LNPT	Rango DE en segundos	Rango CV
26	0.063-18.94	0.054-1.28
210	0.04-9.43	0.054-0.24
1680	0.12-9.45	0.07-0.17

El valor de LNPI que optimiza el rendimiento depende de la configuración inicial, la cantidad de procesos y el valor del parámetro LNPT (es decir, el tamaño del mensaje de trabajo).

La Figura 7.19 muestra para cada valor de LNPT y configuración de cluster, la cantidad de configuraciones que alcanzan rendimiento óptimo con un LNPI determinado. Es importante destacar que en ningún caso se alcanzó rendimiento óptimo con LNPI = 1. Esto indica que el parámetro que se incorporó en la versión propia de HDA* MPI es de importancia, siendo un aporte original en el área.



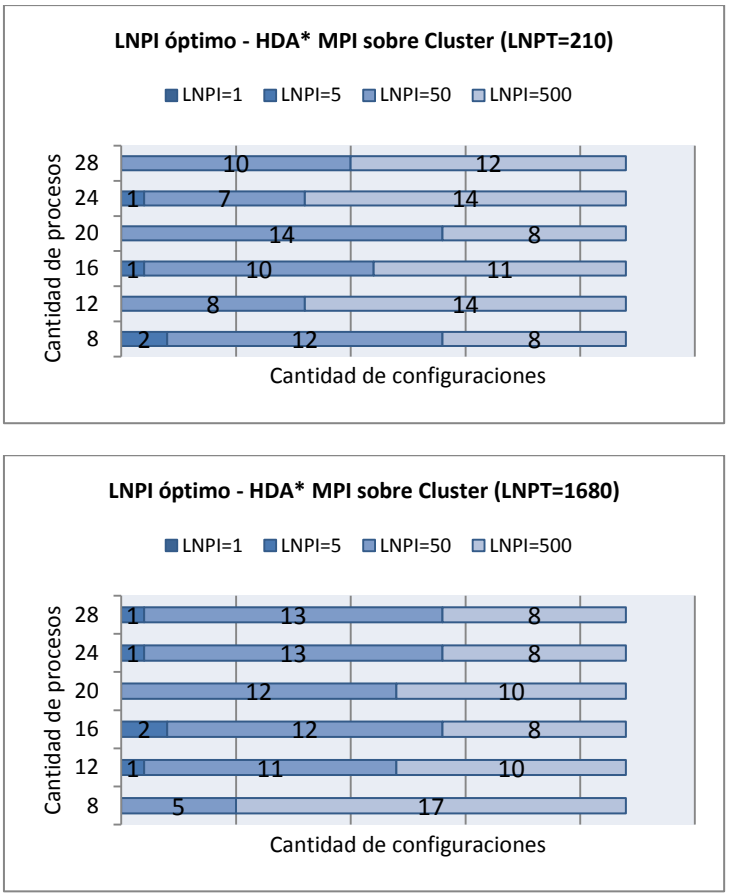


Figura 7.19. De arriba hacia abajo: Cantidad de configuraciones que alcanzan el rendimiento óptimo con un LNPI particular para LNPT=26, LNPT=210 y LNPT=1680.

Para LNPT=26, el valor que optimiza el rendimiento para las configuraciones en general es 50; las configuraciones que alcanzaron el óptimo con LNPI=500 son algunas de las más difíciles (60, 88, 105, 106 y 104), sin embargo a medida que aumenta la cantidad de procesadores la cantidad de configuraciones que prefirieron dicho valor disminuyó; las configuraciones que alcanzaron el óptimo con LNPI=5 son algunas de las menos complejas (3, 21 y 49).

Para LNPT=210 y LNPT=1680, la optimización del rendimiento surge con valores de LNPI 50 o 500. Contrastando con el caso anterior, no está clara la preferencia entre estos valores.

De lo anterior se podrían mantener las conclusiones de la Sección 7.3.1.1.1:

- Cuando LNPT es chico, los procesos enviarán muchos mensajes de trabajo conteniendo pocos nodos, por ello debe aumentarse la frecuencia de recepción de mensajes (valor bajo de LNPI) para permitir incorporar nuevos nodos que están entre los mejores globales. En caso de elegir un valor de LNPI muy alto, los procesos estarían realizando excesivo trabajo especulativo sobre nodos locales en cada etapa de procesamiento, sin incorporar nuevos nodos, lo cual aumenta el Overhead de Búsqueda[44], empeorando aún más al incrementar la cantidad de procesos.

[44] Las excepciones ocurren para algunas configuraciones más complejas y cuando se utilizan pocos procesos. Dichas configuraciones tienen gran cantidad de nodos con igual costo. Esta característica

- Cuando LNPT es grande, los mensajes contendrán muchos nodos y serán más bien escasos, con lo cual al disminuir la frecuencia de recepción (valor de LNPI alto) ya no se harán continuas verificaciones por llegada de mensajes que resulten fallidas. Cuando se elige en este caso un valor de LNPI bajo, no afecta tanto al rendimiento por utilizar comprobaciones asincrónicas de llegada de mensajes. No se notan variaciones significativas del Overhead de Búsqueda al variar LNPI.

En comparación con HDA* Pthreads, para el algoritmo HDA* MPI el parámetro LNPI toma mayor importancia ya que varía la frecuencia en la que el proceso verifica arribos de mensajes con costo de mejor solución.

7.3.2.1.2 Límite de Nodos Por Transferencia (LNPT)

El parámetro *Límite de Nodos Por Transferencia (LNPT*) indica la cantidad de nodos que contendrá como máximo cada mensaje de trabajo.

Con el objetivo de analizar el efecto del parámetro *LNPT* sobre el tiempo de ejecución, se tomaron todas las *muestras promedio* que surgen de las ejecuciones limitando *LNPT* a 26 nodos (mensajes de 1KB), 210 nodos (mensajes de 8KB) y 1680 nodos (mensajes de 64KB). A continuación, se agruparon las *muestras promedio* con igual configuración, cantidad de cores y *LNPI*, es decir cada grupo contiene las *muestras promedio* que difieren únicamente en el parámetro *LNPT*. Para cada *grupo* se calculó la Desviación Estándar (*DE*) y el Coeficiente de Variación (*CV*) del tiempo de ejecución, lo que proporciona una medida de cuánto difiere el tiempo de ejecución de una *muestra promedio* del grupo respecto al *tiempo promedio grupal*. Este procedimiento se realizó para las ejecuciones que ubican 4 procesos por máquina.

En general, los resultados arrojados revelan que la *DE* de los grupos se encuentra entre 0.045 y 24.37, esto indica que variar el parámetro *LNPT* entre los valores definidos provoca que los tiempos de ejecución de las *muestras promedio* se desvían en a lo sumo 24.37 segundos de su *promedio grupal*. Vale destacar que el 46.21% de los grupos tiene una *DE* menor a 1 y el 23.10% por debajo de 0.5.

Por otro lado, los valores generales de *CV* obtenidos para los grupos oscilan entre 0.019 y 1.15, es decir el tiempo de ejecución de una *muestra promedio* del grupo se desvía entre un 1.9% y un 115% de la *media grupal* al variar el parámetro *LNPT* entre los valores fijados. Sólo el 24.4% de los grupos tienen CV por debajo de un 0.1.

Por los valores altos de CV se concluye que el parámetro LNPT tiene influencia sobre el tiempo de ejecución. Esto se debe a que LNPT impacta en el congestionamiento del medio de comunicación, pero también tendrá influencia en la actividad de los procesos y en el balance de calidad de los nodos. Se mantienen las conclusiones de la Sección 7.3.1.1.2:

---

provoca que no influya el hecho de que el proceso no incorpore nodos frecuentemente y realice en cambio trabajo especulativo.

- Un valor bajo de LNPT provocará que los procesos formen mensajes de tamaño chico, esto incrementa la cantidad de mensajes, los cuales serán enviados de un proceso a otro por MPI vía red, y genera un *overhead* por la manipulación de buffers asociados a los mensajes.
- Valores muy altos de LNPT también pueden degradar el rendimiento dado que si un proceso empaqueta demasiados nodos para un proceso destino podría influir en la ociosidad de éste último cuando el mismo posee pocos nodos para procesar o cuando al momento se encuentra inactivo por falta de trabajo. También podría estar demorando la transmisión de nodos de mejor calidad que aquellos que actualmente el destinatario está procesando, contribuyendo a un incremento del *Overhead de Búsqueda.*

En la mayoría de los casos el valor de LNPT que optimiza los tiempos de ejecución para cada configuración y cantidad de cores es 210, es decir mensajes de 8KB. Las excepciones ocurren con algunas configuraciones poco complejas que prefirieron LNPT=26 a medida que aumenta la cantidad de procesos, ya que con valores más altos de LNPT crece el *Overhead de Búsqueda*; y también con algunas de las configuraciones más complejas cuando se utilizan pocos procesos, las cuales alcanzan rendimiento óptimo con LNPT=1680 a causa de la obtención de menor *Overhead de Búsqueda* o menor *Desbalance de Carga*. Optimizan su rendimiento con LNPT=26 las configuraciones: 21 con 12 procesos; 100, 21 y 101 con 16 procesos; 100, 21, 101 y 82 con 20 procesos; 100, 33, 21, 101 y 82 con 24 procesos; 100, 33, 21, 101, 82, 59 y 53 con 28 procesos. Optimizan su rendimiento con LNPT=1680 las configuraciones: 88, 102, 106, 104 con 8 procesos; 104 con 12 procesos.

De lo anterior se concluye que para las configuraciones menos complejas, a medida que aumenta la cantidad de procesadores es óptimo usar mensajes de trabajo más chicos (disminuir LNPT). Para las configuraciones más complejas y pocos procesos, en ocasiones se alcanza el rendimiento óptimo con mensajes de trabajo más grandes (aumentar LNPT). Sin embargo, para esta arquitectura el rendimiento óptimo a medida que escala la carga de trabajo y aumenta el número de procesos se obtiene con mensajes de trabajo de tamaño medio (8KB).

Las Figuras 7.20, 7.21, 7.22, 7.23, 7.24 y 7.25 muestran el Speedup según la configuración inicial, utilizando 8, 12, 16, 20, 24 y 28 procesos respectivamente, para cada valor de LNPT, teniendo en cuenta que la *muestra promedio seleccionada* por configuración y cantidad de cores es aquella cuyo LNPI optimiza el rendimiento. De manera similar, las Figuras 7.26, 7.27, 7.28, 7.29, 7.30 y 7.31 muestran la Eficiencia según la configuración inicial, para 8, 12, 16, 20, 24 y 28 procesos respectivamente, para cada valor de LNPT.

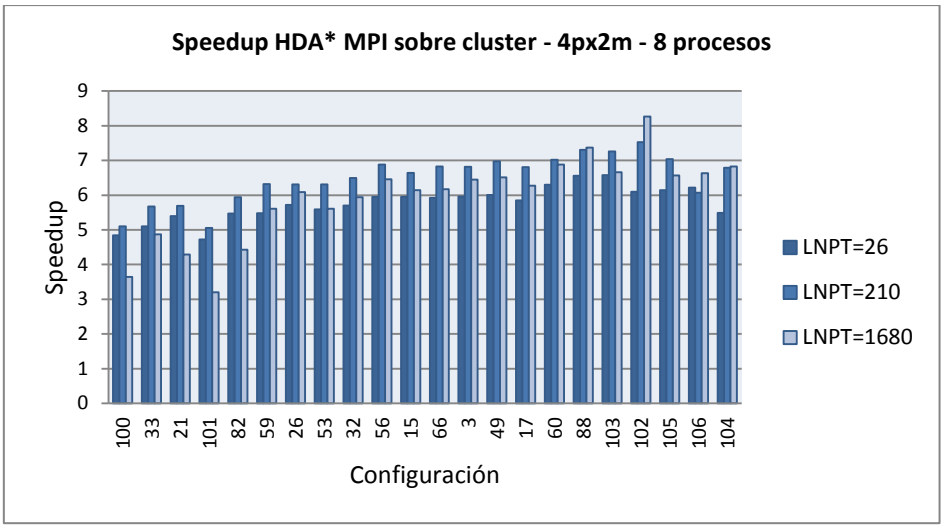


Figura 7.20. Speedup obtenido por HDA* MPI según LNPT, para cada configuración y 8 procesos.

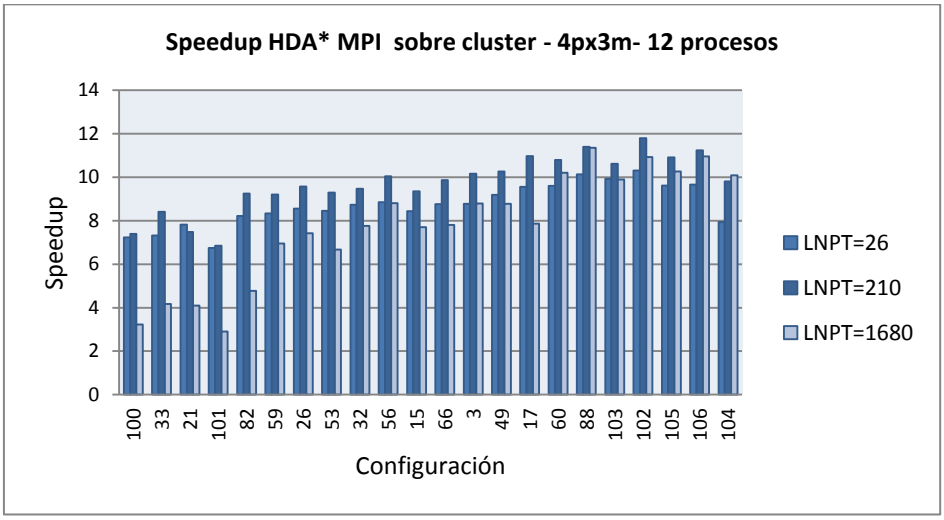


Figura 7.21. Speedup obtenido por HDA* MPI según LNPT, para cada configuración y 12 procesos.

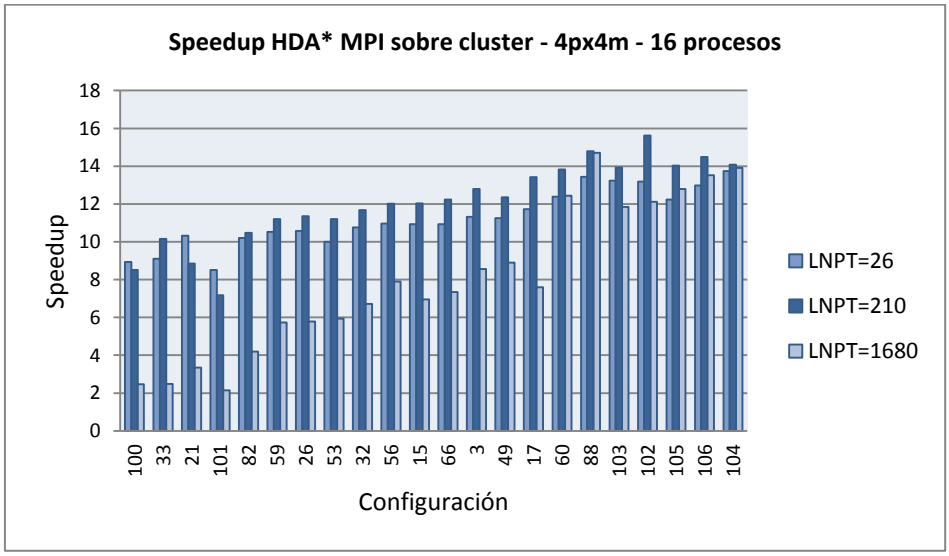


Figura 7.22. Speedup obtenido por HDA* MPI según LNPT, para cada configuración y 16 procesos.

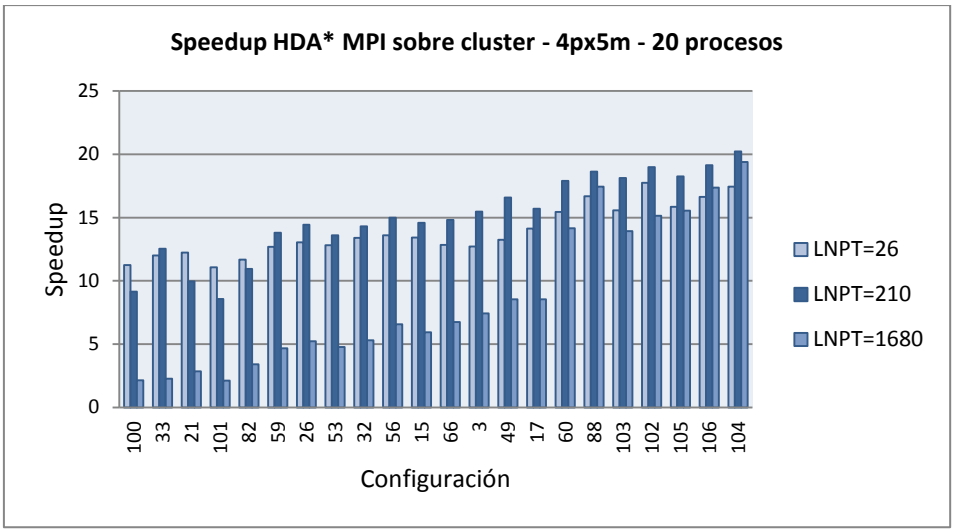


Figura 7.23. Speedup obtenido por HDA* MPI según LNPT, para cada configuración y 20 procesos.

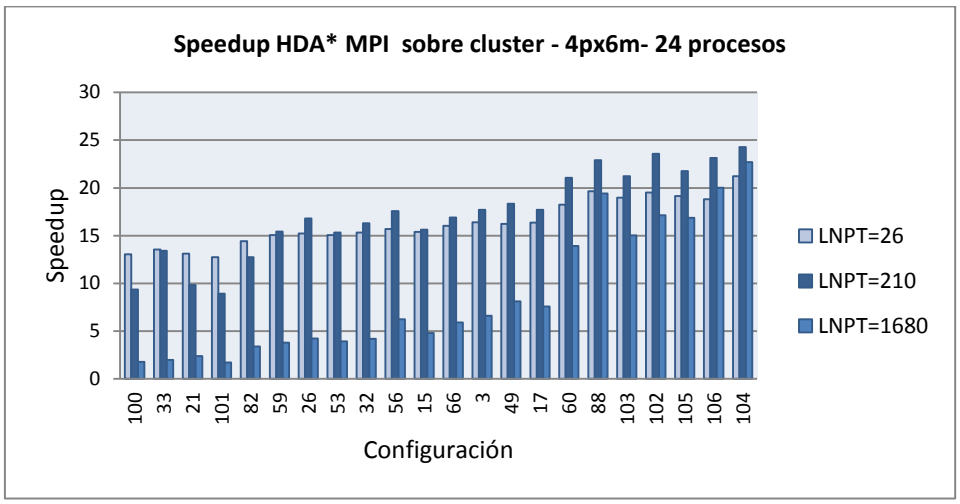


Figura 7.24. Speedup obtenido por HDA* MPI según LNPT, para cada configuración y 24 procesos.

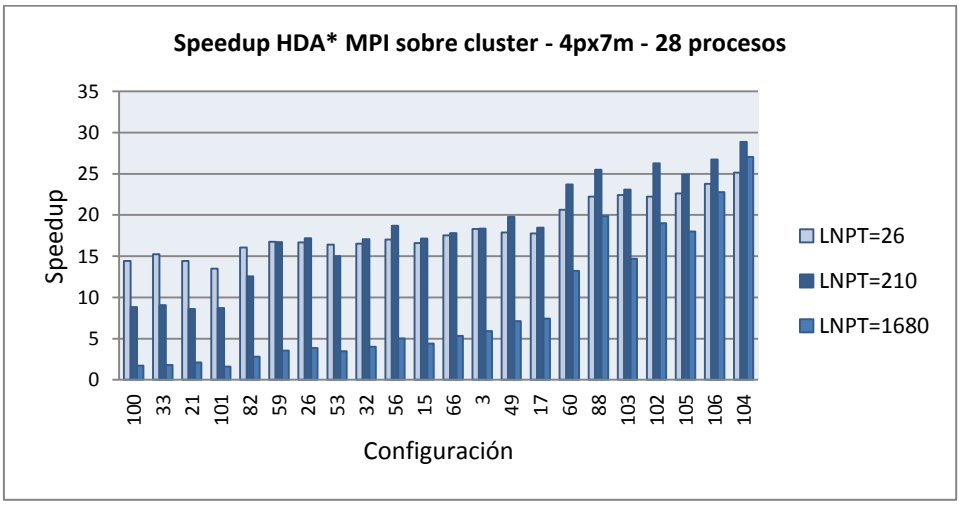


Figura 7.25. Speedup obtenido por HDA* MPI según LNPT, para cada configuración y 28 procesos.

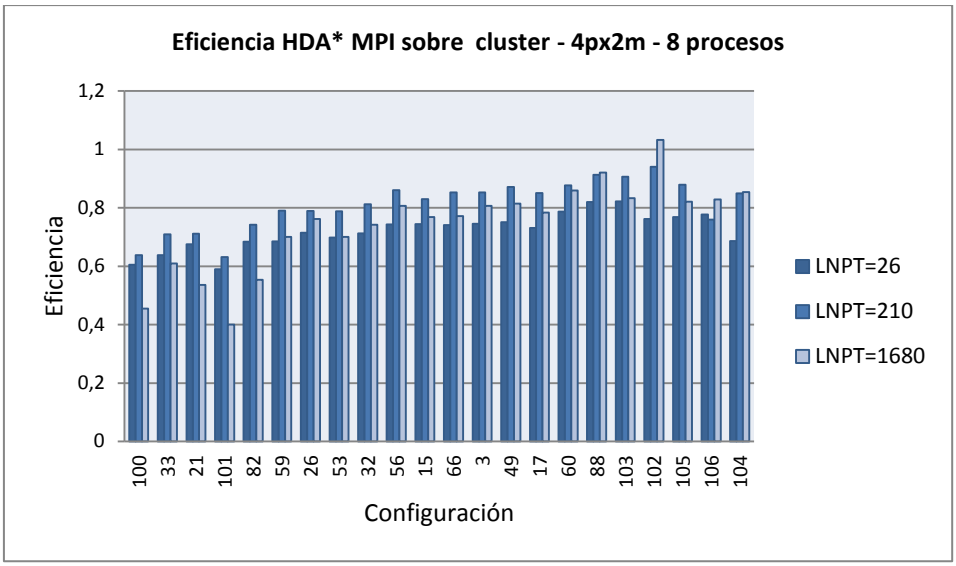


Figura 7.26. Eficiencia obtenida por HDA* MPI según LNPT, para cada configuración y 8 procesos.

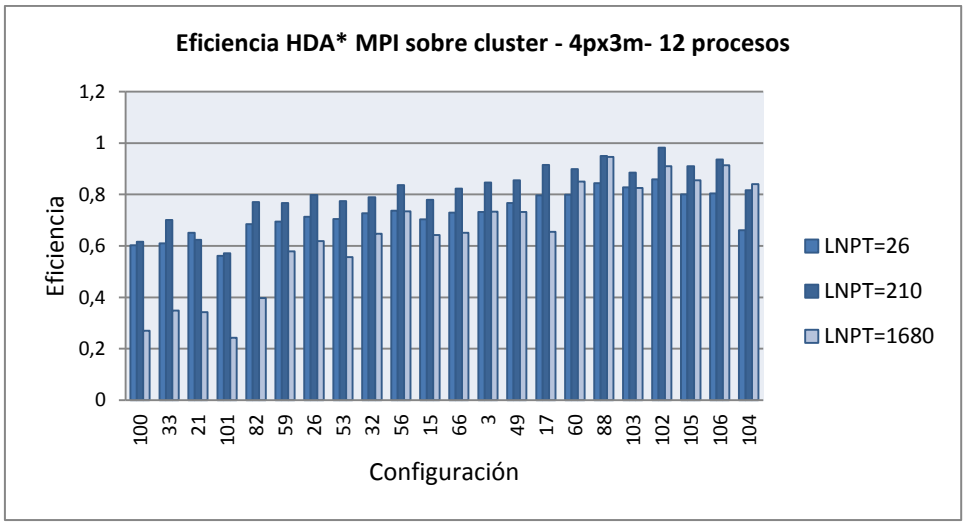


Figura 7.27. Eficiencia obtenida por HDA* MPI según LNPT, para cada configuración y 12 procesos.

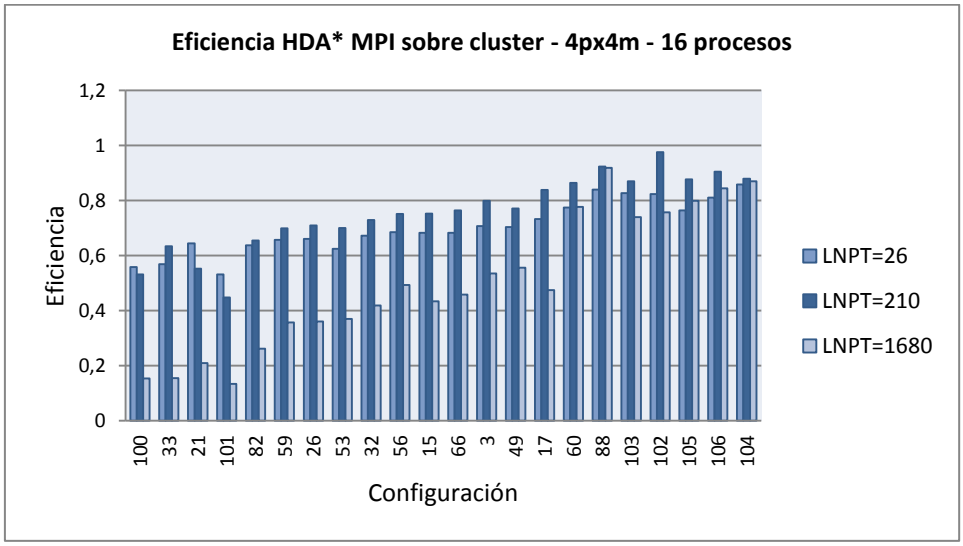


Figura 7.28. Eficiencia obtenida por HDA* MPI según LNPT, para cada configuración y 16 procesos.

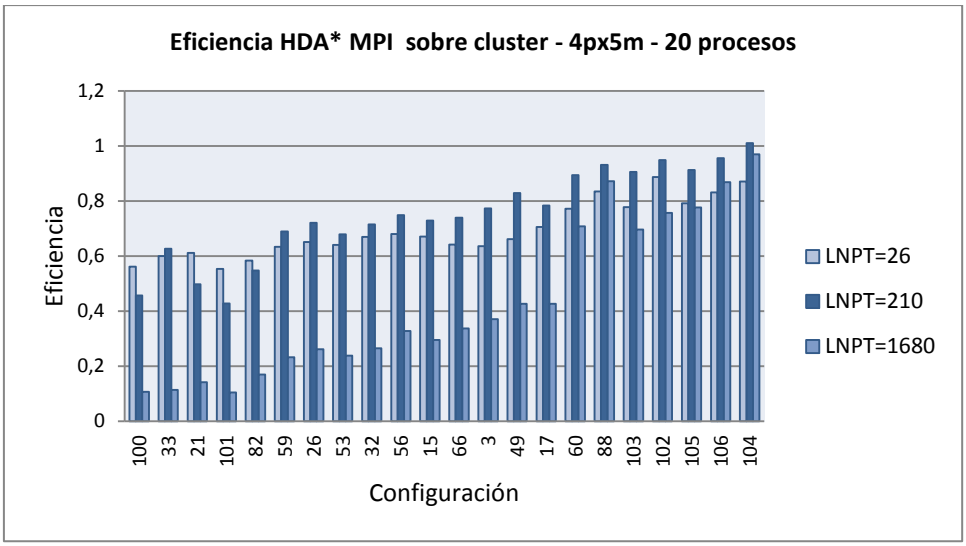


Figura 7.29. Eficiencia obtenida por HDA* MPI según LNPT, para cada configuración y 20 procesos.

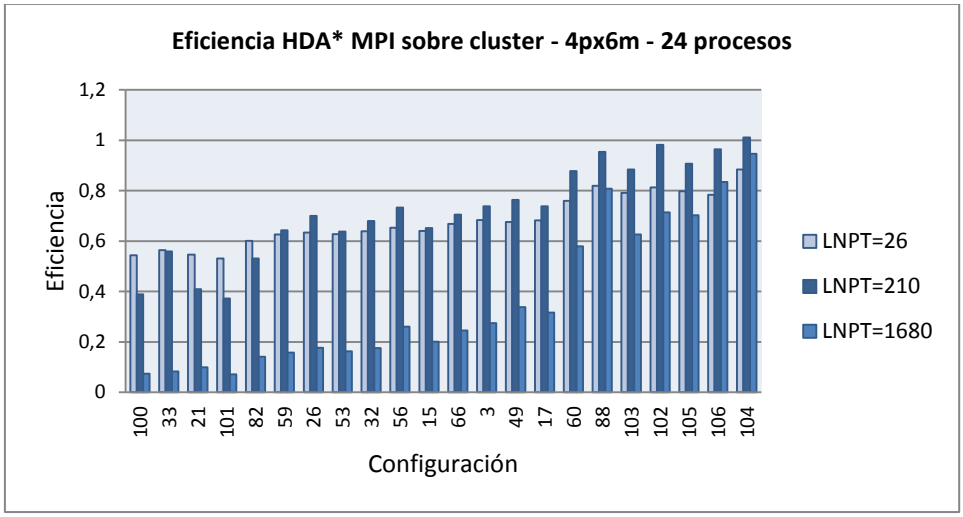


Figura 7.30. Eficiencia obtenida por HDA* MPI según LNPT, para cada configuración y 24 procesos.

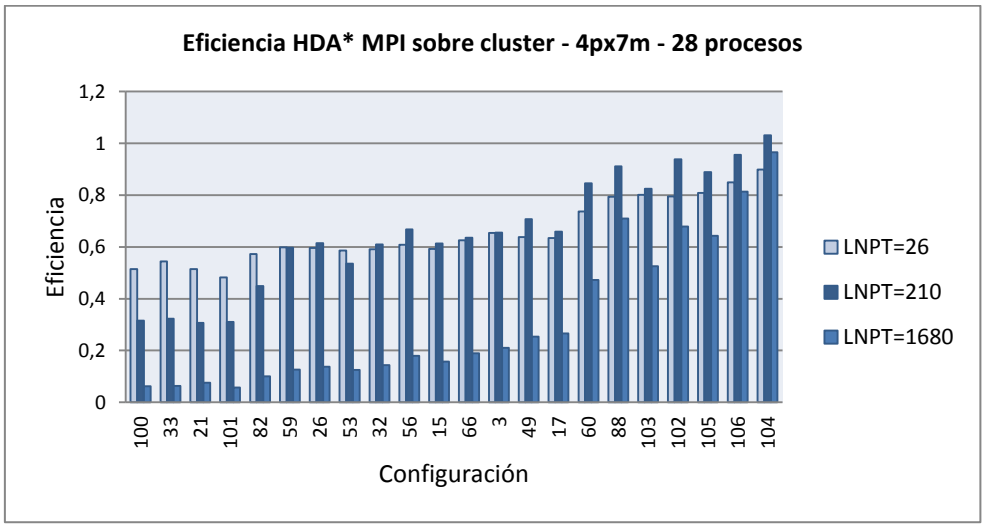


Figura 7.31. Eficiencia obtenida por HDA* MPI según LNPT, para cada configuración y 28 procesos.

La Tabla 7.8 resume los rangos en los que se encuentran el Speedup y la Eficiencia, según el valor LNPT y la cantidad de procesos, tomando las *muestras promedio* con LNPI óptimo para cada LNPT.

Tabla 7.8. Speedup y Eficiencia por cada LNPT y cantidad de procesos, para HDA* MPI sobre cluster

LNPT	Procesos	Speedup	Eficiencia
26	**8**	4.72 - 6.58	0.59 - 0.82
210	**8**	5.05 - 7.53	0.63 - 0.94
1680	**8**	3.20 - 8.26	0.40 - 1.03
26	**12**	6.74 - 10.31	0.56 - 0.86
210	**12**	6.85 - 11.79	0.57 - 0.98
1680	**12**	2.90 - 11.35	0.24 - 0.95
26	**16**	8.52 - 13.73	0.53 - 0.86
210	**16**	7.16 - 15.62	0.45 - 0.98
1680	**16**	2.15 - 14.70	0.13 - 0.92
26	**20**	11.07 - 17.76	0.55 - 0.89
210	**20**	8.57 - 20.22	0.43 - 1.01
1680	**20**	2.11 - 19.40	0.11 - 0.97
26	**24**	12.73 - 21.23	0.53 - 0.88
210	**24**	8.92 - 24.27	0.37 - 1.01
1680	**24**	1.72 - 22.71	0.07 - 0.95
26	**28**	13.49 - 25.16	0.48 - 0.90
210	**28**	8.60 - 28.86	0.31 - 1.03
1680	**28**	1.61 - 27.05	0.06 - 0.97

7.3.2.2 Desvío Estándar del Tiempo de Ejecución

Con el objetivo de conocer la variación de los tiempos de ejecución de una muestra con respecto a la *muestra promedio* teniendo en cuenta una configuración inicial, cantidad de procesadores y parámetros fijos, se calculó la Desviación Estándar *DE* y el Coeficiente de Variación *CV,* que permitirá interpretar mejor la *DE*. Este procedimiento se realizó para las ejecuciones que ubican 4 procesos por máquina.

Considerando *todas* las pruebas ejecutadas con LNPT = 26, se comprobó que el CV se encuentra entre 0.0086 y 0.19, es decir el tiempo de ejecución de una muestra difiere de su *muestra promedio* en un máximo de 19%. Asimismo, el 93.56% de las pruebas realizadas tienen un CV por debajo del 10% por lo que se puede decir que el no determinismo y asincronismo presente en el algoritmo paralelo no provocan una variación significativa en el tiempo de ejecución.

La Tabla 7.9 muestra el rango en el que varía la *DE* y el *CV* para las pruebas ejecutadas con distintos valores de LNPT. De lo observado se puede concluir que el CV disminuye al incrementar LNPT, por lo que el tiempo de resolución de una *muestra* varía en menor medida respecto a la *muestra promedio*.

Tabla 7.9. DE y CV de las muestras para las pruebas con 4 procesos por máquina, clasificados por LNPT

LNPT	Rango DE en segundos	Rango CV
26	0.0091 -15.02	0.0086 - 0.19
210	0.012 - 6.62	0.0068 - 0.14
1680	0.024 - 4.58	0.008 - 0.11

De manera similar, las ejecuciones que ubican 8 procesos por máquina exhiben CV menores al 13%.

7.3.2.3 Análisis de Rendimiento

Para analizar el rendimiento del algoritmo se tomaron en cuenta aquellas pruebas experimentales de las secciones anteriores que optimizaron los resultados. Se seleccionó para cada configuración inicial y cantidad de procesos la *muestra promedio* que minimiza el tiempo de resolución, es decir aquella cuyos valores de parámetros *LNPI* y *LNPT* optimizan el rendimiento. Este procedimiento se realizó para las pruebas que ubican 4 y 8 procesos por máquina.

Para evaluar la escalabilidad del algoritmo se ordenaron las distintas *muestras promedio seleccionadas* para cada configuración inicial según la carga secuencial de trabajo de ésta (tiempo secuencial). En este sentido escalar el problema significa aumentar la cantidad de nodos procesados o generados. Por otro lado, la arquitectura escala aumentando la cantidad de cores/procesos usados para resolver el problema.

La Figura 7.32 muestra el Speedup obtenido por la *muestra promedio seleccionada* para cada configuración inicial utilizando 8, 12, 16, 20, 24 y 28 procesos, para configuraciones de cluster que colocan 4 procesos por máquina, mientras que la Figura 7.33 muestra la Eficiencia obtenida. La Tabla 7.10 muestra los rangos en que varía el Speedup y Eficiencia para distinta cantidad de procesos utilizada.

Analizando los datos presentados, se concluye que si se mantiene la carga de trabajo (configuración inicial) y se aumenta el número de cores/procesos, el Speedup obtenido es mejor lo que indica que el problema se resuelve en menor tiempo. Sin embargo, en general esta mejora no mantiene la Eficiencia en un valor constante. La desmejora en la eficiencia se debe a factores tales como: partes secuenciales en especial al inicio y fin del cómputo, sincronización, comunicación, tiempo ocioso, desbalance de carga, aumento del overhead de búsqueda, entre otros.

El Speedup superlineal obtenido por la configuración 102 y 8 procesos se debe a la obtención de *Overhead de Búsqueda* negativo. Los demás casos se deben a la disminución en la cantidad de elementos en las estructuras Lista Abierta y Lista Cerrada, lo que provoca una aceleración en las operaciones realizadas sobre éstas.

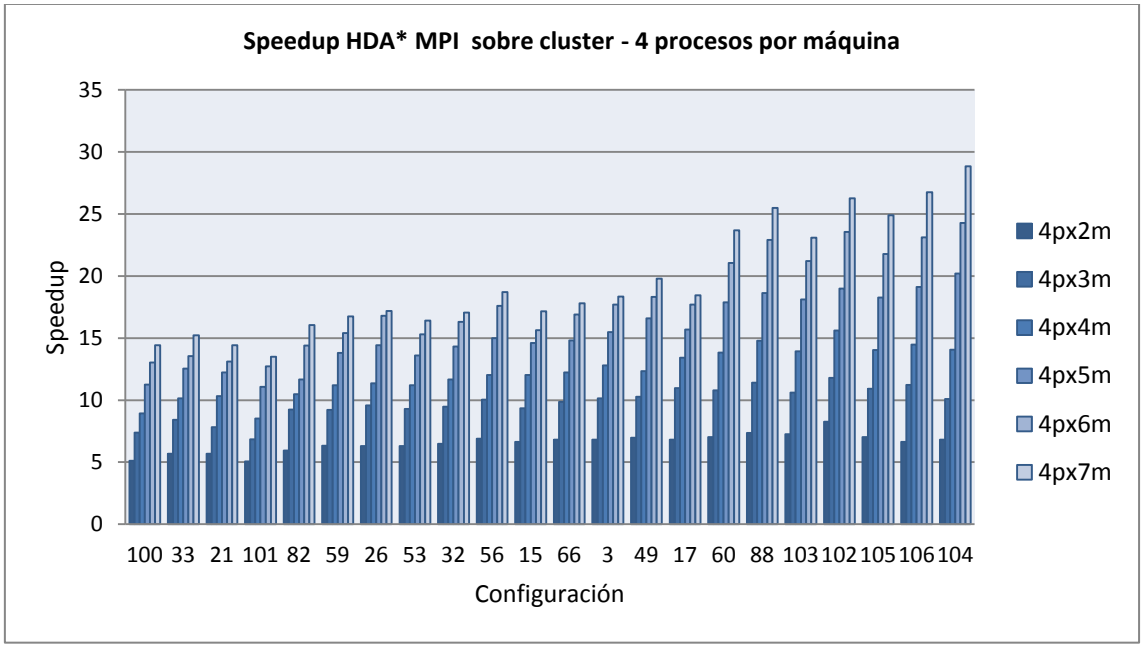


Figura 7.32. Speedup obtenido por HDA* MPI sobre cluster, para configuraciones de cluster que colocan 4 procesos por máquina.

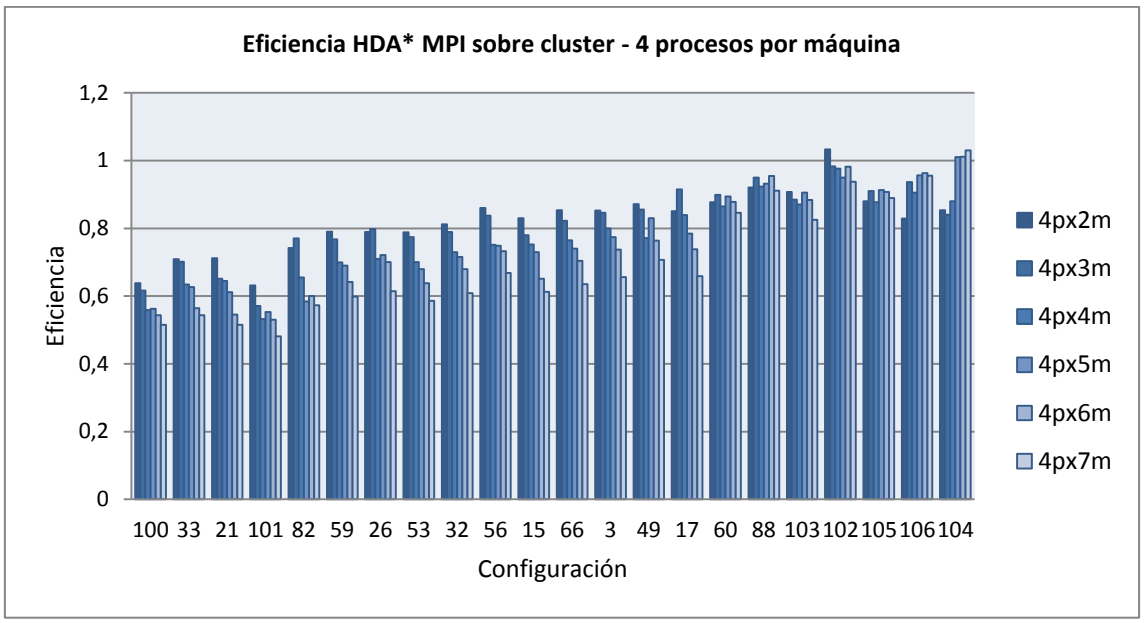


Figura 7.33. Eficiencia obtenida por HDA* MPI sobre cluster, para configuraciones de cluster que colocan 4 procesos por máquina.

Tabla 7.10. Rangos de Speedup y Eficiencia obtenido por HDA* MPI sobre cluster, para configuraciones de cluster que colocan 4 procesos por máquina.

Procesos/ Procesadores	Rango Speedup	Rango Eficiencia
8	5.05 - 8.26	0.63 -1.03
12	6.85 - 11.79	0.57 - 0.98
16	8.52 - 15.62	0.53 - 0.98
20	11.07 - 20.22	0.55 - 1.01
24	12.73 - 24.27	0.53 - 1.01
28	13.49 - 28.86	0.48 - 1.03

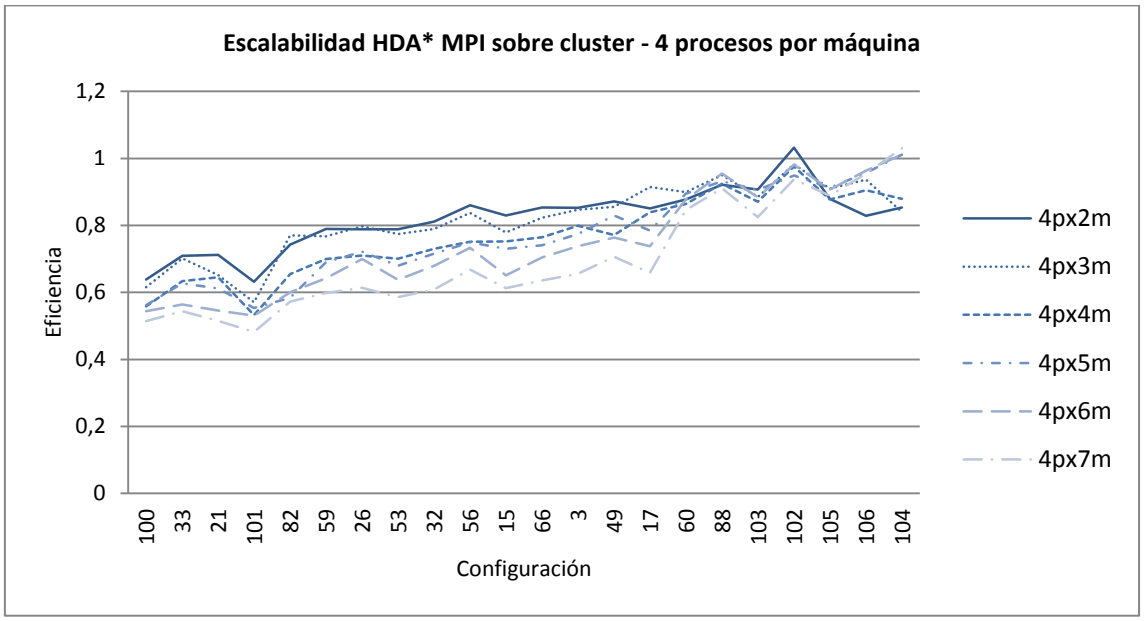


Figura 7.34. Eficiencia obtenida por HDA* MPI sobre cluster, para configuraciones de cluster que colocan 4 procesos por máquina, al escalar la cantidad de procesos y carga de trabajo.

La Figura 7.34 grafica la Eficiencia a medida que aumenta la carga de trabajo y cantidad de procesos, para las pruebas que ubican 4 procesos por máquina. En la misma se observa que cuando escala el problema manteniendo fijo el número de *cores* usados en general la Eficiencia mejora o se mantiene constante dado que el *overhead* es menos significativo en el tiempo total de procesamiento.

De los resultados presentados se concluye que el comportamiento expuesto por el algoritmo cuando se ejecuta con configuraciones de cluster que ubican 4 procesos por máquina es propio de un sistema paralelo escalable, donde la eficiencia puede mantenerse en un valor constante cuando se escala la carga de trabajo y la arquitectura.

De manera similar, la Figura 7.35 muestra el Speedup obtenido por la *muestra promedio seleccionada* para cada configuración inicial utilizando 16, 24, 32, 40, 48 y 56 procesos, para configuraciones de cluster que colocan 8 procesos por máquina, mientras que la Figura 7.36

muestra la Eficiencia obtenida. La Tabla 7.11 muestra los rangos en que varía el Speedup y Eficiencia para distinta cantidad de procesos utilizada.

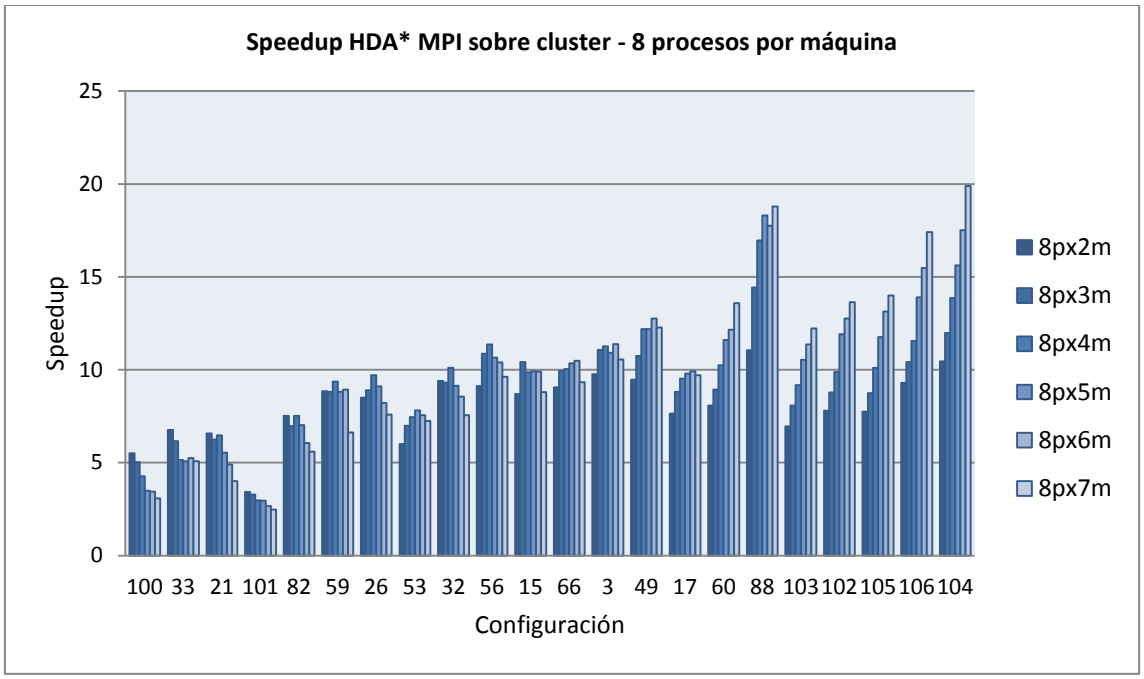


Figura 7.35. Speedup obtenido por HDA* MPI sobre cluster, para configuraciones de cluster que colocan 8 procesos por máquina.

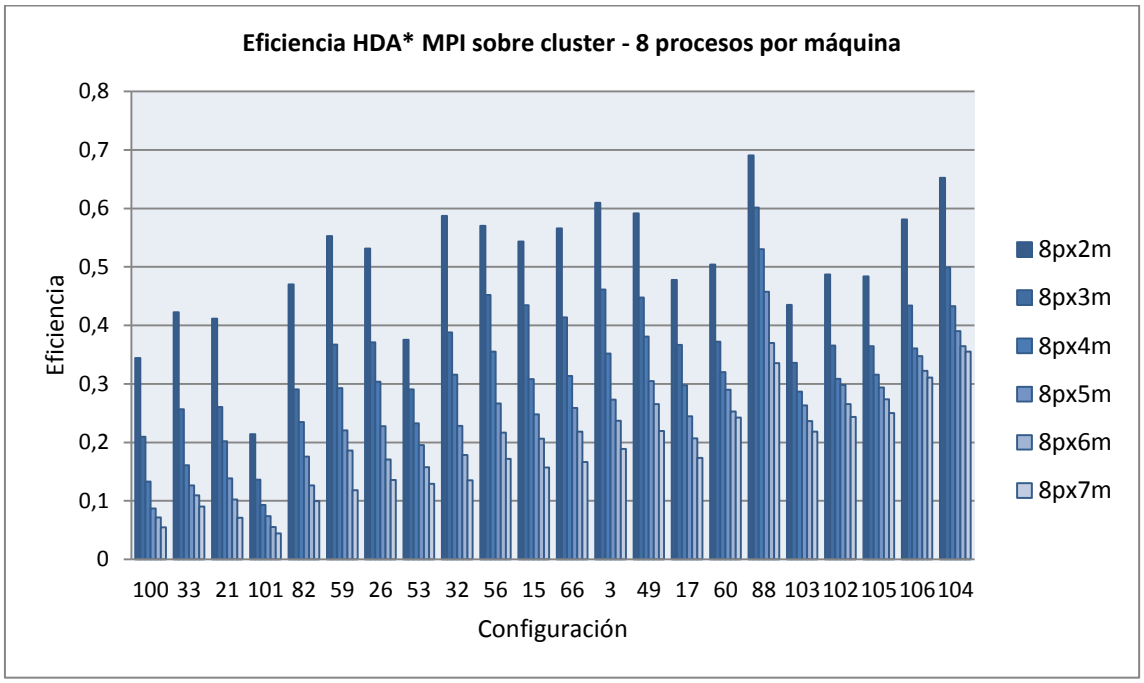


Figura 7.36. Eficiencia obtenida por HDA* MPI sobre cluster, para configuraciones de cluster que colocan 8 procesos por máquina.

Tabla 7.11. Rangos de Speedup y Eficiencia obtenido por HDA* MPI sobre cluster, para configuraciones de cluster que colocan 8 procesos por máquina.

Procesos	Rango Speedup	Rango Eficiencia
16	3.42 - 11.05	0.21 - 0.69
24	3.28 - 14.44	0.14 - 0.60
32	2.98 - 16.97	0.09 - 0.53
40	2.96 - 18.31	0.07 - 0.46
48	2.66 - 17.76	0.06 - 0.37
56	2.47 - 19.89	0.04 - 0.36

De la Figura 7.35 se observa que al añadir cores/procesos para las configuraciones no tan complejas, el speedup crece hasta cierto punto y luego decae, lo que significa que a partir de ese punto el hecho de añadir cores/procesos incrementa el tiempo de cómputo. En la Figura 7.36 se muestra para cada configuración una caída abrupta de la eficiencia a medida que se añaden procesadores. La Tabla 7.11 exhibe valores decrecientes de Eficiencia mínima y máxima obtenida.

Los indicadores anteriores muestran que el comportamiento del algoritmo cuando se ejecuta sobre configuraciones de cluster que colocan 8 procesos por máquina no es escalable.

En la Figura 7.37 se puede observar una comparación del Speedup obtenido por el algoritmo cuando se ejecuta sobre configuraciones de cluster 4px4m y 8px2m (16 procesos). De modo similar, la Figura 7.38 muestra una comparación del Speedup obtenido por el algoritmo cuando se ejecuta sobre configuraciones de cluster 4px6m y 8px3m (24 procesos). Claramente, el rendimiento obtenido cuando se ubican 8 procesos por máquina es muy inferior al obtenido cuando se ubican 4 procesos por máquina.

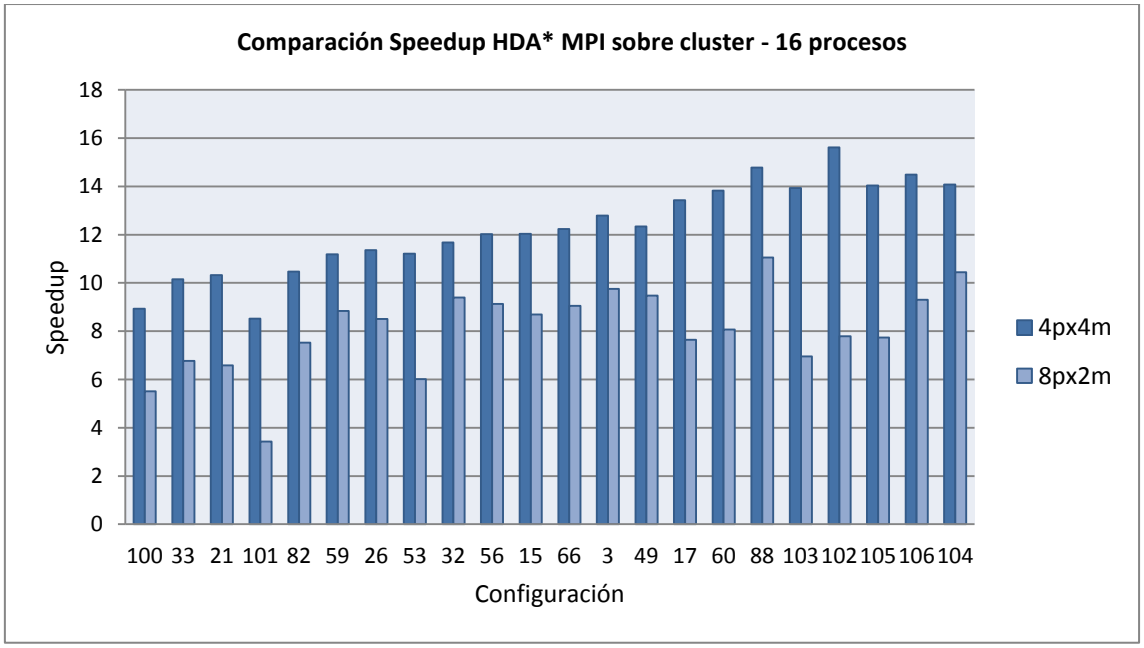


Figura 7.37. Speedup obtenido por HDA* MPI sobre configuraciones de cluster 4px4m y 8px2m

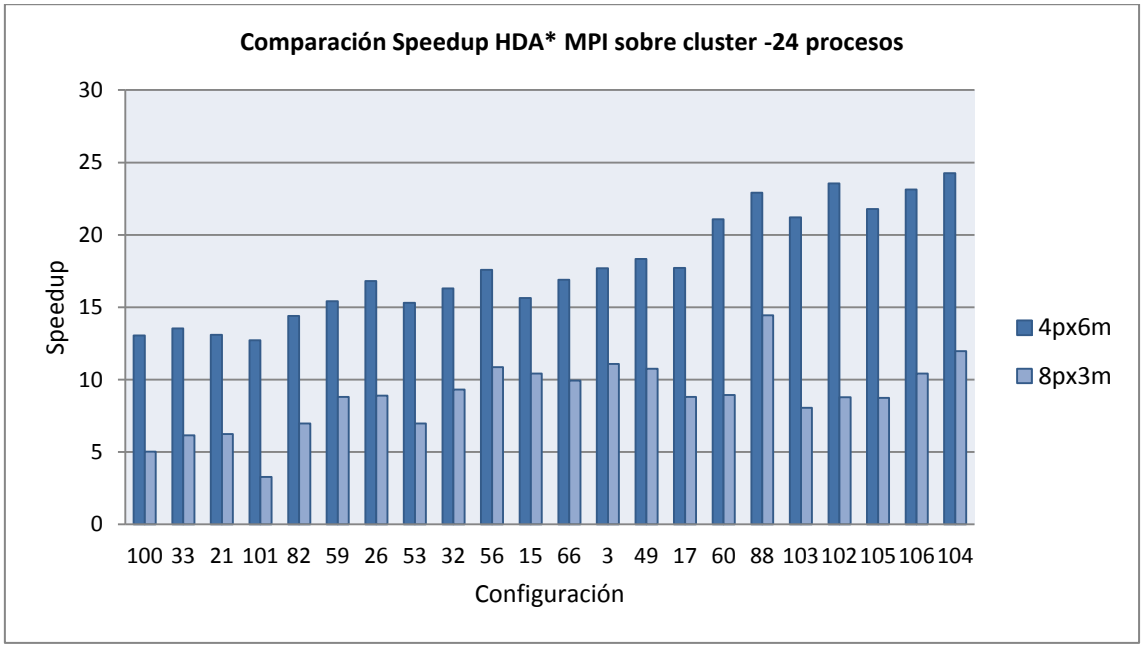


Figura 7.38. Speedup obtenido por HDA* MPI sobre configuraciones de cluster 4px6m y 8px3m

7.3.2.3.1 Factores de Overhead

Esta sección analiza el Overhead de Búsqueda (OB) y Desbalance de Carga (DC) para las pruebas evaluadas en la Sección 7.3.2.3, ejecutadas sobre configuraciones de cluster que colocan 4 procesos por máquina. Las Figuras 7.39 y 7.40 muestran el Overhead de Búsqueda y Desbalance de Carga respectivamente.

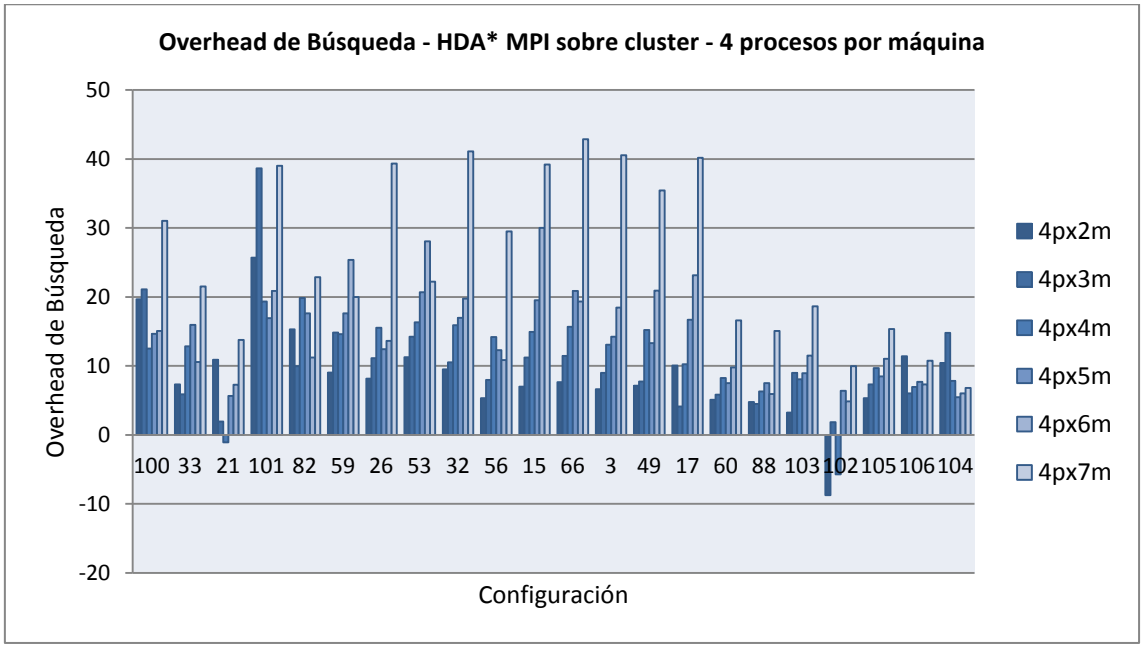


Figura 7.39. Overhead de Búsqueda obtenido por HDA* MPI sobre configuraciones de cluster que colocan 4 procesos por máquina

En general, las pruebas exhiben un OB por debajo del 20%. Las pruebas que traspasan dicho límite corresponden a configuraciones poco complejas y gran cantidad de procesadores. Al mantener fija una instancia de entrada y al aumentar la cantidad de procesadores, el OB tiende a crecer. Sin embargo, al aumentar la complejidad de la instancia de entrada, el OB tiende a disminuir.

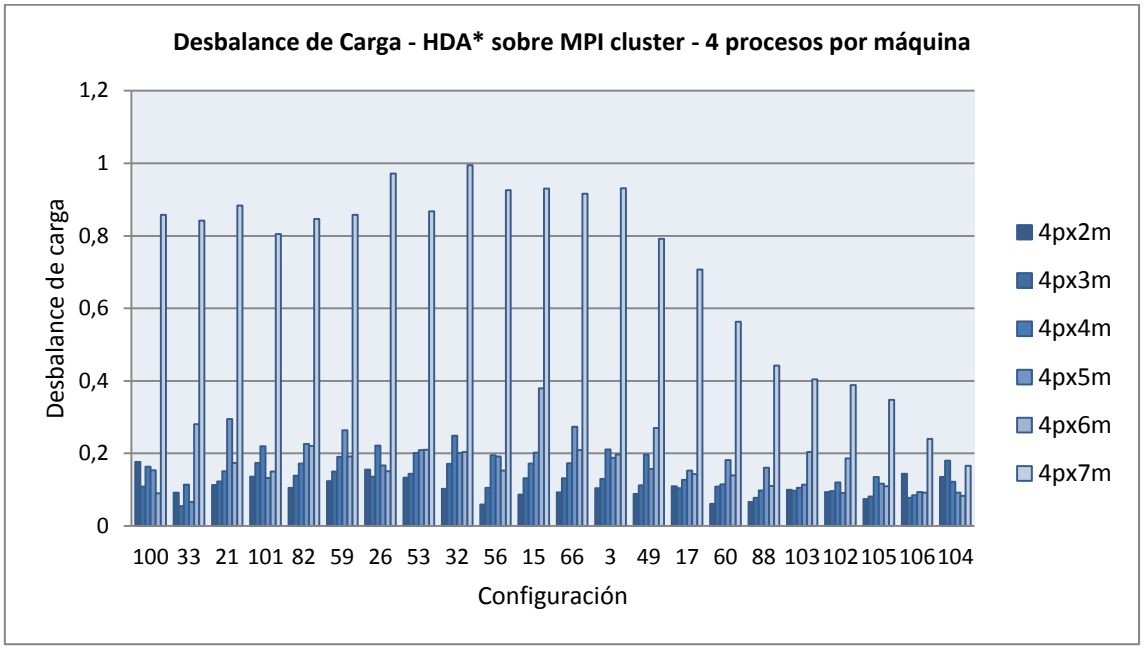


Figura 7.40. Desbalance de Carga obtenido por HDA* MPI sobre configuraciones de cluster que colocan 4 procesos por máquina

Por otro lado, el DC en general se mantiene cercano al 0,2. En general, las pruebas que superaron dicho valor son aquellas que utilizan 28 procesos. Se puede observar que al mantener fija una instancia de entrada y al aumentar la cantidad de procesadores, el DC tiende a crecer. Sin embargo, al aumentar la complejidad de la instancia de entrada, el DC tiende a disminuir.

7.4 Comparación del consumo de memoria de HDA* Pthreads y HDA* MPI sobre máquina multicore

El objetivo de esta sección es comparar el consumo de memoria de las aplicaciones HDA* MPI y HDA* Pthreads cuando se ejecutan sobre una máquina multicore, utilizando 4 y 8 cores de la arquitectura. Para realizar las mediciones de consumo de memoria de las aplicaciones se recurrirá a información recolectada por el Sistema Operativo.

El Sistema Operativo *Linux* almacena información estadística para cada proceso en ejecución bajo el directorio /proc/pid, donde pid es el identificador del proceso (Kerrisk, 2014). En particular, el directorio /proc/pid/status provee la siguiente información básica que permitirá

identificar al proceso en sí y el consumo de memoria por parte de éste. Dicha información puede ser accedida por el proceso en tiempo de ejecución. Los campos de interés son:

- *Tgid:* Identificador del proceso.
- *Pid:* Identificador del thread.
- *PPid:* Identificador del proceso padre.
- *Threads:* Número de threads del proceso que contiene a éste thread.
- *VmSize:* Virtual memory size (KB). Contabiliza todas las alocaciones en memoria virtual (archivos, memoria compartida, memoria de pila). Su valor se incrementa cada vez que se realiza un *alloc* y decrece cada vez que una dirección virtual es liberada.
- *VmPeak:* Peak virtual memory size (KB). Valor máximo de *VmSize* en la ejecución hasta el momento.
- *VmRSS:* Resident set size (KB). Su valor se incrementa cuando la memoria es accedida y decrece cuando se transfieren páginas de RAM a disco.
- *VmHWM:* Peak resident set size (KB). Valor máximo de *VmRSS* en la ejecución hasta el momento.
- *VmLib:* Shared library code size (KB).

El campo a tener en cuenta para medir el consumo de memoria es *VmPeak,* ya que su valor al final de la aplicación mide la máxima cantidad de memoria virtual alocada durante la ejecución.

En una aplicación multiproceso, por ejemplo una aplicación MPI, cada proceso debe acceder a su propia información desde /proc/pid/status; luego si se requiere conocer el total de memoria utilizada se deberá realizar una reducción de la información obtenida.

En una aplicación multihilo, tal es el caso de aplicaciones que utilizan Pthreads, solamente un hilo (por ejemplo el maestro) debe acceder a la información contenida en /proc/pid/status, ya que la misma es referida al proceso que contiene a los threads.

Se crearon dos programas simples, uno que utiliza MPI y otro que utiliza Pthreads cuya tarea es crear N procesos o threads respectivamente, cada proceso o thread toma su rank o id. Cada proceso del programa MPI toma la información de /proc/pid/status y la envía a la salida estándar. Por otro lado, sólo el thread maestro del programa Pthreads toma la información de /proc/pid/status y la envía a la salida estándar. La ejecución de estos programas servirá para evaluar la cantidad de memoria reservada inicialmente por aplicaciones que utilizan estas bibliotecas de programación paralela.

Se ejecutaron ambos programas de prueba sobre una de las 7 máquinas de la computadora paralela mencionada al principio del capítulo; las pruebas utilizan 4 u 8 procesos/threads.

El programa simple que utiliza Pthreads arrojó un valor de *VmPeak* igual a 158.16MB para 4 threads y 190.18MB para 8 threads. Por su parte, los valores totales de *VmPeak* arrojados por el programa simple MPI fueron 608.14MB para 4 procesos y 1216.31MB para 8 procesos.

De los resultados anteriores se concluye que inicialmente el programa simple Pthreads reduce la reserva de memoria en un 73.99% para 4 cores y un 84.36% para 8 cores respecto al programa simple MPI.

La gran cantidad de memoria utilizada por MPI se debe a que éste realiza una reserva importante de espacio al comienzo de la aplicación para gestionar comunicaciones, la cual se incrementa con el número de procesos en el sistema.

A continuación, se realizaron mediciones de consumo de memoria de las aplicaciones HDA* Pthreads (utilizando Jemalloc configurado con 256 *arenas*, espera activa y la técnica *Memory Pool)* y HDA* MPI (utilizando Jemalloc configurado con 256 *arenas*) cuando se ejecutan sobre una máquina multicore.

Las pruebas experimentales tienen en cuenta las 100 configuraciones iniciales y la configuración final propuestas por (Korf, 1985). Para cada configuración inicial y grupo de parámetros (entrada) se realizaron 10 ejecuciones, en cada ejecución se obtuvo el valor *VmPeak* al final de la aplicación (siguiendo la estrategia antes planteada según la aplicación), y luego se calculó el *valor promedio de VmPeak* para esos datos de entrada. Los parámetros utilizados son: cantidad de threads/procesos (entre 4 y 8); LNPI={1,5,50,500}; LNPT se limitó a 210.

De la comparación de los valores promedio de *VmPeak* para cada conjunto de datos de entrada se observa una reducción a favor de HDA* Pthreads que va entre 13.51% a 80.92% respecto a HDA* MPI, mientras que los valores de reducción absolutos en MB están entre 408.26 a 1560.42.

Tomando en cuenta las pruebas que utilizan 4 cores, las reducciones porcentuales a favor de HDA* Pthreads van de 13.51% a 75%, mientras que las reducciones absolutas en MB varían entre 408.26 y 867.86. Por otro lado, si se tienen en cuenta las pruebas que utilizan 8 cores, las reducciones porcentuales varían entre 28.06% y 80.92%, mientras que las reducciones absolutas en MB están en el rango 1112.02 y 1560.42.

Esta diferencia en el uso de memoria a favor de HDA* Phtreads respecto a HDA* MPI constituye un punto a favor para la realización de un algoritmo HDA* híbrido, ya que dentro de una máquina el algoritmo HDA* Pthreads utiliza menos memoria que HDA* MPI.

7.5 Comparación del rendimiento de HDA* Pthreads y HDA* MPI sobre máquina multicore

Esta sección realiza una comparación del rendimiento obtenido por HDA* Pthreads, presentado en la Sección 7.2.5, y HDA* MPI, presentado en la Sección 7.3.1.3, cuando la ejecución se realiza sobre una máquina multicore.

Para cada algoritmo se consideraron las *muestras promedio seleccionadas* para cada configuración y cantidad de procesos/hilos utilizados (es decir se seleccionaron las pruebas cuyo valor de *LNPI* mejoran el rendimiento). El algoritmo HDA* Pthreads fue configurado

utilizando espera activa y la técnica *Memory Pool*. Ambos algoritmos fueron configurados para utilizar Jemalloc con 256 *arenas* y *LNPT* = 210, ya que se ha demostrado alcanzan mejor rendimiento.

La Figura 7.41 muestra el Speedup alcanzado por HDA* Pthreads y HDA* MPI, variando la configuración inicial y utilizando 4 cores, mientras que la Figura 7.42 muestra el Speedup alcanzado por ambos algoritmos, para las mismas configuraciones, cuando se utilizan 8 cores. Las Figuras 7.43 y 7.44 muestran la Eficiencia obtenida para los algoritmos utilizando 4 y 8 cores respectivamente. Asimismo, las Figuras 7.45 y 7.46 grafican la Eficiencia obtenida para cada algoritmo evaluado a medida que escala el trabajo, para 4 y 8 cores respectivamente. En todas las gráficas las configuraciones fueron ordenadas según la carga de trabajo secuencial de la misma.

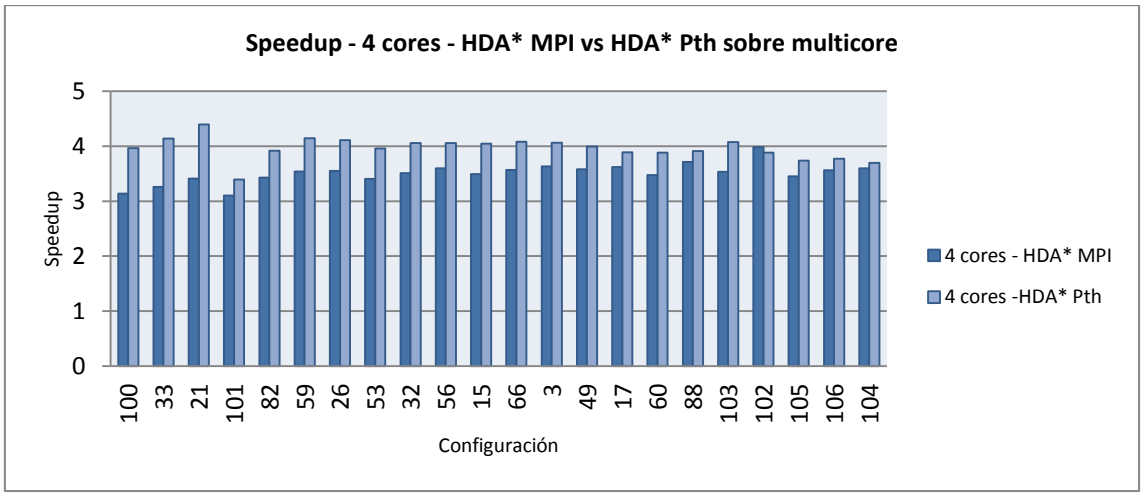


Figura 7.41. Comparación del Speedup obtenido para HDA* Pthreads y HDA* MPI con 4 cores y LNPT=210.

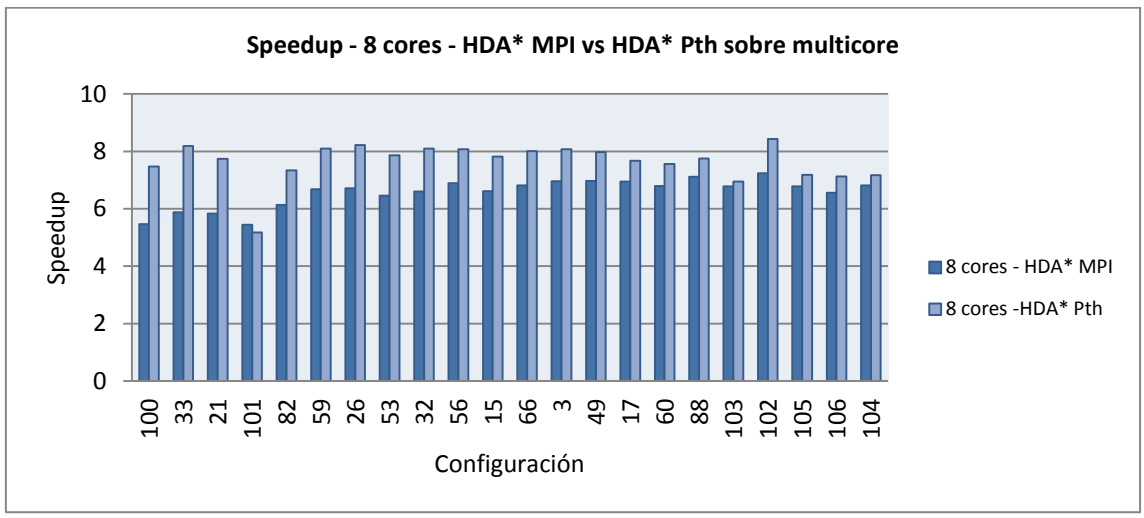


Figura 7.42. Comparación del Speedup obtenido para HDA* Pthreads y HDA* MPI con 8 cores y LNPT=210.

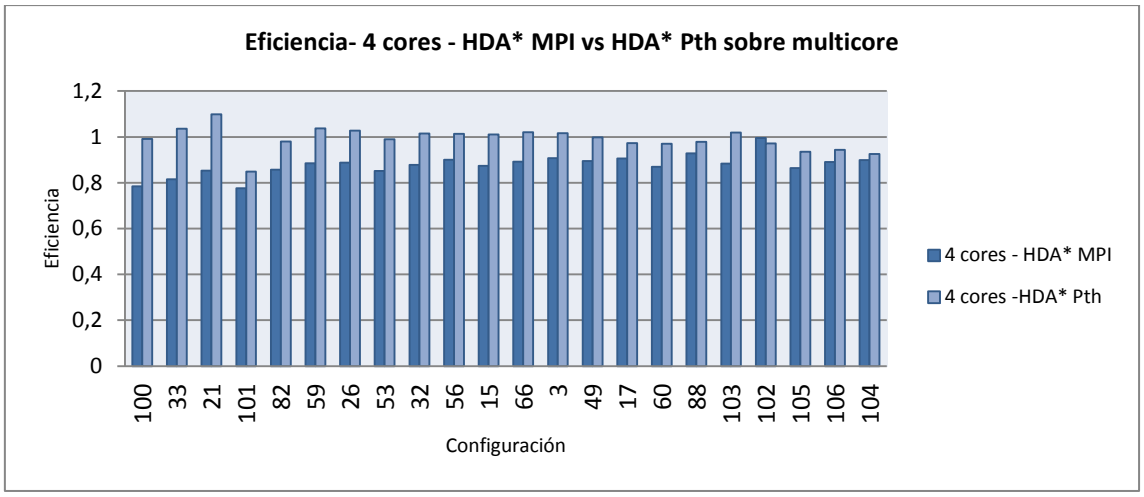


Figura 7.43. Comparación de la Eficiencia obtenida para HDA* Pthreads y HDA* MPI con 4 cores y LNPT=210.

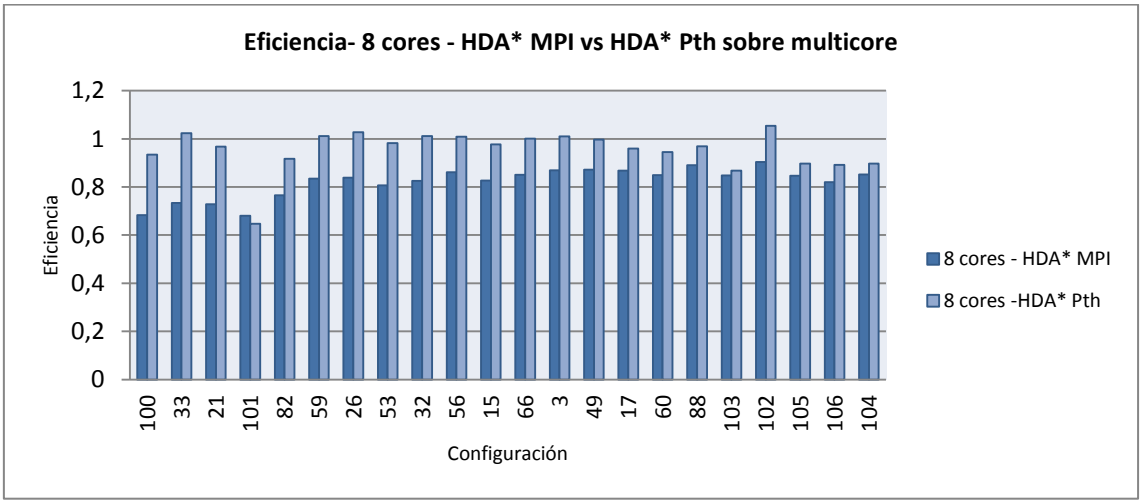


Figura 7.44. Comparación de la Eficiencia obtenida para HDA* Pthreads y HDA* MPI con 8 cores y LNPT=210.

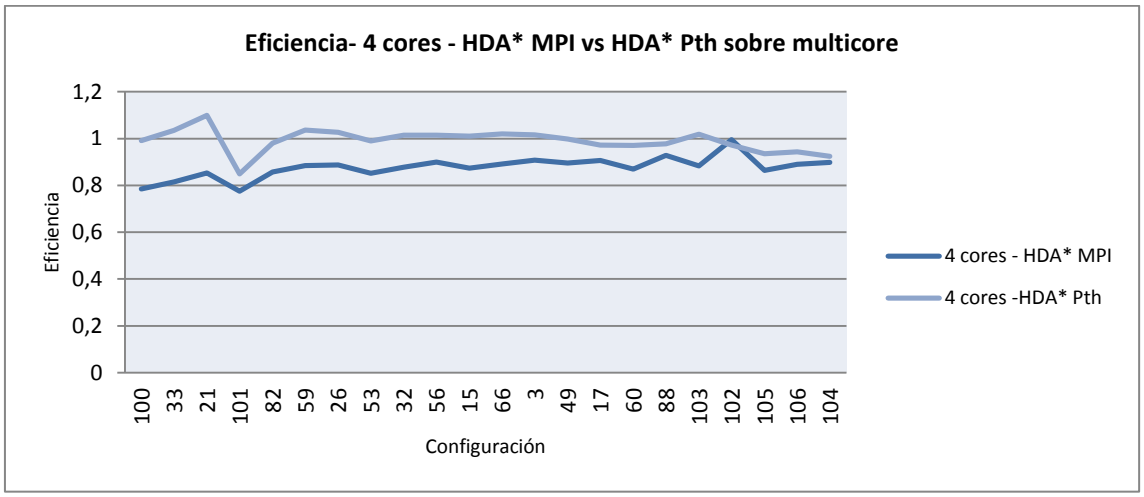


Figura 7.45. Eficiencia obtenida para HDA* Pthreads y HDA* MPI a medida que escala el trabajo para 4 cores.

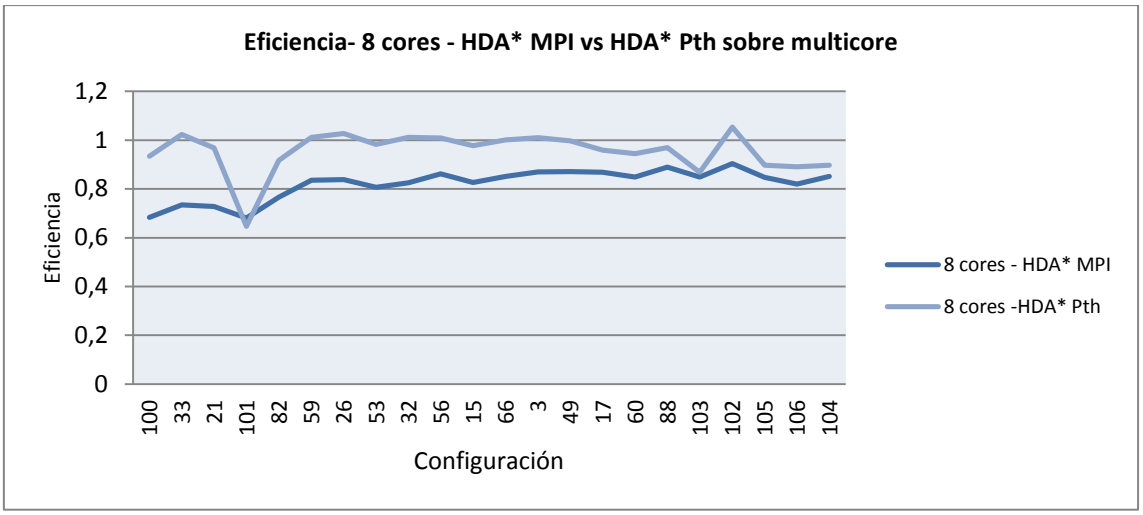


Figura 7.46. Eficiencia obtenida para HDA* Pthreads y HDA* MPI a medida que escala el trabajo para 8 cores.

De las gráficas comparativas se puede observar que, para la mayoría de los casos, el algoritmo HDA* Pthreads obtiene mejor rendimiento respecto a HDA* MPI cuando se ejecuta sobre una máquina multicore.

7.6 Discusión y conclusiones

En este capítulo se analizaron las implementaciones de los algoritmos propuestos en el Capítulo 6.

Para comenzar, se evaluó el rendimiento del algoritmo secuencial cuando se utilizan distintas heurísticas y se comprobó que la afinación de su calidad produce mejoras en el rendimiento. También se demostró que la incorporación del gestor de memoria Jemalloc, configurado para usar 256 *arenas*, acelera los tiempos de ejecución. Por último, se documentó que la técnica "*Memory Pool"* no trae beneficios en el algoritmo secuencial.

A continuación, se analizó el rendimiento (tiempo de ejecución) del algoritmo HDA* para memoria compartida (HDA* Pthreads) cuando se utiliza distintas técnicas de espera y la técnica *Memory Pool,* demostrando que:

- El modo de espera (activa o pasiva) no influye en el rendimiento del algoritmo.
- La técnica "*Memory Pool"* tiene efectos positivos sobre el rendimiento.

Siguiendo con el algoritmo HDA* Pthreads, se evaluó el efecto de los parámetros LNPI y LNPT sobre el rendimiento, demostrando lo siguiente:

- El parámetro LNPI, cuando su valor varía entre {1, 5, 50, 500}, no provoca cambios drásticos en el rendimiento del algoritmo. Al aumentar excesivamente su valor, sí se producen desmejoras significativas.

- El parámetro LNPT tiene influencia sobre el rendimiento. En particular para la arquitectura y configuraciones iniciales utilizadas el rendimiento óptimo se alcanza cuando se realizan intentos de transferencia luego de haber acumulado 210 nodos (LNPT = 210). Esto significa un aporte respecto al algoritmo original que tras cada generación de nodo intenta transferir el mismo.

A continuación, se comprobó que el no determinismo y asincronismo presentes en el algoritmo no provocan variaciones drásticas en el rendimiento de una ejecución a otra, cuando se mantienen los mismos datos de entrada y configuración de las estructuras.

Luego, se evaluó el rendimiento (*Speedup y Eficiencia)* obtenido por HDA* Pthreads, para LNPT=210, escalando la carga de trabajo y la arquitectura. De los resultados se concluyó que el algoritmo exhibe un comportamiento propio de un sistema paralelo escalable. Asimismo se analizaron los factores que tienen mayor influencia en el rendimiento (Desbalance de Carga y Overhead de Búsqueda).

Respecto al algoritmo HDA* para memoria distribuida (HDA* MPI), comenzó por evaluarse su comportamiento cuando se ejecuta sobre una máquina multicore. En particular, se analizó la influencia de los parámetros LNPI y LNPT, quedando demostrado lo siguiente:

- El parámetro LNPI, que varía la frecuencia de recepción de mensajes de trabajo y de costos de mejor solución, tiene influencia sobre el tiempo de ejecución cuando su valor alterna entre {1,5,50,500}. También se comprobó que su influencia es mayor para tamaños de mensaje de trabajo chico (valor bajo de LNPT). El valor óptimo de LNPI depende de la configuración inicial y valor de LNPT utilizado. Es importante destacar que en ningún caso LNPI = 1 optimizó el rendimiento, esto indica que el parámetro que se incorporó en la versión propia de HDA* para memoria distribuida es de importancia, siendo un aporte original en el área.
- El parámetro LNPT tiene influencia sobre el tiempo de ejecución, repercute en el grado de congestionamiento del medio de comunicación, actividad de los procesos y balance de calidad de los nodos. En particular para esta arquitectura multicore, el valor que mejora los resultados es 210 (mensajes de 8KB)

A continuación, se comprobó que el no determinismo y asincronismo presentes en el algoritmo no provocan variaciones drásticas en el rendimiento de una ejecución a otra sobre esta arquitectura, cuando se mantienen los mismos datos de entrada y configuración de las estructuras.

Luego, se analizó el rendimiento del algoritmo HDA* MPI cuando se ejecuta sobre una máquina multicore fijando LNPT en 210 y tomando en cuenta el valor de LNPI que optimiza el rendimiento para cada configuración y cantidad de cores, escalando la carga de trabajo y la arquitectura. De los resultados se concluyó que el algoritmo exhibe un comportamiento propio de un sistema paralelo escalable.

Asimismo, se ejecutó el algoritmo HDA* para memoria distribuida (HDA* MPI) sobre distintas configuraciones de cluster (ubicando 4 procesos por máquina u 8 procesos por máquina, variando la cantidad de máquinas) y distintas instancias de entrada.

Se evaluó el efecto de los parámetros LNPI y LNPT sobre el rendimiento para las pruebas que colocan 4 procesos por máquina, quedando demostrado que ambos parámetros tienen influencia sobre el tiempo de ejecución. En particular:

- El valor de LNPI que optimiza el rendimiento depende de la configuración inicial, la cantidad de procesos y el valor del parámetro LNPT (es decir, el tamaño del mensaje de trabajo). Es importante destacar que en ningún caso LNPI = 1 optimizó el rendimiento, esto indica que el parámetro que se incorporó en la versión propia de HDA* MPI es de importancia, siendo un aporte original en el área.
- En la mayoría de los casos el valor de LNPT que optimiza los tiempos de ejecución para cada configuración y cantidad de cores es 210, es decir mensajes de 8KB. Las excepciones ocurren con algunas configuraciones poco complejas que prefirieron LNPT=26 a medida que aumenta la cantidad de procesos, ya que con valores más altos de LNPT crece el *Overhead de Búsqueda*; y también con algunas de las configuraciones más complejas cuando se utilizan pocos procesos, las cuales alcanzan rendimiento óptimo con LNPT=1680.

Luego se evaluó el rendimiento (Speedup y Eficiencia) obtenido por el algoritmo para distintas configuraciones de cluster, aumentando la carga de trabajo y la cantidad de procesos/cores. Los resultados indican que el algoritmo HDA* MPI ejecutado sobre configuraciones de cluster que ubican 4 procesos por máquina exhibe un comportamiento propio de un sistema paralelo escalable.

Por último, se comparó el rendimiento y la cantidad de memoria utilizada por *HDA* Pthreads* y *HDA* MPI* cuando se ejecutan sobre una máquina multicore. Se concluyó que HDA* Pthreads obtiene mejor rendimiento, tanto para 4 y 8 cores, y utiliza menor cantidad de memoria RAM. Estos resultados indican que se obtendría un beneficio al convertir HDA* en una aplicación híbrida ya que utilizaría menor cantidad de memoria por máquina y mejoraría el rendimiento general.

CAPÍTULO 8: ALGORITMO PARALELO HDA* HÍBRIDO

En el Capítulo 5 se estudiaron diversos algoritmos que paralelizan A*, siendo HDA* (*Hash Distributed A**) (Kishimoto, et al., 2013) uno de los más difundidos en la actualidad. Este algoritmo está implementado utilizando únicamente la biblioteca MPI y sus autores realizan un análisis experimental del mismo sobre arquitecturas *multicore* y *cluster de multicore*.

Por otro lado, el trabajo (Burns, et al., 2010) estudia una implementación de HDA* utilizando threads y herramientas de sincronización provistas por la biblioteca Pthreads y analizan su rendimiento sobre una máquina *multicore*. Asimismo, presenta un algoritmo llamado *Parallel Best-NBlock First (PBNF)* cuya implementación no es sencilla y con el cual obtienen en ocasiones un s*peedup* muy superior al teóricamente posible*,* ya que la estrategia de paralelización utilizada provoca que se optimice el manejo de las estructuras *Lista Abierta* y *Lista Cerrada* respecto al algoritmo secuencial clásico A* contra el que comparan; sin embargo, exhibe mayor overhead de sincronización que la versión de HDA* presentada en el mismo trabajo.

Por su simpleza y buena escalabilidad el algoritmo HDA* es de interés actual. Por lo antes expuesto, en el Capítulo 6 se estudiaron las implementaciones de dos versiones propias del algoritmo HDA*, HDA* para memoria compartida (*HDA* Pthreads*) y HDA* para memoria distribuida (*HDA* MPI*), basadas en las propuestas de (Burns, et al., 2010) y (Kishimoto, et al., 2013) respectivamente. Luego, el análisis experimental realizado en el Capítulo 7 demostró que HDA* Pthreads utiliza menor cantidad de memoria y exhibe mejor rendimiento en comparación a HDA* MPI cuando se ejecuta sobre una máquina multicore.

Además, en el Capítulo 4 se relacionaron los distintos modelos de programación paralela (*paso de mensajes* y *variables compartidas*) con las arquitecturas sobre las cuales pueden ejecutarse las aplicaciones que los utilizan. Siguiendo esta línea, se hizo hincapié en las ventajas que podrían obtenerse al adaptar una aplicación programada puramente con paso de mensajes para que utilice ambos modelos, cuando la misma será ejecutada sobre un *cluster de multicore.*

Este capítulo propone un algoritmo HDA* híbrido, basado en lanzar un proceso por cada máquina del *cluster de multicore*, los cuales se encargarán de crear threads que serán responsables de realizar el procesamiento. El algoritmo utiliza comunicación a través de variables compartidas entre los threads de una máquina y comunicación por paso de mensajes entre los procesos residentes en distintas máquinas del *cluster*.

La Sección 8.1 estudia los distintos modelos de aplicaciones híbridas. La Sección 8.2 desarrolla distintas opciones para el algoritmo HDA* Híbrido propuesto. Por último, la Sección 8.3 presenta las conclusiones del capítulo.

8.1 Aplicaciones Paralelas Híbridas

Una aplicación paralela híbrida es aquella que combina dos o más modelos de programación paralela, lo cual puede ser provechoso desde el punto de vista del rendimiento obtenido cuando la aplicación paralela equivalente que sólo utiliza un modelo no se adapta totalmente a la arquitectura subyacente. Por ejemplo si se tiene un *cluster de multicore* y se ejecuta sobre éste una aplicación paralela que sólo utiliza OpenMP (añadiendo una capa de software que trate a la memoria físicamente distribuida como un único espacio de direcciones) o una aplicación paralela que sólo utiliza MPI, el sistema paralelo obtenido puede no ser óptimo en cuanto a rendimiento. (Torquati, et al., 2013)

Existen distintos tipos de aplicaciones híbridas que combinan comunicación y sincronización por paso de mensajes y por variables compartidas (Dongarra, et al., 2002):

- Sólo el thread maestro del proceso realiza la comunicación vía paso de mensajes y los demás threads únicamente se encargan del procesamiento de datos. En este sentido, se paraleliza mediante threads aquellas partes de la aplicación que no requieren comunicación entre distintos procesos.
- Cualquier thread del proceso puede enviar mensajes pero sólo un thread del proceso se encarga de la recepción de los mismos, el cual además será responsable de integrar los datos recibidos en la memoria compartida, notificar a los threads sobre la existencia de nuevos datos para procesar y crear nuevos threads en caso de ser necesario.
- Cualquier thread puede recibir mensajes dirigidos hacia el proceso. Esta alternativa es adecuada cuando cualquier thread del proceso puede trabajar sobre los datos entrantes. Será responsabilidad del programador la prevención de carreras entre ellos.
- Cualquier thread puede enviar mensajes a otro thread del sistema o recibir mensajes dirigidos a él específicamente. Estas aplicaciones deben tener en cuenta que las operaciones de comunicación punto a punto de MPI identifican procesos y no threads.

Entre las ventajas de utilizar técnicas híbridas se encuentran (Hager & Wellein, 2011):

- Reúso de datos en caches compartidas: los datos compartidos entre threads que residen en una misma máquina y que poseen algún nivel de cache en común, podrán ser cargados en cache ante el primer acceso por parte de un thread y luego podrán ser accedidos múltiples veces desde cache por los demás threads.
- Eliminación del overhead de comunicación introducido por MPI: el overhead de comunicación puede crecer rápidamente con el número de procesos y llevará a incrementar el tiempo de ejecución.
- Reducción de la cantidad de memoria utilizada por la aplicación: MPI reserva espacio para buffers que serán manejados por las capas de comunicación, el cual se incrementa con el número de procesos.

Asimismo algunas aplicaciones que utilizan sólo paso de mensajes pueden beneficiarse al adaptar el esquema de comunicación entre threads de la misma máquina para utilizar variables compartidas; o podrían exhibir también una aceleración en la convergencia del algoritmo.

8.2 Algoritmo HDA* Híbrido

8.2.1 Síntesis

El algoritmo HDA* Híbrido propuesto se basa en lanzar un proceso por cada máquina del *cluster de multicore*. Cada proceso inicializa el entorno MPI según el grado de interoperabilidad requerido entre threads y MPI[45]. El proceso 0 cargará el estado inicial, el estado final y la Tabla Zobrist desde archivo, y luego comunicará a los demás procesos la Tabla Zobrist y la tabla que indica la posición final de cada ficha.

Cada proceso inicializa aquellos datos que compartirán los threads que serán lanzados en dicha máquina. Por ejemplo: puntero a la mejor solución encontrada localmente, el costo de la mejor solución encontrada globalmente, colas de entrada, variable *fin* para comunicarse la finalización del cómputo, entre otras. Luego cada proceso crea los threads que se ejecutarán junto con el *thread maestro* sobre la máquina.

Cada thread posee su propia Lista Abierta y Lista Cerrada[46], inicialmente vacías. El thread comenzará inicializando datos propios y luego realizará una serie de iteraciones hasta que se le avise que se ha encontrado la solución óptima globalmente. En cada iteración realizará una serie de etapas que se verán en detalle más adelante. Cuando el thread genera un nodo, calcula la clave de *Zobrist* para conocer el identificador del proceso y del thread que debe procesarlo[47]. Cuando el nodo pertenece al thread que lo generó, dicho thread realizará la detección de duplicados e insertará el nodo en su Lista Abierta en caso que sea adecuado. Cuando el nodo pertenece a otro thread de la misma máquina, el esquema de comunicación es el mismo que aquel utilizado para el algoritmo HDA* Pthreads (estudiado en el Capítulo 6). Cuando el nodo pertenece a un thread que reside en otra máquina se debe establecer un *esquema de comunicación por trabajo vía paso de mensajes* (también se requiere comunicación entre procesos por otros motivos como la detección de terminación, comunicación de costos de soluciones y aviso de finalización de cómputo).

Cuando se encuentra la mejor solución globalmente, termina el cómputo por parte de los threads pero queda activo el *thread maestro* en cada máquina quien tendrá acceso a las Listas Cerradas conteniendo los nodos trabajados por los demás threads. Los procesos MPI comenzarán una etapa para recuperar la solución en forma distribuida, tal y como se realizó en el algoritmo HDA* MPI (Capítulo 6). Una vez obtenida la solución será informada por el proceso 0, y por último los procesos finalizan el entorno MPI[48].

[45] Utilizando MPI_Init_threads

[46] La Lista Cerrada será utilizada exclusivamente por el thread pero es globalmente accesible. De este modo, el *thread maestro* podrá actuar sobre estos datos para recuperar la solución una vez finalizado el cómputo.

[47] Con una clave de 64 bits, se puede identificar el proceso dueño con los 32 bits menos significativos y el thread dueño con los 32 bits más significativos.

[48] Utilizando MPI_Finalize

8.2.2 Esquema de comunicación vía mensajes

Como se estudió en el Capítulo 6, los mensajes enviados entre procesos en el algoritmo HDA* MPI se clasifican en:

- Mensaje de trabajo, que contiene nodos del grafo intercambiados para balancear la carga de trabajo y su calidad.
- Mensaje con costo de mejor solución encontrada por otro proceso del sistema.
- Mensaje correspondiente al algoritmo de detección de terminación (token).
- Mensaje de aviso de fin del cómputo.

En el algoritmo HDA* Híbrido la información descripta con anterioridad se puede dividir en dos categorías:

- General: concierne a todos los threads de la máquina.
- Particular: concierne a un thread de la máquina.

Los mensajes con costo de mejor solución encontrada, los mensajes correspondientes al token de terminación y los mensajes de aviso de fin de cómputo son generales, es decir la información compete a todos los threads de la máquina. Estos mensajes tendrán como destinatario a un proceso, por lo que los threads del mismo deben excluirse al recibir mensajes correspondientes a un tipo evitando así carreras entre ellos. Por ejemplo, el *thread maestro* será encargado de realizar recepciones de mensajes de finalización y token de terminación; por otra parte, los mensajes con costo de mejor solución pueden ser recibidos por cualquier thread, apoyándose en las primitivas de exclusión mutua.

El uso de mensajes generales reduce la cantidad de mensajes en tránsito respecto a HDA* implementado con MPI puramente. Se justificará utilizando como ejemplo el algoritmo HDA* MPI cuando trabaja con 8 procesos por cada una de las 7 máquinas del cluster, habiendo en total 56 procesos. En este caso cuando un proceso encuentra una solución, debe comunicar su costo a cada uno de los 55 procesos pares del sistema. En el esquema híbrido planteado existirían 7 procesos y cada uno tendría 8 threads; cuando un thread de una máquina encuentra una solución, envía un mensaje con su costo a cada proceso par – siendo 6 mensajes en total. Algo similar ocurre con el mensaje correspondiente al token de terminación distribuida, el cual será pasado de un proceso a otro. Utilizando el ejemplo anterior, en HDA* MPI son 56 procesos los que deben pasarse el token y en HDA* Híbrido sólo son 7. Por otro lado, cuando el proceso 0 detecta la terminación global del sistema, en HDA* MPI deberá enviar 55 mensajes avisando el fin del cómputo, en cambio en HDA* Híbrido sólo enviaría 6 mensajes.

Por otro lado, los mensajes de trabajo pueden ser generales o particulares según el *esquema de comunicación por trabajo vía paso de mensajes* elegido:

- Mensajes de trabajo con nodos para destinatarios múltiples: en un mensaje se agrupan nodos que pertenecen a distintos threads del proceso destinatario. Utilizando este esquema el mensaje será de categoría general, será destinado a un proceso particular y los threads del proceso destinatario deberán excluirse al intentar recibir mensajes de este tipo

para evitar carreras entre ellos. Aquel thread que reciba el mensaje deberá detectar a qué thread corresponde cada nodo e incluirlo en la cola de salida propia correspondiente, para luego intentar transferir los nodos residentes en las colas de salida no vacías a las colas de entrada correspondientes.

- Mensajes de trabajo con nodos para destinatario particular: el mensaje de trabajo contiene nodos que pertenecen a un thread particular de un proceso. En este caso el mensaje será de carácter particular. Este esquema permite una comunicación *entre threads* del sistema, en lugar de procesos.

Además el esquema de comunicación deberá definir si los threads de una máquina usarán *buffers comunes* para acumular nodos dirigidos a un proceso o un thread particular de otra máquina, requiriendo en este caso exclusión mutua para el acceso a cada buffer, o si cada thread tendrá *buffers locales* sobre los que acumulará nodos dirigidos a un proceso o thread particular de otra máquina, siendo los buffers de uso exclusivo por lo que no requiere mecanismos de sincronización.

De lo anterior surgen cuatro estrategias:

- Estrategia buffers locales y destinatario particular: cada thread del proceso MPI tiene un buffer de envío de trabajo para cada thread par del sistema.
- Estrategia buffers locales y destinatarios múltiples: cada thread del proceso MPI tiene un buffer de envío de trabajo para cada proceso par del sistema.
- Estrategia buffers comunes y destinatario particular: los threads del proceso MPI comparten una estructura de buffers de envío de trabajo, habiendo uno para cada thread par del sistema.
- Estrategia buffers comunes y destinatarios múltiples: los threads del proceso MPI comparten una estructura de buffers de envío de trabajo, habiendo uno para cada proceso par del sistema.

Las alternativas primera y tercera fueron descartadas por razones que se exponen a continuación. MPI permite intercambio de mensajes entre procesos y no entre threads, esto se debe a que las operaciones de envío de mensajes poseen un parámetro que identifica al proceso destino. Para sustentar las estrategias que utilizan *mensajes de trabajo para destinatario particular* el programador debería implementar *tags* especiales de manera que un mensaje que arriba a un proceso pueda ser distinguido como dirigido a un thread particular –debiendo también identificarse el *tipo de mensaje* en el *tag*. La implementación de este estilo de comunicación puede hallarse en (Intel, 2008). Sin embargo, cuando el *thread maestro* del proceso se encuentre ocioso y deba esperar por mensajes de trabajo, token de terminación, costo de mejor solución o aviso de finalización del cómputo no podrá usar el *wildcard* MPI_ANY_TAG estableciendo que espera un mensaje de cualquier tipo ya que podría cometerse el error de recibir un mensaje de trabajo dirigido a otro thread. Incluso podría darse una condición de carrera entre el *thread maestro* y el thread al cual va dirigido el mensaje de trabajo ya que cualquiera puede quedar bloqueado cuando, habiendo detectado la llegada del mensaje y estando a punto de ejecutar una operación de recepción, el mensaje ha sido recibido por el otro thread.

8.2.3 Esquema de comunicación vía variables compartidas

Los threads de una máquina comunicarán los nodos cuyo procesamiento corresponde a alguno de ellos utilizando el esquema de comunicación de HDA* Pthreads (estudiado en el Capítulo 6) que se basa en el uso de colas de entrada globales y colas de salida locales para evitar bloqueos.

Además los threads de la máquina deberán comunicarse para: detectar la terminación local del cómputo, es decir el estado en que todos los threads de la máquina se encuentran ociosos y no hay nodos pendientes de ser consumidos en las colas de entrada; conocer los costos de mejores soluciones encontradas al momento; conocer el final del cómputo.

8.2.4 Detección de terminación local y Detección de terminación global.

La detección de terminación se realizará en dos etapas. La primera etapa consiste en detectar la terminación local del cómputo entre los threads de una máquina, es decir el estado en el cual todos los threads de la máquina están ociosos y no quedan nodos en las colas de entrada pendientes de ser consumidos. Una vez detectado el fin del cómputo local, el *thread maestro* de cada máquina comenzará la segunda etapa que consiste en detectar terminación global, es decir el estado en el cual no existen mensajes de trabajo en tránsito y todos los threads del sistema están ociosos. Se aplicarán para esto los conceptos estudiados en los Capítulos 5 y 6, en los cuales se detalla la forma de detectar terminación de cómputo distribuido en arquitecturas de memoria distribuida y de memoria compartida.

Para detectar terminación localmente, los threads de la máquina deben mantener cada uno un estado (*estado_thread*) que represente un color (*blanco* o *negro*) y un contador de nodos transferidos y consumidos (*contador_thread*). Al principio *estado_thread* será inicializado en blanco y *contador_thread* en 0. Ante operaciones de transmisión de nodos a otro thread o consumo de nodos desde su cola de entrada, el thread deberá actualizar dichas variables (tal y como especifica el algoritmo de Dijkstra y Safra estudiado en el Capítulo 5), con las siguientes modificaciones: cada vez que el thread deposita nodos en la cola de entrada de otro thread incrementará su contador en la cantidad de nodos depositados; cada vez que un thread consume nodos de su cola de entrada disminuye su contador en la cantidad de nodos consumidos.

La suma de los contadores de cada thread es la cantidad de nodos residentes en todas las colas de entrada, que están pendientes para ser consumidos. Dicha suma será calculada en forma distribuida entre todos los threads. Para ello, los threads compartirán los datos correspondientes a un token de terminación local, compuesto por un estado, un contador y un identificador del thread que posee el token al momento. El token de terminación se transmite de un thread a otro, los cuales se organizan en forma de anillo.

Cuando un thread ocioso posee el token local debe actualizar sus datos, según corresponda siguiendo el algoritmo de terminación, y pasarlo al thread siguiente. Esto último implica cambiarle el valor al campo dueño del token avisándole al thread sucesor que es el nuevo dueño. Por otra parte, el thread 0 realiza la tarea diferenciada de verificar si se dan las

condiciones de terminación local, y en caso contrario inicia una nueva vuelta para la detección de terminación.

Por otra parte, para detectar la terminación global del cómputo se utilizará el algoritmo clásico de Dijkstra y Safra (Capítulo 5). Cada proceso tendrá un estado (*estado_proceso*) y un contador (*contador_proceso*) de mensajes enviados y recibidos. Los *threads maestro* de cada proceso se pasarán el mensaje *token de terminación* que contiene información básica (estado y contador) e información adicional (costo de la mejor solución global, dirección de memoria de la mejor solución global, identificador del proceso que encontró la mejor solución global) que permitirá realizar la recuperación de la solución óptima encontrada. El orden de transmisión del token es en forma de *anillo*. Ante operaciones de envío o recepción de mensajes por parte de un thread del proceso se deberán actualizar las variables *contador_proceso* y *estado_proceso*, tal como especifica el algoritmo Dijkstra y Safra. Dado que cualquier thread del proceso puede enviar o recibir un mensaje, se deberá mantener la consistencia de dichas variables.

Cuando el *thread maestro* de una máquina, habiendo detectado terminación local, posee el token de terminación global, deberá actualizar dicho token y pasarlo al siguiente proceso. El *thread maestro* del proceso 0 tendrá la tarea diferenciada de detectar si se dan las condiciones para alcanzar la terminación global, en dicho caso avisa a los demás procesos vía mensaje y a los demás threads de su misma máquina cambiando el valor de una variable compartida *fin*. En caso de no detectar terminación, iniciará una nueva vuelta.

8.2.5 Variables globales a los threads de un proceso

- id_proceso: identificador o *rank* del proceso.
- Colas de entrada: una cola de entrada por cada thread en ejecución sobre la máquina.
- mejor_solución y costo_mejor_solución: llevarán el puntero a la mejor solución hallada localmente y el costo de la mejor solución conocida respectivamente. Ambas deberán estar protegidas por un lock ya que dos o más threads de la máquina podrían querer actualizarlas en el mismo instante.
- Información referente al token local de terminación (contador, estado y dueño). No serán protegidas dado que sólo un thread a la vez podrá actualizar estos valores.
- Información referente al algoritmo de detección de terminación global (estado_proceso y contador_proceso). Estas variables deberán ser actualizadas ante operaciones de envío y recepción de mensajes por parte de los threads, por lo tanto deberán ser protegidas para mantener su consistencia.
- Variable *fin*, la cual indicará el fin del cómputo global.
- Buffers de envío de trabajo, en caso de utilizar la estrategia de “buffers comunes”, en la que los threads comparten los buffers para acumular nodos dirigidos a un mismo proceso. Habrá un buffer para cada proceso del sistema. Dado que varios threads pueden querer depositar nodos en el mismo buffer al mismo tiempo, cada buffer estará protegido por un lock.

8.2.6 Variables locales a cada thread de un proceso

- id_thread: identificador del thread.
- Lista Abierta y Lista Cerrada[49].
- Colas de salida: una cola de salida para cada thread par en la máquina.
- Información referente al algoritmo de detección de terminación local (estado_thread y contador_thread). Estas variables deberán ser actualizadas ante operaciones transferencia y consumo de nodos local.
- Buffers de envío de trabajo, en caso de utilizar la estrategia de "buffers locales". El thread tendrá un buffer para cada proceso del sistema donde acumulará nodos dirigidos a un mismo destinatario.

8.2.7 Procesamiento

El procesamiento estará dividido en una serie de etapas, algunas de estas serán ejecutadas únicamente por el *thread maestro* de cada máquina (inicialización y finalización) y otras por todos los *threads* existentes en el sistema (búsqueda).

Entre las etapas a ejecutar por el *thread maestro* de cada máquina se encuentran:

- Inicialización del entorno MPI.
- Inicialización de variables globales compartidas por los threads de la máquina.
- Creación de threads. Una vez realizada, cada uno pasará a ejecutar el algoritmo de búsqueda (inclusive el thread maestro).
- Recuperación distribuida de la solución una vez terminado el cómputo.
- Finalización del entorno MPI.

Entre las etapas a ejecutar por cada *thread* del sistema se encuentran la inicialización de variables locales y el procesamiento (búsqueda) hasta recibir el aviso de finalización del cómputo global (que provocará el cambio de valor en la variable *fin*). La búsqueda consiste en las siguientes sub-etapas:

- *Recepción de trabajo:* los nodos pueden ingresar a través de dos vías (a) por vía local, es decir son nodos depositados en la cola de entrada del thread (b) por vía global, es decir a través de mensajes dirigidos al proceso. Primero, el thread realiza un intento para consumir nodos de su cola de entrada, tal y como se realizó en HDA* Pthreads. Luego, verifica la llegada de mensajes de trabajo y en caso de constatar un arribo lo recibe; estas dos acciones requieren exclusión entre threads para garantizar la no ocurrencia de carreras, pero en ningún momento se obliga al thread a esperar. Cuando el thread recibe un mensaje de trabajo, evaluará cada nodo para determinar a qué thread de la máquina corresponde su procesamiento y lo almacenará en la cola de salida local correspondiente (en caso de estar dirigido a otro thread) o realizará la detección de duplicados para luego insertarlo en su Lista Abierta cuando sea adecuado (en caso de que sea un nodo propio).

[49] La Lista Cerrada será declarada en el ámbito global para que, luego de detectar terminación, el thread maestro de la máquina pueda acceder a sus datos. Aun así, durante la etapa de búsqueda será de uso exclusivo del thread.

Cuando el thread detecta que la cola de salida para un thread determinado alcanzó el límite LNPT_MC (Límite de Nodos por Transferencia en Memoria Compartida) intentará transferir todos los nodos existentes en la misma a la cola de entrada del thread destino.

- *Recepción de costos de soluciones encontradas por otros procesos:* el thread verifica el arribo de mensajes con costo de mejor solución encontrada por otro proceso y en caso de constatar un arribo lo recibe. Estas dos acciones requieren exclusión entre threads, pero en ningún momento se obliga al thread a esperar. En caso de recibir el mensaje, actualiza cuando sea adecuado la variable *costo_mejor_solución* (luego de tomar el lock respectivo). Asimismo, el thread debe proceder a vaciar su Lista Abierta cuando el nodo abierto de menor costo tiene valor $\hat{f}$ mayor o igual a *costo_mejor_solucion*.

- *Procesamiento de nodos:* el thread trabajará a lo sumo sobre LNPI (Límite de Nodos por Iteración) nodos de su Lista Abierta. Cuando extrae un nodo verifica si su costo es al menos *costo_mejor_solución,* en cuyo caso vacía la Lista Abierta ya que los nodos en ella conducirían a soluciones sub-óptimas. En caso contrario, comprueba si el nodo representa la solución y en tal situación vacía su Lista Abierta y actualiza *mejor_solución* y *costo_mejor_solución* luego de haber tomado el lock que protege dichos datos y consultado nuevamente si el costo de la solución hallada no supera *costo_mejor_solución* (tal y como se especificó para el algoritmo HDA* Pthreads). En caso de haber encontrado una nueva mejor solución parcial, el thread debe enviar un mensaje a cada proceso del sistema conteniendo el costo de la solución encontrada.
  Cuando el nodo extraído no representa la solución, el thread lo ubica en la Lista Cerrada y lo expande generando sus sucesores. Para cada nodo sucesor calcula la clave de *Zobrist* con el objetivo de conocer el identificador del proceso y del thread que debe procesarlo. Cuando el nodo pertenece al thread que lo generó, dicho thread realizará la detección de duplicados e insertará el nodo en la Lista Abierta en caso que sea adecuado. Cuando el nodo pertenece a otro thread de la misma máquina, el esquema de comunicación es el mismo que aquel utilizado para el algoritmo HDA* Pthreads (estudiado en el Capítulo 6), es decir el nodo es depositado en la cola de salida local correspondiente y en caso de que dicha cola supere el límite LNPT_MC se realizará un intento para transferir los nodos residentes a la cola de entrada del thread destino[50]. Cuando el nodo pertenece a un thread que reside en otra máquina, copia sus datos en el buffer de envío correspondiente al proceso destino, y en caso que el buffer supere el límite LNPT_MD (Límite de Nodos por Transferencia en Memoria Distribuida) realizará el envío del mensaje en forma asincrónica – tal y como se especificó en el Capítulo 6. En caso de utilizar la estrategia de "buffers comunes" debe tomar el lock asociado al buffer para hacer el depósito del nodo y envío de mensaje.
  La transferencia de un nodo vía memoria compartida implica sólo una copia de punteros. La transferencia de un nodo vía mensaje implica la copia de los datos en el buffer, liberación del espacio dinámico que ocupaba en el proceso origen (free), asignación de

[50] Cuando un thread es el primero en depositar nodos sobre la cola de entrada de otro thread, es decir en ese momento la cola de entrada está vacía, debe realizar un aviso por si éste último estaba ocioso esperando el depósito.

espacio dinámico en el proceso destino (malloc) y copia de datos desde el espacio buffer al espacio dinámico en el proceso destino.

- *Etapa espera:* el thread ingresa a la etapa de espera cuando quedó ocioso por haberse vaciado su Lista Abierta en alguna etapa previa. Las tareas a realizar son:

  - *Transferencia de nodos residentes en buffers no vacíos:* cuando se utiliza la estrategia "*buffers locales"*, el thread envía *mensajes de trabajo* para aquellos procesos destino cuyo *buffer de envío de trabajo* contenga nodos que no fueron comunicados. Cuando se utiliza la estrategia "*buffers comunes"*, el último thread de la máquina en entrar a la etapa de espera será el encargado de realizar el envío de mensajes de trabajo conteniendo nodos residentes en buffers no vacíos[51]. En cualquier caso, el envío de mensajes es asincrónico.
  - *Transferencia de nodos residentes en colas de salida no vacías*: el thread realizará el envío de los nodos residentes en las colas de salida no vacías en forma bloqueante ya que está ocioso y no tiene trabajo.
  - Espera local: el thread esperará por los siguientes eventos locales hasta recibir el aviso de finalización global del cómputo (a través de un cambio de valor en la variable *fin*) o poseer trabajo en su Lista Abierta.
    - Llegada del token local correspondiente al algoritmo de detección de terminación.
    - Depósito de trabajo en la cola de entrada propia.

    Adicionalmente, el *thread maestro* terminará esta etapa al detectar el fin del cómputo local. La manera de proceder ante estos eventos es idéntica a la que se explicó en el Capítulo 6 para el algoritmo HDA* Pthreads.

  - Espera global (sólo para el *thread maestro*): cuando el *thread maestro* de la máquina detectó el fin del cómputo local, esperará por alguno de los siguientes mensajes (utilizando la función MPI_Probe):
    - Mensaje con trabajo. El thread deberá proceder como se explicó con anterioridad en este capítulo, es decir depositará los nodos pertenecientes a él mismo en su Lista Abierta cuando sea adecuado y colocará los nodos pertenecientes a otros threads de la máquina en la cola de salida local correspondiente, para luego transferir su contenido a las colas de entrada respectivas.
    - Mensaje conteniendo el token de terminación global.
    - Mensaje de aviso de fin del cómputo global.
    - Mensaje con costo de mejor solución.

    La etapa de espera global finaliza cuando se recibió un mensaje de trabajo, ya que activaría nuevamente a uno o más threads de la máquina, o al recibir el fin del cómputo global.

    En los últimos tres casos, el *thread maestro* debe proceder como se explicó en el Capítulo 6 para el algoritmo HDA* MPI. Cuando el thread maestro recibe el mensaje de

[51] Otra alternativa sería que cada thread ocioso envíe los nodos contenidos en buffers no vacíos, pero esto entorpecería el trabajo de los threads que actualmente se encuentran activos.

finalización global del cómputo, debe dar aviso a los demás threads de la máquina, que se encuentran esperando este evento.

Una vez que se detectó el fin del cómputo, el *thread maestro* de cada máquina quedará activo y se comunicará con sus pares para realizar la recuperación de la solución.

El envío y recepción de mensajes de trabajo por parte de un thread implica actualizar las variables *estado_proceso* y *contador_proceso*, tal y como lo especifica el algoritmo de terminación de Dijkstra y Safra (Capítulo 5).

Cuando un thread consume nodos desde su cola de entrada o transfiere nodos a la cola de entrada de otro thread, debe actualizar las variables *estado_thread* y *contador_thread* tal y como se especificó anteriormente en el capítulo.

8.3 Discusión y conclusiones

En este capítulo se analizaron distintos modelos de aplicaciones paralelas híbridas y las ventajas que podrían obtenerse al adaptar una aplicación paralela programada utilizando un único paradigma de programación paralela hacia un esquema híbrido.

Los buenos resultados obtenidos, tanto en rendimiento como en uso de memoria, por el algoritmo *HDA* Pthreads* respecto al algoritmo *HDA* MPI* cuando se ejecutan sobre una máquina multicore indican que se obtendría un beneficio al convertir HDA* en una aplicación híbrida, cuando la arquitectura subyacente es un *cluster de multicore.*

Por lo anterior, se sentaron las bases para el algoritmo HDA* Híbrido y se analizaron los beneficios potenciales que traería esta adaptación:

- *Reducción de mensajes enviados:* en particular se disminuye la cantidad de mensajes catalogados *generales*, que son aquellos que transportan el aviso de fin del cómputo, el token correspondiente al algoritmo de detección de terminación y el costo de solución parcial encontrada.
- *Reducción de uso de memoria en cada máquina:* al disminuir la cantidad de procesos, disminuye la cantidad de memoria reservada por MPI para sustentar las comunicaciones. Asimismo, los threads de una máquina pueden compartir una única copia de estructuras comunes, lo cual reduce la cantidad de memoria utilizada y permite agilizar el acceso a las mismas cuando ya están cargadas en caches compartidas por varios threads.
- *Eliminación de la serialización de datos para la comunicación de nodos dentro de cada máquina:* la transferencia de un nodo vía mensaje implica la copia de los datos en el buffer de envío, liberación del espacio dinámico que ocupaba en el proceso origen (free), asignación de espacio dinámico en el proceso destino (malloc) y copia de datos desde el espacio de buffer al espacio dinámico en el proceso destino; en cambio la transferencia de un nodo vía memoria compartida implica sólo una copia de punteros. La adaptación hacia un esquema híbrido implica tener N threads en cada máquina en vez de N procesos, los cuales se comunicarían vía memoria compartida eliminando la necesidad de serializar datos a ser comunicados entre threads de la máquina.

Entre las desventajas visibles se encuentra la complejidad de programación.

CAPÍTULO 9: CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURO

En esta tesis se estudiaron diversos algoritmos de búsqueda sobre grafos que permiten encontrar soluciones a problemas combinatorios. El trabajo se centró en la implementación de dos versiones propias del algoritmo paralelo HDA*: HDA* MPI y HDA* Pthreads. La primera es apta para arquitecturas de memoria distribuida y la segunda para arquitecturas de memoria compartida. Ambas versiones incorporan parámetros y/o técnicas que optimizan el rendimiento respecto a los algoritmos originales propuestos por otros autores.

Se analizó el rendimiento obtenido (speedup y eficiencia) por HDA* Pthreads y HDA* MPI cuando se ejecutan sobre una máquina multicore, y el alcanzado por HDA* MPI cuando se ejecuta sobre un cluster de multicore, caracterizando en cada caso la influencia de los parámetros de cada algoritmo y los valores que optimizan los tiempos de ejecución. El análisis se realizó tomando como caso de estudio el problema *15-Puzzle*, utilizando instancias de entrada de distinta complejidad y variando la cantidad de procesadores de la arquitectura. El estudio realizado demuestra que ambos algoritmos exhiben un comportamiento propio de un sistema escalable. También, se presentó un análisis de cantidad de memoria utilizada por cada algoritmo cuando se ejecutan sobre una máquina multicore.

Los resultados aportados por esta tesis sugieren que se obtendría un potencial beneficio al adaptar el algoritmo HDA* para convertirlo en una aplicación híbrida cuando será ejecutado sobre un *cluster de multicore*, de forma que utilice herramientas de comunicación y sincronización por memoria compartida entre los threads que se ejecutan en la misma máquina, y paso de mensajes para comunicar procesos que residen en distintas máquinas del cluster. En consecuencia, se sentaron las bases para el algoritmo HDA* Híbrido.

Entre los trabajos a realizar en un futuro se encuentran:

- Experimentación con otras *funciones* de asignación de estados a procesos/threads:

Los algoritmos presentados en esta tesis utilizan una *función hash global* para asignar cada estado a un procesador, con la cual se logra la detección y poda de duplicados *absoluta,* ya que los nodos que representan a un mismo estado serán asignados al mismo procesador, y se balancea la carga de trabajo dinámicamente. La denominación "global" implica que los nodos generados por un procesador pueden pertenecer a cualquier otro procesador del sistema, requiriendo una comunicación global.

En el Capítulo 5 se comentó el trabajo (Mahapatra & Dutt, 1993), que estudia las desventajas que trae la utilización de esta estrategia ya que genera mayor cantidad de conflictos en la red e incrementa la latencia, para el problema *TSP* cuando se ejecuta sobre un multicomputador Hipercubo. En particular, propone el uso de la técnica *Local Hashing* para limitar la comunicación de nodos a un subconjunto de procesadores de la arquitectura. Sin embargo, los problemas que se adaptan a esta técnica son aquellos donde los nodos que representan un mismo estado aparecen en el mismo nivel, no siendo el caso del *Puzzle*.

Asimismo, el Capítulo 5 repasa el trabajo (Burns, et al., 2010) donde se analiza la misma problemática, concentrándose en el algoritmo HDA* para memoria compartida cuando se ejecuta sobre una máquina multicore. Los autores indican que ante la expansión de un nodo por parte de un thread, los sucesores generados son asignados a cualquier thread del sistema, cuando podría asignarse la mayor parte al thread que los generó o a un subconjunto de threads vecinos, disminuyendo la comunicación y contención en el acceso a las colas de entrada.

Para resolver este problema, los autores proponen abstraer el espacio de estados, dividiéndolo en bloques o *estados abstractos*. De esta manera, en vez de asignar estados individuales a los threads, se le asignan *estados abstractos* (los cuales están compuestos de varios estados individuales). Cuando un thread genera los sucesores de un nodo, aquellos que pertenezcan al mismo estado abstracto que su padre serán asignados a éste mismo thread. Asimismo, se ejemplificó una posible abstracción para el problema del *Puzzle*.

Se plantea como trabajo futuro aplicar la técnica de abstracciones al algoritmo HDA* para memoria compartida presentado en esta tesis y realizar un análisis de la escalabilidad del mismo.

Del mismo modo, está técnica puede aplicarse a los algoritmos HDA* para memoria distribuida y HDA* Híbrido presentados en esta tesis. Se propone utilizar la *función de abstracción* para asignar *estados abstractos* a las máquinas. De esta manera, cuando un proceso/thread de una máquina genere un nodo perteneciente al mismo *estado abstracto* que su padre, el procesamiento de dicho nodo corresponderá a algún proceso/thread de la misma máquina, lo que posibilitará limitar las transferencias vía red. Luego, podría utilizarse una *función hash global* para asignar dicho estado a un proceso/thread específico de la máquina para delegarle su procesamiento.

- Experimentación con otros problemas:

En el trabajo (Kumar, et al., 1988) se caracterizan los problemas *Puzzle*, *VCP* y *TSP* según el histograma de costos que generan durante la búsqueda. Dicho histograma se construye contabilizando para cada costo (eje x), los nodos procesados con dicho costo durante la búsqueda (eje y). Los problemas *Puzzle* y *VCP* poseen características similares, a saber: a medida que la búsqueda avanza se incrementa en forma abrupta la cantidad de nodos con igual costo. Esta característica provoca que se encuentre una gran cantidad de nodos con costo igual al de la solución óptima, lo cual lleva a veces a la obtención de *Overhead de Búsqueda* negativo. En contraste a lo anterior, el problema *TSP* exhibe una baja cantidad de nodos por cada costo procesado. Estas diferencias provocan que los problemas *Puzzle* y *VCP* se adapten mejor a estrategias distribuidas, ya que la pérdida de la guía heurística global no afectará en gran medida.

Se plantea estudiar el rendimiento alcanzado por los algoritmos presentados en esta tesis utilizando el problema TSP como caso de estudio.

- Analizar el comportamiento de los algoritmos de búsqueda presentados sobre otras arquitecturas.

Se pretende estudiar el comportamiento de los algoritmos presentados sobre máquinas multicore con mayor cantidad de procesadores. Asimismo, se propone analizar el rendimiento del algoritmo HDA* para memoria distribuida sobre *clusters* con características distintas al utilizado en esta tesis (mayor cantidad de máquinas, máquinas con distinta arquitectura, otro tipo de interconexión de red, etc).

- Predicción de rendimiento de los algoritmos de búsqueda presentados

Las técnicas de predicción de rendimiento permiten conocer el grado de influencia que tienen los parámetros del algoritmo y la máquina paralela sobre el rendimiento, sin tener que realizar múltiples ejecuciones para cada nueva instancia de entrada y grupos de parámetros.

Los algoritmos de búsqueda son aplicaciones cuyo rendimiento depende del tamaño del problema y de los datos de entrada en sí, por lo tanto resultan difíciles de predecir. Por otro lado, al paralelizar dichos algoritmos se añaden más factores que producen incertidumbre sobre el rendimiento alcanzable, tales como: el número de procesadores, el balance de carga dinámico, la comunicación entre procesos/threads, la contención en el acceso a recursos de hardware y software, entre otros.

Existe una metodología general (Fritzsche, et al., 2008) que permite predecir el rendimiento de este tipo de aplicaciones, la cual se basa en recolectar tiempos de ejecución para un número significativo de instancias del problema. Luego, se apoya en la minería de datos para buscar patrones y/o relaciones entre los datos recolectados con el objetivo de detectar los parámetros principales que afectan el rendimiento. Dichos parámetros se modelan numéricamente produciendo una fórmula analítica del tiempo de ejecución. A partir de la ecuación obtenida se puede predecir cómo se comportará el algoritmo para una nueva instancia de entrada.

- Adaptación de los algoritmos para encontrar soluciones *w-admisibles*

Dependiendo del estado inicial, la búsqueda de una solución óptima puede tomar un tiempo de cómputo excesivo, aun aplicando paralelismo. En algunos casos puede ser aceptable o preferible una solución sub-óptima que pueda ser alcanzada rápidamente.

Los algoritmos de búsqueda presentados en esta tesis pueden adaptarse para reducir la complejidad en tiempo, procesando menor cantidad de nodos comprometiendo en parte la calidad de la solución obtenida.

Weighted A* (WA*) es una generalización de A* (Pohl, 1970). Se basa en utilizar una función de costo $\hat{f}(n) = w_g \hat{g}(n) + w_h \hat{h}(n)$, ponderando el valor de $\hat{g}$ y $\hat{h}$ con un peso constante w_g y w_h respectivamente. Si se define $w = w_h/w_g$se obtiene una función equivalente $\hat{f}(n) = \hat{g}(n) + w\,\hat{h}(n)$. Dar a w un valor mayor a 1 provoca que la búsqueda tome la dirección que se estima más inmediata hacia una solución, ya que favorece el procesamiento de los nodos cercanos a la solución (con menor valor de $\hat{h}$). Aun siendo $\hat{h}$ admisible, al añadir el peso la función de costo se vuelve inadmisible, por lo que las soluciones encontradas pueden no ser

óptimas. Por otro lado, se conoce que si se utiliza una heurística $\hat{h}$ admisible, la solución encontrada por WA* será de costo menor o igual a $w\ C^*$, siendo C* el costo de la solución óptima para la instancia, es decir será *w-admisible*. (Ebendt & Drechsler, 2009)

A medida que se incrementa el valor de w, se encontrará la solución más rápido (procesando menor cantidad de nodos) a expensas de empeorar la calidad de la solución obtenida. (Korf, 1993)

Se propone adaptar los algoritmos presentados en esta tesis para encontrar soluciones subóptimas, evaluando la calidad de las soluciones obtenidas y el rendimiento alcanzado.

- Implementación del algoritmo HDA* Híbrido.

El Capítulo 8 de esta tesis sienta las bases para el algoritmo HDA* Híbrido y analiza las ventajas que traería su ejecución sobre un *cluster de multicore*, cuando se lo compara contra el algoritmo HDA* para memoria distribuida presentado.

Se propone implementar el algoritmo HDA* Híbrido, utilizando las bibliotecas MPI y Pthreads, y realizar un análisis de rendimiento para distintas configuraciones de cluster, variando la complejidad de las instancias iniciales. Luego, se plantea comparar los resultados obtenidos contra los alcanzados por el algoritmo HDA* MPI, con el propósito de cuantificar el beneficio alcanzado. Asimismo, se propone realizar una comparación de la cantidad de memoria utilizada por ambos algoritmos.

APÉNDICE A: CORRECTITUD DEL ALGORITMO DE VERIFICACIÓN DE SOLUBILIDAD PARA EL N^2-1 PUZZLE

Como se ha expuesto en la Sección 3, el espacio de estados para el problema del Puzzle N^2-1 crece en orden factorial, siendo su tamaño $N^2!$ debido a que cualquier ordenamiento de las fichas puede ser tomado como un tablero válido. Sin embargo, el espacio de estados posee dos componentes conexas de tamaño $N^2!/2$ y sólo existirá solución al problema si ambos estados (inicial y final) se encuentran en la misma componente.

Se ha presentado en la Sección 3.2.1 un algoritmo que, a partir de un tablero inicial y un tablero final, decide si el tablero final puede ser alcanzado desde el tablero inicial. Este algoritmo se basa en contar el "número de inversiones" de las fichas de cada tablero y luego compara la paridad de ambos resultados.

En este apéndice se estudia informalmente la correctitud del algoritmo propuesto para la verificación de solubilidad.

A.1 Configuraciones legales e ilegales

Teniendo en cuenta un estado final en particular se puede decir que hay dos grupos de configuraciones: las configuraciones *legales*, es decir aquellos estados que pueden convertirse en el estado final tras aplicar acciones legales, y las configuraciones *ilegales,* a saber aquellos estados que no tienen solución debido a que se encuentran en una componente conexa distinta a la del estado final y se obtienen a partir de intercambiar las posiciones dos fichas vecinas de un tablero legal (a excepción del hueco).

A.2 Fórmula de solubilidad

Sea n_i la cantidad que denota el *número de inversiones* de la ficha i, es decir la cantidad de piezas con menor número que aparecen después de ella, ya sea en la misma fila a su derecha, o en cualquier fila inferior, se plantean las siguientes proposiciones:

A.2.1 Proposición para N par

Si NT = $n_2 + n_3 + \ldots n_{(N^2-1)} + h_i$, donde h_i es el número de fila del hueco, empezando a contar la primera fila como 1. *NT mod 2* se mantiene invariante luego de un movimiento legal.

Prueba:

Los movimientos horizontales del hueco no cambian el ordenamiento de las fichas, por lo que no agregan ni restan inversiones, es decir no modifican NT. Al contrario, los movimientos verticales si modifican NT.

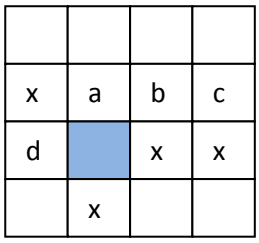


Figura A.1: Tablero con N par.

Teniendo en cuenta el tablero general de la Figura A.1, supongamos se quiere mover el hueco hacia arriba, movimiento que disminuye h_i en una unidad. La ficha "a" pasa a estar después de N-1 fichas (en este caso 3).

- Si las 3 fichas eran mayores que "a", ahora hay 3 inversiones más. Por lo tanto a NT se le resta 1 unidad, por el movimiento del hueco a una fila anterior, y se le suma 3 inversiones, manteniendo la paridad de NT.

- Si había 2 fichas mayores que "a" y una menor, ahora hay dos inversiones más, y dado que "a" se posiciona después de la ficha que era menor se resta una inversión a n_a. Además a NT se le resta 1 (porque disminuye h_i). Por lo anterior NT mantiene el valor y la paridad.

- Si había 3 fichas menores delante de "a", se restarían 3 inversiones a n_a, y 1 unidad por el desplazamiento del hueco manteniendo la paridad de NT.

En general si T es el valor de la ficha que se mueve, supongamos que r fichas de las N-1 en cuestión son menores que T, y (N-1)-r son mayores. Sea V el número de inversiones antes del movimiento, y W el número de inversiones después del movimiento.

$$W = V – r + (N - 1 - r) -1 = V - 2r + N – 2$$

- Si V era par, se le agrega/resta un número par, dejando a W par.

- Si V era impar, se le agrega/resta un número par, dejando a W impar.

De lo anterior se concluye que un movimiento del hueco hacia arriba no cambia la paridad. Lo mismo se puede comprobar para un movimiento del hueco hacia abajo, y estando la ficha que se mueve en cualquier columna.

A.2.2 Proposición para N impar

Si NT = $n_2 + n_3 + \ldots n_{(N^2-1)}$. NT mod 2 se mantiene invariante luego de un movimiento legal.

Prueba:

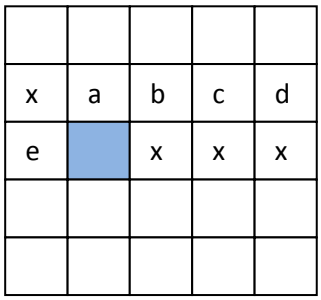


Figura A.2: Tablero con N impar.

Teniendo en cuenta el ejemplo de la Figura A.2, la prueba es similar a la expuesta en la sección anterior. Los movimientos verticales son los únicos que cambian el ordenamiento de las fichas y en consecuencia modifican NT.

- Supongamos que las fichas {b, c, d , e} son mayores que "a". Al mover el hueco hacia arriba, "a" pasará a estar después de éstas fichas, por lo que se debe sumar a NT cuatro inversiones, y en consecuencia se mantiene la paridad de NT.

- Supongamos que 3 fichas de {b,c,d,e} son mayores que "a", y 1 es menor. Dado que "a" pasa a estar delante de la ficha menor luego del movimiento, a NT se le resta una inversión. Dado que "a" pasa a estar delante de las 3 fichas mayores, se le suma tres inversiones a NT. Por lo anterior, NT mantiene la paridad.

- Supongamos que 2 fichas de {b,c,d,e} son mayores que "a", y 2 son menores. NT se mantiene (ya que se restan 2 inversiones porque "a" pasa a estar delante de las dos fichas menores luego del movimiento, y se le suman dos inversiones, ya que "a" pasa a estar delante de las dos fichas mayores).

En general si T es el valor de la ficha que se mueve, supongamos que r fichas de las N-1 en cuestión son menores que T, y (N-1)-r son mayores. Sea V el número de inversiones antes del movimiento, y W el número de inversiones después del movimiento. Haciendo una generalización similar a la expuesta en la sección anterior:

$$W = V - r + (N - 1 - r) = V - 2r + N - 1.$$

Como (N – 1) es par y 2r es par: si V era impar, luego del movimiento legal W será impar; si V era par entonces luego del movimiento W permanece par.

A.3 Conclusiones

De lo estudiado anteriormente, se puede decir:

- Si el tablero final tiene NT par, luego de cada movimiento NT se mantiene par. Entonces, a partir de un tablero con NT par se puede alcanzar sólo tableros con NT par.
- Si el tablero final tiene NT impar, luego de cada movimiento NT se mantiene impar. Entonces, a partir de un tablero con NT impar se puede alcanzar sólo tableros con N impar.

Por lo tanto, el grafo asociado al Puzzle es bipartito: posee dos componentes conexas, una consiste en tableros con permutación par y otra contiene tableros con permutación impar.

APÉNDICE B: FUNCIÓN DE ZOBRIST

La *Función de Zobrist* (Zobrist, 1970) (Millington & Funge, 2009) es una *función hash* utilizada por los algoritmos de búsqueda sobre espacios de estados con forma de grafo, en particular para problemas cuya representación abstracta del estado es un tablero, y es esencial para la implementación de la detección de duplicados.

La *función* asocia una clave a cada estado del espacio de estados del problema. Dado que la cantidad de bits utilizada para representar la clave está limitada a 64 por las arquitecturas actuales, una misma clave podrá ser asignada a dos estados distintos si la cantidad configuraciones del espacio de estados es mayor a 2^{64}. La clave asociada a un estado será utilizada por el algoritmo de búsqueda para acceder a la *Tabla Hash* que se utiliza para implementar la Lista Cerrada y a aquella asociada a la Lista Abierta durante la etapa de *detección de duplicados*, permitiendo rápidamente conocer si el estado fue generado con anterioridad.

La propiedad principal de esta función reside en que genera valores totalmente diferentes para dos estados consecutivos, los cuales exhiben poca variación ya que se diferencian sólo en la ubicación de una ficha; esto se debe a que las claves no están relacionadas con las posiciones de las fichas permitiendo que su uso como *Función Hash* de lugar a pocas colisiones.

La *Función de Zobrist* requiere recibir como entrada la representación del estado del problema. Por ejemplo para el Puzzle la información necesaria sería el tablero y la posición del hueco (hi,hj). A partir de esta información calcula y retorna la clave asociada a dicho estado.

El cálculo requiere del uso de una tabla inicializada con números aleatorios de 64 bits, a la cual se referirá como *Tabla Zobrist* de aquí en adelante, que posee 3 dimensiones: una por cada dimensión del tablero (X e Y) y una que se corresponde con la ficha que se puede almacenar en un casillero del tablero (Z). Por otro lado, el rango de cada dimensión depende del problema, por ejemplo: para el 8-Puzzle las dimensiones X e Y tendrán rango de 0 a 2, es decir coinciden con las dimensiones X e Y del tablero, y la dimensión Z tendrá rango de 0 a 7 (correspondiente a los posibles valores de las fichas).

Algoritmo Funcion-Zobrist-Completo
Entrada: representación del estado (tablero) , Tabla Zobrist
Salida: clave asociada al estado
Iniciar *Clave* en cero
Para cada posición (x,y) del tablero recibido realizar
    Sea *Ficha* el valor almacenado en la posición (x,y) del tablero
    Sea *NroAleatorio* el valor almacenado en la posición (x,y,ficha) de *Tabla Zobrist*
    Si ficha no representa un casillero desocupado
        Asignar a *Clave* el resultado de la operación *Clave* XOR *NroAleatorio*

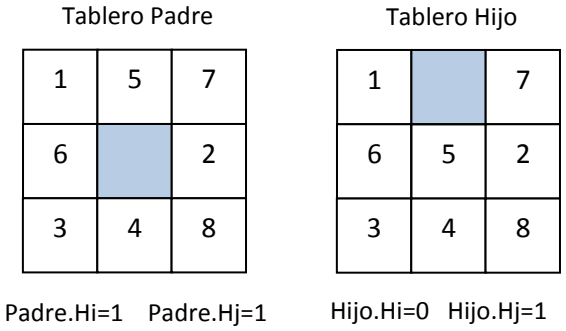
Retornar *Clave*

Figura B.1. Algoritmo Funcion-Zobrist-Completo

La Figura B.1 muestra el algoritmo para calcular la Función de Zobrist.

Teniendo la clave asociada al estado padre, la clave del estado sucesor puede calcularse de forma incremental ya que su cálculo varía sólo en los valores obtenidos de la *Tabla Zobrist* correspondientes a las posiciones de las fichas involucradas en el movimiento realizado.

La Figura B.2 muestra un tablero del 8-Puzzle (izquierda) y el tablero generado a partir de realizar un movimiento hacia arriba del hueco (derecha); además muestra las operaciones a realizar para calcular la clave del tablero sucesor (NEWVAL) en forma incremental a partir de la clave asociada al tablero padre (OLDVAL).



NEWVAL = OLDVAL **XOR** TablaZobrist[hijo.hi, hijo.hj, tableroPadre[hijo.hi, hijo.hj]]

NEWVAL = NEWVAL **XOR** TablaZobrist [padre.hi, padre.hj, tableroHijo[padre.hi, padre.hj]]

Figura B.2. Movimiento generado en el Puzzle-8. A izquierda el estado padre. A derecha el estado hijo generado a partir de realizar el movimiento del hueco hacia arriba. Abajo se muestran las operaciones para el cálculo incremental de la clave del tablero hijo a partir de la clave del tablero padre.

APÉNDICE C: CONFIGURACIONES UTILIZADAS DEL 15-PUZZLE

En este apéndice se muestran las configuraciones iniciales utilizadas para el análisis de los algoritmos. En todas las ejecuciones la configuración final utilizada es: **0** 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Número de Configuración	Configuración	Costo Solución
1	14 13 15 7 11 12 9 5 6 **0** 2 1 4 8 10 3	57
2	13 5 4 10 9 12 8 14 2 3 7 1 **0** 15 11 6	55
3	14 7 8 2 13 11 10 4 9 12 5 **0** 3 6 1 15	59
4	5 12 10 7 15 11 14 **0** 8 2 1 13 3 4 9 6	56
5	4 7 14 13 10 3 9 12 11 5 6 15 1 2 8 **0**	56
6	14 7 1 9 12 3 6 15 8 11 2 5 10 **0** 4 13	52
7	2 11 15 5 13 4 6 7 12 8 10 1 9 3 14 **0**	52
8	12 11 15 3 8 **0** 4 2 6 13 9 5 14 1 10 7	50
9	3 14 9 11 5 4 8 2 13 12 6 7 10 1 15 **0**	46
10	13 11 8 9 **0** 15 7 10 4 3 6 14 5 12 2 1	59
11	5 9 13 14 6 3 7 12 10 8 4 **0** 15 2 11 1	57
12	14 1 9 6 4 8 12 5 7 2 3 **0** 10 11 13 15	45
13	3 6 5 2 10 **0** 15 14 1 4 13 12 9 8 11 7	46
14	7 6 8 1 11 5 14 10 3 4 9 13 15 2 **0** 12	59
15	13 11 4 12 1 8 9 15 6 5 14 2 7 3 10 **0**	62
16	1 3 2 5 10 9 15 6 8 14 13 11 12 4 7 **0**	42
17	15 14 **0** 4 11 1 6 13 7 5 8 9 3 2 10 12	66
18	6 **0** 14 12 1 15 9 10 11 4 7 2 8 3 5 13	55
19	7 11 8 3 14 **0** 6 15 1 4 13 9 5 12 2 10	46
20	6 12 11 3 13 7 9 15 2 14 8 10 4 1 5 **0**	52
21	12 8 14 6 11 4 7 **0** 5 1 10 15 3 13 9 2	54
22	14 3 9 1 15 8 4 5 11 7 10 13 **0** 2 12 6	59
23	10 9 3 11 **0** 13 2 14 5 6 4 7 8 15 1 12	49
24	7 3 14 13 4 1 10 8 5 12 9 11 2 15 6 **0**	54
25	11 4 2 7 1 **0** 10 15 6 9 14 8 3 13 5 12	52
26	5 7 3 12 15 13 14 8 **0** 10 9 6 1 4 2 11	58
27	14 1 8 15 2 6 **0** 3 9 12 10 13 4 7 5 11	53
28	13 14 6 12 4 5 1 **0** 9 3 10 2 15 11 8 7	52
29	9 8 **0** 2 15 1 4 14 3 10 7 5 11 13 6 12	54
30	12 15 2 6 1 14 4 8 5 3 7 **0** 10 13 9 11	47
31	12 8 15 13 1 **0** 5 4 6 3 2 11 9 7 14 10	50
32	14 10 9 4 13 6 5 8 2 12 7 **0** 1 3 11 15	59
33	14 3 5 15 11 6 13 9 **0** 10 2 12 4 1 7 8	60
34	6 11 7 8 13 2 5 4 1 10 3 9 14 **0** 12 15	52
35	1 6 12 14 3 2 15 8 4 5 13 9 **0** 7 11 10	55
36	12 6 **0** 4 7 3 15 1 13 9 8 11 2 14 5 10	52
37	8 1 7 12 11 **0** 10 5 9 15 6 13 14 2 3 4	58
38	7 15 8 2 13 6 3 12 11 **0** 4 10 9 5 1 14	53
39	9 **0** 4 10 1 14 15 3 12 6 5 7 11 13 8 2	49

40	11 5 1 14 4 12 10 **0** 2 7 13 3 9 15 6 8	54
41	8 13 10 9 11 3 15 6 **0** 1 2 14 12 5 4 7	54
42	4 5 7 2 9 14 12 13 **0** 3 6 11 8 1 15 10	42
43	11 15 14 13 1 9 10 4 3 6 2 12 7 5 8 **0**	64
44	12 9 **0** 6 8 3 5 14 2 4 11 7 10 1 15 13	50
45	3 14 9 7 12 15 **0** 4 1 8 5 6 11 10 2 13	51
46	8 4 6 1 14 12 2 15 13 10 9 5 3 7 **0** 11	49
47	6 10 1 14 15 8 3 5 13 **0** 2 7 4 9 11 12	47
48	8 11 4 6 7 3 10 9 2 12 15 13 **0** 1 5 14	49
49	10 **0** 2 4 5 1 6 12 11 13 9 7 15 3 14 8	59
50	12 5 13 11 2 10 **0** 9 7 8 4 3 14 6 15 1	53
51	10 2 8 4 15 **0** 1 14 11 13 3 6 9 7 5 12	56
52	10 8 **0** 12 3 7 6 2 1 14 4 11 15 13 9 5	56
53	14 9 12 13 15 4 8 10 **0** 2 1 7 3 11 5 6	64
54	12 11 **0** 8 10 2 13 15 5 4 7 3 6 9 14 1	56
55	13 8 14 3 9 1 **0** 7 15 5 4 10 12 2 6 11	41
56	3 15 2 5 11 6 4 7 12 9 1 **0** 13 14 10 8	55
57	5 11 6 9 4 13 12 **0** 8 2 15 10 1 7 3 14	50
58	5 **0** 15 8 4 6 1 14 10 11 3 9 7 12 2 13	51
59	15 14 6 7 10 1 **0** 11 12 8 4 9 2 5 13 3	57
60	11 14 13 1 2 3 12 4 15 7 9 5 10 6 8 **0**	66
61	6 13 3 2 11 9 5 10 1 7 12 14 8 4 **0** 15	45
62	4 6 12 **0** 14 2 9 13 11 8 3 15 7 10 1 5	57
63	8 10 9 11 14 1 7 15 13 4 **0** 12 6 2 5 3	56
64	5 2 14 **0** 7 8 6 3 11 12 13 15 4 10 9 1	51
65	7 8 3 2 10 12 4 6 11 13 5 15 **0** 1 9 14	47
66	11 6 14 12 3 5 1 15 8 **0** 10 13 9 7 4 2	61
67	7 1 2 4 8 3 6 11 10 15 **0** 5 14 12 13 9	50
68	7 3 1 13 12 10 5 2 8 **0** 6 11 14 15 4 9	51
69	6 **0** 5 15 1 14 4 9 2 13 8 10 11 12 7 3	53
70	15 1 3 12 4 **0** 6 5 2 8 14 9 13 10 7 11	52
71	5 7 **0** 11 12 1 9 10 15 6 2 3 8 4 13 14	44
72	12 15 11 10 4 5 14 **0** 13 7 1 2 9 8 3 6	56
73	6 14 10 5 15 8 7 1 3 4 2 **0** 12 9 11 13	49
74	14 13 4 11 15 8 6 9 **0** 7 3 1 2 10 12 5	56
75	14 4 **0** 10 6 5 1 3 9 2 13 15 12 7 8 11	48
76	15 10 8 3 **0** 6 9 5 1 14 13 11 7 2 12 4	57
77	**0** 13 2 4 12 14 6 9 15 1 10 3 11 5 8 7	54
78	3 14 13 6 4 15 8 9 5 12 10 **0** 2 7 1 11	53
79	**0** 1 9 7 11 13 5 3 14 12 4 2 8 6 10 15	42
80	11 **0** 15 8 13 12 3 5 10 1 4 6 14 9 7 2	57
81	13 **0** 9 12 11 6 3 5 15 8 1 10 4 14 2 7	53
82	14 10 2 1 13 9 8 11 7 3 6 12 15 5 4 **0**	62
83	12 3 9 1 4 5 10 2 6 11 15 **0** 14 7 13 8	49
84	15 8 10 7 **0** 12 14 1 5 9 6 3 13 11 4 2	55
85	4 7 13 10 1 2 9 6 12 8 14 5 3 **0** 11 15	44
86	6 **0** 5 10 11 12 9 2 1 7 4 3 14 8 13 15	45
87	9 5 11 10 13 **0** 2 1 8 6 14 12 4 7 3 15	52
88	15 2 12 11 14 13 9 5 1 3 8 7 **0** 10 6 4	65
89	11 1 7 4 10 13 3 8 9 14 **0** 15 6 5 2 12	54
90	5 4 7 1 11 12 14 15 10 13 8 6 2 **0** 9 3	50

91	9 7 5 2 14 15 12 10 11 3 6 1 8 13 **0** 4	57
92	3 2 7 9 **0** 15 12 4 6 11 5 14 8 13 10 1	57
93	13 9 14 6 12 8 1 2 3 4 **0** 7 5 10 11 15	46
94	5 7 11 8 **0** 14 9 13 10 12 3 15 6 1 4 2	53
95	4 3 6 13 7 15 9 **0** 10 5 8 11 2 12 1 14	50
96	1 7 15 14 2 6 4 9 12 11 13 3 **0** 8 5 10	49
97	9 14 5 7 8 15 1 2 10 4 13 6 12 **0** 11 3	44
98	**0** 11 3 12 5 2 1 9 8 10 14 15 7 4 13 6	54
99	7 15 4 **0** 10 9 2 5 12 11 13 6 1 3 14 8	57
100	11 4 **0** 8 6 10 5 13 12 7 14 3 1 2 9 15	54
101	15 14 12 **0** 11 13 2 8 7 10 6 9 3 5 4 1	71
102	15 14 13 12 11 10 6 8 9 5 **0** 1 3 2 7 4	72
103	11 15 12 **0** 14 10 2 13 7 6 9 8 3 5 4 1	73
104	11 15 13 12 14 10 6 2 7 9 5 4 3 8 1 **0**	74
105	15 14 13 12 10 6 **0** 8 2 11 9 1 3 7 5 4	75
106	15 14 13 12 10 9 8 11 7 6 5 1 3 2 4 **0**	76

BIBLIOGRAFÍA

Aho, A., Ullman, J. & Hopcroft, J., 1983. *Data Structures and Algorithms.* Primera Edición ed. Boston(MA): Addison-Wesley Longman Publishing Co.

Akhter, S. & Roberts, J., 2006. *Multicore Programming. Increasing Performance through SoftwareMulti-threading.* Primera Edición ed. Intel Press..

Andrews, G., 1999. *Foundations of Multithreaded, Parallel, and Distributed Programming.* San Jose(CA): Addison Wesley.

Baker, M. & Buyya, R., 1999. Cluster Computing at a Glance. En: *High Performance Cluster Computing: Architectures and Systems.* NJ: Prentice Hall.

Bauer, B., 1994. *The Manhattan Pair Distance Heuristic for the 15-Puzzle,* Paderborn: Universität-GH Paderborn.

Brüngger, A., 1998. *Solving Hard Combinatorial Optimization Problems in Parallel: Two Cases Studies.* Zurich: Hartung-Gorre.

Burns, E., Lemons, S., Ruml, W. & Zhou, R., 2010. Best-First Heuristic Search for Multicore Machines. *Journal of Artificial Intelligence Research,* Volumen 39, pp. 689-743.

Burns, E., Lemons, S., Zhou, R. & Ruml, W., 2009. *Best-First Heuristic Search for Multi-Core Machines.* Pasadena(CA), IJCAI Organization, pp. 449-455.

Butenhof, D., 1997. *Programming with POSIX Threads.* Boston(MA): Addison Wesley.

Culberson, J. & Schaeffer, J., 1998. Pattern databases. *Computational Intelligence,* 14(3), pp. 318-334.

Cung, V. & Le Cun, B., 1994. An efficient implementation of parallel A*. *Proceedings of the First Canada-France Conference on Parallel and Distributed Computing. Lecture Notes in Computer Science,* Volumen 805, pp. 153-168.

Dijkstra, E. W., 1959. A note on two problems in connexion with graphs. *Numerische Mathematik,* 1(1), pp. 269-271.

Dijkstra, E. W., 1987. *Shmuel Safra's version of termination detection EWD-Note 998*.

Dijkstra, E. W., Feijen, W. H. J. & Van Gasteren, A. J. M., 1983. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters,* 16(5), pp. 217-219.

Dongarra, J. y otros, 2002. *Sourcebook of Parallel Computing.* Primera Edición ed. San Francisco(CA): Morgan Kaufmann.

Dutt, S. & Mahapatra, N., 1993. *Parallel A* Algorithms and Their Performance on Hypercube Multiprocessors..* Newport Beach (CA), IEEE Computer Society, pp. 797-803.

Ebendt, R. & Drechsler, R., 2009. Weighted A* search - unifying view and application. *Artificial Intelligence,* 173(14), p. 1310–1342.

Edelkamp, S. & Schrödl, S., 2011. *Heuristic Search: Theory and Applications.* Waltham (MA): Morgan Kaufmann.

Engelschall, R., 2006. *The GNU Portable Threads.* [En línea]
Available at: http://www.gnu.org/software/pth/
[Último acceso: 11 02 2014].

Evans, J., 2006. *A Scalable Concurrent malloc(3) Implementation for FreeBSD.* Ottawa, The FreeBSD Foundation.

Evett, M., Hendler, J., Mahanti, A. & Nau, D., 1991. *PRA*: massively parallel heuristic search,* College Park (MD): University of Maryland at College Park.

Evett, M., Hendler, J., Mahanti, A. & Nau, D., 1995. PRA*: massively parallel heuristic search. *Journal of Parallel and Distributed Computing,* 25(2), pp. 133-143.

Feijen, W. & Van Gasteren, A., 1999. Shmuel Safra's Termination Detection Algorithm En: *On a method of multiprogramming (Monographs in Computer Science).* New York(NY): Springer.

Felner, A. y otros, 2011. Inconsistent heuristics in theory and practice. *Artificial Intelligence,* 175(9-10), pp. 1570-1603.

Fitch, R., Butler, Z. & Rus, D., 2005. *Reconfiguration Planning Among Obstacles for Heterogeneous Self-Reconfiguring Robots.* Barcelona, IEEE, pp. 117 - 124.

Fritzsche, P., Rexachs, D. & Luque, E., 2008. *A General Approach to Predict the Performance Order of TSP Family Problems.* Chipre, Springer.

FSF Free Software Foundation, I., 2014. *The GNU C Library.* [En línea]
Available at: https://www.gnu.org/software/libc/manual/html_node/
[Último acceso: 13 02 2014].

Garg, V., 2004. Detecting Termination and Deadlocks. En: *Concurrent and Distributed computing in Java.* Hoboken(NJ): John Wiley & Sons.

Ghemawat, S. & Menage, P., 2014. *TCMalloc : Thread-Caching Malloc.* [En línea]
Available at: http://goog-perftools.sourceforge.net/doc/tcmalloc.html
[Último acceso: 03 02 2014].

Gloger, W., 2014. *Wolfram Gloger's malloc homepage.* [En línea]
Available at: http://www.malloc.de/en/
[Último acceso: 03 02 2014].

Grama, A., Gupta, A., Karypis, G. & Kumar, V., 2003. *Introduction to Parallel Computing.* Segunda Edición ed. Harlow(Essex): Pearson. Addison Wesley.

Grama, A., Gupta, A. & Kumar, V., 1993. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures.. *Parallel & Distributed Technology: Systems & Applications, IEEE,* 1(3), pp. 12-21.

Grama, A. & Kumar, V., 1999. State of the Art in Parallel Search Techniques for Discrete Optimization Problems. *IEEE Transactions on Knowledge and Data Engineering,* 11(1), pp. 28-35.

Grove, D., 2011. *Multicore Application Programming. For Windows, Linux and Oracle (c) Solaris. Developers' Library.* Boston(MA): Pearson Education.

Gudaitis, M., 1994. *Multicriteria Mission Route Planning Using a Parallel A* Search,* Wright-Patterson AFB (OH): Air University.

Gue, K., Furmans, K., Seibold, Z. & Uludag, O., 2013. GridStore: A Puzzle-Based Storage System With Decentralized Control. *IEEE Transactions on Automation Science and Engineering,* 11(2), pp. 429-438.

Gustafson, J., 1990. *Fixed Time, Tiered Memory, and Superlinear Speedup.* Charleston, SC, IEEE Computer Society Press, pp. 1255-1260.

Hager, G. & Wellein, G., 2011. *Introduction to High Performance Computing for Scientists and Engineers.* Boca Raton(FL): CRC Press. Taylor & Francis Group.

Hansen, E. A. & Zhou, R., 2007. Anytime heuristic search. *Journal of Artificial Intelligence Research,* 28(1), pp. 267-297.

Hansson, O., Mayer, A. & Yung, M., 1985. *Generating Admissible Heuristic by Criticizing Solutions to Relaxed Models,* New York (NY): Columbia University.

Hart, P., Nilsson, N. & Raphael, B., 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics,* 4(2), pp. 100-107.

Helmert, M., 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research,* Volumen 26, pp. 191-246.

Hennessy, J. & Patterson, D., 2011. *Computer Architecture: A Quantitative Approach.* Fifth Edition ed. Waltham(MA): Morgan Kaufmann.

Herlihy, M. & Shavit, N., 2008. *The Art of Multiprocessor Programming.* Waltham(MA): Morgan Kauffman. Elsevier.

Hunt, G., Michael, M., Parthasarathy, S. & Scott, M., 1996. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters,* 60(3), pp. 151-157.

Hwang, K., Fox, G. & Dongarra, J., 2012. *Distributed and Cloud Computing. From Parallel Processing to the Internet of Things.* Waltham(MA): Morgan Kaufmann. Elsevier.

Intel, 2007. *Intel Ark.* [En línea]
Available at: http://ark.intel.com/products/33079/Intel-Xeon-Processor-E5405-(12M-Cache-

2_00-GHz-1333-MHz-FSB
[Último acceso: 24 02 2014].

Intel, 2008. *OpenMP* / MPI Hybrid Programming.Multi-core Programming for Academia. Intel Software College.* Intel.

Intel, 2010. *Intel Ark.* [En línea]
Available at: http://ark.intel.com/products/47925
[Último acceso: 24 02 2014].

Johnson, W. & Storey, W., 1879. *Notes on the 15-Puzzle.* Baltimore (MD), The Johns Hopkins University Press, pp. 397-404.

Jones, D., 1989. Concurrent operations on priority queues. *Communications of the ACM,* 32(1), pp. 132-137.

Kerrisk, M., 2014. *Linux Programmer's Manual.* [En línea]
Available at: http://man7.org/linux/man-pages/man5/proc.5.html
[Último acceso: 08 05 2014].

Kishimoto, A., Fukunaga, A. & Botea, A., 2009. *Scalable, Parallel Best-First Search for Optimal Sequential Planning.* Thessaloniki, AAAI Press, pp. 201-208.

Kishimoto, A., Fukunaga, A. & Botea, A., 2013. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence,* Volumen 195, p. 222–248.

Kobayashi, Y., Kishimoto, A. & Watanabe, O., 2011. *Evaluations of Hash Distributed A* in Optimal Sequence Alignment.* Barcelona, AAAI Press, pp. 584-590.

Korf, R., 1985. Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence,* 27(1), pp. 97-109.

Korf, R., 1993. Linear-space best-first search. *Artificial Intelligence,* 62(1), pp. 41-78.

Korf, R., 2000. *Recent Progress in the Design and Analysis of Admissible Heuristic Functions.* Londres, Springer-Verlag, pp. 45-55.

Korf, R. & Taylor, L., 1996. *Finding Optimal Solutions to the Twenty-Four Puzzle.* Portland (OR), MIT Press, pp. 1202-1207.

Korf, R., Zhang, W., Thayer, I. & Hohwald, H., 2005. Frontier search. *Journal of the ACM,* 52(5), pp. 715-748.

Kshemkalyani, A. D. & Singhal, M., 2008. *Distributed Computing. Principles, Algorithms, and Systems.* Cambridge: Cambridge University Press.

Kumar, V., Ramesh, K. & Rao, V. N., 1988. *Parallel Best-First Search of State-Space Graphs: A Summary of Results.* St. Paul (MN), AAAI Press, pp. 122-127.

LocklessInc, 2014. *Lockless Memory Allocator Technical Information.* [En línea]
Available at: http://locklessinc.com/technical_allocator.shtml
[Último acceso: 03 02 2014].

Lucke, R., 2004. *Building Clustered Linux Systems.* Primera Edición ed. Upper Saddle River(NJ): Prentice Hall.

Mahapatra, N. R. & Dutt, S., 1993. *Scalable Duplicate Pruning Strategies for Parallel A* Graph Search.* Dallas (TX), IEEE, pp. 290-297.

Mans, B. & Roucairol, C., 1990. *Concurrency in Priority Queues for Branch and Bound Algorithms,* Rocquencourt: INRIA.

Manzini, G. & Somalvico, M., 1990. *Probabilistic performance analysis of heuristic search using parallel hash tables.* Ft. Lauderdale (FL), Baltzer.

Mattern, F., 1987. Algorithms for distributed termination detection. *Distributed Computing,* 2(3), pp. 161-175.

Michael, M., 2004. *Scalable lock-free dynamic memory allocation.* Washington, DC, ACM, pp. 35-46.

Millington, I. & Funge, J., 2009. *Artificial Intelligence for Games.* Segunda Edición ed. Burlington(MA): Morgan Kaufmann. Elsevier.

MPIForum, 2014. *MPI Documents.* [En línea]
Available at: http://www.mpi-forum.org/docs/
[Último acceso: 11 02 2014].

Nilsson, N., 1998. *Artificial Intelligence: A New Synthesis.* San Francisco(CA): Morgan Kaufmann.

OpenMP-ARB, 2014. *The OpenMP® API specification for parallel programming.* [En línea]
Available at: http://openmp.org/wp/
[Último acceso: 27 02 2014].

Patrick, B., Almulla, M. & Newborn, M., 1992. An upper bound on the time complexity of iterative-deepening-A*. *Annals of Mathematics and Artificial Intelligence,* 5(2-4), pp. 265-277.

Pearl, J., 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Boston(MA): Addison-Wesley Longman Publishing Co., Inc.

Pohl, I., 1970. Heuristic search viewed as path finding in a graph. *Artifitial Intelligence,* 1(3-4), pp. 193-204.

Quiin, M., 1994. *Parallel Computing: Theory and Practice.* New York(NY): McGraw-Hill Higher Education.

Ramakrishnan, R. & Gehrke, J., 1999. *Database Management Systems.* Segunda Edición ed. McGraw-Hill Education.

Rao, N. V. & Kumar, V., 1988. Concurrent access of priority queues. *IEEE Transactions on Computers,* 37(12), pp. 1657-1665 .

Rao, V. & Kumar, V., 1988. *Superlinear speedup in parallel state-space search.* Pune, India, Springer, pp. 161-174.

Ratner, D. & Warmuth, M., 1990. The (n2−1)-puzzle and related relocation problems. *Journal of Symbolic Computation,* 10(2), pp. 111-137.

Ratner, D. & Warrnuth, M., 1986. *Finding a shortest solution for the NxN extension of the 15-puzzle is intractable.* Philadelphia (PA), AAAI Press, pp. 168-172.

Rauber, T. & Rünger, G., 2010. *Parallel programming for multicore and clusters systems.* Londres: Springer-Verlag Berlin Heidelberg.

Romein, J., Bal, H., Schaeffer, J. & Plaat, A., 2002. A performance analysis of transposition-table-driven work scheduling in distributed search. *IEEE Transactions on Parallel and Distributed Systems,* 13(5), pp. 447 - 459.

Romein, J., Plaat, A., Bal, H. & Schaeffer, J., 1999. *Transposition Table Driven Work Scheduling in Distributed Search.* Orlando (FL), AAAI Press, pp. 725-731.

Russel, S. & Norvig, P., 2003. *Artificial Intelligence: A Modern Approach.* Segunda edición ed. NJ: Prentice Hall.

Sanz, V., Naiouf, M. & De Giusti, A., 2011. *Parallel Optimal and Suboptimal Heuristic Search on multicore clusters.* Las Vegas, WORLDCOMP, pp. 673-679.

Stallings, W., 2006. *Organización y Arquitectura de Computadores.* Séptima edición ed. Madrid: Prentice-Hall.

Sundell, H. & Tsigas, P., 2005. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. *Journal of Parallel and Distributed Computing,* 65(5), pp. 609-627.

Torquati, M., Bertels, K., Karlsson, S. & Pacull, F., 2013. *Smart Multicore Embedded Systems.* New York(NY): Springer.

Vidal, V., Vernhes, S. & Infantes, G., 2011. *Parallel AI Planning on the SCC.* Potsdam, Universitätsverlag Potsdam, pp. 15-20.

Zhang, Z. y otros, 2009. *A* Search with Inconsistent Heuristics.* San Francisco (CA), Morgan Kaufmann Publishers Inc, pp. 634-639.

Zhou, R. & Hansen, E., 2004. *Structured Duplicate Detection in External-Memory Graph Search.* San Jose (CA), AAAI Press, pp. 683-688.

Zhou, R. & Hansen, E., 2007. *Parallel Structured Duplicate Detection.* Vancouver, AAAI Press, pp. 1217-1223.

Zobrist, A., 1970. *A New Hashing Method with Application for Game Playing,* Madison: University of Wisconsin.