Ajuste de rendimiento del algoritmo HDA* para máquinas multicore.

Victoria Sanz[1,2], Armando De Giusti[1,2], Marcelo Naiouf[1]
[1]III-LIDI, Facultad de Informática, UNLP, La Plata, Buenos Aires, Argentina. [2]CONICET.
{vsanz, degiusti, mnaiouf}@lidi.info.unlp.edu.ar

Resumen. Este trabajo analiza el rendimiento alcanzado por una versión propia del algoritmo paralelo HDA* para arquitecturas de memoria compartida, que permite encontrar soluciones a problemas de optimización combinatoria, ajustando el valor de los parámetros del mismo. La implementación se realizó utilizando *Pthreads*, el gestor de memoria dinámica *Jemalloc*, y el Puzzle-15 como caso de estudio. El trabajo experimental se enfoca en analizar la desviación de los tiempos cuando se ejecuta el algoritmo sobre una máquina con procesadores multicore, para distintas instancias del problema, variando la cantidad de hilos/cores utilizados y los parámetros propios del mismo. Por último, se presenta un análisis del rendimiento alcanzado al aumentar la carga de trabajo y la cantidad de hilos/cores seleccionando los valores óptimos para cada parámetro según la instancia de entrada.

Palabras clave: HDA*; Multicore; Ajuste de rendimiento; Escalabilidad; Problemas de optimización combinatoria.

1 Introducción

En el área de Inteligencia Artificial los algoritmos de búsqueda heurística 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*) [1], 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 ordenados por la función $\hat{f}$ (*lista abierta*), y otra con los nodos 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* [2] es una de las variantes más utilizadas de *BFS* ya que garantiza encontrar soluciones de costo óptimo. Para ello la función de costo $\hat{f}$ 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 heurística 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.

Hasta el momento distintos autores han propuesto diversas técnicas para paralelizar los algoritmos *BFS*, que difieren en cómo manipulan las listas abierta y 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 [3].

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 un limitante en el rendimiento [3] [4]. 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 [3], y en particular estudios actuales han demostrado que no obtiene un rendimiento competitivo sobre máquinas multicore [5].

Para solventar el problema anterior, cada proceso/thread puede tener su propia lista abierta y lista cerrada local (*estrategia descentralizada*) y realizar una búsqueda cuasi independiente. Esta estrategia se adapta bien tanto a arquitecturas de memoria compartida y 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 equiparar 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 requieren 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**) [6] paraleliza A* aplicando una *estrategia descentralizada* y utiliza la *función hash de Zobrist* [7] para asignar cada estado 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 fue implementado haciendo uso de la biblioteca de paso de mensajes *MPI* y comunicación asincrónica, siendo apto para ser ejecutado tanto en arquitecturas de memoria distribuida como arquitecturas de memoria compartida.

Por otra parte, el trabajo [5] presenta una adaptación del algoritmo HDA* desarrollado haciendo uso de las herramientas 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, y analizan su rendimiento 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 [8]. Luego presentan el algoritmo *PBNF* cuyo objetivo es permitir que los threads trabajen en períodos libres de sincronización. Si bien obtienen mejor rendimiento con *PBNF*, el algoritmo es complejo y no paraleliza A* puramente, por lo que obtienen speedup superlineal en algunos casos.

Asimismo, el trabajo [9] desarrolla una versión propia del algoritmo HDA* para arquitecturas de memoria compartida, basada la propuesta de [5], diferenciándose en:

- La detección de terminación en forma *descentralizada*, mediante una adaptación del algoritmo propuesto por *Dijkstra y Safra* [10].
- La incorporación del parámetro *LNPI (Limite de Nodos por Iteración),* que indica la cantidad límite de nodos a procesar por iteración del algoritmo. Así la versión difiere del algoritmo original el cual expande un único nodo por iteración.
- 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.
- La utilización de la técnica *Memory Pool* para prevenir la degradación de rendimiento causada por operaciones alloc-free en una relación productor-consumidor entre distintos threads.

El trabajo experimental demuestra el beneficio de la técnica *Memory Pool* y analiza la escalabilidad del algoritmo sobre una máquina multicore, considerando para cada instancia de entrada el valor de LNPI que optimiza el rendimiento y un LNPT fijo.

2 Contribución

Este trabajo presenta un análisis del rendimiento alcanzado por el algoritmo paralelo HDA* para arquitecturas de memoria compartida, presentado por los autores en [9], cuando se ejecuta sobre una máquina con procesadores multicore, para distintas instancias del problema Puzzle-15 [11], variando la cantidad de hilos/cores y el valor de los parámetros LNPI y LNPT. De esta manera, se determina el beneficio que trae la incorporación dichos parámetros. Asimismo, se presenta un análisis del speedup y eficiencia al aumentar la carga de trabajo y la cantidad de hilos/cores seleccionando los valores óptimos de LNPI y LNPT según la instancia de entrada.

3 Algoritmo A* secuencial

El algoritmo A* [2] es una variante de la técnica de búsqueda *Best First Search*, en la cual cada nodo *n* se valúa de acuerdo con el costo de alcanzar el mismo a partir de la raíz del árbol de búsqueda $\hat{g}(n)$ y una heurística que estima el costo para ir desde *n* hasta un nodo solución $\hat{h}(n)$. De este modo la función de costo será $\hat{f}(n)=\hat{g}(n)+\hat{h}(n)$.

Si la heurística es *admisible* (es decir que nunca sobreestima el costo real), el algoritmo A* encontrará una solución óptima.

El algoritmo mantiene una lista de nodos no explorados (*lista abierta*), ordenada de acuerdo al valor de la función $\hat{f}$, y otra lista de nodos ya explorados (*lista cerrada*), utilizada para evitar ciclos en el grafo de búsqueda. Inicialmente la lista abierta contiene un sólo elemento, el nodo inicial, y la lista cerrada está vacía.

En cada paso, el nodo con menor valor $\hat{f}$ (el nodo más prometedor) es removido de la lista abierta y es examinado. Si dicho nodo es la solución, el algoritmo termina. En caso contrario, el nodo es expandido (generando los nodos hijos a partir de aplicar movimientos legales) e insertado en la lista cerrada. Cada nodo sucesor es insertado en la lista abierta si no se encuentra en la lista cerrada o lista abierta, o se encuentra en alguna de ellas pero su valor de costo mejora al anterior (esta verificación es conocida como *detección de duplicados)*.

Una vez encontrado el nodo que representa el estado final, se puede obtener la secuencia de acciones aplicadas en el camino óptimo siguiendo la secuencia de punteros al padre de sus nodos.

4 Algoritmo HDA* sobre Arquitecturas de Memoria Compartida

El algoritmo propuesto en el trabajo [5] fue implementado utilizando Pthreads y Jemalloc, y se basa en lo siguiente:

- Cada thread posee: su propia lista abierta y lista cerrada; una cola de entrada 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; y una cola de salida 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 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 accesos posteriores 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, transfiere todos los punteros a nodos quedando la cola de salida local vacía. 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 (es decir, no tiene nodos para seguir trabajando).

La asignación de estados a threads se realizó a través de la *Función de Zobrist.* Las colas de entrada y colas de salida fueron implementadas como arreglos dinámicos conteniendo punteros a nodo.

5 Implementaciones

5.1 A* Secuencial

La estructura seleccionada para implementar la lista abierta es una *MinHeap* [12] cuyo contenido esta indexado por una *Tabla Hash Extensible* [13]. Esta estructura permite ordenar los nodos por el valor arrojado por la función $\hat{f}$, de manera que las operaciones de inserción de un nodo, eliminación del nodo con menor valor $\hat{f}$ y decremento de prioridad de un nodo se pueden realizar en orden logarítmico; a su vez permite realizar búsquedas en orden constante para evaluar la existencia de un nodo que representa un estado particular. Por otra parte, se utilizó una *Tabla Hash Extensible* [13] para implementar la lista cerrada, la cual permite que las operaciones de inserción/eliminación de un nodo y búsqueda para evaluar la existencia de un nodo que representa un estado particular sean en orden constante.

Las claves asociadas a los elementos (*nodos*) almacenados en las *Tablas Hash* se obtienen calculando la *Función de Zobrist* [7] sobre la representación del estado.

5.2 HDA* para memoria compartida

Se implementó una versión propia del algoritmo HDA* para memoria compartida similar al estudiado en la Sección 4, utilizando la biblioteca *Pthread*. La asignación de estados a threads se realizó a través de la *Función de Zobrist.*

Cada *thread* realizará una búsqueda A* en el ámbito local, manteniendo sus listas abierta y cerrada locales. La estrategia de comunicación de nodos se basa en el uso de colas de entrada y colas de salida.

Para evitar encargar a un thread la tarea de detectar el estado de terminación, verificando los estados de sus pares y el estado de las colas de entrada, se realizó una adaptación del algoritmo de terminación de *Dijkstra y Safra* permitiendo que todos los threads colaboren para dicho propósito. Cada thread mantiene un *estado* y un *contador* de nodos enviados y recibidos – en vez de la cantidad de "comunicaciones" realizadas[1]. El token de terminación se representará con una variable compartida, llevando un *contador* de nodos en tránsito, un *estado* y el *identificador* del thread que posee el token al momento; los datos correspondientes al token no son protegidos ya que sólo un thread podrá modificar los mismos en un momento dado. La finalización del cómputo se avisará a través de una variable compartida *fin.*

Para posibilitar la poda de nodos que pertenecen a caminos subóptimos, los threads comparten un puntero a la mejor solución encontrada al momento por todos los threads (*mejor_solucion*) y su costo (*costo_mejor_solucion*), ambas variables deben ser protegidas ya que dos threads pueden encontrar dos soluciones distintas y querer actualizar estos valores al mismo tiempo.

El código a ejecutar por todos los threads es idéntico, sólo al thread 0 se le encarga 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.

---

[1] Las colas de entrada contabilizan la cantidad de nodos que almacenan (dimensión lógica), por ello se calcula la cantidad de nodos en tránsito en vez de la cantidad de "depósitos" que no fueron recibidos aún. Esta es una modificación al algoritmo de *Dijkstra y Safra.*

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 cuyo 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 (Límite de Nodos por Iteración)* nodos 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 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*[2]. Cuando el nodo extraído no era la solución se lo inserta en la lista cerrada, se lo expande generando sus sucesores y para cada sucesor se calcula la *Función de Zobrist* para 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 cantidad de nodos almacenados supera el valor *LNPT* (*Límite de Nodos por Transferencia*) intenta tomar el lock de la cola de entrada del thread destino y en caso de obtenerlo en forma inmediata transfiere los nodos almacenados dejando vacía la cola de salida propia[3]. 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 en ese momento la cola estaba vacía, debe avisarle por si estaba ocioso esperando este depósito.
- *Etapa 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 y esperará avisos correspondientes a alguno de los siguientes eventos hasta que haya recibido trabajo o detectado el fin del cómputo:
  - *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:* el thread debe actualizar las variables compartidas del token, siguiendo el algoritmo de terminación, 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 realiza una nueva vuelta para la detección de terminación.
  - *Depósito de trabajo en la cola de entrada propia:* se 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

[2]Necesario ya que dos threads pueden encontrar dos soluciones con costo distinto e intentar actualizar los datos compartidos en el mismo instante. Si no se verifica nuevamente la condición, podría quedar almacenada una solución subóptima.

[3]Esta es una diferencia respecto a la versión de HDA* propuesta por Burns, que tras cada generación de nodo correspondiente a otro thread intenta tomar el lock de su cola de entrada para transferirlo en forma individual.

a *costo_mejor_solucion* se realiza la detección de duplicados insertándolo en la lista abierta en caso que sea adecuado.

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 consume trabajo desde su cola de entrada, incrementando/disminuyendo el contador local en la cantidad de nodos depositados/ consumidos respectivamente.

Para solventar la degradación de rendimiento cuando se produce una relación alloc-free entre threads, se incorporó un pool de punteros a nodo para cada thread (*Memory Pool*) en el cual se almacenan punteros a nodo que el thread quiera "liberar" para su posterior reúso, evitando que al realizar un free se esté accediendo a las estructuras asignadas por el gestor de memoria dinámica a otro thread lo que provocaría contención en el acceso a las mismas.

6 Resultados experimentales

Para el trabajo experimental se utilizó una máquina que posee dos procesadores Intel ® Xeon ® E5620 y 32 GB de RAM (DDR3 1066 Mhz). Cada procesador cuenta con 4 *cores* físicos de 2.4 Ghz; cada *core* posee dos caché L1 de 64KB para datos e instrucciones respectivamente y una caché L2 de 256KB; a su vez todos los *cores* del procesador comparten una caché L3 de 12MB. Cada procesador posee un controlador de memoria, por lo que el diseño de memoria de la máquina es NUMA, y utiliza una interconexión *QPI* de 5.86 GT/s.

Las pruebas se realizaron tomando en cuenta 16 configuraciones iniciales del Puzzle-15 utilizadas en [14] - numeradas 3, 15, 17, 21, 26, 32, 33, 49, 53, 56, 59, 60, 66, 82, 88, 100; 6 de las configuraciones que tienen mayor cantidad de pasos para su solución [15], numeradas de 101 a 106; y la configuración final propuesta en [14].

Los algoritmos secuencial y paralelo utilizan el gestor de memoria dinámica *Jemalloc* configurado con 256 arenas y la función heurística H4. El algoritmo paralelo fue compilado para utilizar la técnica *Memory Pool* y espera activa. Se comprobó en [9] que estas técnicas mejoran el rendimiento de ambos algoritmos.

El algoritmo secuencial es determinista. Para cada configuración inicial se realizaron 10 ejecuciones y se promedió el tiempo de resolución en segundos.

Por otra parte, el algoritmo paralelo es no determinista. Para cada configuración inicial y cada grupo de parámetros se obtuvieron 100 muestras. Los valores de los parámetros utilizados son: la cantidad de cores/threads entre 4 y 8; *LNPI*={1,5,50,500}; *LNPT*={26,210,1680}[4]. Luego se promediaron los tiempos en segundos arrojados por las 100 ejecuciones realizadas para cada configuración y conjunto de parámetros, a lo que se llamará *muestra promedio*.

En las distintas pruebas se utilizó afinidad para asociar cada thread a un *core* exclusivo utilizando la función *sched_setaffinity()*. En aquellas pruebas con 4 threads se ubicó 1 par de threads en cada procesador de la máquina, y en aquellas pruebas con 8 threads se ubicó 1 thread por cada *core* físico de la máquina.

[4] Los valores de LNPT elegidos corresponden a 1KB (26 nodos), 8KB (210 nodos) y 64KB (1680 nodos) de datos respectivamente

6.1 Influencia del parámetro LNPI sobre el rendimiento

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* y se agruparon aquellas con igual configuración, cantidad de cores y *LNPT*, es decir cada grupo contiene las *muestras promedio* que difieren únicamente en el parámetro *LNPI*. Para cada *grupo* se calculó el Coeficiente de Variación (*CV*) a partir de la Desviación Estándar (*DE*) 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, para LNPT=26 los valores 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, y se destaca que el 95.5% de los grupos tienen *CV* por debajo de un 0.03; para LNPT=210 el *CV* oscila entre 0.0009 y 0.036, teniendo el 95.5% de los grupos un CV por debajo de 0.03; para LNPT=1680 el *CV* varía entre 0.001 y 0.062, y el 97.7% de los grupos tiene un CV por debajo de 0.03.

Por los bajos valores de *CV* obtenidos se concluye que existe poca variación en los tiempos de ejecución al cambiar el valor de *LNPI* entre los definidos. El parámetro influye en la frecuencia en la que un thread intenta tomar nodos de su cola de entrada:

- 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 no se fuerza al thread a esperar.
- Ante intentos *menos frecuentes* de acceso a la cola de entrada (valores 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 aumentará el tiempo de ejecución a causa de un incremento en el Overhead de Búsqueda[5]. Esta hipótesis fue comprobada realizando ejecuciones para las 22 configuraciones, utilizando *Memory Pool,* espera activa, limitando *LNPT* a 26, y *LNPI* en 10000.

6.2 Influencia del parámetro LNPT sobre el rendimiento.

El parámetro *LNPT* indica la cantidad de nodos que el thread debe acumular en la cola de salida para un thread destino antes de intentar la transferencia de los mismos.

Para comprobar si el uso del parámetro *LNPT* trae ventajas en el rendimiento, se ejecutó el algoritmo presentado limitando *LNPT* a 1. Se compararon los tiempos de ejecución de las *muestras promedio* que utilizan LNPT=1 y LNPT=26 respectivamente, teniendo en cuenta la misma configuración, *LNPI* y cantidad de cores. El 97.7% de las *muestras promedio* que utilizan LNPT=26 presentan una mejora en los tiempos de ejecución de entre 0.24% y 10.61%; las *muestras* restantes

[5] El Overhead de Búsqueda calcula el porcentaje de nodos que el algoritmo paralelo expande de más respecto al algoritmo secuencial y se calcula con la fórmula 100x(NP/NS -1), donde NP= cantidad de nodos procesados por el algoritmo paralelo y NS = cantidad de nodos procesados por el algoritmo secuencial.

(2.3%) presentaron una desmejora de entre 0.39% y 6.13%. A partir estos resultados queda demostrado el beneficio que trae la utilización del parámetro *LNPT*.

Luego, con el objetivo de analizar el grado de 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 el Coeficiente de Variación (*CV*) a partir de la Desviación Estándar (*DE*) 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 valores de *CV* obtenidos para los grupos oscilan entre 0.0035 y 0.26, es decir el tiempo de ejecución de una *muestra promedio* de un grupo se desvía entre un 0.35% y un 26% de la *media grupal* al variar el parámetro *LNPT* entre los valores fijados. 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 nodos:

- 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 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 general el valor de LNPT que optimiza el rendimiento para cada configuración y cantidad de cores es 210 (excepto para la configuración 102 y 4 cores cuyo valor de LNPT óptimo fue 1680). Las Figuras 1 y 2 muestran el speedup y eficiencia para cada configuración, utilizando 4 y 8 cores y LNTP=210; 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. Para 4 cores el speedup varía entre 3.39 y 4.39, y la eficiencia entre 0.85 y 1.099. Para 8 cores el speedup se encuentra entre 5.17 y 8.42, y la eficiencia entre 0.65 y 1.05.

Los casos de superlinealidad observados se deben a la obtención de Overhead de Búsqueda negativo (el algoritmo paralelo procesó menor cantidad de nodos que el algoritmo secuencial) o a que las listas abierta y cerrada de cada thread contienen menor cantidad de elementos respecto al algoritmo secuencial, provocando una aceleración en las operaciones realizadas sobre éstas.

7 Conclusiones y líneas de trabajo futuro

Se presentó una versión propia del algoritmo HDA* para memoria compartida y se analizó la influencia de los parámetros LNPI y LNPT sobre el rendimiento. Se analizó el rendimiento al aumentar la carga de trabajo y la cantidad de cores seleccionando

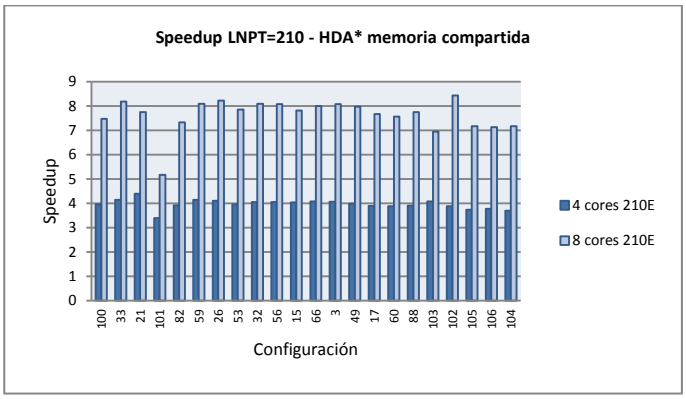

Figura 1. Speedup alcanzado por HDA*, LNPT =210.

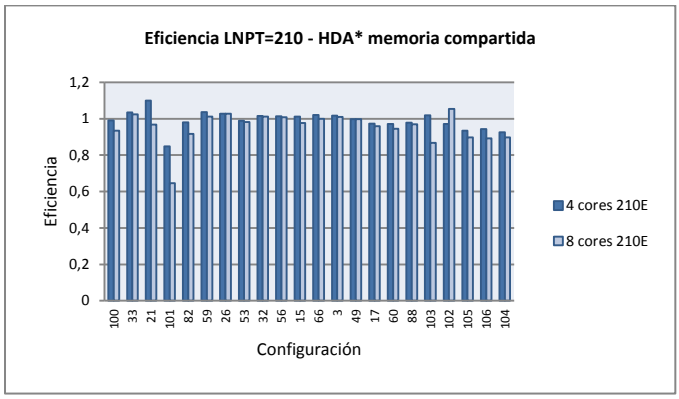

Figura 2. Eficiencia alcanzado por HDA*, LNPT =210.

los valores óptimos de LNPT para cada instancia de entrada del problema Puzzle-15. Las líneas de trabajo futuro se basan en comparar el rendimiento del algoritmo presentado contra el alcanzado por una versión propia de HDA* para memoria distribuida.

Referencias

[1] S. Russel y P. Norvig, *Artificial Intelligence: A Modern Approach*, Segunda edición. New Jersey, USA. Prentice Hall, 2003.

[2] P. Hart, N. Nilsson, y Raphael B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 1, pp. 100-107, 1968.

[3] V. Kumar, K. Ramesh, y V. N. Rao, "Parallel Best-First Search of State-Space Graphs: A Summary of Results", *Proceedings of the 10th Nat. Conf. AI, AAAI*, 1988, pp. 122-127.

[4] V. Cung y B. Le Cun, "An efficient implementation of parallel A*", *Proceedings of the First Canada-France Conference on Parallel and Distributed Computing*, Montréal, 1994, pp. 153-168.

[5] E. Burns, S. Lemons, W. Ruml, y R. Zhou, "Best-First Heuristic Search for Multicore Machines.", *Journal of AI Research*, vol. 39, pp. 689-743 , 2010.

[6] A. Kishimoto, A. Fukunaga, y A. Botea, "Evaluation of a simple, scalable, parallel best-first search strategy", *Artificial Intelligence*, pp. 222–248, 2013.

[7] A. Zobrist, "A New Hashing Method with Application for Game Playing", Computer Sciences Department, University of Wisconsin, Reporte Técnico 88, 1968.

[8] J. Evans, "A Scalable Concurrent malloc(3) Implementation for FreeBSD," in *Proceedings of the 3rd annual Technical BSD Conference*, Ottawa, 2006.

[9] V. Sanz, A. De Giusti, y M. Naiouf, "On the Optimization of HDA* for Multicore Machines. Performance Analysis", *Proceedings of the PDPTA*'14, Las Vegas, USA. En prensa.

[10] E. Dijkstra, "Shmuel Safra's version of termination detection EWD-Note 998", 1987.

[11] D. Ratner y M. Warmuth, "The (n2−1)-puzzle and related relocation problems", *Journal of Symbolic Computation*, vol. 10, no. 2, pp. 111-137, 1990.

[12] A. Aho, J. Ullman, y J. Hopcroft, *Data Structures and Algorithms*, Primera Edición. Boston, MA, USA. Addison-Wesley Longman Publishing Co, 1983.

[13] R. Ramakrishnan y J. Gehrke, *Database Management Systems*, Segunda Edición. McGraw-Hill Education, 1999.

[14] R. Korf, "Depth-first Iterative-Deepening: An Optimal Admissible Tree Search", *Artificial Intelligence*, pp. 97-109, 1985.

[15] A. Brüngger, *Solving Hard Combinatorial Optimization Problems in Parallel: Two Cases Studies*. Zurich, Suiza, 1998.