On the Optimization of HDA* for Multicore Machines. Performance Analysis.

Victoria Sanz[1], Armando De Giusti[2], Marcelo Naiouf
III- LIDI, School of Computer Sciences, UNLP, La Plata, Buenos Aires, Argentina

Abstract – *Combinatorial optimization problems are interesting due to their complexity and applications, particularly in robotics.*
This paper deals with a parallel algorithm suitable for shared memory architectures, based on the HDA algorithm (Hash Distributed A*), which allows finding solutions to combinatorial optimization problems. The implementation was carried out using the shared memory programming tools provided by the Pthreads library, the Jemalloc memory allocator and taking the N^2-1 Puzzle as study case.*
The experimental work focuses on analyzing the speedup and efficiency achieved by the parallel algorithm when running on a computer with multi-core processors, for different instances of the problem and varying the amount of threads/cores used. Finally, the scalability obtained with increasing workload and number of threads/cores used is analyzed.

Keywords: Parallel Heuristic Search; HDA*; Multicore; Combinatorial Problems; Scalability.

1 Introduction

In the area of Artificial Intelligence, heuristic search algorithms are used as the basis to solve combinatorial optimization problems that require a sequence of actions that minimize a goal function and allow transforming an initial configuration (which represents the problem to be solved) into a final configuration (which represents the solution).

One of the most used search algorithms for that purpose is known as *Best First Search* (*BFS*) [1], which browses the graph that represents the state space of the problem using a cost function $\hat{f}$ to value the nodes, which is in part composed of some heuristic information, that will guide the search faster to the solution and will reduce the nodes to be considered. The algorithm is different from the conventional methods because the graph is implicit and generated dynamically, i.e. nodes are created as the search progresses. During the process, it keeps two data structures: one for the unexplored nodes ordered by the function $\hat{f}$ (*open list*), and the other for the already explored nodes (*closed list*) used to avoid processing the same state repeatedly. In each iteration, the most promising node available on the open list is removed (according to function $\hat{f}$), it is included on the closed list and legal actions are applied to it to generate successor nodes which will be added to the open list under certain conditions. The search continues until a node that represents the solution is removed from the open list.

The A* algorithm [2] is one of the most commonly used *BFS* variants because it guarantees finding optimal cost solutions. To that end, the cost function $\hat{f}$ contains known cost information of the path from the initial node to the current node and heuristic information to estimate the unknown cost of the path from the current node to the solution node, which can never overestimate the actual cost; in this way, the search is guided to firstly process the most promising paths.

On the other hand, over the last years the development of parallel heuristic search algorithms has been promoted because the high requirement of computing power and memory, as a consequence of the exponential or factorial graph growth, makes its resolution on a single-core processor difficult. Moreover, it is common to find multi-core machines today, so the sequential applications should be adapted to take advantage of the computing power that this architecture provides.

So far, different authors have presented several techniques to parallelize *BFS* algorithms, which vary according to how they manipulate the open and closed list and in the load balancing strategy used among processors during the execution. The chosen technique will depend on the architecture and the problem to solve [3].

On a shared memory architecture, the simplest strategy is to keep only one open list and only one closed list shared by all the threads (*centralized strategy*). This implies a thread synchronization process to ensure data structure consistency, which will limit performance [3][4]. Although the open and closed lists can be implemented through data structures that allow concurrent access to different portions in order to reduce resource contention, several authors have shown that this technique will only bring improvements for problems with high heuristic computation time [3], and especially current studies have shown that it does not get a competitive performance on multi-core machines [5].

In order to solve the previous problem, each process/thread is equipped with its own local open and closed lists (*decentralized strategy*) and performs a quasi-independent search. This strategy is suitable either for shared memory or distributed memory architectures. However, communication among the processes/threads is needed due to the following reasons:

- As only one process/thread has the initial node on its open list at the beginning and the graph is generated at run time, the workload should be distributed dynamically.
- The nodes located on the processor's open list might not be the global best ones, so it will be necessary to equalize the nodes quality between processors.
- Duplicate nodes (nodes representing the same state) can be generated by different processes/threads. If the duplicate

[1] Fellow, CONICET.
[2] Principal Researcher, CONICET.

detection procedure is only performed by the process/thread which has generated the node and/or by that which has received the node owing to load balancing, the detection and pruning of duplicate nodes will be *partial* because another process/thread may have a node representing the same state on its open or closed list. However, if *absolute* detection and pruning is required, strategies that assign each state to a particular processor will be needed.

- The termination criterion should be modified, as there are multiple inconsistent open lists and, as a consequence of dynamic load balancing, there may be some graph nodes that are being communicated between processes/threads.
- The costs of the partial solutions found should be communicated in order to use them to prune the paths that lead to suboptimal cost solutions.

In this sense, the HDA* algorithm (*Hash Distributed A**) [6] parallelizes A* using the *decentralized strategy* and it applies *Zobrist´s hash function* to assign each state to a unique process; in this way, when a process generates a node, the owner process can be identified and the node is transferred to it. This mechanism allows balancing the workload, leveling node quality, and pruning duplicates in an *absolute* way, as the nodes representing a same state are always sent to the same process. The algorithm was implemented using the *MPI* message passing library and asynchronous communication, so the algorithm can be executed either on distributed or shared memory architectures.

On the other hand, the research carried out by [5] presents an adaptation of the HDA* algorithm developed using the shared memory programming tools provided by the *Pthreads* library; in this way, it is possible to eliminate some inefficiencies that arise when the original HDA* algorithm is run on a shared memory machine. The Jemalloc library [7] is used to avoid performance degradation due to contention in the access to the data structures managed by the dynamic memory allocator, caused by the frequent alloc/free operations. Algorithm performance is analyzed on a multicore machine. Moreover, a technique to create a state space abstraction that allows assigning state blocks to the threads, instead of individual states as it occurs with *Zobrist´s hash function*, is included in the algorithm. Then, the *PBNF* algorithm that allows threads to work during synchronization free periods is presented. The experimental work is done in part considering 250 easy instances of the 15-Puzzle, using the Sum of the Manhattan Distances heuristic, and an analysis of the speedup obtained as the architecture scales is carried out. Although a better performance is obtained with the *PBNF* algorithm, the algorithm is complex and does not use the same approach as the serial A*, so a *superlinear speedup* is obtained in some cases.

A common problem that causes performance degradation in multi-threaded applications, which frequently perform allocation and deallocation operations, is the producer-consumer relation that arises due to alloc-free operations carried out by different threads, which creates a need for synchronization to keep the consistency of the structures assigned to each thread by the memory allocator. In order to improve this, it is suggested to incorporate a pool of pointers to node in every thread (*Memory Pool*) to prevent thread A from freeing memory that is allocated by thread B; instead, thread A will store those pointers for future reuse.

Based on the above, the HDA* algorithm is still interesting due to its simplicity. The focus of this paper is the incorporation of techniques to optimize the HDA* algorithm for its execution on multicore, which may lead to a better performance, and to carry out a scalability analysis when the workload and the amount of processors are increased.

2 Contribution

This paper presents a parallel algorithm suitable for shared memory architectures, based on the HDA* algorithm, which allows finding optimal solutions to combinatorial optimization problems, in this case to the N^2-1 Puzzle. In this sense, the algorithm is similar to the one proposed in [5] but with the following differences: it incorporates an algorithm to detect termination in a decentralized way, which is an adaptation of the algorithm proposed by *Dijkstra and Safra* [8] [9]; threads accumulate a customizable quantity of nodes addressed to another thread before attempting their transfer, i.e. there are no transfers after each node generation; and a technique called *Memory Pool* is used to avoid performance degradation caused by alloc-free operations in a producer-consumer relation among different threads.

The contributions are:

- Carrying out experimental work running the HDA* parallel algorithm proposed, suitable for shared memory, on a multicore processor machine, using different initial configurations of the problem and varying the amount of threads/cores used, analyzing the performance obtained (*speedup*, *efficiency*) in each case.
- Carrying out a comparison between the performance obtained by the parallel algorithm when active waiting or passive waiting is used while the thread is idle.
- Documenting the benefits obtained when using the *Memory Pool* technique.
- Carrying out a scalability analysis of the algorithm when the workload and amount of threads/cores used are increased.

3 Characterization of the N^2-1 Puzzle

The N^2-1 Puzzle problem consists in N^2-1 pieces numbered from 1 to N^2-1 placed on an N^2 sized board [10]. Each square of the board contains one piece, so there is only one empty square.

- A legal movement implies moving the empty square to an adjacent position, either horizontally or vertically, by moving the piece that was in the newly emptied square to the previous position of the empty square.
- The objective of the puzzle is applying legal movements until the initial board becomes the selected final board. The solution to the problem should be the one that

minimizes the number of movements required to achieve the final configuration from the initial given configuration.

1.1. Heuristics

Heuristic search algorithms use information about the problem to guide the search process, so they value the nodes based on the application of a *heuristic function*. Thus, they process first the node that looks more promising. The heuristic value of a node is an estimate and indicates how close it is to the solution node.

A more polished heuristic will carry out estimates that are closer to the real cost; therefore, the algorithms that use it will need to process less nodes [1].

The heuristic used by the algorithms presented for the resolution of the Puzzle problem is a variation of the sum of the Manhattan distance of the pieces with the addition of linear conflict detection among pieces, the detection of the last movements applied, and an analysis of corner pieces. The definition can be found in [11].

4 Sequential A* algorithm

The A* algorithm [2] is a variation of the *Best First Search* technique where each node *n* is valuated in accordance to the cost of reaching it from the root of the search tree $\hat{g}(n)$ and a heuristic that estimates the cost to go from *n* to a solution node $\hat{h}(n)$. Thus, the cost function will be $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$. If the heuristic is *admissible* (i.e., it never overestimates the real cost), the algorithm will always find an optimal solution.

The algorithm keeps a list of unexplored nodes (*open list*) ordered by the value of function $\hat{f}$, and another list of already explored nodes (*closed list*) used to avoid loops in the search graph. Initially, the open list contains only one element, the initial node, and the closed list is empty.

In each step, the node with the lowest $\hat{f}$ value (the most promising node) is removed from the open list and examined. If the node is the solution, the algorithm ends. Otherwise, the node is expanded (generating the children nodes by applying legal movements) and added to the closed list. Each successor node is added to the open list if it does not appear on either list, or if it does but its cost value improves that of the previous node (this verification is known as *duplicate detection*).

Once the node that represents the final state has been found, the sequence of actions taken on the optimal path can be obtained by following the sequence of pointers to each parent node.

5 HDA* algorithm for shared memory architectures

The HDA* algorithm suitable for shared memory architectures proposed in [5] is based on the following:

- Each thread has its own open and closed lists.
- Each thread has an input queue known globally where the rest of the threads will deposit nodes that must be processed by this thread. The input queue must be protected by a lock to keep its consistency.
- Each thread has a local output queue for each peer thread, which does not need to be protected since it will be for thread's own use to avoid obstructions.
- When a thread t_i generates a node that belongs to another thread t_j, it must be communicated by adding it to t_j's input queue at some point. In order to do this, the thread tries to take the lock associated to t_j's input queue. When the lock is obtained immediately, node transfer is done by copying the pointer, and then the lock is released (this enables subsequent access to the queue by another thread). Otherwise, the pointer is added to the local output queue for t_j (there is no waiting time associated with this operation).
- After thread t_i carries out a certain number of node expansions from its open list:

- For each non-empty local *output queue*, the thread tries to communicate the stored nodes on it to its owner thread. In order to do this, the thread tries to take the lock associated to the input queue of the corresponding thread. If the lock is obtained, all the pointers to node are transferred, leaving the local output queue empty. Otherwise, it is not forced to wait.
- The thread tries to consume the nodes left by other threads on its own input queue. To do this, the thread must take the lock but it is only forced to wait if its open list is empty (in this case, it does not have any nodes to keep on working).

Fig. 1 shows the communication scheme of HDA* algorithm for shared memory. Here the thread's main local structures can be seen (open list, closed list, output queues) and also the global input queues. We can observe that thread 0 and thread 3 have generated a node that corresponds to thread 2; both of them attempt to take the lock associated to target thread's input queue. On the one hand, thread 3 gets the lock immediately, copies the pointer and releases the lock. On the other hand, thread 2 does not get the lock immediately, so it adds the pointer to its local output queue for the target thread.

The implementation was carried out with the tools provided by the Pthreads Library and Jemalloc, the dynamic memory allocator. The allocation of states to threads was done through the *Zobrist Function*. The input and output queues were implemented as a dynamic array that contain pointers to node.

6 Implementations

The implementations of the following algorithms were carried out in C Language. The compilation was done through Gcc, phase in which the memory allocator that is going to be used can be selected (in this case, it was *ptmalloc* [12] or *Jemalloc* [7]).

6.1 Sequential A*

The selected structure to implement the open list is a *MinHeap* [13] whose content is indexed by an *Extensible*

Hash Table [14]. This structure allows nodes to be ordered according to the $\hat{f}$ function, so the operations of inserting a node, removing the node with the lowest $\hat{f}$ value and decreasing the priority of a node can be carried out in logarithmic order; at the same time, it enables carrying out searches to determine the existence of a node that represents a particular state in constant order. Additionally, an *Extensible Hash Table*[14] was used to implement the closed list, which allows the operations of insertion/ removal of a node and searching to determine the existence of a node that represents a particular state occur in constant order.

The keys associated to the elements (nodes) stored in the *Hash Tables* are obtained by calculating the *Zobrist Function* [15] over the representation of the state. The *Zobrist Table* that was used is loaded from file and it will be the same for all the runs and instances of the problem selected as a case of study. The key will be represented with a 64-bit integer, which leads the function to assign the same key to different states of the search space (this happens because the number of possible problem states can be much higher than 2^{64}). This case is not frequent during the run of the search algorithm.

The kind of heuristic functions used is that which calculates the estimation of the cost directly, taking as input the representation of the state. Consequently, the heuristic is problem-dependent and can be selected before the compilation phase; this enables experimenting with different heuristic functions and analyzing the performance obtained.

6.2 HDA* for shared memory

A version of the HDA* algorithm suitable for shared memory similar to the one studied in Section V was implemented. Threads and synchronization mechanisms provided by the *Pthread* library were used. The assignment of states to threads was carried out through the *Zobrist Function*.

Each *thread* will perform an A* search locally, keeping its local open and closed lists. The node communication strategy is based on the use of input and output queues.

To avoid making an only thread detect the termination state by checking the state of the other threads and the state of their input queues, an adaptation of the *Dijkstra and Safra's* termination detection algorithm was carried out, allowing all the threads to cooperate with such purpose. Each thread keeps a *state* (or *color)* and a *counter* of sent and received nodes - instead of the number of "communications" that were done[1]. The *termination token* will be represented with a shared variable with the following information: a counter of nodes in transit, a *state* (or *color*), and the identifier of the thread that owns the token at the moment; the data that corresponds to the token are not protected since only one thread will be able to modify them at a given point in time. The end of the computation will be communicated through a shared variable *end*.

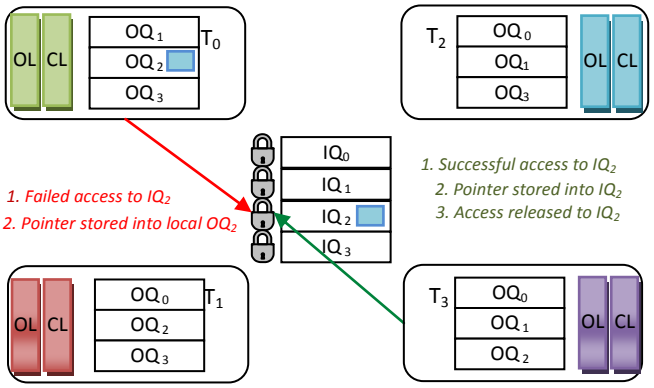

Fig. 1. Communication scheme of HDA* algorithm suitable for shared memory

With the aim of making possible the pruning of nodes that will lead to suboptimal solutions, the threads share a pointer to the best solution found so far by all the threads (*best_solution*) and its cost (*best_solution_cost*). Both variables must be protected, since two threads can find two different solutions and try to update these values at the same time.

The code that all the threads will run is identical, only thread 0 will be in charge of the additional tasks of generating the initial node and adding it to the input queue of its owner thread, initializing the common structures, detecting the termination state, and recovering from the shared memory the steps sequence that represents the solution to the problem once computation is finished.

Each thread will carry out a series of iterations until it detects the end of the computation (through a change in the value of the variable *end*). In each iteration, the following phases are performed:

- *Phase of node consumption from input* queue: the thread checks whether its own input queue is not empty. In that case, it tries to take the lock associated to the queue. When it obtains the access immediately, it takes all the pointers to nodes that were deposited on the queue, releases the lock, and then for each node whose cost is lower than *best_solution_cost* the thread performs the duplicate detection process adding them to the open list as appropriate.
- *Processing phase*: the thread processes at least *LNPI* (*Limit of Nodes per Iteration*) nodes from its open list. When the thread removes a node, it verifies if its cost is at least *best_solution_cost*. If it is so, the thread empties the open list since the nodes on it will lead to suboptimal solutions. Otherwise, it checks if the node represents the solution and in that case it updates *best_solution* and *best_solution_cost*, after having taken the lock that protects them and having consulted again if the node cost is less than *best_solution_cost*[2]. When the removed node is

[1] The input queues do not count how many times a deposit was done over them, but the total amount of nodes that they store (logical dimension). Because of that, the amount of nodes in transit is calculated instead of the amount of "communications" or "deposits" that have not been received yet. This is a modification of the *Safra and Dijskstra* algorithm.

[2] This is necessary because two threads can find two solutions with different cost at the same time. If a solution had not been found yet or if the two solutions that have just been found improve the current partial solution, when the threads try to obtain the lock to update the shared data, the thread with the best solution could update the data first, and then the second thread could

not the solution, it is inserted into the closed list, it is expanded (in this way successors are generated), and then for each successor the *Zobrist Function* is calculated so as to know which thread has to process it. When the node belongs to the thread that generated it, the thread carries out the duplicate detection and adds it on the open list as appropriate. Otherwise, the thread places the node in the local output queue for the target thread; when the amount of stored nodes on the output queue is higher than the limit *LNPT* (*Limit of Nodes per Transference*), the thread tries to take the lock of the target thread's input queue and, if it obtains the lock immediately, it transfers the stored nodes leaving the output queue empty[3]. A node transfer simply means a pointer copy. When a thread is the first one that deposits nodes on the input queue of another thread (i.e., at that moment the input queue was empty) it must inform the action, just in case the other thread was idle waiting for work.

- *Idle phase*: after the processing phase, if the thread stands idle because its open list is empty, it will send nodes stored on each non-empty output queue, and it will wait for any of the following events:

- *End of calculation*: thread 0 detected the *termination state* and it changed the value of the *end* variable, so that this state can be known.
- *Termination token arrival*: the thread must update the shared variables that correspond to the token, based on the termination algorithm, and pass it to the following thread, which means that the thread must change the value of the *token owner* field (informing the successor thread that it is the new owner). On the other hand, thread 0 verifies if the termination conditions are given. If this is so, the value of the *end* variable changes. Otherwise, it starts a new round to detect termination.
- *Work deposit on its own input* queue: the thread must obtain the lock associated to its input queue, it must take all the pointers to nodes that were deposited on the queue (leaving the input queue empty), and release the lock. For each node whose cost is lower than *best_solution_cost* the duplicate detection is carried out, adding the node to the open list as appropriate.

The termination detection algorithm involves updating the variable *state* (or *color*) and the *counter* of the thread every time it adds nodes to the input queue of another thread or every time it removes nodes from its own input queue, either increasing or decreasing the local counter of nodes that were deposited and received respectively.

To resolve performance degradation when an alloc-free relation between threads happens, a *pool* of pointers to node (*Memory Pool*) was incorporated to each thread, where the pointers to node that the thread wishes to "set free" for a further use are stored. This technique prevents access by thread A to the structures assigned to another thread B by the dynamic memory allocator, when the former wants to "free" a pointer allocated by the latter, situation that would produce contention.

Finally, the possibility to compile the algorithm to carry out a wait in a *passive* or *active* way when the thread stands idle was incorporated.

7 Experimental results

For the experimental work, a machine with two Intel ® Xeon ® E5620 [16] processors was used. Each processor has *four* 2.4 Ghz physical *cores*. Each *core* has two L1 caches of 64 KB for data and instructions respectively and one L2 cache of 256 KB. At the same time, all processor cores share a L3 cache of 12 MB. Each processor has a memory controller, therefore, the machine's memory design is NUMA and it uses a *QuickPath Interconnect (QPI)* interconnection of 5.86 GT/s. The machine has 32 GB of RAM, DDR3 1066 Mhz.

The tests were carried out taking into account the 100 initial configurations of 15-Puzzle used by [17], numbered from 1 to 100. Ten of the configurations with more steps for their solution [18] were also taken into account in the parallel algorithm scalability analysis, since for some of them resolution time is considerable. These were numbered from 101 to 110.

7.1 Sequential A*

7.1.1 Effect in the use of Jemalloc

The sequential algorithm was run with the 100 initial configurations and the final configuration suggested by [17] using the heuristic function presented in Section II.A and varying the dynamic memory allocator between *ptmalloc* and *Jemalloc*. *Jemalloc* was configured to work with 256 arenas, as this configuration will be used during the parallel algorithm tests.

For each initial configuration, 10 runs were performed and the runtime in seconds for each test was calculated. From the results, it can be observed that the average runtime of each configuration obtained from the samples that use *Jemalloc* presents a reduction ranging between 0.9% and 15.8% with respect to the average runtime for the same configurations using *ptmalloc*.

As it has been proved that when *Jemalloc* (configured to work with 256 arenas) is used with sequential A*, algorithm performance improves, the above mentioned allocator will be used in the tests from now on.

7.1.2 Effects in the use of Memory Pool technique

The sequential algorithm was run with the 100 initial configurations and the final configuration proposed by [17], using the heuristic function presented in Section II.A, *Jemalloc* (configured to use 256 arenas) and the *pool* of pointers to node "*Memory Pool*". For each initial configuration, 10 tests were run. Then, the average runtime in seconds obtained for each configuration was compared with

obtain the lock to carry out its update. If the condition about the cost is not verified again, the second thread could make effective the update and a suboptimal solution would be stored.

[3] This is different to the version proposed by Burns in which after each node generation that belongs to another thread t_j, the thread tries to take the lock associated to t_j's input queue. Moreover, here when the lock is obtained all the nodes stored on the output queue for the target thread are communicated, this is another difference with Burns' version.

the results presented in the previous section that do not use the *"Memory Pool"* technique.

From the comparison, it can be observed that the "*Memory Pool" technique* does not bring any advantage for the sequential application: 17 configurations suffered an increment of their average runtime of about 2% and 9%; 15 configurations increased their average runtime between 1% and 2%; 43 configurations achieved a modest increase in its average runtime which goes between 0% and 1%; finally, 25 configurations reduced their average runtime between 0% and 6.45%.

Generally, relevant variations in performance achieved by instances with a significant runtime are not observed. Thus, the sequential results obtained without using the "*Memory Pool*" technique will be used for the performance analysis of the parallel algorithm.

7.2 HDA* for shared memory

The HDA* parallel algorithm is nondeterministic, i.e. when different experimental samples are taken for the same initial/ final configuration and the same parameters, the results obtained by the algorithm may be different. That is possible because an initial configuration can have multiple optimal solutions and, as threads distribute the space of states dynamically among themselves, the nodes processed by a thread will vary depending on how asynchronous events occur in the system.

In the tests, affinity was used to allocate each thread to an exclusive *core* using the function *sched_setaffinity*() [19]. In those tests with 4 threads 1 pair of threads was allocated to each machine processor, and in those tests with 8 threads 1 thread was allocated to each physical *core* of the machine.

The selected initial configurations are those used in Section VII.A whose sequential runtime is of at least 5 seconds[4]. For performance analysis, the configurations numbered from 101 to 106 were also taken into consideration; sequential and parallel tests for configurations 107, 108, 109 and 110 exhausted available RAM memory and were therefore aborted.

The Jemalloc memory allocator, configured to work with 256 arenas, and the heuristic function presented in Section II.A were used. For each initial configuration and each parameter group, 100 samples were obtained. The parameters are: the amount of cores/threads, whose values vary between 4 and 8; *LNPI* between 1, 5, 50 and 500; *LNPT* was set in 26 nodes. Then, the *average runtimes* resulting from the 100 runs for the same configuration and set of parameters, which will be called *average sample,* were obtained.

7.2.1 Passive waiting vs. active waiting

Two test sets were run using *active waiting* and *passive waiting* respectively, *LNPT was* limited to 26 and the *Memory Pool* technique was used.

Average runtimes brought by the test sets do not show an apparent benefit for any particular waiting technique. This may be due to algorithm asynchronism, as most times threads perform attempts to take locks and in case they do not get it they keep on working. Additionally, each thread is run on an exclusive core. Therefore, either waiting actively or resorting to the Operating System to perform a passive wait does not cause drastic changes in performance.

7.2.2 Effects in the use of Memory Pool technique

Two test sets were run including the *Memory Pool* technique or not; *LNPT* was limited to 26, and *active waiting was used*.

Average runtimes of the test set that uses *Memory Pool* reduced the average runtimes of the test set that does not use that technique between 4.5% and 12.82%. Generally, the reduction in the average runtime for the samples with 4 threads is between 4.5% and 8.64%, while the reduction in tests with 8 threads is between 6.43% and 12.82%.

Therefore, the advantage of this technique for reducing contention in the access to structures assigned to each thread by the dynamic memory allocator, in cases with an existing producer-consumer relation between threads by alloc-free operations, is shown.

7.2.3 Performance Analysis

The experimental tests discussed in the previous section, which optimized the results, were considered to assess parallel algorithm performance[5]. Moreover, tests were carried out for configurations 101 to 106 following the same strategy. Then, for each configuration and number of threads, the *average sample* that minimizes average runtime, i.e. the sample whose *LNPI* parameter value optimizes performance, was selected.

To assess algorithm scalability, the average samples selected for each configuration were organized according to their sequential workload (sequential time). In this sense, escalating the problem means increasing the number of processed or generated nodes. On the other hand, the architecture is escalated by increasing the number of cores used to solve the problem.

Fig. 2 shows the Speedup obtained by the average sample selected for each configuration using 4 cores and 8 cores, while Fig. 3 shows the Efficiency obtained. For tests with 4 cores, the Speedup obtained varies from 2.95 to 4.01, while Efficiency ranges from 0.73 to 1.0034. Tests with 8 cores show a Speedup between 5.14 and 8.15, and Efficiency between 0.64 and 1.018.

Both average samples that obtained a superlinear Speedup present a negative Search Overhead[6] (-2.92 for the average sample with higher Speedup with 4 cores and -9.86 for the average sample with higher Speedup with 8 cores). Therefore, the parallel algorithm processes fewer nodes than the sequential algorithm. This situation is possible for this class of algorithms due to the causes explained in [20].

[4] Configurations are as follows: 3, 15, 17, 21, 26, 32, 33, 49, 53, 56, 59, 60, 66, 82, 88, 100

[5] The tests of interest are those that use active waiting, limiting LNPT in 26 and using the *Memory Pool* technique.

[6] The Search Overhead represents the percentage of increment in the number of nodes expanded by the parallel algorithm against the sequential algorithm and it is calculated with the formula 100x(NP/NS -1), where NP= number of nodes processed by the parallel algorithm and NS = number of nodes processed by the sequential algorithm.

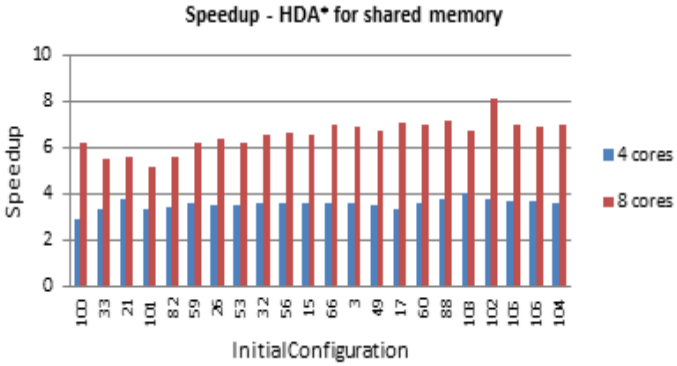

Fig. 2. Speedup achieved by the HDA* algorithm for shared memory, by configuration

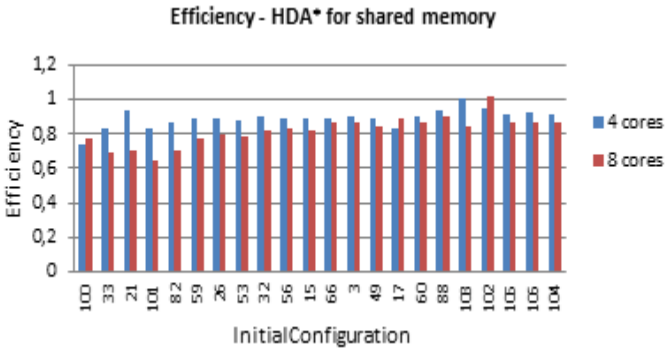

Fig. 3. Efficiency achieved by the HDA* algorithm for shared memory, by configuration

After analyzing the results shown in Fig. 2 and 3, it can be concluded that, for the same workload (initial configuration), if the number of cores is increased, the Speedup obtained is better. This proves that the problem is solved faster as more cores are used. However, efficiency does not normally remain constant. This decrease in efficiency is due to different factors, such as sequential parts especially at the beginning and at the end of computation, synchronization, idle time, load unbalance, search overhead increase, among other factors.

It is observed that when the problem is escalated maintaining the same number of processors, efficiency generally improves or remains constant as *overhead* is less significant on total processing time.

8 Conclusions and future lines of work

A version of the HDA* algorithm that is suitable for shared memory architectures and incorporates an effective technique to avoid performance degradation when there is a producer-consumer relation between various threads due to alloc-free operations was presented. The algorithm was run taking the Puzzle problem as study case and a more polished heuristic with respect to the classical one. On the other hand, it was proved that using active or passive waiting when the thread becomes idle is irrelevant, as there are no significant variations in performance.

This paper shows a scalability analysis of the parallel algorithm on a machine with multicore processors. From the results obtained, it can be concluded that the behavior exhibited is typical of a scalable parallel system, where efficiency can be kept constant when workload and architecture are escalated.

Future lines of work focus on contrasting the algorithm presented in this paper against HDA* for distributed memory (implemented exclusively with MPI), comparing the performance achieved and the amount of memory used.

9 References

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

[2] P. Hart, N. Nilsson and B. Raphael, "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 and V. N. Rao, "Parallel Best-First Search of State-Space Graphs: A Summary of Results", de Proceedings of the 10th Nat. Conf. AI, AAAI, 1988.

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

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

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

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

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

[9] E. W. Dijkstra, W. H. J. Feijen and A. J. M. Van Gasteren, "Derivation of a termination detection algorithm for distributed computations", Information Processing Letters, vol. 16, pp. 217-219, 1983.

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

[11] R. Korf and L. Taylor, "Finding Optimal Solutions to the Twenty-Four Puzzle", Proceedings of the Thirteenth National Conference on Artificial Intelligence, 1996.

[12] W. Gloger, "Wolfram Gloger's malloc homepage", 2014. http://www.malloc.de/en/.

[13] Aho, J. Ullman and J. Hopcroft, Data Structures and Algorithms, First Edition, Boston, MA: Addison-Wesley Longman Publishing Co, 1983.

[14] R. Ramakrishnan and J. Gehrke, Database Management Systems, Second Edition, McGraw-Hill Education, 1999.

[15] A. L. Zobrist, "A New Hashing Method with Application for Game Playing", University of Wisconsin, Madison, 1968.

[16] Intel Ark, 2010. http://ark.intel.com/products/47925.

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

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

[19] D. Gove, Multicore Application Programming. For Windows, Linux and Oracle (c) Solaris. Developers' Library, Boston, MA: Pearson Education, 2011.

[20] Grama, A. Gupta, G. Karypis and V. Kumar, Introduction to Parallel Computing (Second Edition), Edinburgh Gate, Harlow, Essex: Pearson, 2003.