Scalability analysis of Hash Distributed A* on commodity cluster: results on the 15-puzzle problem

Victoria Sanz (1,2), Armando De Giusti (1,2) and Marcelo Naiouf (1)
(1) III-LIDI. School of Computer Science, UNLP, Argentina
(2) CONICET. Ministry of Science, Technology and Productive Innovation, Argentina
Email:{vsanz,degiusti,mnaiouf}@lidi.info.unlp.edu.ar

Abstract—**The A* algorithm is generally used to solve combinatorial optimization problems, but it requires high computing power and a large amount of memory. In this sense, Hash Distributed A* (HDA*) parallelizes A* in order to benefit from the computing power and the accumulated memory provided by clusters. The parallelization is done by applying a decentralized strategy and using a hash function to distribute nodes among processes. In this paper, we present a detailed implementation of HDA* using MPI, which includes a parameter that determines the number of nodes to be computed by each process per iteration of the algorithm. The experimental work is carried out on a commodity cluster, using the 15-puzzle as study case. Our experimental results reveal that the included parameter favors performance. Finally, we present a performance analysis of HDA*, as the problem size and the number of processors increase, which indicates that the algorithm scales well.**

Keywords:Hash Distributed A*, Commodity Cluster, Scalability, 15-Puzzle.

I. Introduction

In the area of Artificial Intelligence, heuristic search algorithms are used as the basis to solve combinatorial optimization problems that require finding 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 widely used search algorithms for that purpose is A* [1], [2], a variant of *Best-First Search*, which browses the graph that represents the state space of the problem using a cost function $\hat{f}$ to value the nodes, in order to process the most promising paths first. To that end, function $\hat{f}$ contains known cost information of the path from the initial node s to the current node n ($\hat{g}$), as well as heuristic information to estimate the unknown cost of the path that goes from the current node n to the solution node t ($\hat{h}$), which can never overestimate the actual cost. The algorithm is different from conventional methods because the search tree is implicit and dynamically generated. During the process, it keeps two data structures: one for the unexplored nodes sorted by function $\hat{f}$ (*open list*), and another for the already explored nodes (*closed list*) used to avoid processing the same state multiple times. In each iteration, the most promising node (according to $\hat{f}$) available on the open list is removed, it is added to the closed list, and legal actions are applied to it to generate successor nodes that will be added to the open list under certain conditions (verification known as *duplicate detection*). The search process continues until the node that represents the solution is removed from the open list.

The major drawback of A* is that it requires high computing power and a large amount of memory, as a consequence of the exponential or factorial growth of the state space of the problem. Therefore, over the last decades, the development of parallel A* algorithms has been promoted which, in particular, may benefit from the computing power and the accumulated memory provided by clusters.

In this sense, Hash Distributed A* (HDA*) [3] parallelizes A* by applying a *decentralized strategy* (i.e. each processor has its own open and closed lists and performs a quasi-independent search) and using a hash function to assign each state of the problem to a single processor. In this way, when a processor generates a node, it determines who the owner is and transfers the node to that owner. This mechanism allows balancing the workload and pruning *duplicate nodes* (i.e. nodes representing the same state) in an *absolute* way, as they are always sent to the same processor. The implementation of HDA* proposed by these authors uses MPI and asynchronous communication. Although they carried out an extensive experimental work on an *HPC cluster* with Infiniband interconnection, for applications with different heuristic computation time such as the domain-independent planning and the Sliding Puzzle, the tests carried out on a conventional cluster with Ethernet connection did not take into account the latter application.

In this paper, we present a detailed implementation of HDA*, which includes the *LNPI* (Limit of Nodes per Iteration) parameter that determines the number of nodes to be computed by each process per iteration of the algorithm. In this sense, our version differs from the original algorithm, since in the latter each process computes a single node per iteration. The implementation was carried out using MPI and the 15-Puzzle was selected as study case. The experimental work is focused on analyzing the speedup and efficiency obtained by the parallel algorithm when it is run on a cluster of multi-core machines connected through Ethernet (a *commodity cluster*), as the problem size and the number of processors increase. Our experimental results reveal that the included parameter favors performance. Finally, this analysis shows that the implemented version scales well.

II. Related work

So far, different authors have presented various techniques to parallelize *A**, which vary in the method used to manage the open and closed lists and the strategy used to balance load among processors during the execution. The technique to be chosen will depend on the architecture and the problem to solve [4].

The commonly used parallelization technique is known as *decentralized strategy* [5], and it is based on the following: each processor has its own local open and closed lists, and carries out a quasi-independent search. This strategy is suitable both for shared memory and distributed memory architectures. However, communication among processors is needed due to the following reasons: the search tree is generated at run time, therefore, the workload should be distributed dynamically; *duplicate nodes* can be generated by different processors and should be pruned in order to prevent processors from performing redundant work (it is possible to achieve an *absolute* duplicate detection and pruning by using a hash function that assigns each state to a particular processor); the termination criterion should be modified because if the search is ended when a solution is found, there will be no guarantee that such solution is the best one; the costs of the partial solutions found so far should be communicated in order to use them to prune the paths that lead to suboptimal cost solutions.

The earliest parallel A* algorithms based on the *decentralized strategy* used receiver/sender initiated load balancing algorithms [4], [5], [6], [7]. However, those techniques cause duplicate nodes to arise on the open or closed lists of different processors in the system, because they usually involve carrying out a *partial* duplicate detection and pruning, i.e., that procedure is performed only by the donor processor and the recipient processor. Therefore, redundant work is carried out by different processors, which in turn increases the *Search Overhead*[1] and the amount of RAM consumed by the parallel algorithm, situation that is even worse as more processors are used. Consequently, an *absolute* duplicate detection and pruning procedure should be applied to obtain higher performance.

In this sense, the Hash Distributed A* algorithm (HDA*) [3] parallelizes A* by applying a *decentralized strategy* and using *Zobrist's hash function* [8] to assign each state of the problem to a single processor. Thus, when a processor generates a node that does not belong to it, it determines who the owner is and transfers the node to that owner. This mechanism allows balancing the workload and pruning duplicates in an *absolute* way, as the nodes representing the same state are always sent to the same processor.

The implementation of HDA* proposed by these authors uses MPI and asynchronous communication, so the algorithm is suitable for execution both on distributed memory and shared memory architectures. To avoid congestion in the communication medium caused by messages being sent containing a single node, the algorithm uses the technique proposed in [9], which involves packing in one message a given number of nodes whose recipient is the same before sending the message (in this paper, we refer to this value as *LNPT*, or *Limit of Nodes per Transfer*).

The experimental work carried out by these authors uses the domain-independent *Fast Downward planner* [10], placing HDA* as the search mechanism. On the other hand, it uses the *24-Puzzle* problem, a specific application where processing a state is faster, together with a *disjoint pattern database* heuristic [11], [12], but they exclude the time required for reading these data from the disc. The authors note that the speed of processing a state significantly affects the *efficiency* of the parallel search algorithm: when processing a state is expensive, the impact of parallelization-related overheads, such as communication and synchronization, tends to decrease. For this reason, studying the efficiency achieved by HDA* for various types of applications running on different architectures is of interest.

Consequently, the authors analyze the speedup and efficiency achieved by HDA* on a single, 32GB RAM multicore machine, for planning problems. However, they do not assess performance for the *24-Puzzle* since the instances they used exhaust RAM before finding a solution.

Additionally, the authors study the scalability of HDA* for the applications mentioned above on a multicore cluster with Infiniband interconnection (*HPC cluster*). They show that HDA* achieves good performance and scales relatively well for complex planning problems that require large amounts of RAM. On the other hand, the performance obtained for the *24-Puzzle* application is not as satisfactory as in the previous case, and it rapidly degrades when increasing the number of cores in the architecture.

Similarly, they assess the scalability of HDA* for planning applications on a multicore cluster with Ethernet connection (*commodity cluster*), obtaining an almost linear relative speedup. However, they do not evaluate the performance on this architecture for the *24-Puzzle*.

HDA* is currently interesting due to its simplicity and good scalability. On the other hand, the Sliding Puzzle has recently gained relevance because it is related to real problems such as moving pallets with an automated guided vehicle in high-density storage warehouses [13]. Also, research centers usually have clusters formed by connecting multicore machines through conventional networks such as Ethernet. It is for all these reasons that the study of HDA* behavior on this type of clusters for solving the Sliding Puzzle is an open research line.

III. Hash Distributed A*

Hash Distributed A* (HDA*) [3] parallelizes A* based on a *decentralized strategy*. It was programmed using exclusively the MPI message passing library and asynchronous communication; therefore, it is suitable for execution both on distributed

[1]The Search Overhead represents the percentage of nodes that the parallel algorithm expands in excess as compared to the sequential algorithm. It is calculated using the formula 100x(NP/NS -1), where NP is the number of nodes processed by the parallel algorithm and NS is the number of nodes processed by the sequential algorithm.

memory as well as shared memory architectures. It uses a *hash function* to assign states to processes, thus it implicitly achieves load balancing and absolute duplicate detection (two nodes representing the same state are sent to the same process, which in turn checks for duplicates in its local structures).

Initially, the relevant process adds the initial node to its open list. Then, each process carries out iterations until a global optimal solution is reached. In each iteration:

1) It checks if one or more nodes have been received through messages. If so, for each node received, it carries out the *duplicate detection* to determine if the node must be added to the open list or if it should be discarded.
2) If no messages containing nodes were received, the process selects a node from its open list (the one with the lowest $\hat{f}$-value) and expands it, generating successor nodes. Then, for each successor, it calculates the *hash value* to identify the *owner process* and, if the node belongs to another process, it sends asynchronically the node to its owner through a message.

To reduce the communication overhead, the idea proposed in [9] is used, which involves packing within a single message a given number of nodes whose recipient is the same (in this paper, this number is referred to as *Limit of Nodes per Transfer*, or *LNPT*). The number of nodes to be packed depends on factors such as communication medium speed and number of processors, among others.

To obtain a uniform distribution of nodes, which is necessary to achieve an effective load balancing, *Zobrist's hash function* [8] is used. On the other hand, to detect the end of the computation, i.e., to know the state in which all processes are idle and there are no messages in transit, *Mattern's time algorithm* [14] is used.

IV. Implementation of Hash Distributed A*

We developed our own version of the HDA* algorithm, which is described in this section. For its implementation, the following tools were used: MPI; asynchronous communication; non-blocking query for a message's arrival (MPI_Iprobe) when the process is not idle, blocking query (MPI_Probe) otherwise; the termination detection algorithm proposed by Dijkstra and Safra [15], since we opted for not increasing the size of each *work message* sent with additional information[2]; *Zobrist's hash function* to assign nodes to processes; and the technique proposed in [9] to package a given number of nodes (*LNPT*) before sending the *work message* to its recipient process.

Each process carries out an A* search locally and communicates with its peers for any of the following reasons: sending/receiving *work messages* containing nodes, sending/receiving the *costs of solutions* found, sending/receiving the *termination token*, sending/receiving *termination notification* messages.

[2]This algorithm is similar to Mattern's time algorithm [14], [16]. Both are based on the same idea of counting messages. The main difference between them is that Dijkstra and Safra's algorithm is based on the *double wave* approach, whereas Mattern's time algorithm is based on the *single wave* approach (at the expense of increasing the amount of control information or augmenting every message with a time stamp).

Locally, each process has its open and closed lists, which will be empty at first, the cost of the best global solution known so far (*best_solution_cost*), the best solution found by the process (*best_solution*), the data required by the termination detection algorithm, and a variable that changes its value when the computation reaches its end (*end*). For the purpose of packing several nodes in a message before sending them to their owner, processes are equipped with a buffer for each peer process (*send_buffer*); each buffer will contain node records, its physical dimension is known as *LNPT* (Limit of Nodes per Transfer), and its logical dimension must be kept updated to know the number of nodes that have been accumulated.

Although the code is the same for all processes, process 0 is in charge of: loading the initial configuration; generating the initial node, which will be added to the open list of this process (if the node belongs to it) or which will be sent to the corresponding owner process; and carrying out the termination detection, which involves sending the termination notification to the other processes.

Each process performs a series of iterations until receiving the termination notification; at that moment the optimal solution has been reached. In each iteration, the following stages are carried out:

1) *Work message reception stage*: the process checks, in a non-blocking manner (MPI_Iprobe), if *work messages* containing nodes have arrived. If so, it receives each message and, for each node record whose cost is lower than *best_solution_cost*, it carries out the following actions: it allocates space in dynamic memory; it assigns the record received to the allocated space; it performs the duplicate detection adding the node to the open list as appropriate.
2) *Cost message reception stage*: the process checks, in a non-blocking manner (MPI_Iprobe), if *cost messages* containing the cost of a better solution found have arrived. If so, it receives the messages and updates the local variable *best_solution_cost* as appropriate. Now, if the cost of the best open node (according to function $\hat{f}$) is at least *best_solution_cost*, the process empties its open list since the nodes stored in it would lead to suboptimal solutions.
3) *Processing stage*: the process expands at most *LNPI* (Limit of Nodes per Iteration) nodes from its open list. For each extracted node, the process checks if its cost is at least *best_solution_cost*. If so, the process empties the open list. Otherwise, it checks if the node represents the solution and in that case it empties the open list and updates *best_solution* and *best_solution_cost*. When the extracted node is not the solution, it is added to the closed list, it is expanded (i.e., *successor nodes* are generated) and then, for each successor, the *Zobrist function* is calculated to determine its corresponding

owner process. When the successor belongs to this very process, it carries out the duplicate detection and adds the node to its open list as appropriate. Otherwise, the process adds the node record to the *send_buffer* for the destination process and, if that buffer is full (i.e., it already has *LNPT* nodes), it sends the *work message* asynchronically.

4) *Idle stage*: the process goes into this stage when its open list is empty. If a new solution was found in the *processing stage*, the process sends the solution cost to its peers. It then sends *work messages* to those destination processes whose *send_buffer* contains nodes that were not communicated. Finally, it remains waiting (MPI_Probe) for any type of message: (1) work message, (2) cost message (3) token message, (4) termination notification message. The process ends this stage when it has nodes on its open list, as a result of having received a work message, or when it receives the termination notification message. Messages of types (1) and (2) are processed in a similar way as described above; messages of the type (3) are processed based on the termination detection algorithm (knowing that the process is idle and it must send the token to the next process or assess the termination condition in the case of Process 0); messages of the type (4) are processed by changing the value of variable *end*, and by doing so, algorithm ends.

Each *work message* that is sent or received is processed as indicated by the termination detection algorithm. Three additional fields are added to the token message to make solution retrieval possible: cost of the best global solution found so far, ID of the process that found that solution, and memory address for the solution node. Before sending the token, the process must update these fields with its own information if it has the best solution so far.

When computation ends, the solution is retrieved in a distributed manner, obtaining the sequence of actions that allows transforming the initial state into the final state.

V. Experimental results

Experimental tests were carried out on a cluster composed by 7 machines connected through 1Gb Ethernet. Each machine has two Intel Xeon E5620 processors and 32GB RAM. Each processor has four 2.4Ghz physical cores. Each processor has a memory controller and uses a 5.86 GT/s *QPI* connection.

A* and HDA* were configured to use the *Jemalloc* memory allocator (with 256 arenas) and a heuristic that is a variation of the *Sum of Manhattan Distances (SMD)* of the tiles with the addition of linear conflicts detection, the detection of the last moves applied and an analysis of corner tiles [17]. We showed in [18] that this configuration improves the performance of A* versus using the default memory allocator (*Ptmalloc*) and the *SMD* heuristic.

A* was run on a single machine of the previous cluster.

HDA* was run on various cluster configurations, i.e., varying the number of machines used between 2 and 7. For all tests, 4 processes were assigned to each machine used[3] (two processes per processor in the machine). The parameters and values used were: number of processes/cores (8, 12, 16, 20, 24, 28), LNPI (1, 5, 50, 500) and LNPT (26, 210, 1680). The LNPT values correspond to work messages whose size is 1KB, 8KB and 64KB, respectively.

The tests considered the 15-Puzzle instances used in [19] whose sequential execution time is of at least 5 seconds (numbered 3, 15, 17, 21, 26, 32, 33, 49, 53, 56, 59, 60, 66, 82, 88, 100) and six of the 10 configurations that are part of the test suite proposed in [20] (numbered 101-106 in this paper). Thus, the 22 configurations used present different levels of complexity, which is measured in terms of the number of nodes processed by A*, and varies between 1 and 103 million processed nodes.

We have selected the instances mentioned above because they are solvable on a single 32GB RAM machine. Serial runtimes can not be measured for hard problems, for example *Korf's 24-puzzle instances*, using our architecture because A* exhausts RAM before finding a solution. The same problem arises when running HDA* on a single 32GB RAM multicore machine for those instances [3]. While it may be possible to solve some *Korf's 24-puzzle instances* on our cluster and measure *relative* speedup and efficiency as in [3], as future work we intend to analyze the scalability of our version of HDA*, implemented using MPI, on a multicore machine for the Sliding Puzzle problem, and to compare the results achieved with those presented here and those obtained by our optimized version of HDA* for multicore machines, implemented using Pthreads [18], [21].

HDA* is non-deterministic, i.e., when multiple runs are carried out using the same input data (initial and final states), the results generated by the algorithm may be different. For this reason, 10 tests were carried out for each cluster configuration, initial configuration and set of parameters. The data obtained with these 10 runs were then averaged, which will be referred to as *mean sample*.

In the following sections, we analyze the impact of the LNPI and LNPT parameters on the performance of HDA*, and then we assess its scalability when the parameters values that experimentally improved performance are used.

A. Limit of Nodes per Iteration (LNPI)

The LNPI parameter determines how many nodes a process must expand per algorithm iteration, i.e., it establishes the interval for checking the arrival of *work messages* and *cost messages*.

To analyze the impact of this parameter on execution time, all *mean samples* obtained from runs carried out for LNPT=26 (i.e., 1KB *work messages*) were taken; and those with the same initial configuration and number of cores were grouped. That

[3]It was observed that when 8 processes per machine are used, the performance obtained is poor. This is because each machine has a single network controller and therefore bottlenecks are caused by network I/O; this is even worse because services communicate through the same network.

TABLE I
SD AND CV RANGES SORTED BY LNPT

LNPT	SD range (seconds)	CV range
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

is, each group contained the *mean samples* whose only difference among them was the LNPI parameter value. For each group, *Standard Deviation* (SD) and *Coefficient of Variation* (CV) of execution time were calculated, since these values measure how much the execution time of a *mean sample* in the group tends to deviate from the *group mean time*.

From the data obtained for LNPT=26, it was observed that only 60% of the groups have SD values below 1, and 39% below 0.5. Similarly, only 11% of the groups have CV values below 0.1.

The procedure described above was also applied to *mean samples* obtained from runs carried out for LNPT=210 and LNPT=1680 (i.e., 8KB and 64KB *work messages*, respectively).

Table I shows the ranges of SD and CV values for the groups, sorted by the LNPT value. As it can be seen, when *work messages* contain few nodes (small LNPT), the LNPI value has a greater impact on execution time, since there is a greater difference in the execution times of the *mean samples* in each group (greater CV). The results obtained indicate that there is a significant variation in execution time when changing the value of the LNPI parameter among those defined (1, 5, 50 or 500), mainly for LNPT=26 and LNPT=210.

The LNPI value that improves performance depends on the initial configuration, the number of processes and the LNPT value.

For LNPT=26, the LNPI value that improves overall performance is 50. The configurations that presented improvements with LNPI=500 are some of the most complex ones (60, 88, 105, 106 and 104); however, as the number of cores increases, the number of configurations that favored such value decreased. The configurations that presented improvements with LNPI=5 are some of the less complex ones (3, 21 and 49) with a large number of cores.

For LNPT=210 and LNPT=1680, performance improves with LNPI=50 or LNPI=500. In contrast with the previous case, there is no clear preference for either of these two values.

From the data presented above, the following can be concluded:

1) When LNPT is small, processes will send numerous *work messages* containing few nodes. If LNPI is set to a very high value, processes will carry out too much speculative work on their local nodes in each *processing stage*, without adding newly arrived nodes, increasing the *Search Overhead*, which is even worse as the number of processes increases. Therefore, the frequency of checking for message's arrival has to be increased (low LNPI) to allow the addition of nodes that are among the global best ones.
2) When LNPT is large, processes will send few *work messages* containing many nodes. If LNPI is set to a high value, the number of checks for message's arrival, that are likely to fail, decreases. Otherwise, if LNPI is set to a low value, performance is not affected that much because non-blocking checks are used. No significant variations in *Search Overhead* are observed with LNPI changes.

It should be noted that setting LNPI to 1 never resulted in improved performance. This indicates that the parameter that was added to our own version of HDA* is indeed relevant and an original contribution in the area.

B. Limit of Nodes per Transfer (LNPT)

The LNPT parameter indicates the maximum number of nodes that will be included in each *work message*.

For the purpose of analyzing the effect of the LNPT parameter on execution time, all *mean samples* obtained from runs limiting LNPT to 26 nodes (1KB messages), 210 nodes (8KB messages) and 1680 nodes (64KB messages) were taken. Then, all *mean samples* with identical initial configuration, number of cores and LNPI values were grouped; i.e., each group contained *mean samples* whose only difference was the value for their LNPT parameter. For each group, *Standard Deviation* (SD) and *Coefficient of Variation* (CV) of execution time were calculated, since these values measure how much the execution time of a *mean sample* in the group tends to deviate from the *group mean time*.

In general, the results obtained show that the SD for the groups is between 0.045 and 24.37. This indicates that when varying the LNPT parameter between the values defined, *mean samples* execution times tend to deviate at most in 24.37 seconds from their *group mean*. It should be noted that 46.21% of the groups has a SD value below 1, and 23.10% has a SD value below 0.5.

On the other hand, the general CV values obtained for the groups range from 0.019 to 1.15; i.e., the execution time of a *mean sample* in the group tends to deviate between 1.9% and 115% from the *group mean* when the LNPT parameter changes between the values defined. Only 24.4% of the groups have CV values below 0.1.

Based on the high CV values, it is concluded that the LNPT parameter affects execution time. This is because this parameter impacts network traffic, process activity and volume of speculative work carried out. The following can be inferred:

1) A low value of LNPT will cause processes to generate *small work messages*, which increases the number of work messages that are sent from one process to another over the network using MPI, which in turn generates an overhead for handling buffers associated to communications and network congestion.
2) Very high values of LNPT can also degrade performance because, if a process packs too many nodes for a recipient process, it could cause the latter's idleness when it

TABLE II
SPEEDUP AND EFFICIENCY FOR EACH LNPT VALUE AND NUMBER OF PROCESSES

LNPT	Processes/Cores	Speedup	Efficiency
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

has only few nodes to process or when it is currently idle due to lack of work. The process could also delay the transmission of higher quality nodes (according to $\hat{f}$) than those that are currently being processed by the recipient, which would increase the *Search Overhead*.

Table II summarizes the ranges for Speedup and Efficiency, sorted by LNPT value and number of processes/cores. For each LNPT, initial configuration and number of cores, the *mean sample* with the best performance was selected (i.e., that whose LNPI value reduces execution time).

In most of the cases, the value of LNPT that improves execution times for each configuration and number of cores is 210, i.e., 8KB *work messages*. There are some exceptions with some low-complexity configurations as the number of processes scales, for which performance improved using LNPT=26, since higher values of LNPT increased *Search Overhead*. Other exceptions were observed with some of the more complex configurations when only a few processes were used; in these cases, the best performance was obtained with LNPT=1680 because it produces a lower *Search Overhead* or a lower *load unbalancing*.

The following configurations had their best performance with LNPT=26: 21 with 12 processes; 100, 21 and 101 with 16 processes; 100, 21, 101 and 82 with 20 processes; 100, 33, 21, 101 and 82 with 24 processes; 100, 33, 21, 101, 82, 59 and 53 with 28 processes. The following configurations had their best performance with LNPT=1680: 88, 102, 106 and 104 with 8 processes; 104 with 12 processes.

From the above, it can be concluded that, for less complex configurations, as the number of processors increases it is better to use *smaller work messages* (smaller LNPT). In the case of more complex configurations and only a few processes, the best performance is occasionally obtained with *larger work messages* (larger LNPT). However, empirically for this architecture, as the workload and the number of processes/cores scale, better performance is obtained when using 8KB *work messages* (i.e., LNPT= 210).

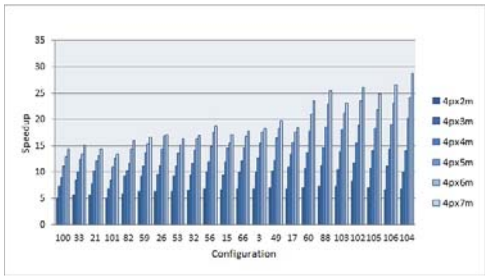

Fig. 1. Speedup obtained by HDA* on cluster

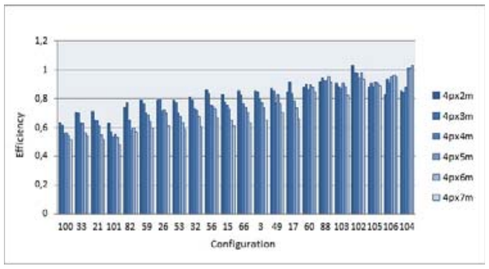

Fig. 2. Efficiency obtained by HDA* on cluster

C. Performance analysis

To analyze the performance of HDA*, for each initial configuration and number of processes, the *mean sample* that minimized resolution time was selected; that is, the sample whose LNPI and LNPT values resulted in the best performance.

To assess algorithm scalability, the various selected *mean samples* were sorted by configuration complexity (that is, based on the number of nodes processed by A*, which is related to sequential execution time). In this sense, scaling the problem means increasing the number of processed /generated nodes. On the other hand, the architecture is scaled by increasing the number of cores/processes used to solve the problem.

Figure 1 shows the speedup obtained by the *mean sample* selected for each initial configuration using 8, 12, 16, 20, 24 and 28 processes/cores (4 processes per machine), while Figure 2 shows the efficiency obtained.

After analyzing the data presented, it can be concluded that, if workload is constant (initial configuration) and the number of cores/processes is increased, speedup improves, meaning that less time is required to solve the problem. However, this improvement does usually not keep efficiency at a constant value. The decrease in efficiency is due to factors such as: sequential portions of the algorithm, particularly at the beginning and at the end of the computation, synchronization, communication, idle time, load unbalancing, increased search overhead, and so forth.

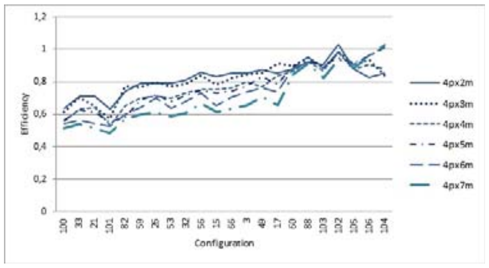

Fig. 3. Scalability for HDA* on cluster

The superlinear speedup obtained for the configuration 102 and 8 processes was due to the fact that the parallel algorithm processed a lower number of nodes than the sequential algorithm, which is possible in this type of search algorithms[4], [5]. The remaining cases are due to the decrease in the number of elements in the open-list and closed-list structures, which causes an acceleration of the operations carried out on them.

Figure 3 shows the efficiency as the workload (problem complexity) and the number of processes/cores increase. As it can be observed, when problem complexity is scaled and the number of cores used is constant, the efficiency generally improves or remains constant because the overhead is less significant for total processing time.

Based on the results presented above, it can be concluded that the behavior presented by the algorithm, when it is run on cluster configurations with 4 processes per machine, is typical of a scalable parallel system, where the efficiency can be kept at a constant value as both problem size and number of processors are increased.

VI. Conclusions and future work

We developed our own version of HDA*, which differs from the original version in that it includes the LNPI (*Limit of Nodes per Iteration*) parameter, which indicates the maximum number of nodes to be processed in each iteration.

The effect of the LNPI and LNPT parameters (the latter determines the size of *work messages*) on performance was analyzed. It was concluded that the LNPI value that improves performance depends on the initial configuration, the number of processes and the size of *work messages* (LNPT). It was noted that performance never improved by processing one node per algorithm iteration (LNPI=1), as done in the original version, which indicates that the parameter added to our version favors performance. On the other hand, it was established that, empirically for this architecture, as the problem size and number of processes/cores increase, better performance is obtained when using *8KB work messages* (LNPT=210).

Finally, algorithm scalability was assessed. The results obtained indicate that the parallel algorithm, if run on cluster configurations with 4 processes per machine, presents the typical behavior of scalable parallel systems.

As for future work, we intend to compare performance achieved and memory consumed by HDA* for shared-memory architectures and HDA* for distributed-memory architectures, when they are run on a multicore machine. The former algorithm was presented in [18], [21] and implemented with Pthreads, and the latter was introduced in this paper and implemented with MPI. The results will allow assessing if there are any potential benefits of converting HDA* into a hybrid application that uses programming tools for shared and distributed memory when the underlying architecture is a multicore cluster.

References

[1] Hart et al. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 1968; 4(2):100-107.
[2] Russel et al. *Artificial Intelligence: A Modern Approach* 2nd ed. Prentice Hall: New Jersey, 2003.
[3] Kishimoto et al. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence* 2013; 195: 222-248.
[4] Kumar et al. Parallel Best-First Search of State-Space Graphs: A Summary of Results. *Proceedings of the 7th Nat. Conf. AI*. AAAI:1988; 122-127.
[5] Grama et al. *Introduction to Parallel Computing* 2nd ed. Pearson: Harlow, 2003.
[6] Dutt et al. Parallel A* Algorithms and Their Performance on Hypercube Multiprocessors. *Proceedings of Seventh International Parallel Processing Symposium*. IEEE Computer Society:1993; 797-803.
[7] Sanz et al. Parallel Optimal and Suboptimal Heuristic Search on multicore clusters. *Proceedings of The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*. CSREA Press:2011; 673-679.
[8] Zobrist. *A New Hashing Method with Application for Game Playing*. Computer Sciences Department, University of Wisconsin: Madison, 1968. Technical Report 88.
[9] Romein et al. A performance analysis of transposition-table-driven work scheduling in distributed search. *IEEE Transactions on Parallel and Distributed Systems* 2002; 13(5): 447-459.
[10] Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 2006; 26: 191-246.
[11] Culberson et al. Pattern databases. *Computational Intelligence* 1998; 14(3): 318-334.
[12] Korf. Recent Progress in the Design and Analysis of Admissible Heuristic Functions. *Proceedings of the 4th International Symposium on Abstraction, Reformulation, and Approximation*. Springer:2000; 45-55.
[13] Gue et al. GridStore: A Puzzle-Based Storage System With Decentralized Control. *IEEE Transactions on Automation Science and Engineering* 2014; 11(2): 429-438.
[14] Mattern. Algorithms for distributed termination detection. *Distributed Computing* 1987; 2(3):161-175.
[15] Dijkstra. *Shmuel Safra's version of termination detection*. Department of Computer Sciences, University of Texas: Austin, 1987. EWD-Note 998.
[16] Mittal et al. A family of optimal termination detection algorithms. *Distributed Computing* 2007; 20(2):141162.
[17] Korf et al. Finding Optimal Solutions to the Twenty-Four Puzzle. *Proceedings of the Thirteenth National Conference on Artificial Intelligence* AAAI:1996; 1202-1207.
[18] Sanz et al. On the Optimization of HDA* for Multicore Machines. Performance Analysis. *Proceedings of the 2014 International conference on Parallel and Distributed Processing Techniques and Applications*. CSREA Press:2014; 625-631.
[19] Korf. Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence* 1985; 27(1): 97-109.
[20] Brüngger. *Solving Hard Combinatorial Optimization Problems in Parallel: Two Cases Studies*. Swiss Federal Institute of Technology Zurich: Zurich, 1998. Diss. ETH No. 12358.
[21] Sanz et al. Performance tuning of the HDA* algorithm for multicore machines. *Computer Science and Technology Series.XX Argentine Congress of Computer Science. Selected Papers*. EDULP: La Plata,2015.