Available online at www.sciencedirect.com

Performance Evaluation 60 (2005) 223–236

PERFORMANCE EVALUATION
An International Journal

www.elsevier.com/locate/peva

Performance evaluation of parallel iterative deepening A* on clusters of workstations

Abdel-Elah Al-Ayyoub

Faculty of Computer Studies, Arab Open University, P.O. Box 18211, Bahrain

Available online 7 December 2004

Abstract

In this paper we investigate the performance of distributed heuristic search methods based on a well-known heuristic search algorithm, the *iterative deepening A** (IDA*). The contribution of this paper includes proposing and assessing a distributed algorithm for IDA*. The assessment is based on space, time and solution quality that are quantified in terms of several performance parameters such as generated search space and real execution time among others. The experiments are conducted on a cluster computer system consisting of 16 hosts built around a general-purpose network. The objective of this research is to investigate the feasibility of cluster computing as an alternative for hosting applications requiring intensive graph search. The results reveal that cluster computing improves on the performance of IDA* at a reasonable cost.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Heuristic search; Parallel and distributed computing; IDA*; Performance evaluation

1. Introduction

The problem of finding a path connecting two predefined points in an implicitly defined network is a computationally exhaustive problem that exists in many real life applications ranging from image processing to robotics and from discrete optimization to automatic theorem proving. Heuristic search is one of the known path-finding methods applicable to implicit graphs. The search begins by applying a set of generators on the start node and then to its successor and so on, until a part of the implicit graph sufficient to include a goal node is generated. The search terminates with a path connecting the start node to the goal node.

E-mail address: a_ayyoub@arabou-jo.edu.jo.

0166-5316/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.peva.2004.10.009

The heuristic search is characterized by the tuple (σ, τ, ξ, f), where σ is the start node, τ is the final node, $\xi = \{g_1, g_2, \ldots, g_k\}$ is the set of generators used to expand nodes, and f is the evaluation function. For each node n the function f gives the estimated cost of the solution path from σ to τ constrained to go through n. This function, referred to as $f(n)$, is composed of a known part $g(n)$ that represents the cost of the path from σ to n plus an estimated part $h(n)$ that represents to the cost of the path from n to τ. The later part, called the heuristic function, is based on application-specific heuristics that are obtained from the problem domain. A heuristic search method is said to be admissible (obtains shortest path solutions) if the employed heuristic function is upper bounded by the actual cost [8]. In some domains, the inadmissible heuristic search functions produce faster and reasonably acceptable solutions [9].

The various ways of selecting nodes for expansion define the different search algorithms. Breadth-first, depth-first, and best-first are the most common methods of graph search. Iterative-deepening search is a combination of depth-first and breadth-first search methods. It performs a series of depth-first searches that operate with successively extended search depths. This approach was introduced as a way to reduce the space complexity of best-first searches from exponential to linear. In a following work, the approach was proved to be efficient as it exploits previously gained search information to improve the search efficiency [4].

The IDA* algorithm uses information from previous iterations to increase the cut-offs in the current iteration. The successive iterations do not correspond to the increased search depth, but to increased cost bound of the currently investigated path. The algorithm performs depth-first search cutting-off all nodes that exceed a fixed cost bound. At the beginning, the cost bound is set to $f(\sigma)$. Then, for each iteration, the bound is increased to the minimum f-value that exceeds the previous bound.

One nice property of this algorithm is its capability to reduce space complexity while preserving optimality. As a consequence, IDA* can be applied in domains were many other algorithms fail [4]. However, the shallow tree paths are re-examined many times because the algorithm does not retain information on the entire search space. This might lead to enormous amounts of repeated search efforts, which could render the search ineffective. Motivated by this observation and also by the inherent parallelism is the successive iterative-deepening, we investigate in this paper the effect of having more than one computer executing the different iterations simultaneously using what is called *parallel window search* [10].

The rest of the paper is organized as follows: in the coming section some related work is outlined. Next, the cluster-based IDA* is described. Then, a framework for performance assessment is presented and used to evaluate the cluster-based IDA*. Finally, some experimental results on solving a large sample from the 15-puzzle search space on a cluster of 16 hosts are presented.

2. Previous work

Though the first use of IDA* was documented in [14], the iterative deepening A* algorithm has been rediscovered a number of times in the literature. The often-cited research work in [4] has analyzed the algorithm along three dimensions: time, space and quality. The authors have also shown that IDA* is asymptotically optimal and superior to its standard bread-first and depth-first counterparts in the above three dimensions [4].

The empirical results in [15] report insignificant effect of the redundant search induced by the repeated iterations of the IDA* algorithm when applied to problem domains such as two-person games. Other versions of IDA* proposed as extensions to the well-known A* algorithm can be found in [9].

In [7], the authors propose an improved IDA* which uses the concept of perimeter search (a set of nodes surrounding the goal). In this improved version, the search "shoots" towards a more visible goal (the perimeter). The authors evaluated two versions of this algorithm, the unidirectional and the bidirectional perimeter-based IDA*. Both versions outperform the original IDA* on the 15-puzzle domain [7].

Another improvement to the IDA* is reported in [13], where cycle-transposition tables and successor reordering information is used to reduce the size of the generated search space by half on the 15-puzzle problem and by three quarters on the traveling salesman problem.

Parallelizing IDA* is another possible way of improvement. There are two common approaches for parallelizing IDA* that have been discussed in the literature. The first approach, referred to as *parallel window search* [5,10], distributes the *iteration* space to the available processors. Each processor then searches its sub-space independently. The second approach, referred to as *distributed tree search*, divides the *search* space vertically into "equal" size sub-trees and then each processor performs its own IDA*. This approach has been investigated in [6,11,12]. The pros and cons of these two approaches will be discussed in the next section.

The two approaches (parallel window search and distributed tree search) can be combined so each iteration in a parallel window search can be searched in a distributed tree search manner, or each part of the tree in a distributed tree search can be searched in a parallel window search manner. This combined approach was tested on Sequent Balance 21000, Intel iPSC Hypercube, and BBN Butterfly using the 15-puzzle and showed to achieve reasonable speed up [11].

3. Cluster-based formulation of IDA*

In a parallel window search, iterations are distributed to processors so each processor independently performs depth-first searches at increasing depth thresholds. Once a processor exhausts the search with a given depth it restarts the search with a larger depth until one of the processors finds a goal node.

A parallel window search does not guarantee optimal solutions since a solution found by a processor is provably optimal only if all other processors have also exhausted their search space until that iteration and no better solution has been found. This is a consequence of the fact that a processor working up to a shallower depth may find a solution with a lower cost after another processor working deeper finds one.

With distributed tree search, we may have each processor perform IDA* in its sub-tree or have all available processors perform distributed tree search on one iteration of IDA*. Other combinations can be thought out which are basically intermixing parallel window search with distributed tree search. Since all processors work with the same cost bound, each processor performs a depth-first search on its part of the tree. At the end of each iteration, a master processor determines the new threshold for the next iteration and then restarts the parallel depth-first search with this new threshold. The search terminates when one of the processors finds a goal node and broadcasts a termination request to all other processors.

The main bottleneck in the second approach (distributed tree search) is the communication time involved in finding the next threshold. A costly communication operation such as *exchange-minimum* needs to be performed in order to find and distribute a minimum f value to all the processors. One method to achieve exchange-minimum is as follows: all processors send the minimum among all f values exceeded the current threshold in their part of the tree to the master processor, which after receiving these partial minimum values finds the minimum and then broadcasts it back to all processors. This method of implementation is useful for cluster environments, which are mostly based on broadcast networks such as the Ethernet. Another method of implementing exchange-minimum operations is through repeated neighbor-to-neighbor exchanges (neighboring processors exchanging partial minimums δ times, where δ is the diameter of the network). This method is useful for point-to-point interconnected computers such as the hypercube and the mesh.

Furthermore, intermixing parallel window search with distributed tree search adds on the amount of duplicate work. In distributed tree search, processors searching in their part of the search space might replicate other processors' work. Adding to it the amount of redundant search involved in IDA*, the second approach turns out to be less appealing.

The communication overhead becomes an obstacle on problem domains that require moderate amounts of search on parallel environments with modest communication speeds. One way to get around this obstacle is by using incremental thresholds. The next threshold is equal to the previous one plus a constant factor. Of course this might add useless searches that could have been skipped.

The first approach introduces lesser additional redundant work, yet it has solution quality concerns as discussed earlier in this section. There is a possibility that a processor searching deep will find a non-optimal solution before a processor searching shallow encounters a better solution in its tree.

In parallel window search, the error in the obtained solution quality is bounded by the number of processors if an incremental threshold is used (unit increments). In case a trivial termination condition is used (terminate once a solution is found) the upper bound on this error could be lower or could be higher if the thresholds are evaluated using an exchange-minimum procedure rather than using incremental thresholds. The error analysis naturally depends on the problem domain and also on the employed heuristic function.

In this paper, a form of parallel window search algorithm tailored for cluster computers will be used. This algorithm is implemented in a master-slave model and can be explained as follows: one of the host computers (the master) evaluates $f(\sigma)$ then spawns one copy of IDA* in each of the available hosts (slaves). The first copy is spawned with $f(\sigma)$ as a cost threshold, the second copy is spawned with $f(\sigma)+k$, the third copy is spawned with $f(\sigma)+2k$, and so on (k is the iteration size or the depth of the search). The slaves start performing depth first search until either a goal is found or the search is exhausted. If a goal is found, a termination request is broadcasted so that all spawned IDA* copies are terminated. If the search is exhausted without encountering a goal, then a new threshold request is sent to the master followed by a block-receive till the new threshold is received. Figs. 1 and 2 outline the steps for the master and the slave tasks.

The two algorithms given above can be implemented in a variety of ways to reduce the master-slave coordination overhead. For instance, each slave can determine its initial threshold as a function of the start node and the host identifier. Therefore, the threshold $f(\sigma)+l*k$ for the search tree in the host l can be determined by the host itself. All it needs is a function that maps hosts IP addresses to integer numbers from 1 to H. The next threshold can be determined by using global semaphore to gain access to a predefined shared threshold which is initially set to $f(\sigma)+H*k$. The termination command (a trivial one) can also be managed through semaphores. Once a slave finds a goal, it sets a flag for other slaves to test before attempting new search iteration.

The function *Serve* executed by the master retrieves the next message from a receive buffer (*receive_queue*). This buffer contains two types of messages sent by the slave hosts. A message can be a request for a new threshold or a declaration of goal found. The second type of messages requires the master to evaluate quality tests before an appropriate action is taken. In our experiments, the search is terminated

```
Algorithm Master
    Step 1: Spawn Slave(f(σ) + l*k) on the host l, where l is the host identifier.
    Step 2: global threshold = f(σ) + H*k.
    Step 3: Serve(receive_queue, message).
    Step 4: Case message of
        searchexhausted : Spawn Slave(global threshold += k) on the host l,
        where l is the message sender.
        goalfound :apply quality tests and then terminate all running slave tasks
        if the quality tests pass.
 Step 5: Exit.
End Master.
```

Fig. 1. The master task.

```
Algorithm Slave
    Step 1: Receive (on blocking mode) the search threshold from the master task.
    Step 2: Perform IDA* until either a goal is found or the search tree is exhausted.
    Step 3: Send the search status to the master task.
    Step 4: Exit.
End Slave.
```

Fig. 2. The slave task.

once such a message appears in the *receive_queue*. The consequence of this will be demonstrated in the following sections.

The constant k determines the iteration size (the depth of the search). Larger k values mean less communication time, but at the expense of the solution quality. The upper bound on the error is proportional to kH, where H is the number of processors in the cluster. Recall that the last spawned copy of IDA* gets $f(\sigma) + kH$ as a threshold.

The execution time of the above parallel setup of IDA* consists of three parts, these are search time, communication time, and idle time. An experimental assessment for the above algorithm should measure these times separately. In what follows, we present an experimental assessment for the above algorithm on the 15-puzzle and using a typical academic installation of a cluster computer system consists of Pentium IV workstations connected by 100 mbps Ethernet segments.

4. A framework for performance assessment

The 15-puzzle is a traditionally challenging problem which is still beyond today's high performance computing. In our experiments, we generated 11,321 samples distributed as shown in Fig. 3. Some of these samples can be solved easily and some can hardly be solved in a reasonable amount of time. Given the large sample over which the algorithms will be assessed, a time limit is set so an algorithm aborts if no solution is found within this time limit. The selected time limit is equal to the time it

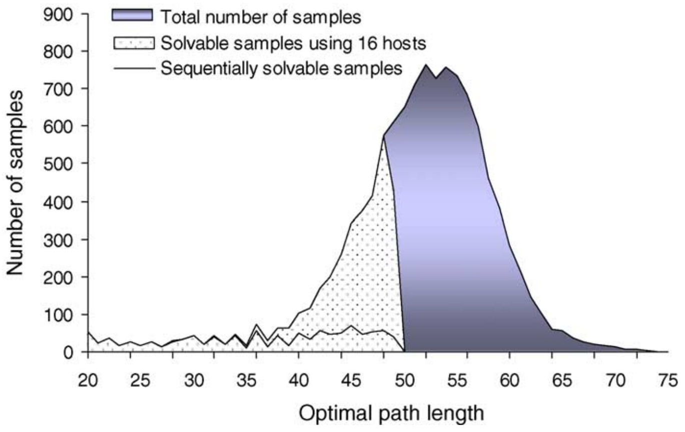

Fig. 3. The 15-puzzle samples.

takes till the physical memory is exhausted. Once the physical memory is exhausted, the search becomes very slow and hence the time needed to complete a problem would take days. Hence, the logic behind this limit is to achieve this experiment on an academic cluster used by the university communities for different purposes. In order to get correct timing results, the cluster has been dedicated to experiments; no other processes are running other than IDA* copies at the experimentation time.

The above figure shows the total number of samples generated from each category using a random sample generator and then solved using disjoint databases [3], which are specialized heuristics for quick 15-puzzle solvers. Fig. 3 also shows the number of samples solvable, within a time limit, using sequential IDA*. The percentage of solvable samples from each category is rather small compared to that of the percentage of solvable samples using 16 hosts. Several samples after 48 moves can be solved using the 8 or 16 hosts; however none could be solved sequentially. Therefore, in the subsequent figures, we only show performance parameters obtained from samples ranging from 20 to 48 moves in order to obtain consistent comparisons.

Among the targeted performance parameters are the standard ones related to execution time, required space, and solution quality which are captured by the main search activities such as node expansion, node duplicate checking, heuristic distance calculation, number of nodes generated, the lengths of the obtained solutions, execution time, communication time, and process blocking (idle) time.

The experiments are repeated on 1, 8 and 16 Pentium IV hosts (1.7 GHz, MB128 RAM, full-cache, and GB40 HD) connected by 100 mbps Ethernet segments. On each of the 11,031 (3677 samples on three cluster configurations) the targeted performance parameters are collected in three databases with a accumulative size of $3677 + 8 \times 3677 + 16 \times 3677 = 91{,}925$ records. Each record contains the performance parameters recorded by a host upon termination. In fact 174,028 records have been collected from which we only took 91,925 records for consistent study (records generated for samples ranging from 20 to 48 moves).

This large number of records has been processed in various ways to obtain two main sets of processed data. These are averages and averaged accumulates; which are plotted in the subsequent figures. In order to simplify the interpretation of the data sets we use the following notation:

$S_m = \{\sigma_1^m, \sigma_2^m, \ldots, \sigma_k^m\}$ the set of k samples at depth m, where m is the minimum number of moves from σ_i^m to τ, $1 \le i \le k$

$S = \{S_1, S_2, \ldots, S_j\}$ the set of sample sets

$R^l_{\sigma_i^m}$ the record generated by the host l when solving the sample σ_i^m from S_m, $1 \le i \le k$

$T^H_{\sigma_i^m} = \sum_{h=1}^{H} R^l_{\sigma_i^m}$ the *accumulate* record generated by the H hosts in the cluster after solving the sample σ_i^m from S_m, $1 \le i \le k$, where H is the number of hosts in the cluster

$A^H_{\sigma_i^m} = \dfrac{T^H_{\sigma_i^m}}{H}$ the *average* record generated by a host in the cluster when solving the sample σ_i^m from S_m, $1 \le i \le k$

$A^H_{S_m} = \dfrac{\sum_{\sigma_i^m \in S_m} A^H_{\sigma_i^m}}{|S_m|}$ the *average* (expected) record generated by a host in the cluster when solving a sample from S_m

$T^H_{S_m} = \dfrac{\sum_{\sigma_i^m \in S_m} T^H_{\sigma_i^m}}{|S_m|}$ the *averaged accumulate* (expected) record generated by the H hosts in the cluster when solving a sample from S_m

$A^H = \dfrac{\sum_{S_m \in S} A^H_{S_m}}{|S|}$ the average record (expected) generated by a host in the cluster when solving any sample from any set in S

$T^H = \dfrac{\sum_{S_m \in S} T^H_{S_m}}{|S|}$ the averaged accumulate record (expected) generated by the H hosts in the cluster when solving any sample from any set in S

The difference between (and the purpose of) these averages and averaged accumulates can easily be noticed if we think of the cluster as a single computer with multiple processing units or as separate machines with single processors. So these different terms give more flexibility to interpret and use the obtained results. The information in all of the subsequent figures is based on $A^H_{S_m}$ averages for $20 \le m \le 48$, unless otherwise stated.

Using the above described data sets, several cost-performance judgments can be made. For instance, search effectiveness for IDA* can be represented in terms of the average number of nodes generated and the optimal path lengths [2]. Similarly, the solution quality is judged by the obtained solutions compared to the optimal ones [1,2]. Several other observations can be made as will be explained in the next section.

5. Experimental results

In this section, the experimental results are presented. First, the raw data is presented and justified. Next, the processed data will give more meaningful insight on the distributed implementation of IDA*.

The real execution times are shown in Figs. 4 and 5. The samples are plotted in two categories; easy to medium (S_{20} to S_{35}) and hard (S_{36} to S_{48}) samples. The figures suggest that clustered IDA* runs faster with larger number of hosts especially on hard samples. The gain in execution time when solving easy problems is small; however the gain becomes more apparent as the problem size increases. This is expected since easy problems are solved rather quickly which implies dominance of startup and

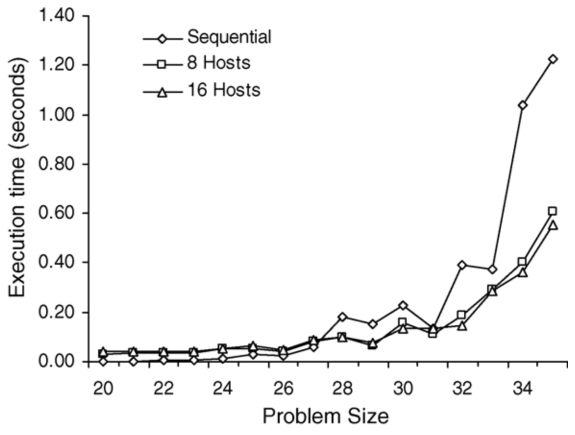

Fig. 4. Execution time on easy to medium problems.

master-slave coordination over search time. This fact is depicted more clearly in Fig. 6. In easy problems, hosts exhaust their search trees quickly and hence master-slave coordination happen more often (recall that $f(\sigma)$ for easy problems is relatively small) and hence hosts starting earlier will mob the master with threshold requests leaving little chance for other hosts to start searching. Some of the hosts may not even get the chance to start before one of the early hosts finds a solution, rendering these late hosts idle during the whole execution period. This explains the increase in idle times when solving easy problems. Therefore, it is wiser to use smaller number of hosts when attempting easy problems.

The effect of the host idle times on the overall execution time is shown in Fig. 7. The figure shows the percentage of host waiting time over the total execution time. Generally, hosts are busy most of the time with marginal exception on easy problems. The information in the figure suggest that as the problem size increases the hosts get busier searching on separate iterations of the IDA*.

The communication time roughly ranges from 0.01 to 0.03 s. Fig. 8 suggests minor increase in the communication time as the problem size increases. The communication curves might seem to grow for

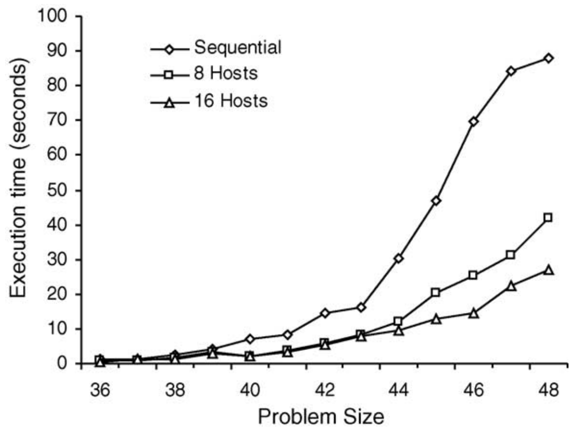

Fig. 5. Execution time on hard problems.

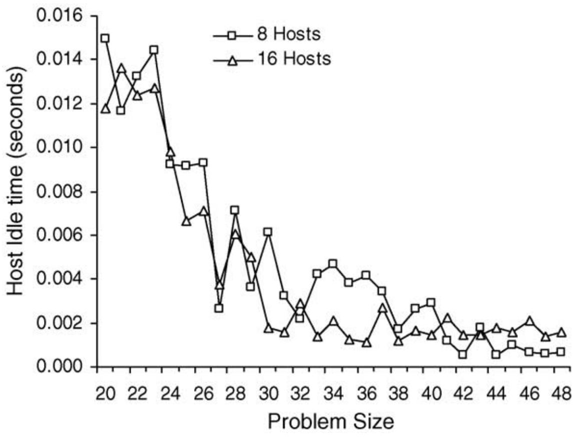

Fig. 6. Idle time.

larger problem sizes; however the time scale in the figure conveys no significant communication overhead on the overall execution time. Of course, this conclusion is valid for large problem sizes where parallel solutions are more appealing.

The communication overhead shown in Fig. 9 is calculated as the percentage of communication time over the total execution time. This percentage becomes negligible when the problem size increases. Recall that $f(\sigma)$ in hard samples is relatively larger and hence it will take sometime before a host exhausts its search tree and then communicates for the next threshold.

The speed up gain approaches 5 in its the best case as Fig. 10 suggests. There is no gain in the execution time when the problem size is below 28. The speed up improves after 28 even though there are declines in the speed up for some samples with sizes above 28. The explanation for this decline lay in the employed heuristic function (linear conflict in this case). These samples turn out to be easy to solve in most of them; hence the sequential IDA* solves these samples quickly. We have to bear in mind that parallel computing is more appealing when the amount of computation time is large, which is not the case in these samples.

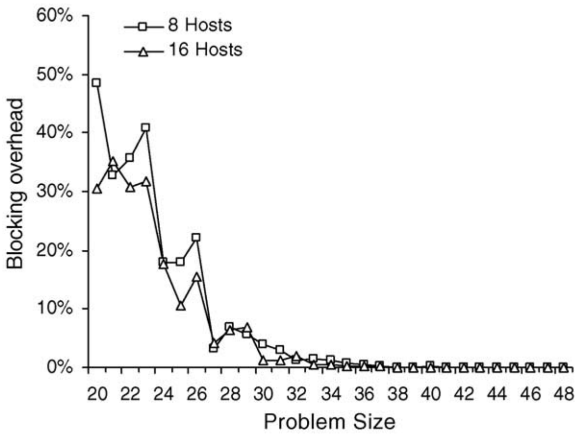

Fig. 7. Blocking overhead.

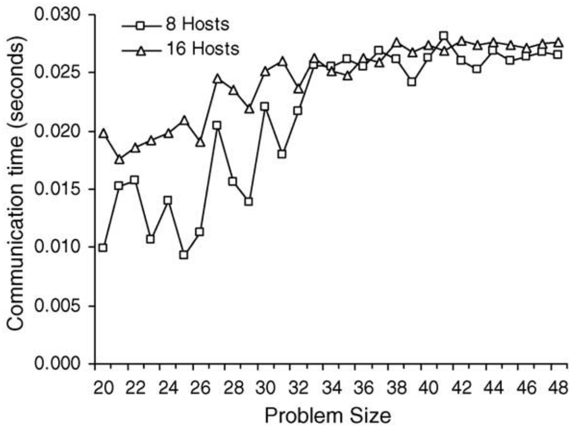

Fig. 8. Communication time.

Search focus measures the search effectiveness in terms of search percentage that was directed towards the goal [1,2]. This measure is determined by the ratio of the $\beta\pi/\Gamma$, where β is the branching degree, π is the length of the optimal solution path, and Γ is the total number of nodes generated until a goal is found. Generally, IDA* performs poor in terms of search focus as the number of nodes generated is quite large, rendering the algorithm useless for problem domains or target machines with costly node generation processes.

Fig. 11 shows the search focus for the sequential and the clustered IDA*. The poor performance for large problem sizes is inherited from the original IDA*. Distributing the search iterations over a larger number of hosts did not improve on the search focus of IDA* algorithm.

The IDA* algorithm when used with admissible heuristic functions is optimal [4]. The algorithm gradually deepens the search until it encounters a solution, which should then be an optimal one, otherwise the search would have been terminated earlier with a shorter solution. Unfortunately, this is not the case with clustered IDA*. Hosts are allowed to search deeper even if other hosts searching shallower might

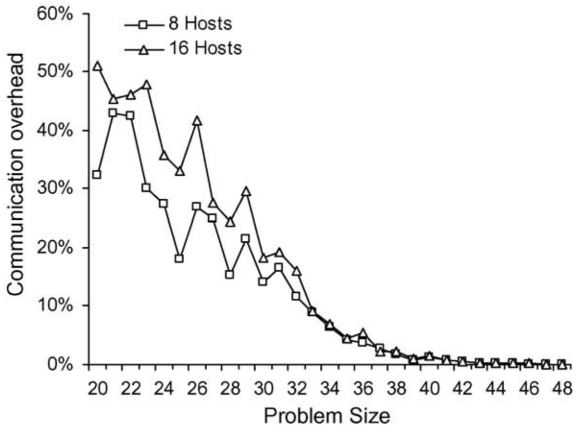

Fig. 9. Communication overhead.

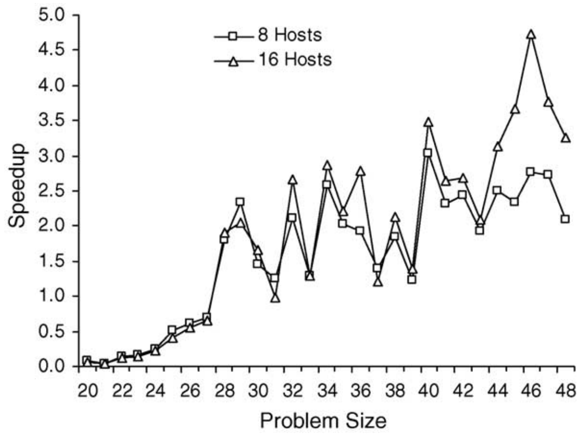

Fig. 10. Gained speedup.

find a shorter solution. Enforcing the optimality constraints would serialize the search process. On the other hand relaxing these constrains might compromise the solution quality.

The effect of relaxing the optimality constraints on the solution quality is illustrated in Fig. 12. Increasing the number of hosts would normally decrease the solution quality as explained earlier. The figure shows up to 30% decrease in the solution quality when 16 hosts are used. In fact this is not as bad as it looks if we consider the gained speed up. There are real-time applications where quick and partially optimal solutions are acceptable. In such situations clustered IDA* are viable alternatives.

The size of the generated search space is another concern. In some problem domains where node generation is costly; the total number of generated nodes is a deciding factor. Fig. 13 shows the number of nodes generated by the sequential and the clustered IDA*. The figure shows the average number of generated nodes (based on $A^H_{S_m}$) and the averaged accumulate number of generated nodes (based on $T^H_{S_m}$). The clustered IDA* generates (on average) less nodes than the sequential IDA*. The averaged accumulate

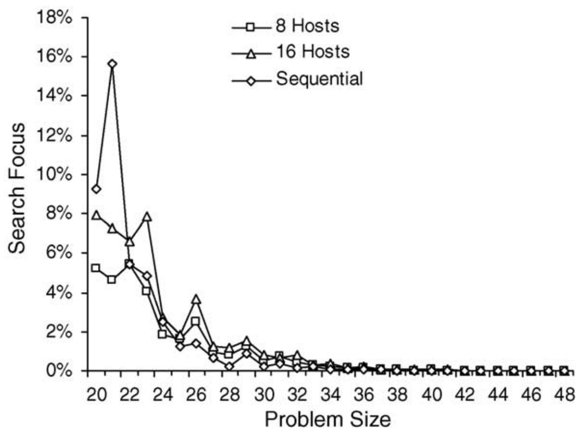

Fig. 11. Search focus.

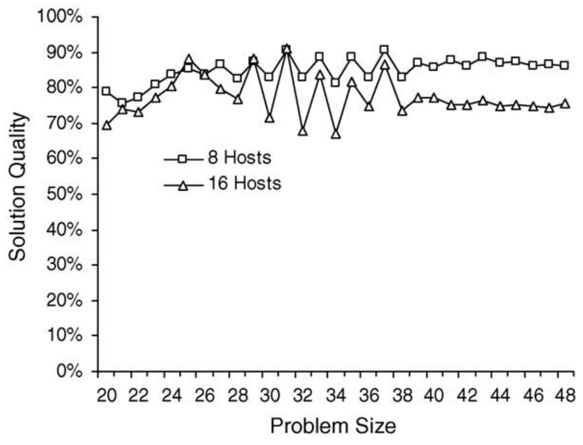

Fig. 12. Solution quality.

number of generated nodes (based on $T_{S_m}^H$) is obviously larger than the sequential IDA*. Furthermore, $T_{S_m}^{16}$ is larger than $T_{S_m}^{8}$ which means that clustered IDA* adds more redundant search when larger number of hosts are used. Table 1 gives a better image on these averages and averaged accumulates.

The information in Table 1 summarizes all the obtained results. The table shows three columns representing averages and averaged accumulates for the sequential and the clustered IDA*. The second and the third columns are based on A^H and the fourth column is based on T^H.

The information in Table 1 has a clear message: the clustered IDA* puts lesser load on individual hosts and runs faster at the expense of the solution quality. The first three rows in the table show that the clustered IDA* expands and generates fewer nodes and induces lower branching factor (the average number of nodes generated when expanding a node). It also runs two to three times faster but with 15–22% loss in the solution accuracy.

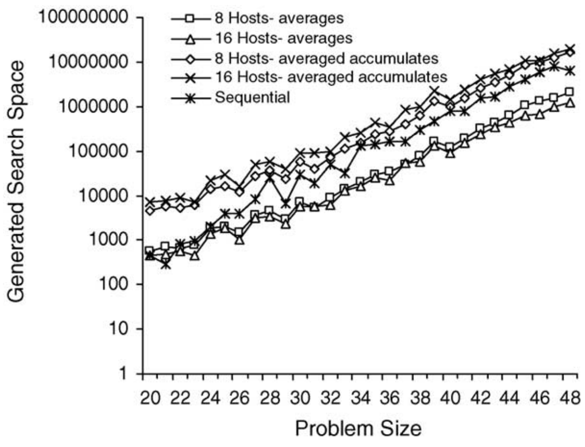

Fig. 13. Generated search space.

Table 1
Summary of performance parameters

Parameter	Sequential	Averages		Averaged accumulates	
		8 Hosts	16 Hosts	8 Hosts	16 Hosts
Nodes expanded	566,523.85	278,278.46	175,748.56	2,226,227.65	2,811,976.99
Nodes generated	1,153,453.81	281,007.52	176,645.33	2,248,060.18	2,826,325.22
Duplicate node checks	1,153,453.81	281,006.52	176,644.33	2,248,052.18	2,826,309.22
Duplicate nodes found	274.66	59.03	32.16	472.22	514.49
Branching factor	2.047	1.065	1.089		
Path length (s)	34.000	39.862	44.248		
Execution time (s)	13.040	5.491	3.886		
Communication time (s)		0.021	0.024		
Idle time (s)		0.005	0.004		
Search time (s)		5.507	3.962		
Focus (%)	1.55	1.07	1.57		
Solution quality (%)	100.00	84.88	77.45		

Table 2
Redundancy induced by parallel search

Parameter	Averages (%)		Averaged accumulates (%)	
	8 Hosts	16 Hosts	8 Hosts	16 Hosts
Nodes expanded	49	31	393	496
Nodes generated	24	15	195	245
Duplicate node checks	24	15	195	245
Duplicate nodes found	21	12	172	187

The more hosts are involved the higher the chance of a reduced quality and an increased amount of redundant or needless search effort. Table 2 shows the factors of redundant search effort compared to the sequential IDA*. The “Averages” column is based in A^H and the “Averaged accumulates” column is based on T^H. The numbers in the Averages column are encouraging since the clustered IDA* induces large reduction in the expected search effort (quarter to half of the sequential IDA*). Even when considering the aggregate (the Averaged accumulates column) the factor is also encouraging. For instance, adding eight hosts to the search process roughly doubles the generated search space which means each host contributes only 12% (roughly) to the redundant work.

6. Conclusions and further work

The clustered IDA* algorithm presented in this paper is based on the well-known parallel window search methods. The version we developed and evaluated relaxes the quality concerns for faster solutions. The overall conclusion obtained from this study can be summarized as follows: compromising 15–22% of the solution quality will double or triple the speed of obtaining solutions. The solution quality can be improved at the expense of the speed. This can be done by introducing the quality concerns such as accepting solutions only if there is no other host attempting iteration at a shallower depth. Of course, this will add extra redundant search effort as the decision to terminate the search after finding a solution will be postponed until the quality test is accomplished. Obviously more empirical results are needed to further justify quality-speed trade-off assessment.

Acknowledgements

The author wishes to express his deep gratitude to the anonymous referees for the reasonable revisions and also for correcting the errors in this paper. The author would also like to thank Basel Mahafza, Mohamed Bata, and Nayef Abu-Ghazaleh for helping in the setup of the cluster environment on which the experiments took place. Many thanks also go the Department of Computer Science at Jordan University of Science and Technology for offering their labs in support of this work.

References

[1] A. Al-Ayyoub, F. Masoud, Heuristic search revisited, J. Syst. Software 55 (2) (2000) 103–113.
[2] A. Al-Ayyoub, F. Masoud, Search quality and effectiveness for intelligent systems, in: Proceedings of the 8th International Conference on Intelligent Systems, Denver, CO, USA, June 24–26, 1999, pp. 146–149.
[3] R. Korf, A. Felner, Disjoint pattern database heuristics, Artif. Intell. J. 134 (2002) 9–22.
[4] R. Korf, Depth-first iterative-deepening: an optimal admissible tree search, Artif. Intell. 27 (1985) 97–109.
[5] V. Kumar, L. Kanal, Parallel branch-and-bound formulation for and/or tree search, IEEE Trans. Pattern Anal. Machine Intell. 6 (1984) 768–778.
[6] V. Kumar, V. Rao, Parallel depth first search, part II: analysis, Int. J. Parallel Program. 16 (6) (1987) 501–519.
[7] G. Manzini, BIDA*: an improved perimeter search algorithm, Artif. Intell. 75 (1995) 347–360.
[8] N. Nilsson, Principles of Artificial Intelligence, Tioga Publishing Company, Palo Alto, USA, 1980.
[9] J. Pearl, Heuristics: Intelligence Search Strategies for Computer Problem Solving, Addison-Wesley, Reading, MA, 1984.
[10] C. Powley, R. Korf, Single-agent parallel window search, IEEE Trans. Pattern Anal. Machine Intell. 13 (5) (1991) 466–477.
[11] V. Rao, V. Kumar, Parallel depth-first search, part I: implementation, Int. J. Parallel Program. 16 (6) (1987) 479–499.
[12] V. Rao, V. Kumar, K. Ramesh, A parallel implementation of iterative-deepening A*, in: Proceedings of the National Conference on Artificial Intelligence, Seattle, Washington, 1987, pp. 878–882.
[13] A. Seinefeld, T. Marsland, Enhanced iterative-deepening search, IEEE Trans. Pattern Anal. Machine Intell. 16 (6) (1994) 701–710.
[14] D. Slate, L. Atkin, CHESS 4.5—The Northwestern University Chess Program, Springer Verlag, New York, 1977.
[15] P. Winston, Artificial Intelligence, Addison Wesley, Reading, MA, 1984.

Dr. Abdel-Elah Al-Ayyoub is an Associate Professor of Computer Science at the Arab Open University. He received his B.Sc. degree in Computer Science in 1986 from Yarmouk University, Jordan. He then joined the Middle East Technical University, Turkey, where he obtained his M.S. and Ph.D. degrees in Computer Engineering in 1987 and 1992, respectively. His areas of interest include e-Learning, artificial intelligence, networks, mobile computing. Dr. Al-Ayyoub is an IEEE Senior Member. He received more than a million US dollars in research grants, won three major prizes in Computer Science, and published over 50 papers in well-known journals and conference proceedings. Dr. Al-Ayyoub supervised Ph.D. and Master students and developed several graduate and undergraduate programs in various fields of Information Technology. Before joining the Arab Open University, Dr. Al-Ayyoub severed in the University of Bahrain, Sultan Qaboos University—Oman, The University of Akron—Ohio, and Jordan University of Science and Technology—Jordan. His experience in teaching extends to 14 years.