A Hybrid Parallel Search Algorithm for Solving Combinatorial Optimization Problems on Multicore Clusters

Victoria Sanz[1,2(✉)], Armando De Giusti[1,3], and Marcelo Naiouf[1]

[1] School of Computer Sciences, III-LIDI, National University of La Plata, La Plata, Argentina
{vsanz,degiusti,mnaiouf}@lidi.info.unlp.edu.ar
[2] CIC, Buenos Aires, Argentina
[3] CONICET, Buenos Aires, Argentina

Abstract. Multicore clusters are widely used to solve combinatorial optimization problems, which require high computing power and a large amount of memory. In this sense, Hash Distributed A* (HDA*) parallelizes A*, a combinatorial optimization algorithm, using the MPI library. HDA* scales well on multicore clusters and on multicore machines. Additionally, there exist several versions of HDA* that were adapted for multicore machines, using the Pthreads library. In this paper, we present Hybrid HDA* (HHDA*), a hybrid parallel search algorithm based on HDA* that combines message-passing (MPI) with shared-memory programming (Pthreads) to better exploit the computing power and memory of multicore clusters. We evaluate the performance and memory consumption of HHDA* on a multicore cluster, using the 15-puzzle as a case study. The results reveal that HHDA* achieves a slightly higher average performance and uses considerably less memory than HDA*. These improvements allowed HHDA* to solve one of the hardest 15-Puzzle instances.

Keywords: Parallel search algorithms · Hybrid programming · Multicore cluster · Combinatorial optimization problems · Hash Distributed A*

1 Introduction

Several search algorithms require high computing power and a large amount of memory, thus, different parallel approaches have been proposed in order to take advantage of the resources of multicore clusters. This is the case of the A* algorithm, a variant of Best-First Search, which is used for solving combinatorial optimization problems. These problems require finding a sequence of actions that minimizes a goal function and allows transforming an initial configuration (i.e., the problem to be solved) into a final configuration (i.e., the solution).

A* [1,2] explores the graph that represents the state space of the problem using a cost function $\hat{f}$ to value the nodes, which is defined as follows: $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$, where $\hat{g}(n)$ is the known cost of the path from the initial

S. Ibrahim et al. (Eds.): ICA3PP 2017, LNCS 10393, pp. 766–775, 2017.
DOI: 10.1007/978-3-319-65482-9_62

node to the current node n and $\hat{h(n)}$ is a heuristic estimate that represents the unknown cost of the path from the current node n to the solution node. In this algorithm the search tree is generated as the search progresses. During the process, it keeps two data structures: one for the unexplored nodes sorted by $\hat{f}$ (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.

Hash Distributed A* (HDA*) [3,4] is a parallel A* algorithm in which each processor has its own open/closed lists and performs a quasi-independent search. It uses a standard hash function to assign each state of the problem to a single processor. This hash-based node distribution scheme allows balancing the load and pruning duplicate nodes (i.e., nodes representing the same state) in an absolute way, as they are always sent to the same processor. This version of HDA* was implemented using the MPI library, thus, it can be run on distributed-memory, shared-memory, or hybrid systems.

Other authors [5–7] adapted HDA* for multicore machines, using 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.

Since current clusters are composed of shared-memory nodes, some applications may benefit from hybrid programming, i.e. by combining message-passing (MPI) with shared-memory programming (Pthreads or OpenMP) [8,9]. To our best knowledge, no hybrid version of HDA* has been proposed until now, to better exploit the computing power and memory of multicore clusters.

In this paper, we present Hybrid HDA* (HHDA*), a hybrid MPI/Pthreads parallel search algorithm based on HDA*. We evaluate the performance and memory consumption of HHDA* on a multicore cluster, using the 15-puzzle as a case study. The results reveal that HHDA* achieves a slightly higher average performance and uses considerably less memory than HDA*. These improvements allowed HHDA* to solve one of the hardest 15-Puzzle instances.

The rest of the paper is organized as follows. Section 2 discusses background and related work. Section 3 introduces HDA* (HDA* MPI) and HDA* for shared-memory architectures (HDA* Pthreads). Section 4 describes the Hybrid HDA* algorithm. Section 5 shows our experimental evaluation. Finally, Sect. 6 presents the main conclusions and some ideas for future research.

2 Related Work

Today, many commodity clusters are composed of shared-memory machines. Applications to be run on these systems can be developed by using only message-passing or by combining message-passing with shared-memory programming (hybrid programming). While the former approach requires less programming effort, the latter may improve performance and reduce the memory used [10].

The most common and efficient way to parallelize A* is to use a decentralized strategy [11]: each process/thread (processor) is equipped with its own local open and closed lists and performs a quasi-independent search. This strategy is suitable both for shared-memory and distributed-memory architectures. However, communication among the processors is needed due to the following reasons: 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 duplicated work; the termination criterion should be modified because if the search is ended when the first 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.

Hash Distributed A* (HDA*) [3] parallelizes A* by applying a decentralized strategy. It was implemented using only MPI and asynchronous communication. It uses a standard hash function to assign states to processors. This hash-based node distribution scheme allows balancing the load and pruning duplicate nodes in an absolute way, as the nodes representing the same state are always sent to the same processor, which performs the duplicate detection procedure. The algorithm works as follows. Periodically, each process P performs the following steps until a global optimal solution is reached: (1) P checks if a message with nodes has arrived. (2) If so, for each node, P determines if the node must be added to the open list or if it should be discarded. (3) If no messages were received, P selects a node from its open list (the one with the lowest $\hat{f}$-value). Then, P expands the node and, for each successor, it calculates the hash value to identify the owner process. If the node belongs to another process Q, P sends a message with the node to Q. To reduce the communication overhead, a given number of nodes whose recipient is the same are packed into a single message.

On the other hand, in [5] the authors adapted HDA* for multicore machines. This version does not have the extra overhead of message-passing between processors (threads) on a shared-memory architecture. Also, it uses less memory as threads share common data structures. The algorithm works as follows. Each thread is given an input queue, where the rest of the threads will deposit nodes that must be processed by this thread, and a local output queue for each peer thread. When a thread t_i generates a node that belongs to another thread t_j, t_i tries to acquire the lock associated with t_j's input queue. If the lock is obtained immediately, the node is transferred and the lock is released. Otherwise, the node is added to the local output queue for t_j. After t_i carries out a certain number of node expansions from its open list: (1) It tries to communicate the nodes stored in each non-empty output queue to the respective thread, but it is never forced to wait on a lock (2) It tries to consume nodes from its input queue (it is only forced to wait when its open list is empty).

We developed our own versions of HDA* (HDA* MPI) and HDA* for multicore machines (HDA* Pthreads), which are summarized in Sect. 3. Implementation details can be found in [4,6] respectively. The former differs from the original version in that it includes a parameter which indicates the maximum number of nodes to be processed per algorithm iteration. We noted that performance does not improve by processing one node per algorithm iteration, as done in the original version. The latter includes a technique to group several nodes before transferring them to the corresponding thread. We observed that this technique reduces the amount of node transfers and mitigates communication and contention.

From the above it can be concluded that significant efforts were made to parallelize the A* algorithm on different parallel architectures. However, neither of the known algorithms considers hybrid programming to better utilize the resources of multicore clusters. It should be noted that, although in [12] a hybrid parallel algorithm is presented for solving combinatorial problems, the parallelization is based on the Weighted A* algorithm (a suboptimal version of A* that trades-off solution quality for search time). However, Hybrid HDA* is based on the A* algorithm and it aims to find optimal solutions.

3 Implementation of the HDA* Algorithms

3.1 HDA* (HDA* MPI)

Each process carries out an A* search locally and communicates with its peers for sending/receiving messages containing nodes, the costs of solutions found and messages that allow detecting termination.

Each process maintains its own open/closed lists, the cost of the best global solution known so far (*best_solution_cost*), the best solution found by the process (*best_solution*), among others. In order to pack several nodes into a single message, the process is equipped with a buffer (*send_buffer*) for each peer process.

Each process P performs the following stages until a global optimal solution is reached:

1. Work message reception stage: P checks if *work messages* containing nodes have arrived. If so, P receives each message and, for each node with $\hat{f} < best\_solution\_cost$, it performs the duplicate detection and adds the node to the open list as appropriate.
2. Cost message reception stage: P checks if *cost messages* containing the cost of a solution have arrived. If so, P receives the messages and updates *best_solution_cost* as appropriate.
3. Processing stage: P extracts nodes from its open list, discarding those with $\hat{f} >= best\_solution\_cost$. When the extracted node represents a solution, P updates *best_solution* and *best_solution_cost* and sends the solution cost to the other processes. Otherwise, P adds the node to its closed list and expands the node. Then, for each successor, P calculates the hash value to determine the owner process. When the successor belongs to P, it adds the node to its open list as appropriate. Otherwise, P adds the node to the *send_buffer* for the destination process and, if the buffer became full, it sends the *work message* asynchronously.

4. Idle stage: P enters this stage when its open list is empty. Firstly, it sends *work messages* to those destination processes whose *send_buffer* is non-empty. Then, it remains waiting for: (1) work messages, (2) cost messages, (3) messages that allow detecting termination. P ends this stage when its open list is non-empty, 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 type (3) are processed based on Dijkstra's termination detection algorithm [13].

When computation ends, the optimal solution (i.e. the sequence of actions that allows transforming the initial state into the final state) is retrieved in a distributed manner.

3.2 HDA* for Multicore Machines (HDA* Pthreads)

Each thread has its own open/closed lists. The node communication strategy is based on the use of input/output queues. All threads share the best global solution found so far (*best_solution*), its cost (*best_solution_cost*), among others.

Each thread t_i performs the following stages until a global optimal solution is reached:

1. Work reception stage: t_i tries to consume nodes from its input queue. If it obtains the lock immediately, it takes all the nodes stored in the queue, releases the lock, and then for each node with $\hat{f} <$ *best_solution_cost*, t_i performs the duplicate detection procedure adding the node to the open list as appropriate.
2. Processing stage: the main difference with HDA* MPI is the way in which nodes are communicated between threads. When t_i generates a node that belongs to another thread t_j, it stores the node in the local output queue for t_j; when the amount of stored nodes reaches a certain limit, t_i tries to acquire the lock associated with t_j's input queue and, if it obtains the lock immediately, it transfers the stored nodes.
3. Idle stage: t_i enters this stage when its open list is empty. Firstly, it transfers the nodes stored in each non-empty output queue. Then, it remains waiting until it receives work or it receives a termination notification from the master thread (to this end, we adapted Dijkstra's termination detection algorithm for shared-memory machines [6]).

When computation ends, the optimal solution is retrieved by the master thread.

4 Hybrid HDA* (HHDA*)

Hybrid HDA* (HHDA*) is based on the HDA* algorithm and its version for multicore machines. HHDA* assigns only one process per machine. Each process (master thread) creates threads that will perform the search procedure, along with the master thread. The proposed algorithm uses communication via shared-variables, among threads on the same machine, and communication via message-passing, among processes on different machines.

All threads on the same machine share the best solution found locally by these threads (*best_solution*), the cost of the best global solution known so far (*best_solution_cost*), among others.

Each thread has: its own open/closed lists, a global input queue, an output queue for each peer thread on the machine, message buffers for inter-process communication, among others.

Each thread t_i performs the following stages until a global optimal solution is reached:

1. Message reception stage: any thread on the machine can receive messages addressed to its process, containing either (1) nodes or (2) the cost of a solution found. In the first case, for each received node, t_i identifies the owner thread and, depending on whether the node belongs to t_i or not (a) it carries out the duplicate detection and adds the node to its open list (as appropriate) or (b) it stores the node in the local output queue for the destination thread. In the second case, t_i updates *best_solution_cost*, as appropriate.
2. Work reception stage (from the input queue): the thread checks the state of its input queue, in order to consume the nodes left by other threads, as in HDA* Pthreads.
3. Processing stage: the main difference with HDA* MPI and HDA* Pthreads is the way in which nodes are communicated among threads. When a generated node belongs to another thread on the same machine, the node is communicated via shared-memory, using input/output queues, as in HDA* Pthreads. When a generated node belongs to a thread running on a different machine, the node is communicated via message-passing. In the last case, each thread has a send buffer for each process in the system, where nodes that must be communicated are stored, as in HDA* MPI.
4. Idle stage: t_i enters this stage when its open list is empty. Firstly, it transfers the nodes stored in each non-empty output queue and each non-empty send buffer to its owner thread/process, via shared-memory or message-passing, respectively. Then, t_i remains waiting until it receives nodes in its input queue or it receives a termination notification from the master thread. The master thread behaves differently: when it detects local termination (i.e., on the machine, using Dijkstra's termination detection algorithm for shared-memory machines [6]), it will wait for: (1) work messages (2) cost messages (3) messages that allow detecting global termination. Messages of types (1) and (2) are processed in a similar way as described above; messages of type (3) are processed based on Dijkstra's termination detection algorithm [13].

When computation ends, the master thread on each machine remains active. Together, they will retrieve the optimal solution in a distributed manner.

5 Experimental Results

Experimental tests were carried out on a cluster composed of 7 machines connected through 1 GB Ethernet. Each machine has two Intel Xeon E5620 processors and 32 GB RAM. Each processor has four 2.4 Ghz physical cores.

The tests considered sixteen 15-Puzzle instances presented in [14] (numbered 3, 15, 17, 21, 26, 32, 33, 49, 53, 56, 59, 60, 66, 82, 88, 100) and six of the 10 configurations proposed by [15] (numbered 101–106 in this paper). These configurations present different levels of complexity.

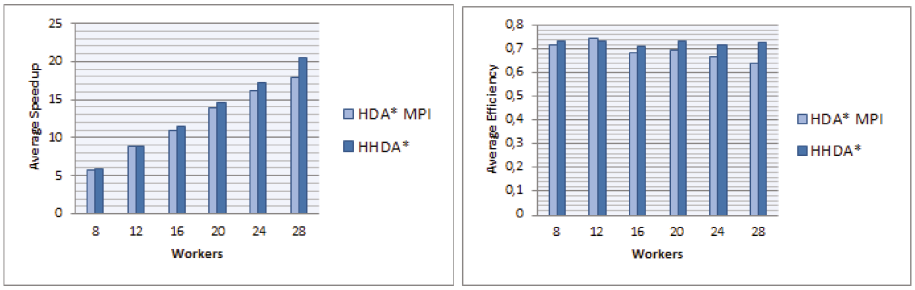
A* was run on a single machine of the previous cluster. HDA* MPI and HHDA* were run on the cluster, varying the number of machines used between 2 and 7. In HDA* MPI, 4 processes/workers were assigned to each machine. In HHDA*, 1 process (master thread) was assigned to each machine, each one will create 3 threads (4 threads/workers per machine).

In this section, we compare the performance achieved (speedup and efficiency[1]) and the amount of memory consumed by HDA* MPI and HHDA*.

5.1 Performance Analysis

Figures 1a and b illustrate the average Speedup and the average Efficiency achieved by HDA* MPI and HHDA*, for different number of workers. The results reveal that, on average, the speedup of HHDA* is similar for 8 and 12 workers, and slightly better for 16, 20, 24 and 28 workers, compared to HDA* MPI. Also, HHDA* exhibits an almost constant average Efficiency, which ranges between 0.71 and 0.73, whereas HDA* MPI shows a decreasing average Efficiency, with values ranging between 0.64 and 0.74.

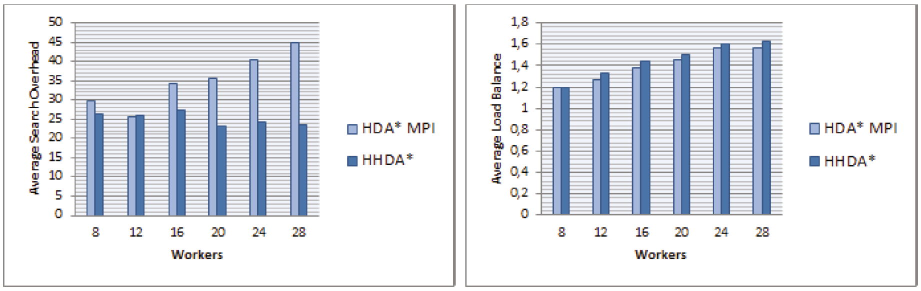
To clarify the improvement in the performance of HHDA*, Figs. 2a and b show the average Search Overhead (SO) and the average Load Balance (LB)



(a) Average Speedup (b) Average Efficiency

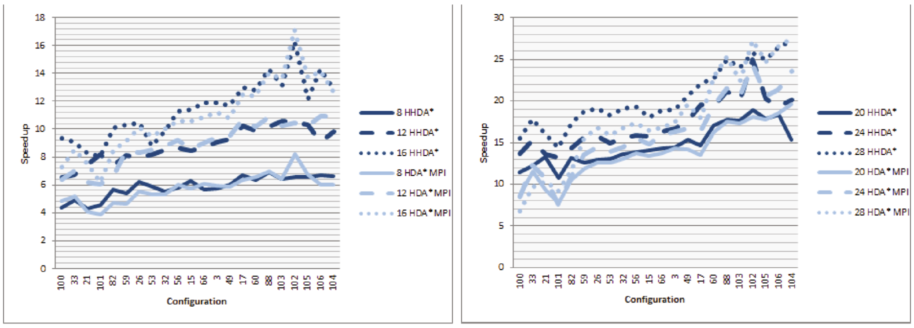
Fig. 1. Performance of HDA* MPI and HHDA*

[1] Efficiency is defined as Sp/N, where Sp is the speedup of the parallel algorithm over the sequential algorithm and N is the number of workers/cores used.



(a) Average Search Overhead (b) Average Load Balance

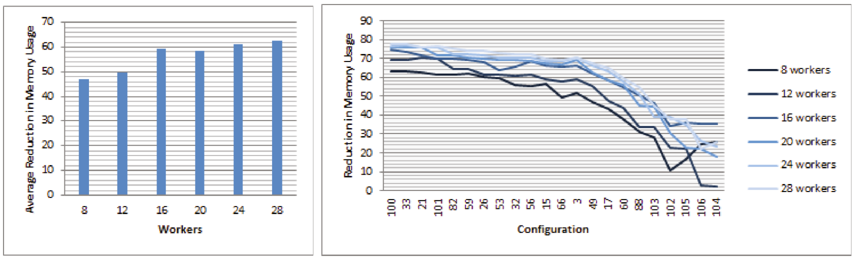
Fig. 2. Search Overhead and Load Balance of HDA* MPI and HHDA*



(a) Speedup for 8, 12 and 16 workers (b) Speedup for 20, 24 and 28 workers

Fig. 3. Speedup of HDA* MPI and HHDA*, sorted by problem complexity

achieved by HDA* MPI and HHDA*, for different number of workers. The definitions of SO and LB can be found in [3]. In general, the results show that HDA* MPI exhibits a higher average SO, compared to HHDA*, which augments as the number of workers increases, and ranges between 29% and 44%. However, HHDA* presents an almost constant average SO, which varies between 23% and 27%. SO arises as a side effect of using multiple inconsistent open lists in a parallel A* algorithm. Since each worker performs a local A*, the nodes expanded by a worker do not necessary represent a global best selection. This occurs because the access to global knowledge is restricted. This, however, has less impact on HHDA*, because threads on the same machine share *best_solution_cost*. When a thread finds a solution or receives a cost message, it updates *best_solution_cost*, so threads on the same machine immediately know this information and use it to prune nodes. Consequently, in HHDA*, the last iterations of the search explore less nodes, compared to HDA* MPI. On the other hand, the average LB is similar for both algorithms.



(a) Avg reduction in memory usage (b) Reduction in memory usage, sorted by problem complexity

Fig. 4. Reduction in memory consumption: HHDA* vs HDA* MPI

In order to determine the improvement in the performance of HHDA* by problem complexity, Figs. 3a and b illustrate the Speedup obtained by both algorithms, for different number of workers and instances (sorted by complexity). As it can be observed, when the problem scales up and the number of workers remains constant, similar values of speedup are obtained for 8 and 12 workers. However, for 16, 20, 24, and 28 workers, HHDA* performs better for some instances. Furthermore, as the number of workers increases, the number of instances which are solved more efficiently by HHDA* increases. Similar conclusions for Efficiency were reached. We observed that a lower SO is obtained by HHDA* for these instances and workers.

5.2 Memory Consumption Analysis

Figure 4a shows the average reduction in memory usage for HHDA*, with respect to HDA* MPI. We observe that the average reduction ranges between 46% and 62%, and it augments as the number of workers increases. Figure 4b illustrates the reduction in memory usage for each instance (sorted by complexity). As it can be seen, when the number of workers is constant, a higher reduction is achieved for the easier instances, and the reduction decreases as the problem scales up. In general, for hard instances, the reduction ranges between 20% and 40%.

The reduction in memory requirements for HHDA* over HDA* MPI allowed solving one of the hardest 15-Puzzle instances[2], presented in [16]. HHDA* solved this instance using 7 machines (224 GB RAM) and 28 workers. A* (1 machine, 32 GB RAM) and HDA* MPI (7 machines, 224 GB RAM, 28 workers) did not solve this instance since both algorithms ran out of memory.

6 Conclusions and Future Work

In this paper we presented HHDA*, a hybrid MPI/Pthread version of the HDA* algorithm for solving combinatorial problems. We compared the performance

[2] 15 14 13 12 10 11 8 9 2 6 5 1 3 7 4 0.

achieved and the amount of memory consumed by HDA* (pure MPI) and HHDA* (MPI/Pthreads). The results revealed that HHDA* achieves a slightly higher performance and consumes less memory, compared to HDA* (pure MPI). These improvements allowed HHDA* to solve one of the hardest 15-Puzzle instances.

As for future work, we plan to parallelize suboptimal search algorithms using our hybrid parallelization strategy.

References

1. Hart, P., et al.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968)
2. Russel, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice Hall, Upper Saddle River (2003)
3. Kishimoto, A., et al.: Evaluation of a simple, scalable, parallel best-first search strategy. Artif. Intell. **195**, 222–248 (2013)
4. Sanz, V., et al.: Scalability analysis of Hash Distributed A* on commodity cluster: results on the 15-puzzle problem. In: Proceedings of PDPTA 2016, 221–230. CSREA Press, Georgia (2016)
5. Burns, E., et al.: Best-first heuristic search for multicore machines. J. Artif. Intell. Res. **39**(1), 689–743 (2010)
6. Sanz, V., et al.: On the optimization of HDA* for multicore machines. Performance analysis. In: Proceedings of PDPTA 2014, pp. 625–631. CSREA Press, Georgia (2014)
7. Sanz, V., et al.: Performance tuning of the HDA* algorithm for multicore machines. In: Computer Science and Technology Series 2015. EDULP, La Plata (2015)
8. Chow, E., et al.: Assessing performance of hybrid MPI/OpenMP programs on SMP clusters. Technical report, UCRL-JC-143957. Lawrence Livermore National Laboratory, California (2001)
9. Rabenseifner, R., et al.: Hybrid MPI, OpenMP parallel programming on clusters of multi-core SMP nodes. In: Proceedings of PDP 2009, pp. 427–436. IEEE Computer Society, Washington, D.C. (2009)
10. Hager, G., et al.: Introduction to High Performance Computing for Scientists and Engineers, 1st edn. CRC Press, Boca Raton (2010)
11. Kumar, V., et al.: Parallel best-first search of state-space graphs: a summary of results. In: Proceedings of AAAI 1988, pp. 122–127. AAAI Press, California (1988)
12. Vidal, V., et al.: Parallel AI planning on the SCC. In: 4th Many-Core Applications Research Community (MARC) Symposium, pp. 15–20. Postsdam University Press (2011)
13. Dijkstra, E.W.: Shmuel Safra's version of termination detection. EWD-Note 998. Department of Computer Sciences, University of Texas, Austin (1987)
14. Korf, R.: Depth-first iterative-deepening: an optimal admissible tree search. Artif. Intell. **27**(1), 97–109 (1985)
15. Brüngger, A.: Solving hard combinatorial optimization problems in parallel: two cases studies. Ph.D. thesis, ETH Zurich, Dissertation ETH No. 12358 (1998)
16. Brüngger, A., et al.: The parallel search bench ZRAM and its applications. Ann. Oper. Res. **90**, 45–63 (1999)