Proceedings 10th Canadian Conference on Artificial Intelligence, AI'94, May 1994, Banff, Canada, 295-302

AIDA* – Asynchronous Parallel IDA*

Alexander Reinefeld and **Volker Schnecke**
PC^2 – Paderborn Center for Parallel Computing
D-33095 Paderborn, Germany
{ar|ossi}@uni-paderborn.de

Abstract

We present AIDA*, a generic adaptable scheme for highly parallel iterative-deepening search on large-scale asynchronous MIMD systems. AIDA* is based on a data partitioning scheme, where the different parts of the search space are processed asynchronously in parallel. Existing sequential solution algorithms can be linked to the AIDA* routines to build a fast, highly parallel search program.

Taking the 15-puzzle as an application domain, we achieved an average speedup of 807 on a 1024 processor system, corresponding to an efficiency of 79% on Korf's [1985] 25 largest problem instances. Specific problem instances yield more than 90% efficiency.

The total time taken by AIDA* to solve Korf's 100 random puzzles on a 1024-node system was 24.2 minutes. This is 5.7 times faster than the most efficient parallel algorithm on a 32 K CM-2 machine, SIDA* by Powley *et al.*

1 Introduction

Heuristic search is one of the most important techniques for problem solving in Artificial Intelligence and Operations Research. Since search algorithms usually exhibit exponential run-time, and sometimes also exponential space complexity, the design of efficient parallel searching methods is of obvious interest.

The backtracking approaches used in AI and OR benefit from a wealth of powerful heuristics that eliminate unnecessary states in the search space without affecting the final result. The most prominent methods include the universal *branch & bound* technique and *dynamic programming*, which examine only branches that are below/above a current upper/lower bound on the solution value. While these schemes are successfully applied in many problem domains, they do not work in domains with

- low solution density,
- high heuristic branching factor,
- poor initial upper/lower bounds on the optimal solution value.

Typical examples include single-agent games like the 15-puzzle [Korf, 1985], VLSI floorplan optimization [Wimer *et al.*, 1988], and some variants of the cutting stock problem [Morabito *et al.*, 1992]. For this kind of applications, there exists a simple and efficient backtracking method, called *Iterative-Deepening A* (IDA*)* [Korf, 1985], that performs a series of independent depth-first searches, each with the cost-bound increased by the minimal amount.

In this paper, we present AIDA*, a parallel implementation of iterative-deepening search on a massively parallel asynchronous MIMD system. AIDA* is based on a data partitioning scheme, where the different parts of the search space are processed asynchronously by the distributed processing elements. A simple, but effective task attraction scheme combined with a weak synchronization mechanism ensures high processor utilization and good scalability for up to more than a thousand processors.

Running on a 1024 processor transputer system, we achieved a speedup of 807 on twentyfive problem instances of the 15-puzzle, corresponding to an efficiency of 79%. Using Korf's [1985] random problem instances as a benchmark suite, AIDA* runs more than five times as fast as the fastest SIMD implementation, SIDA* by Powley *et al.* [1993], which was implemented on a CM-2 with 32 K processing elements. While such a comparison might seem unfair, because a single CM-2 processing element is about 100 times slower than the T805 transputers of our system, there are 32 times more processing elements in the CM-2. Hence one would expect our transputer program to run three times faster. However, we achieved a time improvement by a factor of 5.7, due to faster work-load balancing and almost zero synchronization costs.

In the following, we first discuss the basic ideas of sequential IDA*, give a brief overview about previous parallel approaches, and present the AIDA* algorithm. Most of the paper is devoted to the discussion of our empirical performance results, including an analysis of the various overheads.

2 Iterative-Deepening Search

Iterative-Deepening A (IDA*)* [Korf, 1985] performs a series of independent depth-first searches, each with the cost-bound increased by the minimal amount. Following the lines of the well-known A* heuristic search algorithm [Nilsson, 1980, Pearl, 1985], the total cost $f(n)$ of a node n is made up of the cost already spent in reaching that node $g(n)$, plus a lower bound on the estimated cost of the path to a goal state $h(n)$. At the beginning, the cost bound is set to the heuristic estimate of the initial state, $h(root)$. Then, for each iteration, the bound is increased to the minimum value that exceeded the previous bound, as shown in the following pseudo code:

```
procedure IDA* (n);
bound := h(n);
while not solved do
    bound := DFS(n, bound);

function DFS (n, bound);
if f(n) > bound
    then return f(n);
if h(n) = 0
    then return solved;
return lowest value of DFS(n_i, bound)
         for all successors n_i of n
```

With an admissible (=non-overestimating) heuristic estimate function h, IDA* is guaranteed to find an optimal (shortest) solution path [Korf, 1985]. Moreover, IDA* obeys the same asymptotic branching factor as A* [Nilsson, 1980], if the number of newly expanded nodes grows exponentially with the search depth [Korf, 1985, Mahanti *et al.*, 1992]. This growth rate, the *heuristic branching factor*, depends on the average number of applicable operators per node and the discrimination power of the heuristic estimate h.

3 Applications

Typical application domains for IDA* search include VLSI floorplan optimization, some variants of the cutting stock problem and single-agent games like the 15-puzzle. These problems may be characterized by a high heuristic branching factor, a low solution density and poor information about bounds that can be used to prune the search tree.

We tested the performance of our parallel AIDA* algorithm on one hundred randomly generated problem instances [Korf, 1985] of the 15-puzzle. In its more general $n \times n$ extension, this puzzle is known to be *NP*-complete [Ratner and Warmuth, 1986]. While exact statistics on solving the smaller 8-puzzle are known [Reinefeld, 1993], the 15-puzzle spawns a search space of $16!/2 \approx 10^{13}$ states, which cannot be exhaustively examined. Using IDA*, an average of 10^8 node expansions are needed to obtain a first solution with the popular *Manhattan distance* (the sum of the minimum displacement of each tile from its goal position) as a heuristic estimate function.

4 Parallel Approaches

Previous approaches to parallel iterative-deepening search include parallel window searches, tree decomposition, search space mappings and special schemes for SIMD machines.

Powley and Korf [1991] presented a *Parallel Window Search*, where each processor examines the entire search space, but with a different cost-bound. Depending on the application, this method works only for a hand full of processors (e.g., 5–9 in [Powley and Korf, 1991, p. 475]) and the solution cannot be guaranteed to be optimal.

Kumar and Rao's [1987,1990] parallel IDA* variant is based on a task attraction scheme that shares subtrees (taken from a donator's search stack) among the processors on demand. For a selected problem set they achieved almost linear speedups on a variety of MIMD computers. These favorable results, however, apply only for MIMD systems with small communication diameters, like a 128 processor Intel Hypercube, a 30 processor Sequent Balance and a 120 processor BBN Butterfly. On a 128-node ring topology their algorithm achieved a maximum speedup of 63. From Kumar and Rao's analysis, it is evident that these results do not scale up to systems of, say, some thousand processors.

The algorithm of Evett *et al.* [1990] performs a mapping of the search space onto the processing elements of a SIMD machine. This allows to eliminate duplicate states at the cost of an increased communication overhead.

Two other approaches, SIDA* by Powley *et al.* [1993] and IDPS by Mahanti and Daniels [1993] also run on the CM-2. From these, SIDA* is probably the fastest parallel IDA* implementation, solving all 100 problem instances [Korf, 1985] of the 15-puzzle in 2.245 hours.

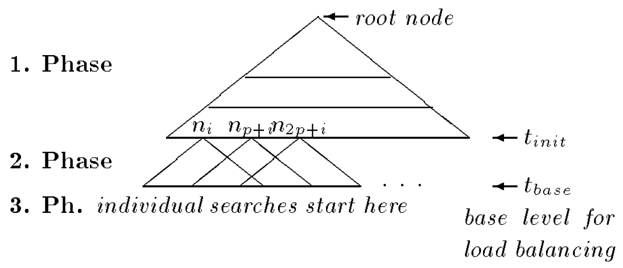

Figure 1: AIDA* Algorithm Architecture

5 AIDA*

In the following we describe *AIDA**, a generic adaptable scheme for highly parallel iterative-deepening search on asynchronous MIMD systems. AIDA* is based on a data partitioning scheme, where the different parts of the search space are processed asynchronously by the fastest available sequential routines running in parallel on the distributed processing elements. Existing sequential search code can be adapted to the parallel AIDA* system by linking the routines for initial tree partitioning, work-load balancing and communication. A simple, but efficient task attraction scheme combined with a 'weak' synchronization mechanism ensures a high processor utilization and good scalability up to some thousand processors.

AIDA* consists of three phases (cf., Fig. 1):

- a short *initial data partitioning phase*, where all processors redundantly expand the first few tree levels in an iterative-deepening manner until a sufficient amount of nodes is generated to keep each processor busy in the next phase,
- an additional *distributed node expansion phase*, where each processor expands its 'own' nodes of the first phase to generate a larger set of, say, some thousand fine grained work packets for the subsequent asynchronous search phase,
- an *asynchronous search phase*, where the processors generate and explore different subtrees in an iterative-deepening manner until one or all solutions are found.

None of these three phases requires a hard synchronization. Processors are allowed to proceed with the next phase as soon as they finished the previous one. Only in the third phase, some mechanism is needed to keep all processors working on about the same search iteration. However, this synchronization is a weak one (as opposed to hard barrier synchronization), allowing the processors to proceed with the next iteration after checking for work in their neighborhood only.

Similar to Newborn's [1988] *unsynchronized iteratively deepening parallel alpha-beta*, each processor carries out an iterative-deepening search on its selected subset of nodes. Our work-load balancing scheme ensures that all processors finish their iterations at about the same time.

5.1 Phase 1: Initial Data Partitioning

Before starting a distributed tree search, each processor must be supplied with a suitable amount of different nodes which can then be further expanded in parallel. This could be achieved in logarithmic time, $O(\log P)$, on P processors, using a binary divide-and-conquer approach. However, since communication on a MIMD-machine is usually an order of magnitude more time-consuming than the node expansion costs[1], AIDA* generates the first few tree levels redundantly on all processors. In the 15-puzzle, the processors perform an iterative-deepening search, saving all nodes of the last search frontier in a local node array, until there are at least $5 \cdot P$ entries. This gives a sufficient number of subtree roots (some 10,000 nodes) while not overflowing the memory resources of our transputer system.

At the end of this phase, duplicate nodes can be eliminated from the node array. In our experiments, however, we found that sorting the node array takes too much time. A total of 30% removed duplicate nodes (cf. [Powley *et al.*, 1993]) at the end of this phase gave only a 10% reduction of the total nodes, which did not pay for the increased overhead.

In practice, the first phase is short, taking less than three seconds (cf., Fig. 4) on the 1024-node system. There is neither communication or synchronization involved in this phase.

5.2 Phase 2: Generating Fine Grained Work Packets

In the second phase, processor P_i takes its nodes $n_i, n_{p+i}, n_{2p+i}, \ldots$ from the frontier node array t_{init} to get a wide-spread distribution of search frontier nodes. The nodes are expanded by applying two IDA* iterations, giving a new search frontier, t_{base}, as shown in Figure 1. At the end of the second phase, the local node arrays of the individual processors contain about 3000 frontier nodes each. These nodes make up the work packets used in dynamic load balancing in the third phase. As before, there is neither synchronization nor communication involved in this phase.

[1]This is especially true for the 15-puzzle with its cheap operator cost.

5.3 Phase 3: Asynchronous Search with Dynamic Load Balancing

The following iterations start on the frontier nodes t_{base}. All processors expand the nodes of their local array up to the current search-threshold. Since the size of the subtrees emanating from the t_{base} nodes is not known *a priori*, dynamic load balancing is required.

Our implementation of AIDA* employs a simple task attraction scheme. The $P = n^2$ processing elements are connected in a $n \times n$ torus topology (i.e. a mesh with wrap-around links in the rows and columns). Each processor is a member of two rings with n elements: the horizontal and the vertical ring.

A processor first expands its local frontier nodes of level t_{base}. When running out of work, it sends a `work_request` in clockwise order along the horizontal link of the torus. The first processor with unexpanded frontier nodes in its array sends a `work` packet back to the requester. If none of the processors on the horizontal ring has work to share, the request continues its path along the ring and eventually returns to the requester, indicating that the current iteration run out of work on this horizontal ring of the torus. The requester now sends a work request along the vertical ring using the same mechanism. If again no processor responds with a work packet, an `out_of_work` message is sent on both rings and this processor starts the next iteration. We call this a *weak synchronization* – as opposed to a hard barrier synchronization. It keeps all processors working at about the same iteration, while not requiring too much idle time [Newborn, 1988]. In practice, our weak synchronization works much like a majority consensus approach. When searching for a first solution, care must be taken that all processors working on shallower iterations finish their search before returning the optimal solution.

Note, that any work package is exclusively owned by a single processor. Whenever a package is transferred to another processor, it changes ownership. This is done with the expectation that all subtrees grow at about the same rate from one iteration to the next. Hence, the load balance will automatically improve during the search. In fact, the number of work packets decreases with increasing search time (cf., Fig. 7).

5.4 Implementation Details

While the above description gives a general outline of the AIDA* scheme, the actual implementation is more sophisticated:

- When a processor is done with its local nodes, it can start a new iteration when it receives an `out_of_work` message or detects a `work_request` with a higher cost threshold on the ring.
- To keep communication costs low, up to five nodes are bundled in a work packet. To avoid the donator from giving away all of its non processed nodes, only half of these are transferred.
- At the end of each iteration the nodes in t_{base} are re-ordered[2]: Medium size subtrees with $avg\_nodes/2 < x < 2 \times avg\_nodes$ are sorted to the end of the array, so that only work packets of average size will be transferred to other processors.
- In the hard problems (with many iterations), the size of work packets can differ by an order of magnitude. We therefore experimented with node splitting and node contraction strategies [Chakrabrti *et al.*, 1989] to adjust the work packets to an average size. Our preliminary results indicate that the additional overhead does not pay off.

6 Empirical Results

We implemented AIDA* on a 1024-node MIMD transputer system, using the 15-puzzle as a sample application. Figures 2 and 3 show the speedup results for two sets of 25 random problem instances with different difficulties. Speedup anomalies (cf. [Kumar and Rao, 1990]) were avoided by searching all nodes of the last (goal) iteration. We call this the *'all-solutions-case'* as opposed to the *'first-solution-case'*, where it suffices to find one optimal solution. Our fifty problem instances are the larger ones from Korf [1985], here sorted to the number of node generations in the 'all-solutions-case'. We also run AIDA* on Korf's fifty smaller problems. However, with an average parallel solution time of 8 seconds, a 1024-node MIMD system cannot be sufficiently utilized, so we did not include the data in this paper.

The speedup S of a parallel algorithm is measured as the ratio of the time taken by an equivalent and most efficient sequential implementation, $T(1)$, divided by the time taken by the parallel algorithm, $T(P)$:

$$S(P) = \frac{T(1)}{T(P)}.$$

Care was taken to use the most efficient sequential algorithm for comparison. Our IDA* is written in C and generates nodes at a rate of 35,000 nodes per second on a T805 transputer, corresponding to 350,000 nodes per second on a SUN Classic Workstation, or 660,000

[2]This is just a partial re-ordering, not a total sort. Nodes are sorted to the average size of all subtrees in the last iteration, avg_nodes.

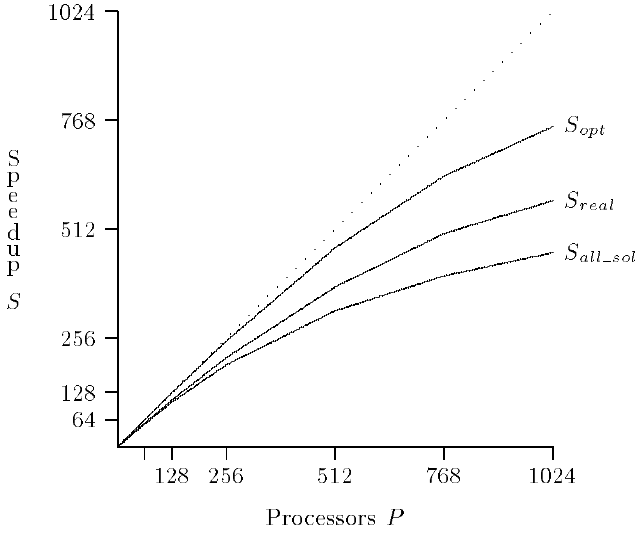

Figure 2: Speedup, prob. #51..75

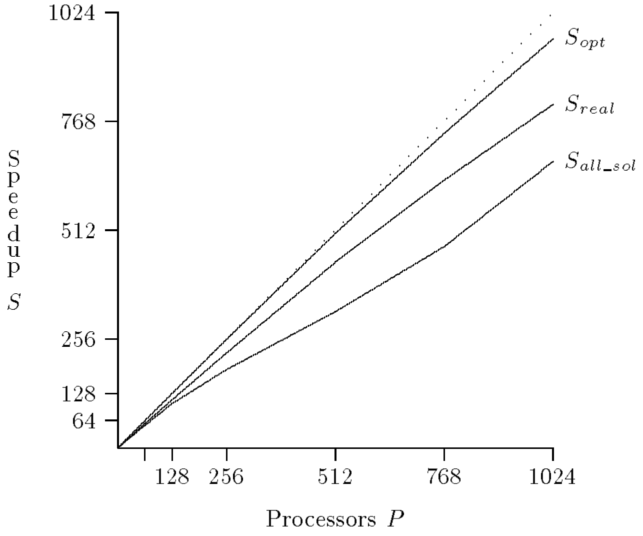

Figure 3: Speedup, prob. #76..100

nodes per second on a SUN SparcStation 10/40. Similar sequential IDA* run-times have been reported by Powley *et al.* [1993].

Figures 2 and 3 show the performance results on a torus topology. For each problem set, three graphs are shown:

S_{opt}: The topmost graph shows the maximum speedup that could be achieved with an optimal parallel algorithm (with zero overheads) after the first phase is done. This is a hypothetical measure to show how much time is taken by the initial data distribution phase.

S_{real}: The middle graph shows the speedup that would be obtained by a search for the *first* solution, one that stops right after one solution has been found. It includes the startup-time overhead, the communication overhead due to load balancing and the weak synchronization between iterations of the third phase.

S_{all_sol}: The bottom graph shows the actual speedup (measured in terms of elapsed time) of the 'all-solutions-case'. Compared to S_{real}, it also contains termination detection overhead and idle times due to processors which are done 'too soon' in the last iterations while others are still working on their last subtree.[3]

As is evident from Figures 2 and 3, good speedups are more difficult to achieve for the small problem instances #51..75 than for the hard ones #76..100. On the 1024-node system, the small problems take only an average of 16 seconds to solve, while the more difficult require three minutes. Hence, the negative effect of the initial work distribution, which is about constant for all problems, does not hamper the overall speedup in the hard problems too much.

7 Overheads in AIDA*

In this section, we analyze the various sources of overheads in more detail.

7.1 Initial Work Distribution

In the first phase, all processors perform a synchronous iterative-deepening search on the first few tree levels, storing all nodes of the last search frontier until there are at least $5 \cdot P$ nodes in each processor's local node array. This gives a sufficient number of work packets while not overflowing the memory resources.

For the larger systems, more nodes must be generated to give every processor a sufficient amount of 'own' nodes to work on. Hence, the CPU time spent in the first phase increases linearly with the system size, as shown in Figure 4. This additional node generation overhead does not reduce the overall efficiency of AIDA* in the large problems #76..100 too much. As shown in Figure 5, less than 1.5% of the total search time is spent in the first phase. Only the small problems #51..75 require up to 10% for the initial work-load distribution. This is just another manifestation of *Amdahl's Law*. The scalability of AIDA* can be improved by reducing the size of

[3] While in the 'first-solution-case' node expansion can be stopped after a first solution is found, all processors must finish searching their current subtree in the 'all-solutions-case'. Due to different work packet sizes, which vary most in the last iteration, some processors might get idle while others are still expanding their last tree. Most of the overheads in S_{all_sol} can be reduced by implementing a stack-splitting strategy as in [Kumar and Rao, 1990].

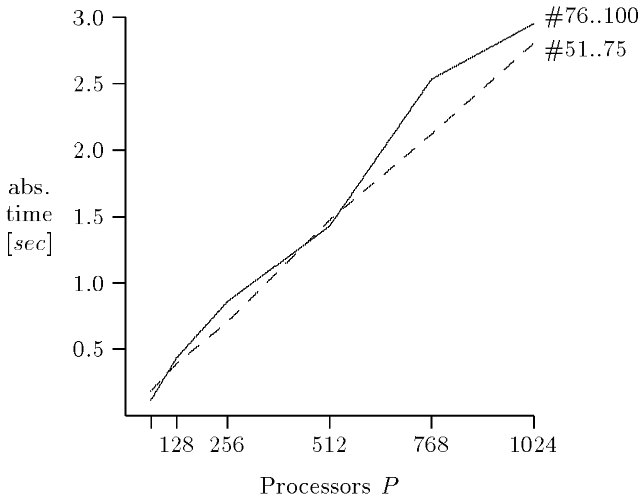

Figure 4: Absolute time of first phase

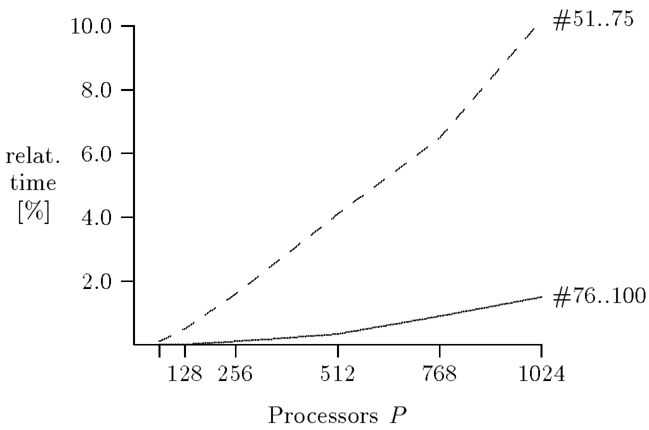

Figure 5: Time of first phase relative to total search time

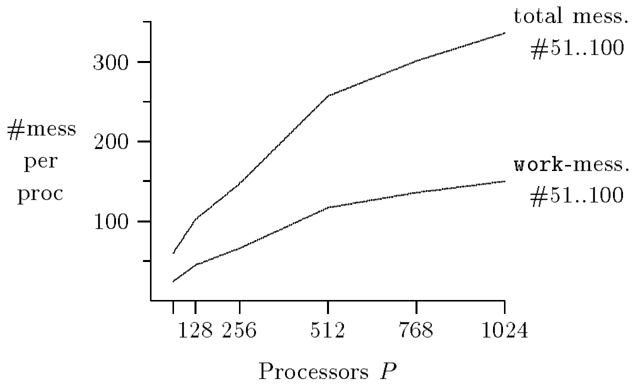

Figure 6: Messages per processor (last iteration only)

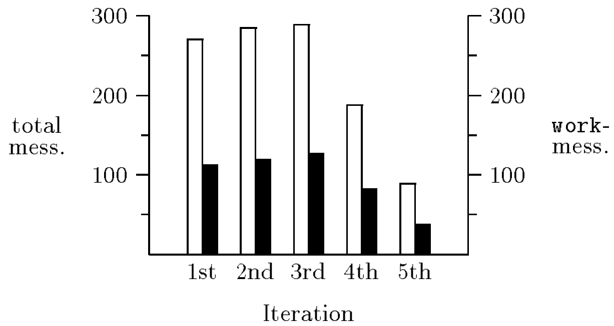

Figure 7: Messages per iteration (1024 procs.)

the first phase or by expanding different subtrees on the parallel processors in a divide-and-conquer approach.

Note, that in the second phase, every processor starts node expansion on its own subtrees, thereby fully exploiting the parallel processing power.

7.2 Communication Overhead

Communication is another source of overhead hampering the performance of parallel algorithms, especially those running on massively parallel systems. Fortunately, AIDA* exhibits a very low communication overhead. Starting with 64 processors, one would expect the communication rate to increase by a factor of sixteen when increasing the system size to 1024 processors. However, as can be seen in Figure 6, the actual number of messages increases only by a factor of six. The curves seem to level off with growing system size, which can be explained by an increased likelihood that `work_requests` are answered in the immediate neighborhood of the idle processor. As a consequence, the average distance between sender and receiver does not increase linearly with the system size.

At the end of an iteration, only a single `work_request` is sent around the ring to indicate that there is no further work available in the current iteration. All other processors on the ring are informed by an `out_of_work` message. They directly start the next iteration without asking for further work.

Moreover, as shown in Figure 7 the communication overhead seems to decrease with increasing search time. This is because the transferred nodes change ownership when being shipped to another processor, thereby constantly improving the global work-load balance. The number of messages that went through a single processor on a 1024-processor system decreases rapidly in the last two iterations. Due to this effect, one can assume an even lower communication overhead in other applications involving more iterations.

7.3 Termination Detection

With the weak synchronization scheme between the iterations, special provision must be taken to ensure that the returned solution is optimal in the 'first-solution-case'. When a goal node is found in iteration i, all processors working on iterations $< i$ must complete their search to prove that no better solution exists.

In the 'all-solutions-case' (subject of this paper) the last iteration is searched to completion until all processors examined all their assigned subtrees. Due to the varying branching degree, the subtree sizes can hardly be estimated in advance. This results in different termination times for the parallel processors. As shown in Figures 2 and 3 (compare the two bottom graphs S_{real} and S_{all_sol}) the time spent in the finishing phase is appreciable. Note however, that this will usually not occur in the search for *one* solution, a case, that is more important in practice.

8 Conclusions and Future Research

In this paper, we presented a universal scheme for asynchronous parallel iterative-deepening search on massively parallel MIMD systems. Any sequential iterative-deepening algorithm can be linked to our generic AIDA* routines without much modification. Compared to a similar parallel MIMD scheme by Kumar and Rao [1990], our method is more general and obeys less communication overhead, especially on MIMD systems with a large diameter.

Moreover, AIDA* proved to be scalable for more than one thousand processors with a high efficiency.

For the 'first-solution-case', we solved Korf's [1985] hundred puzzle instances in 24 minutes, which is 5.7 times faster than SIDA* [Powley *et al.*, 1993] running on a 32K CM-2. Comparing SIDA*'s theoretical speed of 32K processors $\times$ 322 nodes/sec $=$ $10,551,296$ nodes/sec for the whole system with AIDA*'s theoretical speed of 1024 processors $\times$ $35,000$ nodes/sec $=$ $35,840,000$ nodes/sec for our whole system, it is evident, that our MIMD approach makes better use of the processors. This is a remarkable result, considering that our faster machine solves the smallest 70 problems in less than 10 seconds each. Here, losses due to initialization are most pregnant. On the other hand, it is a more ambitious task to keep all processors of a SIMD machine working on relevant parts of the search space.

Our future work includes the solution of the 19-puzzle, where all heuristics that are known up to date, must be put together to obtain a solution within reasonable time limits. At the present time, we have solved 20 smaller[4] problems of 100 random 19-puzzle instances with an average runtime of 34 minutes on the 1024 processor system. VLSI floorplan optimization [Wimer *et al.*, 1988] is another practical application, which we intend to solve with AIDA*.

[4] avg. values: Manhattan distance h = 46, solution path length g = 67.5, 11.8 iterations, 56.2 billion nodes

Appendix: All-Solutions-Case Data

Prob.	Speedup				Time
	S(256)	S(512)	S(768)	S(1024)	T(1024)
51	163.8	278.9	363.3	419.4	11
52	195.2	351.8	492.0	303.9	18
53	209.7	393.7	389.2	453.4	11
54	196.6	360.3	502.1	620.2	9
55	211.8	402.9	430.2	419.5	10
56	196.9	355.1	503.8	597.2	10
57	184.8	341.0	474.8	541.0	11
58	211.0	330.4	446.9	531.9	11
59	213.8	338.3	460.5	563.8	12
60	210.6	403.3	577.2	500.1	13
61	190.5	354.2	323.3	349.1	24
62	221.2	393.8	562.2	707.8	11
63	215.3	359.4	492.4	604.4	13
64	213.3	411.4	457.1	560.8	14
65	208.8	401.4	573.4	728.9	13
66	211.1	317.2	432.1	518.8	18
67	216.2	413.8	473.6	579.1	16
68	219.9	399.0	570.5	730.6	17
69	216.5	415.8	595.3	769.2	14
70	216.2	416.9	599.4	534.9	24
71	211.2	352.2	494.9	616.5	20
72	212.5	427.1	625.3	653.5	21
73	221.4	404.5	576.6	736.5	21
74	222.4	406.5	589.6	754.0	24
75	222.0	382.8	541.3	682.8	23
76	220.3	428.8	624.4	811.9	21
77	218.0	423.5	625.0	801.0	32
78	218.0	413.7	599.7	775.1	34
79	222.0	444.2	536.3	666.0	37
80	221.3	397.1	559.1	717.3	34
81	220.1	431.3	637.1	677.9	42
82	208.3	396.8	580.3	733.0	40
83	220.1	430.5	636.6	662.2	55
84	221.1	434.6	565.7	719.4	46
85	221.4	433.5	562.4	709.7	54
86	219.2	429.7	633.2	828.7	62
87	220.7	432.7	647.2	846.0	66
88	223.1	439.8	602.4	777.2	88
89	220.5	435.7	646.9	852.4	79
90	225.5	435.2	642.4	845.9	80
91	225.7	396.7	576.5	734.8	125
92	226.2	448.7	671.2	890.6	165
93	224.1	445.5	664.6	882.6	144
94	223.6	446.0	665.9	869.9	149
95	225.5	450.7	658.5	870.7	175
96	224.8	448.1	652.7	858.9	207
97	226.3	449.8	672.3	897.4	214
98	235.8	467.6	698.1	924.9	520
99	230.4	452.2	675.0	896.4	579
100	230.4	459.8	687.3	915.3	1300

Acknowledgements

Thanks to Tony Marsland for many valuable comments and for visiting the PC^2 at the right time.

References

[Altmann *et al.*, 1988] E. Altmann, T. A. Marsland, T. Breitkreutz. *Accounting for Parallel Tree Search Overheads.* Procs. Int. Conf. Par. Proc. (1988), 198 – 201

[Chakrabrti *et al.*, 1989] P.P. Chakrabarti, S. Ghose, A. Acharya, S.C. de Sarkar. *Heuristic search in restricted memory.* Art. Intell. 41,2(1989/90), 197 – 221.

[Evett *et al.*, 1990] M. Evett, J. Hendler, A. Mahanti, D.S. Nau. *PRA*: A memory-limited heuristic search procedure for the Connection Machine.* 3rd IEEE Symp. Frontiers Mass. Par. Comp. (1990), 145 – 149.

[Korf, 1985] R.E. Korf. *Depth-first iterative-deepening: An optimal admissible tree search.* Art. Intell. 27(1985), 97 – 109.

[Kumar and Rao, 1990] V. Kumar, V.N. Rao. *Scalable parallel formulations of depth-first search.* In: Kumar, Gopalakrishnan, Kanal (eds.), Parallel Algorithms for Machine Intelligence and Vision, Springer-Verlag (1990), 1 – 41.

[Mahanti *et al.*, 1992] A. Mahanti, S. Ghosh, D.S. Nau, A.K. Pal and L. Kanal. *Performance of IDA* on trees and graphs.* 10th Nat. Conf. on Art. Int., AAAI-92, San Jose, CA, (1992), 539 – 544.

[Mahanti and Daniels, 1993] A. Mahanti, C.J. Daniels. *A SIMD approach to parallel heuristic search.* Art. Intell. 60(1993), 243 – 282.

[Morabito *et al.*, 1992] R.N. Morabito, M.N. Arenales, V.F. Arcaro. *An and-or-graph approach for two dimensional cutting problems.* European J. of OR 58(1992), 263 – 271.

[Newborn, 1988] M. Newborn. *Unsynchronized iteratively deepening parallel alpha-beta search.* IEEE Trans. Pattern Anal. Mach. Int., PAMI-10,9 (1988), 687 – 694.

[Nilsson, 1980] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing, Palo Alto, CA, (1980).

[Pearl, 1985] J. Pearl. *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, (1984).

[Powley and Korf, 1991] C. Powley, R.E. Korf. *Single-agent parallel window search.* IEEE Trans. Pattern Anal. Mach. Int., PAMI-13,5 (1991), 466 – 477.

[Powley *et al.*, 1993] C. Powley, C. Ferguson, R.E. Korf. *Depth-first heuristic search on a SIMD machine.* Art. Intell. 60(1993), 199 – 242.

[Rao *et al.*, 1991] V.N. Rao, V. Kumar, R.E. Korf. *Depth-first vs. best-first search.* 9th Nat. Conf. on Art. Int. AAAI-91, Anaheim, CA, (1991), 434 – 440.

[Rao and Kumar, 1987] V.N. Rao, V. Kumar. *Parallel depth-first search. Part I. Implementation.* Int. J. Par. Progr. 16,6(1987), 479 – 499.

[Ratner and Warmuth, 1986] D. Ratner, M. Warmuth. *Finding a shortest solution for the $N \times N$ extension of the 15-puzzle is intractable.* AAAI-86, 168 – 172.

[Reinefeld and Marsland, 1993] A. Reinefeld, T.A. Marsland. *Enhanced iterative-deepening search.* Univ. Paderborn, FB Mathematik-Informatik, Tech. Rep. 120 (March 1993), to appear IEEE-PAMI.

[Reinefeld, 1993] A. Reinefeld. *Complete solution of the Eight-Puzzle and the benefit of node-ordering in IDA*.* Procs. Int. Joint Conf. on AI, Chambéry, Savoi, France (Sept. 1993), 248 – 253.

[Russell, 1992] S. Russell. *Efficient memory-bounded search methods.* European AI-Conference, ECAI-92, Vienna, (1992), 1 – 5.

[Wimer *et al.*, 1988] S. Wimer, I. Koren, I. Cederbaum. *Optimal aspect ratios of building blocks in VLSI.* Procs. 25th ACM/IEEE Design Automation Conference, 1988, 66 – 72.