Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-89-45

1989-10-01

Parallel Iterative-Deepening Search

David J. Harker

This report describes my implementation of a parallel iterative-deepening A* search algorithm on a NCUBE parallel computer. I present a detailed description of the algorithm, followed by a discussion of its theoretical performance. The actual performance of my implementation is compared to the theoretical model, and a summary of the results is presented.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Harker, David J., "Parallel Iterative-Deepening Search" Report Number: WUCS-89-45 (1989). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/755

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

PARALLEL ITERATIVE-DEEPENING SEARCH

David J. Harker

WUCS-89-45

October 1989

Center for Intelligent Computer Systems
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

ABSTRACT

This report describes my implementation of a parallel iterative-deepening A* search algorithm on an NCUBE parallel computer. I present a detailed description of the algorithm, followed by a discussion of its theoretical performance. The actual performance of my implementation is compared to the theoretical model, and a summary of the results is presented.

This work is supported by the McDonnell Douglas Corporation.

Parallel Iterative-Deepening Search

1. Introduction

Many Artificial Intelligence applications search large solution spaces, often resulting in slow execution, due to the complexity of the search process and the size of the search space. Heuristics, or rules of thumb, can increase the search efficiency by focusing the search in the most promising direction, if the domain consistently converges to a solution in a measurable manner.

The most common heuristic search is the A* algorithm, which always finds the solution of lowest cost, if the heuristic function is admissible, as described in Korf [1]. To eliminate the exponential memory usage of A*, Korf describes a modified version, the depth-first iterative-deepening A* (IDA*). IDA* is described and analyzed in [1]. Rao and Kumar [2, 4, 5, 6] developed a parallel version of the IDA* (PIDA*), which is the basis of this work.

2. Iterative-Deepening A*

In IDA*, the heuristic function of the A* algorithm is used to guide a cost-bounded depth-first search over the solution space. When a branch of the search exceeds the current cost threshold the algorithm backtracks, until all branches are searched. A new, higher, cost bound is determined, and the process begins again. This cycle is repeated until a goal is found.

On the first iteration, the cost threshold is set to the value returned by h(start state), where h() is the heuristic function (see [1] for description). On each successive iteration, the new threshold is set to the minimum of the values which exceeded the current threshold. Korf [1] shows this algorithm to be optimal in terms of time, space, and cost of solution path, for heuristic state-space search. It eliminates the exponential memory usage of the A*, since only the current branch of the depth-first search need be stored in memory.

In this work, the IDA* algorithm has been implemented in C on a Sun 4/280 computer, and has been tested on the 15-puzzle problem [3]. The results are very encouraging, and are summarized in Appendix B.

3. Parallel IDA*

The parallel IDA* algorithm searchs the tree as the sequential version does, but each processor searchs a disjoint portion of the tree, maintaining its own separate stack. When a processor's stack becomes empty, it requests a portion of a stack from a neighbor. At the end of an iteration, the processors must decide what the next cost threshold will be. When a goal is found, the processor finding it sends it to the host, which then tells the other processors to stop. Pseudo-code for PIDA* is in Appendix A, and should be understandable after reading the following detailed explanation of the algorithm.

Initially, the host sends the starting state (in this case, the initial configuration of the tiles) to processor 0. Processor 0 creates a root node, expands the root (finds its children) and begins the search. The other processors begin by requesting work from their immediate neighbors, until some work is received. While they are doing this, they must also be aware of other messages they may receive, such as other nodes asking them for work, or a message from node 0, signaling the end of an iteration. When a processor requests work from another, the reply must be sent back as soon as possible, so that the requesting processor can get work quickly. Thus even while in a loop, asking for work, a processor must respond to *other* processors also requesting work.

These processors requesting work must also be aware of the end of an iteration. Since processor 0 and its immediate neighbors are searching, they eventually may finish an iteration before some processors have any data. At this time, *all* processors need to send their current cost thresholds to processor 0,

which calculates the next threshold and sends it to all other processors, starting the next iteration. Thus the communication between processors is complex, and whenever a processor is sending or receiving messages it must check for *all* possible message types and sources, and respond to them appropriately and quickly, before continuing.

As the search progresses, all processors will have work, since there are thousands of nodes after only a few iterations, and each iteration is now long enough to allow the work to spread. In order to facilitate this spreading of work, each processor maintains a index pointing to the half-way point in its stack. Since each stack grows upwards, the nodes below this half-way point in the stack are in the upper half of the tree. When a processor gives some work to a neighbor, it only gives nodes from this upper half of the tree, to make sure that it gives a substantial amount of work away, yet keeps a substantial amount for itself, to maximize the efficiency of the search. Kumar and Rao [5] experimented with various stack splitting strategies, and found that the one described above performed best, on average.

In this implementation, when a processor sends work to another processor, it removes nodes from its own stack and packs them into a buffer, which is sent to the requester. When this buffer is received, a new stack must be created, using the buffer as the source. Once this stack is built, the processor can begin searching as if it had generated the stack itself. Since this is a complex process, it's use must be minimized, hence the experiments done by Kumar and Rao in [5], attempting to find the best stack splitting strategy.

This work splitting process takes place at the beginning of each iteration, since the algorithm starts over at the root node on each iteration. Eventually, a processor will find a goal. When this happens, the processor sends the result to

the host, which then sends a termination message to the other processors. While the search is running, the host processor of the NCUBE is free to continue working on something else, since it is not involved in the search algorithm at all. So not only is the search done in parallel, but while it is running the host can *also* continue processing, in parallel with the search.

4. Theoretical Performance

Assuming a problem which requires M nodes to be searched before a solution is found, the sequential IDA* algorithm obviously searches each node, one at a time. The parallel algorithm, given the same problem, and N processors, requires M/N nodes be searched by each processor. Given a large calculation to communication ratio, this implies a linear speedup.

Since in a realistically sized problem there are hundreds of thousands, if not millions, of nodes, the communication time is tiny when compared to the computation time. This is intuitively obvious, since the stacks grow very large when there are hundreds of thousands of nodes, minimizing the need for communication, since all processors are searching their huge stacks. After only a few iterations each iteration takes at least several minutes to run (due to exponential growth of the tree), since in an iteration each processor must empty its stack, searching every possible node that does not exceed the current cost threshold. This long iteration time minimizes the time spent in communication, since work requests and transitions between iterations occur very infrequently. A runtime analysis of our implementation verifies this hypothesis.

Thus the implication of linear speedup appears to be plausible, since given a large enough problem, nearly all the time is spent in computation, and the computational load is split evenly among N processors. As an added advantage, in some problem spaces (as described below) the PIDA* algorithm produces a

slightly *superlinear* speedup, although most cases should be "only" linear. Of course, all cases where a linear or superlinear speedup is expected assume a very large search space, providing a sufficiently large computation to communication ratio.

In [2, 4, 5] Kumar and Rao found a superlinear speedup in several cases. They analyze the efficiency of PIDA* in great detail in [5], and the preceding paragraphs are a brief summary of this work. They conclude that the speedup obtained will either be linear (when the distribution of solutions is uniform) or superlinear (when distribution of solutions is non-uniform and no heuristic information on this distribution is available). Briefly, superlinear speedup is possible when some areas of the tree have a high density of solutions, and there exists no heuristic information telling where these areas are. Since it has many searches executing at one time, the PIDA* algorithm nearly always has some processor searching in a high density area, while the sequential algorithm is often *not* searching a high density area, since it only has one search executing, at only one point in the tree. Thus for some problems, superlinear speedup is to be expected, although most problems will be roughly linear.

5. Actual Performance

Before discussing the speedup obtained from the parallelism, we will compare a single processor of the NCUBE and the sequential code on the SUN 4/280, to determine the "real-world" benefits of using a parallel computer. On the 15-puzzle problem described in Appendix B, the single processor of the NCUBE required 393 seconds, and the SUN required 18.0 seconds, so the SUN is 21.8 times faster on this problem.

This is a reflection on the age of the NCUBE, since it is a "first-generation" parallel computer, and each processor has roughly twice the speed of a VAX 750,

much slower than a SUN 4/280. However, the results indicate the *usable* speedup we can expect from the NCUBE on this particular problem, since the usable speedup of a parallel computer is the execution time of the parallel machine compared to the fastest available sequential computer, in this case the SUN. Assuming a linear speedup of 64 (on our 64 processor NCUBE), the execution time on this problem will be 6.1 seconds, which is a usable speedup of 2.9 when compared to the SUN. A huge problem space is needed to obtain a linear speedup, so on smaller problems the SUN may provide a faster overall execution time. This will occur any time the speedup on the NCUBE is less than 21.8, since when the speedup is 21.8, the NCUBE and the SUN have equal execution times of 18 seconds, based on the results described above.

However, the current hypercube machines from Intel and newer machines from NCUBE have significantly more power than the NCUBE used here, so would yield much higher usable speedup. As an aside, the code running on the NCUBE is a direct port of the code on the SUN, so the SUN has no advantage stemming from more highly optimized code. Both the sequential and parallel versions are written in C, and are as optimized as our time and the code's readability allow. The programs use look-up tables whenever possible, and do their own memory management, as described in Appendix B.

6. Parallel Performance

As explained in section 4, a sufficiently large problem should give linear or even *superlinear* speedup. Our results confirm this, on up to four processors. In the interest of practical execution time, we chose five of the smallest instances of the 15-tile problem listed in [1] as our test cases. These selected problems have search spaces ranging in size from 540,000 to 11,861,000 nodes, and have an average solution length of 46. The 100 randomly-generated problems listed in [1]

have search spaces as large as 6 billion nodes, and an average solution length of 53. We chose only these five small problems out of the 100 listed because evaluating each test case involves running the program at least 50 times. This is due to the dynamic nature of the work distribution (based entirely upon the ordering of the arrival of inter-processor messages), which makes the run-times on more than four processors vary widely, so each such run was repeated many times, in order to find an average of the run-times. We typically ran each problem ten times on each configuration of 8, 16, 32 and when possible, 64 processors. The extremely slow execution time on only one processor (the baseline run) was also a factor in choosing only small problems. The tables showing the results for these five problems are in Appendix C, as is a table showing the speedup results averaged across all five problems. The results are incomplete on runs using 64 processors, since exclusive access to the NCUBE was a rarity.

On up to four processors, our implementation produced linear speedup. With two processors, we found an average speedup of 2.058, and on four processors the average speedup was 4.002. The performance began to trail off at eight processors, with a speedup of 6.772. This is still almost 85% efficiency, where efficiency is defined by the actual speedup divided by the number of processors. The tables in Appendix C show the efficiency of each test case. We are pleased by these results, and expect similar linear speedup on larger problem spaces.

7. Conclusion

In conclusion, we have implemented the PIDA* algorithm developed by Korf and Rao in [2, 4, 5, 6] on an NCUBE parallel computer, and verified the linear speedups found by Korf and Rao on other parallel computers. Since we have no access to low-level details of Korf and Rao's software, our independent implementation of the PIDA* algorithm confirms its basic soundness.

Another significant benefit of both IDA* and PIDA* is the very efficient use of memory, and the predictability and constant nature of the memory requirements. In constrast to A*, which uses exponential memory and requires an increasing amount of memory as the search progresses, the IDA* algorithm allocates a fixed amount of memory once, and the amount is determined by the needs of the application program.

The linear speedup and efficient memory usage of PIDA* make it an attractive choice for software developers interested in superb hueristic search performance on parallel computers.

References

[1] Korf, Richard E. (1985). Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. In *Artificial Intelligence*, Elsevier Science Publishers B.V., North Holland. pp. 97-109.

[2] Kumar, V., Rao, V. N., and Ng, H. H. (1989). On the Efficiency of Parallel Depth-First Search, *Manuscript under preparation.* Department of Computer Sciences, University of Texas at Austin. January 19, 1989.

[3] Luger, G. F. and Stubblefield, W. A. (1989). *Artificial Intelligence and the Design of Expert Systems,* The Benjamin/Cummings Publishing Company, Inc, New York. pp. 84-86.

[4] Rao, V. N., Kumar, V., and Ramesh, K. (1987). A Parallel Implementation of Iterative-Deepening-A*, In *Proceedings of the National Conf. on Artificial Intelligence AAAI,* pp. 878-882.

[5] Rao, V. N., and Kumar, V. (1987). Parallel Depth First Search. Part 1. Implementation. In *International Journal of Parallel Programming,* Vol. 16, No 6. December 1987, Plenum Publishing Corporation, Belgium. pp. 479-499.

[6] Rao, V. N., and Kumar, V. (1987). Parallel Depth First Search. Part 2. Analysis. In *International Journal of Parallel Programming,* Vol. 16, No 6. December 1987, Plenum Publishing Corporation, Belgium. pp. 501-519.

Appendix A - Pseudo-code Listings, based upon [4]

Driver Loop on Processor 'i'

```
PROCESSOR[i]

 while (not solutionfound)
  if work is available in stack[i]
    perform Bounded Depth-First Search on stack[i];
  else if (GETWORK = SUCCESS)
    continue search
  else (iteration finished)
    /* determine cost bound for next iteration */
    if (this node is not node 0)
      wait until node 0 is done with its iteration
      send my next_f_bound to node 0
      read new next_f_bound from node 0
    else (this node is node 0)
      signal all other nodes
      collect all other next_f_bounds
      send MIN(all next_f_bounds) to all other nodes
    endif
    initialize stack, depth, f_bound, next_f_bound
      for next iteration
  endif
endwhile
```

Appendix A - Pseudo-code Listings, based upon [4]

Bounded Depth-First Search on Processor 'i'

```
Bounded DFS(startstack,movegen,h)

 /* work is available in the stack and depth, f_bound, and
   next_f_bound have been properly initialized. */

 excdepth = -1;
 while ((not solutionfound) and (depth > 0))
  if there are no children in the top element of the stack
    POP;
    depth = depth - 1;        /* BACKTRACK */
    if (depth < excdepth)
         excdepth = depth / 2;
    endif
  else
    remove nextchild from TOP;
    if (nextchild.cost <= f_bound)
     if nextchild is a solution
          solutionfound = TRUE;
          send result to host;
          send quit message to all other processors;
          QUIT;
     endif
     PUSH (nextchild, movegen(nextchild,h));
     check for requests for work from other nodes
     depth = depth + 1;      /* ADVANCE */
     excdepth = MAX(depth/2, excdepth);
    else
     next_f_bound = MIN(next_f_bound,nextchild.cost);
    endif
  endif
endwhile
```

Appendix A - Pseudo-code Listings, based upon [4]

Getwork() and Givework() on Processor 'i'

GETWORK()

```
for (n = each neighbor of this node)
  ask n for work
  if (receive end-of-iteration message instead)
    /* even if this node has no work, it must
       'synch up' with the others */
    send dummy next_f_bound to node 0
    read new next_f_bound
    continue asking n for work
  endif
  if (n has work to share)
    read buffer sent by n
    use (node state) pairs in buffer to build new stack.
    set internal stack variables to describe new stack.
    return
  endif
endfor
```

GIVEWORK()

```
for (n = each neighbor of this node)
  check for message from n
  if (receive end-of-iteration message instead)
    send next_f_bound to node 0
    read new next_f_bound
    return (to begin next iteration in synch with others)
  endif
  if (n requests work from this node)
    if (this node has no work)
      tell node n 'have no work to share'
    else (this node has work to share)
      remove nodes from each level of
        the top half of the tree
      place these nodes in buffer
      send buffer to node n
    endif
  endif
endfor
```

Appendix B - Results of Sequential IDA* Implementation

Implementation Details

The IDA* algorithm, as described in [2, 4], is implemented in the C language on a SUN 4/280 computer. The implementation does its own memory management, since nodes are constantly being created, then thrown away as the next iteration begins. By allocating one large block of memory once, and building the search stack in this space, many calls to "malloc()" and "free()" are eliminated. The execution time of this implementation was reduced to 45% of the original time when we converted to our own memory management.

Results

When run on the 15-puzzle problem (initial state 79 from the 100 random instances Korf generated in [1]) this implementation found the solution in 18 seconds, searching ~168,000 nodes. A total of ~345,000 nodes were generated, but only those not exceeding the cost threshold (168,000) are searched.

The most significant result is the very efficient memory usage, and the predictability and constant nature of the memory requirements. Only 80K of memory is needed for the search instance mentioned above (the real memory (resident) size of the process is 80K), and this memory size is constant throughout the execution time, for any instance of the 15-puzzle problem.

The application using this search implementation determines the size of the data area. The stack depth is currently fixed at 100, which is sufficient for our test problems. (The largest of Korf's 100 instances of the 15-tile problem searched six Billion nodes with a stack 62 layers deep.) The amount of memory required is obviously domain dependent, since it depends upon the branching factor of the search space, and the size of the domain's "state" data structure.

Appendix C - Results of Parallel IDA* Implementation

Average speedup results, found by averaging the five sets of results also listed in this appendix:

N	Speedup	Efficiency (%)
1	1	100
2	2.058	103
4	4.002	100
8	6.772	85
16	7.396	46
32	9.772	31
64	12.420	19

Results from Problem 79:

N	Runtime	Speedup	Efficiency (%)
1	393	1	100
2	126	3.11	156
4	75	5.25	131
8	35	11.18	140
16	54	7.31	46
32	41	9.69	30
64	33	11.79	18

Results from Problem 47:

N	Runtime	Speedup	Efficiency (%)
1	3872	1	100
2	1667	2.32	116
4	502	7.70	193
8	411	9.42	118
16	324	11.95	75
32	276	14.0	44

Results from Problem 19:

N	Runtime	Speedup	Efficiency (%)
1	1841	1	100
2	831	2.21	110
4	695	2.65	66
8	524	3.51	44
16	347	5.30	33
32	293	6.27	20
64	141	13.05	20

Results from Problem 31:

N	Runtime	Speedup	Efficiency (%)
1	3477	1	100
2	2435	1.43	71
4	1728	2.01	50
8	817	4.25	53
16	399	8.71	54
32	257	13.50	42

Results from Problem 13:

N	Runtime	Speedup	Efficiency (%)
1	6595	1	100
2	5414	1.22	61
4	2738	2.41	60
8	1189	5.50	69
16	1778	3.71	23
32	1224	5.40	17