A Parallel Implementation of Iterative-Deepening-A*

V. Nageshwara Rao, Vipin Kumar and K. Ramesh

AI TR87-46 January 1987

This work was supported by Army Research Office grant #DAAG29-84-K-0060 to the Artificial Intelligence Laboratory and by the Parallel Processing Equipment grant from ONR to the Department of Computer Science at the University of Texas at Austin.
A Parallel Implementation of Iterative-Deepening-A*\textsuperscript{1}

V. Nageshwar Rao, Vipin Kumar\textsuperscript{2} & K. Ramesh
Artificial Intelligence Laboratory
Computer Science Department
University of Texas at Austin
Austin, Texas 78712

ABSTRACT

This paper presents a parallel version of the Iterative-deepening A* (IDA*) algorithm. Iterative-Deepening-A* is an important admissible algorithm for state-space search which has been shown to be optimal both in time and space for a wide variety of state-space search problems. Our parallel version retains all the nice properties of the sequential IDA* and yet does not appear to be limited in the amount of parallelism. To test its effectiveness, we have implemented this algorithm on Sequent Balance 8000 parallel processor to solve the 15-puzzle problem, and have been able to obtain almost linear speedups on the 9 processors that are available on our machine. On machines with larger number of processors, we expect that the speedup will still grow linearly. The parallel version seems suitable even for loosely coupled architectures such as the Hypercube.

\textsuperscript{1} This work was supported by Army Research Office grant \#DAAG29-84-K-0060 to the Artificial Intelligence Laboratory and by the Parallel Processing Equipment grant from ONR to the Department of Computer Science at the University of Texas at Austin.

\textsuperscript{2} Arpanet Address: kumar@sally.UTEXAS.EDU
Telephone: (512)-471-9571 (off)
A Parallel Implementation of Iterative-Deepening-A*

V. Nageshwara Rao, Vipin Kumar and K. Ramesh
Artificial Intelligence Laboratory
Computer Science Department
University of Texas at Austin

1. INTRODUCTION

Search permeates all aspects of AI including problem solving, planning, learning, decision making, natural language understanding. Even though domain-specific heuristic knowledge is often used to reduce search, the complexity of many AI programs can be attributed to large potential solution spaces that have to be searched. With the advances in hardware technology, hardware is getting cheaper, and it seems that parallel processing could be used cost-effectively to speedup search. Due to their very nature, search programs seem naturally amenable to parallel processing. Hence many researchers have attempted to develop parallel versions of various AI search programs (e.g., Game Tree search [KANAL 81] [LEIFKER 85] [FINKEL 82] [FINKEL 83] [MARSLAND 82], AND/OR graph search [KUMAR 84] [KIBLER 83], State-Space Search [RAO 87] [IMAI 79] [KORNFELD 81]).

Even though it may seem that one could easily speedup search N times using N processors, in practice, N processors working simultaneously may end up doing a lot more work than a single processor. Hence the speedup can be much less than N. In fact, early experience in exploiting parallelism in search was rather negative. For example, Fennel and Lesser's implementation of Hearsay II gave a speedup of 4.2 with 16 processors [FENNEL 77] (Kibler and Conery mention many other negative examples in [CONERY 85]). This early experience led to a pessimism that perhaps AI programs in general have very limited effective parallelism.

We have developed a parallel version of Iterative-Deepening-A* (IDA*) [KORF 85] that does not appear to be limited in the amount of parallelism. To test its effectiveness, we have implemented this algorithm to solve the 15-puzzle problem on Sequent Balance 8000 parallel processor, and have been able to obtain almost linear speedup on the 9 processors that are available on the machine. On machines where larger number of
processors are available, we expect that the speedup will still grow linearly.

Iterative-Deepening-A* is an important admissible state-space search algorithm, as it runs in asymptotically optimal time for a wide variety of search problems. Furthermore, it requires only linear storage. In contrast, A* [NILSSON 80], the most known admissible state-space-search algorithm, requires exponential storage for most practical problems [PEARL 84]. Our experience in parallelizing IDA* and A*[RAO 87] has shown that IDA* is more amenable to parallel processing than A* in terms of simplicity and overheads. The parallel version of IDA* is also efficient in storage.

In Section 2, we present an overview of IDA*. In Section 3, we discuss one way of parallelizing IDA* and present implementation details. In Section 4, we present speedup results of our parallel IDA* for solving the 15-puzzle problem on Sequent Balance 8000. Section 5 contains concluding remarks. Throughout the paper, we assume familiarity with the standard terminology (such as "admissibility", "cost-function", etc.) used in the literature on search [NILSSON 80] [PEARL 84].

2. ITERATIVE-DEEPENING-A* (IDA*).

Iterative Deepening consists of repeated bounded depth-first search (DFS) over the search space. In each iteration, IDA* performs a cost-bounded depth-first search, i.e., it cuts off a branch when its total cost \( f = g + h \) exceeds a given threshold. For the first iteration, this threshold is the cost \( f \)-value of the initial state. For each new iteration, the threshold used is the minimum of all node costs that exceeded the (previous) threshold in the preceding iteration. The algorithm continues until a goal is expanded. If the cost function is admissible, then IDA* (like A*) is guaranteed to find an optimal solution.

For exponential tree searches, IDA* expands asymptotically the same number of nodes as A*. It is quite clear that the storage requirement of IDA* is linear with respect to the depth of the solution. For a detailed description of IDA* and its properties, the reader is referred to [KORF 85]. In the following figure we give an informal description of IDA*.
Fig. 1

IDA*(startstate,h,movegen)
/* h is an admissible heuristic function for the problem */
/* movegen(state,fun) generates all sons of state and returns
them ordered according to heuristic function fun. */
/* cb is cost bound for current iteration        */
/* nextcb is cost bound for next iteration     */

nextcb = h(startstate) ;
while (not solutionfound)
{
    cb = nextcb ;
    nextcb = +∞ ;
    PUSH(startstate,movegen(startstate,h)) ;
    depth = 1 ;
    while (depth > 0)
    {
        if there are no children in the TOP element of the stack
            POP ;
        depth = depth - 1 ;    /* BACKTRACK */
        else
            remove nextchild from TOP ;
        if (nextchild.cost \leq cb)
            if nextchild is a solution
                solutionfound = TRUE ;
                QUIT ;
                PUSH(nextchild,movegen(nextchild,h)) ;
                depth = depth + 1 ;    /* ADVANCE */
        else
            nextcb = MIN(nextcb,nextchild.cost) ;
    }
} /* POP, PUSH and TOP are operations on the DFS stack */
/* The elements of the stack are state-children pairs */
/* The children are ordered according to h. This ensures that the the children of a node
are explored in increasing h order */
/* The cost function used is f(n) = g(n) + h(n) */
    { End of Fig. 1 }
3. A PARALLEL VERSION OF ITERATIVE-DEEPENING-A* (PIDA*)

3.1 Basic Concepts.

We parallelize IDA* by sharing the work done in each iteration (i.e., cost-bounded depth-first search) among a number of processors. Each processor searches a disjoint part of the cost-bounded search space in a depth-first fashion. When a process has finished searching its part of the (cost-bounded) search space, it tries to get an unsearched part of the search space from the other processors. When the cost-bounded search space has been completely searched, the processors detect termination of the iteration and determine the cost bound for the next iteration. When a solution is found, all of them quit.

Since each processor searches the space in a depth-first manner, the (part of) state-space available is easily represented by a stack (of node-children pairs) such as the one used in IDA* (see Fig. 1). Hence each processor maintains its own local stack on which it performs bounded DFS. When the local stack is empty, it demands work from other processors. In our implementation, at the start of each iteration, all the search space is given to one processor, and other processors are given null space (i.e., null stacks). From then on, the state-space is divided and distributed among various processors.

The basic driver routine in each of the processors is given in Fig. 2.

**Fig. 2**

```
PROCESSOR(i)
while (not solutionfound)
{
    if work is available in stack[i]
        perform Bounded DFS on stack[i] ;
    else if (GETWORK = SUCCESS )
        continue ;
    else if (TERMINATION = TRUE)
        /* determine cost bound for the next iteration */
        cb = MIN { nextcb[k] / 1 ≤ k ≤ N } ;
        /* k varies over set of processors */
        initialize stack depth,cb and nextcb for the next iteration
```
Since the cost bounds for each iteration of PIDA* are identical to that of IDA*, the first solution found by any processor in PIDA* is an optimal solution. Hence all the processors abort when the first solution is detected by any processor. Due to this it is possible for PIDA* to expand fewer or more nodes than IDA* in the last iteration\(^1\), depending upon when a solution is detected by a processor. Even on different runs for solving the same problem, PIDA* can expand different number of nodes in the last iteration, as the processors run asynchronously. If PIDA* expands fewer nodes than IDA* in the last iteration, then we can observe speedup of greater than \(N\) using \(N\) processors. This phenomenon (of greater than \(N\) speedup on \(N\) processors) is referred to as acceleration anomaly [LAI 83]. In PIDA* at least one processor at any time will be working on a node \(n\) such that everything to the left of \(n\) in the (cost bounded) tree has been searched. Hence discounting overheads due to parallel processing (such as locking, work transfer, termination detection), at any time during the search PIDA* must have searched all parts of the search space that can be searched by IDA* upto that moment plus much more. This guarantees absence of deceleration anomaly (i.e., speedup of less than 1 using \(N>1\) processors) for PIDA*, as PIDA* running on \(N\) processors would never be slower than IDA* for any problem instance.

3.2 Implementation Details.

As illustrated in Fig. 2, PIDA* involves three basic procedures to be executed in each processor: (i) when work is available in the stack, perform bounded DFS; (ii) when no work is available, try to get work from other processors; (iii) when no work can be obtained try to check if termination has occurred. Notice that communication occurs in procedures (ii) and (iii). The objective of our implementation is to see that (i) when work is being exchanged, communication overheads are minimized; (ii) the work is exchanged between processors infrequently; (iii) when no work is available termination is detected

\(^1\) PIDA* expands exactly the same nodes as IDA* upto the last but one iteration, as all these nodes have to be searched by both PIDA* and IDA*. 
quickly. Fig. 3 illustrates the bounded DFS performed by each processor. This differs slightly from bounded DFS performed by IDA* (Fig. 1).

**Fig. 3**

Bounded DFS (startstack, movegen, h)

```c
/* Work is available in the stack and depth, cb, nextcb have been properly initialized */
excdepth[i] = -1;
while ((not solutionfound) and (depth > 0))
{
    if there are no children in the top element of the stack
        POP;
        depth[i] = depth[i] - 1; /* BACKTRACK */
    if (depth < excdepth[i])
        lock stack[i];
        excdepth[i] = depth[i]/2;
        unlock stack[i];
    else
        remove nextson from TOP[i];
    if (nextchild.cost <= cb)
        if nextchild is a solution
            solutionfound = TRUE;
            send abort message to all other processors;
            QUIT;
            PUSH[i] (nextchild, movegen(nextson, h));
            depth[i] = depth[i] + 1; /* ADVANCE */
            excdepth = MAX(deep[i]/2, excdepth[i]);
        else
            nextcb[i] = MIN(nextcb[i], nextson.cost);
}
```

**End of Fig. 3**

To minimize the overhead involved in transferring work from one processor to another, we associate a variable excdepth[i] with the stack of processor i. The processor i which works on stack[i] permits other processors to take work only from below excdepth[i]. (We follow the convention that the stack grows upwards). The stack above excdepth[i] is completely its own and the processor works uninterrupted as long as it is in this region. It can increment excdepth[i] at will, but can decrement it only under
mutual exclusion. Access to regions under excdepth[i] needs mutual exclusion among all processors. A deeper analysis of the program in Fig 3 shows that this scheme gives almost unrestrained access to each processor for its own stack.

The rationale behind keeping excdepth = depth/2 is to see that only a fraction of the work is locked up at any time by processor i. In a random tree with branching factor b, if the stack i has depth d then the fraction of work exclusively available to processor i is \(1/(b^{d/2})\), which is quite small. But this work is big enough for one processor so that it can keep working for a reasonable amount of time before locking the whole stack again. This ensures that work is exchanged between processors infrequently.

The procedure GETWORK describes the exact pattern of exchange of work between processors. The processors of the system are conceptualized to form a ring. Each processor maintains a number named target, the processor from which it is going to demand work next time. Starting at target, GETWORK tries to get work from next few processors in a round robin fashion. If no work is found after a fixed number of retries, FAIL is returned.

Fig. 4

GETWOTRK()
{
  for (j = 0; j < NUMRETRY; j++)
  {
    increment target;
    if (work is available at target below excdepth[target])
      lock stack[target];
    pick work from target.
    unlock stack[target];
    return (SUCCESS);
  }
  return (FAIL);
}
{ End of Fig. 4 }

The termination algorithm is the Ring termination algorithm of Dijkstra [DIJKSTRA 83]. This algorithm suits our implementation very well and it is very efficient.
Due to lack of space, we omit the exact details of the algorithm here.

4. PERFORMANCE RESULTS.

We implemented PIDA* to solve the 15-puzzle problem on Sequent Balance 8000, a shared memory parallel processor. We ran our algorithm on all the thirteen problem instances given in Korf's paper [KORF 85] for which the number of nodes expanded is less than two million\(^2\). Each problem was solved using IDA* on one processor, and using PIDA* on 9, 6 and 3 processors. As explained in the previous section, for the same problem instance, PIDA* can expand different number of nodes in the last iteration on different runs. Hence PIDA* was run 20 times in each case and the speedup was averaged over 20 runs.

The speedup results vary from one problem instance to another problem instance. For the 9 processor case, the average speedup for different problem instances ranged from 3.46 to 16.27. The average speedup over all the instances was 9.24 for 9 processors, 6.56 for 6 processors and 3.16 for 3 processors (Fig. 5). See Table 1 for all the results. Even though for the 13 problems we tried, the average speedup is superlinear (i.e., larger than N for N), in general we expect the avg speedup to be sublinear. This follows from our belief that PIDA* would not in general expand fewer nodes than IDA* (otherwise the time sliced version of PIDA* running on a single processor would in general perform better than IDA*). Our results so far show that PIDA* does not appear to expand any more nodes than IDA* either. Note that the sample of problems we used in our experiment is unbiased (Korf generated these instances randomly). Hence in general we can expect the speedup to be close to linear.

To study the speedup of parallel approach in the absence of anomaly, we modified IDA* and PIDA* (into AIDA* and APIDA*) to find all optimal solutions. This ensures that the search continues for all the search space within the costbound of the final

\(^2\) Two million nodes was chosen as the cutoff, as the larger problems take quite a lot of CPU time. Besides, we were still able to get 13 problems, which is a reasonably large sample size.
iteration in both AIDA* and APIDA*; hence both AIDA* and APIDA* explore exactly
the same number of nodes. In this case the speedup of APIDA* is quite consistently
close to N for N processors for every problem instance (Fig. 6). For 9 processors the
average speedup is 8.36 (max. 8.52, min. 8.1), for 6 processors it is 5.65 (max. 5.7, min.
5.58) and for 3 processors it is 2.82 (max. 2.85, min. 2.8). This shows that our scheme of
splitting work among different processors is quite effective. The speedup is slightly less
than N, because of overheads introduced by distribution of work, termination detection
etc..

5. CONCLUDING REMARKS.

We have presented a parallel implementation of the Iterative-Deepening-A* algo-

rithm. The scheme is quite attractive for the following reasons.

It retains all the advantages of sequential IDA*, i.e., it is admissible, and still has a
storage requirement linear in the depth of the solution. Since parallel processors of
PIDA* expand only those nodes that are also to be expanded by IDA*, conceptually (i.e.,
discounting overheads due to parallel processing,) speedup should be linear. Further-
more the scheme has very little overhead. This is clear from the results obtained for the
all solution case. In the all solution case, both sequential and parallel algorithms expand
exactly the same number of nodes; hence any reduction in speedup for N (> 1) proces-
sors is due to the overheads of parallel processing (locking, work transfer, termination
detection, etc.). Since this reduction is small (speedup is very close to N for N up to 9),
we can be confident that the overhead of our parallel processing scheme is very low. The
effect of this overhead should come down further if PIDA* is used to solve a problem
(e.g. the Traveling Salesman Problem) in which node expansions are more expensive.
Even on 15-puzzle, for which node expansion is a rather trivial operation, the speedup
shows no sign of degradation up to 9 processors. We speculate that speedup as high as 20
or 30 could be obtained for 15-puzzle. For large grain problems (such as TSP) the
speedup could be much more.

Even though we implemented PIDA* on Sequent Balance 8000 (which is a shared
memory machine with processors and memory connected to a high speed bus), we should
be able to run the same algorithm on different parallel processors such as BBN's Butterfly, Hypercube [SEITZ 85] and FAIM-1 [DAVIS 85]. This is significant because the shared bus in multiprocessors such as Sequent Balance tend to saturate beyond 20 or 30 processors. Loosely-coupled architectures such as Hypercube make the interprocess communication more costly, but can be built for a very large number of processors. Since interprocess communication is rather infrequent in our parallel implementation, it appears equally suited for architectures such as Hypercube.
<table>
<thead>
<tr>
<th>prob</th>
<th>9 processors</th>
<th>6 processors</th>
<th>3 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>speedup</td>
<td>standard deviation</td>
<td>speedup</td>
</tr>
<tr>
<td>p1</td>
<td>3.46</td>
<td>0.2</td>
<td>2.38</td>
</tr>
<tr>
<td>p2</td>
<td>13.28</td>
<td>2.02</td>
<td>9.39</td>
</tr>
<tr>
<td>p3</td>
<td>9.69</td>
<td>1.99</td>
<td>6.9</td>
</tr>
<tr>
<td>p4</td>
<td>16.27</td>
<td>8.23</td>
<td>17.3</td>
</tr>
<tr>
<td>p5</td>
<td>6.02</td>
<td>0.38</td>
<td>4.28</td>
</tr>
<tr>
<td>p6</td>
<td>7.48</td>
<td>0.05</td>
<td>5.26</td>
</tr>
<tr>
<td>p7</td>
<td>4.66</td>
<td>0.42</td>
<td>3.86</td>
</tr>
<tr>
<td>p8</td>
<td>12.96</td>
<td>3.45</td>
<td>8.32</td>
</tr>
<tr>
<td>p9</td>
<td>16.6</td>
<td>4.03</td>
<td>6</td>
</tr>
<tr>
<td>p10</td>
<td>5.86</td>
<td>0.55</td>
<td>3.77</td>
</tr>
<tr>
<td>p11</td>
<td>8.52</td>
<td>0.3</td>
<td>7.07</td>
</tr>
<tr>
<td>p12</td>
<td>5.1</td>
<td>0.59</td>
<td>3.18</td>
</tr>
<tr>
<td>p13</td>
<td>10.56</td>
<td>2.06</td>
<td>7.66</td>
</tr>
<tr>
<td>Avg</td>
<td>9.24</td>
<td>6.56</td>
<td>3.16</td>
</tr>
</tbody>
</table>

Table 1: Speedup results for PIDA* on Sequent Balance 8000
for finding one optimal solution for the 15 puzzle
Fig 5: Number of processors vs Avg speedup for PIDA*
Fig 6: Number of processors vs Avg speedup for APIDA*(all solution case)
REFERENCES


