Dynamic Splitting of Decision Trees

S. Farrage
and
T.A. Marsland

Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

E-Mail: tony@CS.UAlberta.CA
Fax: (403)-492-1071

Technical Report
TR93.03
March 1993
(with minor changes, 22 April 1993)

Dynamic Splitting of Decision Trees

S. Farrage
and
T.A. Marsland

ABSTRACT

There are several ways to search decision trees (one and two-person game trees) in parallel, from simple splitting at the root and Principal Variation Splitting, to Baudet's use of aspiration windows. These static schemes are simple and effective, but dynamic methods like Feldmann's "young brothers wait", Hyatt's Dynamic Tree Splitting, and Schaeffer's Distributed Search, though more complex, are even better. In two-person game trees some splitting methods assume that the minimal game tree is being traversed, and so split at the expected ALL nodes (well defined nodes where all successors must be examined). In the search of typical game trees, these ALL nodes are not so easily found.

Here we consider a simple dynamic splitting scheme, which balances the work across the processors without being redundant and without excessive duplication. This report describes a method to curtail excessive searching by simply dividing in half the remaining work along the current solution path, and giving it to another processor. We study a method developed for Parallel IDA* and test a variation of it in a single agent game, hence providing data from a working dynamic work distribution method. We also provide insights into issues that must be considered for an equivalent implementation in two-person games.

1. Introduction

Here we describe a Parallel Depth-First Search method for traversing single agent game trees. It is implemented under the Network Multi-Processor Package (NMP) [2] — a set of subroutines that simplify the use of UNIX sockets and the creation of processes on other machines.

A common parallelization of depth-first search divides the search space into pieces (subtrees) that can be evaluated by different processors. This splitting may be *static*, the division is specified prior to startup, or *dynamic*, so that parts of a processor's work load may be re-allocated at various times throughout the search.

The size and location of each processor's work load is an important factor in determining the efficiency of the parallelization. The major overheads are:

- work distribution: communication cost of distributing work or receiving results.
- synchronization: waiting for other processors to complete work, before being able to continue.
- information deficiency: a lack of data causing extra, unnecessary searching.
- maintenance: cost of extra data structures to support parallelism.

These overheads are not usually independent, since reducing one may increase others.

1.1. Parallel Methods for Decision Trees

In static partitioning, the strategy for dividing the tree is fixed, e.g. the tree may be split at the origin root and the pieces allocated to all processors. This approach usually results in poor speedups because it is difficult to predict in advance how many nodes each processor will search. The branching factor and cutoff rate may vary greatly in different parts of the tree. Processors that receive smaller subtrees soon finish searching, then remain idle, often for a long time, until the others finish. The simplest search method splits the tree at the root (or at some common search depth) so that the subtrees formed can be matched to the available processors. These matches are rarely perfect, leading to unbalanced work. Other classic methods include Baudet's use of aspiration windows [1], and the popular Principal Variation Splitting Method [11]. This latter static scheme led to the development of better but more complex, dynamic work distribution methods.

Dynamic distribution ensures that busy processors decide which work to assign to idle ones. With the dynamic case, it is often possible to make better selections of the nodes to transfer, because more is known about the structure of the tree as the search progresses. This information can lead to the distribution of work loads that are of more equal size, and hence yield better load balancing. It can also reduce the extra search caused by information deficiency, by delaying the transfer of nodes that are unlikely to be searched by the sequential algorithm. At least three independent dynamic schemes have been developed for searching game trees, including Hyatt *et al.*'s DTS (Dynamic Tree Splitting) system [7], Schaeffer's Distributed Search [14], and the use of Feldmann *et al.*'s "young brothers wait" idea [5]. Because these methods were developed for completely different computing systems they are not easily compared, even though they were applied to the same complex computer chess application. Here our intention is to study another simple dynamic parallel method for a single-agent search, with an eye to evolving it into an equally simple two-person game tree search method.

One strategy for dynamic distribution requires that a designated processor (called the *master*) operate as in a sequential program, except that it maintains a list of subtrees that can be searched in parallel by other processors. In a *synchronous* algorithm, the master splits into several *slave* tasks after enough subtrees have been generated. Each slave—one of which could run on the same processor as the master—would search one subtree. As the slaves finish, the master combines their results and prepares for another division of work. In an *asynchronous* algorithm, subtrees can be sent to (and results received from) slave processors at any time. In both synchronous and asynchronous (and even static) methods, there may be multiple levels of master-slave relationships. For example, in the asynchronous case subtasks may split into smaller subtasks while the master continues searching; but in synchronous methods the master waits for the slaves to complete their work.

In coarse-grained systems, asynchronous methods are usually the more efficient because idle processors can receive work at any time; they do not need to wait for sibling tasks to finish. In these schemes, the decision of when to send work may be made by busy processors or by idle ones that are looking for work. In the former case, a busy processor decides that certain nodes should be searched concurrently and simply gives them to other processors. In the latter case, a processor sends work only after receiving a request from an idle processor. This use of idle processors is clearly more efficient when there is significant work to be given away.

Master-slave relationships can be made even more flexible: an idle processor can request work from *any* other processor, thus becoming its slave. The master-slave relationship ends when the slave finishes the work and returns the results to its master. In some domains slaves do not need to return results to their masters, so the master-slave relationship ceases after work is transferred. This freedom increases the likelihood that an idle processor will be able to find work. In centralized schemes, idle slaves cannot obtain work directly from a master that is itself idle, whereas in a decentralized scheme any idle processor can request work. The lack of centralization makes these algorithms appropriate for extremely large systems, because work distribution is not left to an overloaded single processor.

1.2. Previous Results for Single-Agent Games

Here we focus on asynchronous algorithms with decentralized work distribution, where idle processors actively try to find work from all other processors. Several people have developed such algorithms for a variety of search domains. Rao, Kumar and Ramesh [13] implemented a parallel IDA* program and tested it on the 15-puzzle†, using a Sequent Balance 21000, a MIMD machine with memory that is shared among all processors. In their algorithm, one processor initially receives the entire search tree; all others are idle. Kumar and Rao [9] implemented the same parallel IDA* algorithm on a BBN Butterfly, an Intel iPSC hypercube, and 1-ring and 2-ring configurations of processors. In the BBN Butterfly, each processor has its own memory, but can access other processors' memories through a fast switch, thus giving the appearance of shared memory. In the Intel iPSC hypercube, each processor has its own memory, which is inaccessible to other processors.

Kumar and Rao's results [9] show that performance decreases as communication costs increase and connectivity decreases. Their speedups for the Balance and Butterfly computers were almost linear (106 for 115 processors) even in large configurations. Their speedups for the iPSC are still good (115 for 128 processors), but lower than for the Sequent and Butterfly because the iPSC, being distributed, has higher communication costs. The iPSC's reduced connectivity affects performance. Kumar and Rao's speedups for the ring configurations were poor, because the restricted communication paths did not allow work to be distributed efficiently.

Finkel and Manber [6] implemented DIB, a library package that can be used for distributed (general) depth-first search, without having to worry about the extra programming effort required for parallelism. DIB handles process creation, communication, work distribution, and fault tolerance. The user supplies the initial state, a successor-generating function, and a procedure/subroutine to print the results. In DIB, work is initially distributed among the processors. When a processor becomes idle, it requests work from another processor. When a processor receives a request, it sends out work, if any is available. In addition, if a processor cannot find work, it tackles incomplete work that it had previously transferred, providing *fault tolerance* when a processor malfunctions or during network delays. The redundant work has low priority; It is aborted should the

† See Appendix A for a brief description of the 15-puzzle.

results arrive, and is suspended when new non-redundant work is received from another processor. Unlike in Kumar and Rao's program [9], each processor in DIB keeps track of where it sends work and from where it receives work, so that any results can be returned to the appropriate processor.

Finkel and Manber tested DIB on a variety of problems, using distributed systems containing up to 20 VAX 11/750 computers. For the exhaustive search of the 11-queens†† problem the efficiency is almost 100% (even for 16 processors) [6, page 245]. For the traveling salesman problem [10] there was a wider variation, and speedup anomalies were seen. The figures for individual processors [6, page 247] showed that work was balanced, in that each processor spent almost exactly the same amount of time doing useful work. These results are consistent with our own, as presented later. For applications using alpha-beta pruning, information deficiency is a problem, and the speedups are therefore lower (less than 3 for 8 processors) [6, page 248]. Finkel and Manber also discuss alterations [6, page 250], which might improve DIB's performance for more complicated search problems.

It is important to note, however, that information deficiency does not occur in exhaustive searches such as IDA* for the 15-puzzle, and in the *N*-queens problem. In more complex domains, there are tradeoffs between information deficiency and communication and synchronization overheads, and thus the speedups should be lower. In the alpha-beta algorithm, for example, if processors simply send away parts of their trees and then forget about them, information deficiency overhead increases substantially. Feldmann *et al.* [5], on the other hand, use an improved work distribution scheme to implement a parallel version of alpha-beta. To increase the effectiveness of the bounds it is necessary for processors to wait until their slaves have returned results, though these waiting processors can obtain more nodes and search them. Information deficiency is further reduced, but not eliminated, by transferring a node only if at least one of its siblings has already been evaluated [5]. Because parallelism is initiated by idle processors the overheads are reduced. The essence of these methods is to reduce overheads by distributing work only when requested by an idle process, and not when processors decide that they have too much work to do.

Kumar and Rao [9, page 21] analyzed the dependence of performance on problem size, communication cost and network connectivity. Their *isoefficiency function* estimates how big a problem has to become to maintain efficiency in multiprocessor systems. Our implementation follows their analysis for the shared-memory model, while using loosely connected computers with only local memory.

2. Depth-First Search in Parallel

Our parallel depth-first search algorithm is presented in a general framework, though the current implementation lacks some of the operations needed in more complex search domains. The algorithm, named PDFS, uses an asynchronous, decentralized work distribution scheme, and employs N processors, numbered 0 to $N - 1$, each executing the same program. The *root* or *master* processor (processor 0), receives the initial input data,

†† In the *N*-queens problem, the goal is to place *N* queens on an *N* times *N* chessboard in a position in which no two queens attack each other.

which it sends to the others, and also arranges for orderly termination when the search ends. Although not essential, in our case any pair of processors can communicate with each other directly over a network.

All processors run the same program for the duration of the search, and are designated *busy* when searching nodes, and *idle* when not. Initially, the master processor is assigned the entire tree, and is therefore busy, while the others (*slave* processors) are idle. Busy processors search as a sequential program would, except that they can be interrupted by messages from idle ones. Whenever a processor is idle, whether it starts out idle or becomes idle after finishing its work, it instantly tries to obtain more work. This technique is in contrast to many static algorithms, where idle processors simply wait for the master to assign more work.

A unit of work consists of one or more subtrees whose roots are successors of nodes in the current search path. ‘‘Transferring a node’’ means transferring the whole subtree rooted at that node. This strategy does not require large, complicated data structures — only a stack, whose additional memory requirements are linear in the search depth — and so leads to a simple implementation. In addition, PDFS includes a simple mechanism to control the sizes and locations of transferred subtrees, since these factors can affect various overheads substantially. Only subtrees whose height (remaining length to be searched) is greater than a specified limit are selected for transfer. Despite the apparent simplicity, our experiments show that more elaborate work distribution methods are unnecessary, at least for simple problem domains.

To obtain work, an idle processor sends a request to another processor. If the recipient of this message is busy, its searching is interrupted so that it can seek nodes to transfer to the requester. Available nodes are sent and marked as no longer needing search, and the busy processor continues its search where it had left off. Idle processors receiving requests for work, on the other hand, immediately reject them. When the requester receives work, it becomes busy and begins searching. If it receives a rejection message instead of work, it seeks work elsewhere. Idle processors continue to request work from all others, one at a time in a round-robin fashion, until either work arrives or all processors are idle because the search has finished.

The search ends when all processors become idle, that is, the problem is solved. Except for handling extra operations to do with the initial allocation of the entire tree, and some synchronization at the end, the master processor is identical to the slaves. When it becomes idle, the master looks for work, just like any other processor. Alternatively, the master could do only these extra operations, while a separate slave process, running on the same processor, could do the searching. This uniformity among processors eliminates the possibility of any one becoming a bottleneck. Except initially, the master is no more likely to receive requests than is any other processor.

Because the overheads at startup and termination occur only once, regardless of the size of the problem, these extra duties have little effect on the speedup, unless the problem size is too small relative to how many processors are used. The startup overhead could be reduced further by initially partitioning the tree, assigning a few nodes to each processor, but in some pruned search applications this splitting leads to significant nonproductive work, because the processors lack knowledge about bounds on the search.

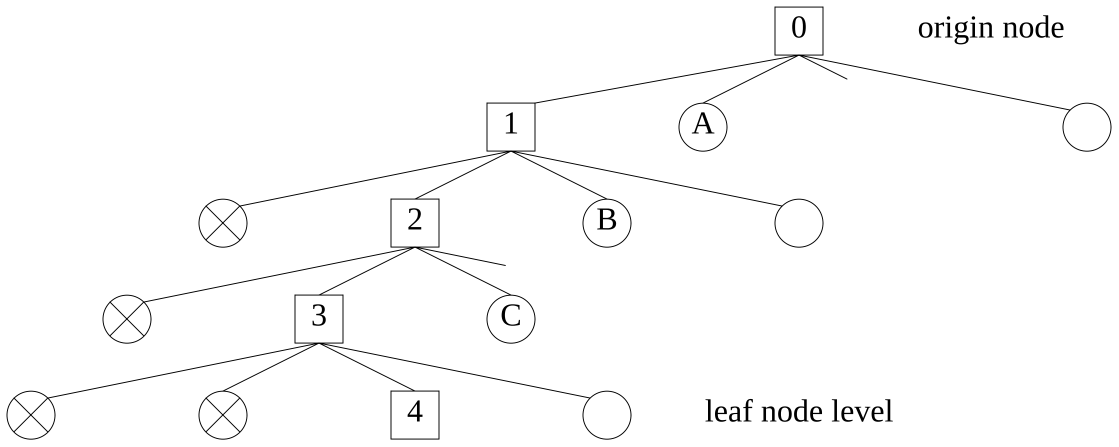


Figure 1: Search tree of sending processor (before transferring nodes)

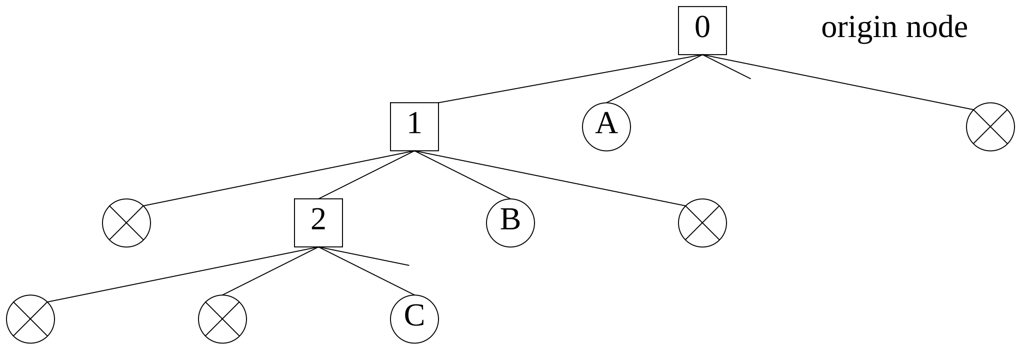


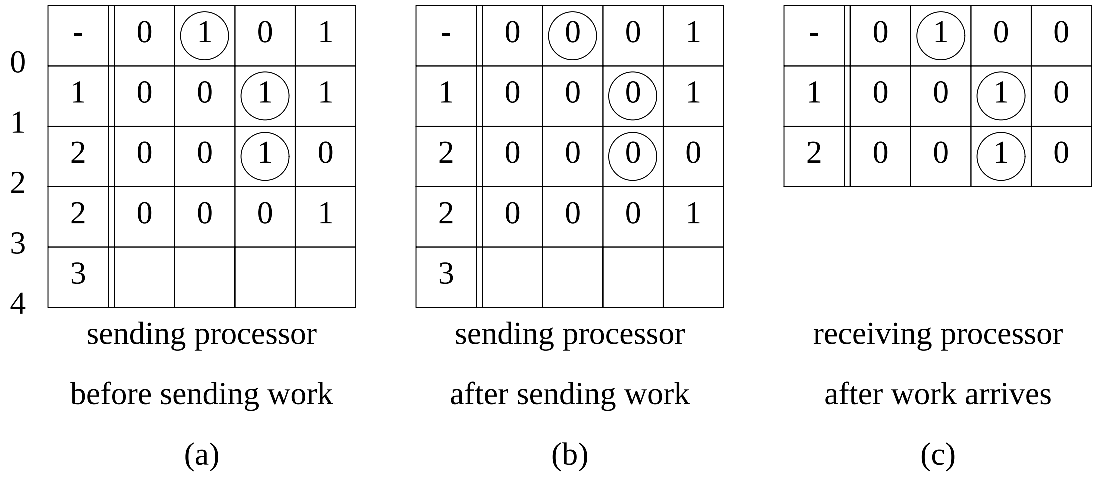
Figure 2: Search tree of receiving processor (after receiving nodes)

2.1. Subtree Selection

Figure 1 shows an example 4-ply tree in which one processor is about to send three subtrees, rooted at the nodes marked A, B and C, to another processor. Figure 2 shows the other processor's tree after receiving these nodes. In both diagrams, the squares represent nodes in the current search path, the crossed-out circles represent roots of subtrees that have been searched or have been allocated to another processor, and the other circles represent nodes (roots of subtrees) that still need to be searched. In this example, each node has at most four successors; the lines that do not lead to circles represent invalid operations (such as retracting the move just made in the 15-puzzle, or exposing a king to check in chess). After transferring nodes A, B and C, the sending processor marks them as unavailable (crossed-out circles).

In a recursive routine, a processor normally accesses only the local variables corresponding to its current invocation. However, when looking for work to distribute, a processor must examine data that corresponds to local variables in other invocations of the recursive subroutine. To allow access to data from those other invocations, it is necessary to maintain extra data structures. PDFS uses an explicit stack, which contains all sharable information. Maintaining this stack increases the node expansion cost because extra operations must be performed whenever entering and exiting the subroutine (i.e. for every node expansion). For example, every time a node is expanded, the parallel program must check and update a bit-mask in the stack that identifies the successors that are available to be transferred. However, these overheads can be reduced substantially by controlling the size of the subtree transferred, as discussed later in Section 2.3 and more completely in Farrage's thesis [4].

To access information about all nodes in its current search path, each processor maintains an explicit stack, whose entries correspond to nodes at depths from 0 to some maximum depth (*MAXDEPTH*) along the current search path. Entries are not needed for greater depths, because work (subtrees) associated with them will not be transferred. Thus for the shortest subtrees the normal data structure for supporting the sequential algorithms can be used, thus eliminating the extra stack overhead for all the nodes beyond *MAXDEPTH* (i.e., nearer the frontier of the search). Some lower limit on the size of the subtree transferred is necessary because the resulting work load must be large compared to the cost of transferring work. Hence the need for the *MAXDEPTH* parameter.



0	-	0	$\textcircled{1}$	0	1
1	1	0	0	$\textcircled{1}$	1
2	2	0	0	$\textcircled{1}$	0
3	2	0	0	0	1
4	3				

sending processor

before sending work

(a)

-	0	$\textcircled{0}$	0	1
1	0	0	$\textcircled{0}$	1
2	0	0	$\textcircled{0}$	0
2	0	0	0	1
3				

sending processor

after sending work

(b)

-	0	$\textcircled{1}$	0	0
1	0	0	$\textcircled{1}$	0
2	0	0	$\textcircled{1}$	0

receiving processor

after work arrives

(c)

Figure 3: Stacks before and after work-transfer

Figure 3 shows three stacks, which correspond to the work transfer shown in Figures 1 and 2. Every row represents a stack entry. The entry in row *k*, as labelled by the numbers on the left, corresponds to the node at depth *k* in the search path. Each stack entry contains two fields: the index of the last move made and a bit-mask corresponding to valid successor nodes. The last-move field (leftmost column) specifies which branch was taken in reaching this node; it enables processors to reconstruct the current state from the initial state by making the appropriate sequence of transitions. Clearly this field is not relevant for the root node at depth 0. The bit-mask specifies whether the corresponding successor is available for search or transferral. A zero bit specifies that there is no need to search this successor because:

(a). The corresponding move cannot legally be made, or

(b). The corresponding successor is being (or has been) searched, or

(c). The corresponding successor has been sent to another processor.

A processor only searches or transfers nodes whose corresponding bit-mask is set to 1. At that time the bit-mask entry is cleared to show that the node is no longer available. Also, only nodes whose distance from the root is less than *MAXDEPTH* are transferred.

When work is transferred, the sending processor duplicates its stack and updates the bit-masks in each copy to show exactly which nodes will be available in each case. In the original stack, the bits corresponding to the root of each transferred subtree are cleared (compare the stacks in Figure 3 (a) and (b)). In the duplicate stack, the bits corresponding to those subtrees that must now be searched are set, and all other bits are cleared, as in the stack in Figure 3 (c) shows. This scheme guarantees that each valid successor is searched exactly once. The duplicate stack is then sent to the idle processor, where it becomes the working stack. In Figure 3, the bits corresponding to transferred nodes are circled. Before beginning its search, the idle processor uses the last-move information to obtain its new current state from the initial state, which it received when PDFS was started.

Although our stack structure allows several disjoint subtrees to be sent to another processor in one message, in the experiments described here only one sibling of a node on the current search path is transferred at a time.

2.2. Implementation

One important issue that affects performance is the normal inability to divide the search tree into about equal pieces. In some applications, it is also hard to predict when a split will yield equal work, because pruning in the subtrees can differ substantially. Static distributions commonly lead to wide variation in work load and hence to a state in which all processors are idle while waiting for the last one to finish [11]. Thus dynamic work distribution schemes like the one described here are essential.

2.2.1. Some Work Distribution Strategies

Kumar and Rao [9, page 8] split the tree into approximately equal-sized pieces, by reassigning half of the unsearched successors of each node in the search path (a ‘‘splitting the tree down the middle’’ strategy). In our experiments, PDFS was allowed to give away only a single subtree at any node, although several subtrees can be transferred at once without additional communications, since the entire stack is always sent. However, the structure used in PDFS is sufficiently general that it can also model Kumar and Rao’s approach, as well as many variations of it, since any subset of these unsearched successors may be sent in one operation.

2.2.2. Work Distribution Method

Deciding which nodes to transfer when distributing work is an important issue because poor choices can increase overheads. The size of the re-allocated subtree affects the efficiency of the parallelization. It follows that the time required to search the received nodes should be large compared to the time required for communication. If the subtree is too small, the requesting processor will quickly finish searching it, then request more work, resulting in extra communication overhead. If the subtree is too big, the sending processor would give away most of its nodes, thus soon becoming idle and having to request work itself, as well being unable to fulfill requests that it may receive before more work arrives.

2.3. Subtree Size Control

To provide some control over the size of the subtrees transmitted the parameters *MINDEPTH* and *MAXDEPTH* are used. A subtree is transferred only if its root node depth is not less than *MINDEPTH* and not greater than *MAXDEPTH* (i.e. the subtree's root node must be located between the horizontal lines in Figure 4). There are usually several subtrees, rooted at various depths, that meet this condition. In the domain used here, a subtree rooted closest to the origin should be chosen because it is probably the biggest. PDFS arbitrarily chooses the leftmost of these shallowest-rooted subtrees for transferral, if there is more than one. One aim of our experiments is to determine optimal ranges for the two working parameters *MINDEPTH* and *MAXDEPTH.*

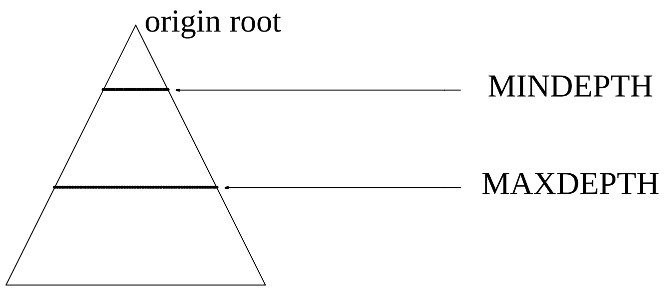


Figure 4: Location of nodes to transfer

As *MAXDEPTH* increases, there will be more frequent transfers of smaller trees, because this limit is now closer to the search frontier. Idle time will also be reduced, because a processor receiving a request for work is more likely to find subtrees to send out. However, a large value of *MAXDEPTH* also increases total communication overhead because smaller subtrees are searched more quickly than larger ones: processors become idle again quickly, resulting in more frequent requests and work transfers. In addition, the transferral of more deeply-rooted subtrees, leads to extra stack operations for those nodes, which in turn increases the cost of node expansion.

Conversely, if *MAXDEPTH* is small, work that could otherwise be distributed is done sequentially while available processors remain idle for long periods. When all nodes less than or equal to *MAXDEPTH* are being worked on, busy processors, which may still have many subtrees to search, are restricted from sending to idle processors subtrees rooted at depth greater then *MAXDEPTH.* Idle processors do nothing except issue requests, which are always rejected because no work is ever available. Similarly, setting *MINDEPTH* too high slows busy processors because they reject requests for work, resulting in an unbalanced work load as evidenced by high idle times for some processors. Not only do these unsatisfied requests slow down the busy processors, but they also show that the work load is highly unbalanced, since at least one processor is searching a large subtree for a long time, while others remain idle.

If *MINDEPTH* is small, a processor might give away a big subtree, and leave little for itself. If *MINDEPTH* is large, all subtrees sent out are forced to be rooted deeper in the tree, thus tending to be smaller, resulting in more communication overhead (messages and interruptions), as is the case with large *MAXDEPTH* values. Our experiments show that the effect of transferring too many nodes at one time is less damaging than the cost of the extra communication overhead, hence small *MINDEPTH* values are preferable.

As often happens in parallel algorithms, there is a tradeoff between overheads: a change that reduces one overhead often increases another. To obtain good speedups, it is necessary to balance these overheads. As our experiments will show, reasonable values of *MINDEPTH* and *MAXDEPTH* yield almost linear speedups in some problem domains, even though the work distribution method is simple.

2.4. Software

The main program (Figure 5) is first run by processor 0, which arranges for the creation of slaves on other processors (also running the same main program, but as a slave). Each processor's work stack is initially emptied. Processor 0 obtains work (the input data) from the user, and then initializes its work stack to show that search of the root node is required. The main `while` loop, which runs for the duration of the search, consists of seeking work (if the processor is idle), searching the subtree it receives, and passing on the termination detection token (see Section 2.5.1) if necessary. The loop ends when the search is finished, in which case the processors synchronize (to allow for an orderly completion), print statistics and exit. In some search domains, it is necessary to combine the results of each processor's search, after which processor 0 outputs this information; in others each processor simply outputs its own results.

Within the `while` loop of Figure 5, the `parallel` procedure is called to search the subtree recursively. This procedure also maintains a bit-mask to identify and avoid successors that have been transferred to other processors. As Figure 6 shows, when the recursion continues beyond *MAXDEPTH*, control switches to `sequential`, which has fewer overheads because it no longer maintains the work stack.

On reaching a frontier node, a busy processor checks to see if any messages are waiting. A message flag is set by an interrupt handler whenever a message arrives. If messages are waiting, the processor temporarily stops searching and calls `poll` to find out where they came from, so it can respond. If a request for work is received, another function is called to examine the stack and find unsearched successors. If an unsearched successor is not found, a rejection message is sent. Otherwise the stack is updated, to show that this successor is no longer available, and it is transferred to the requester. The work allocation function can be modified to implement a wide variety of work distribution methods, some of which may be appropriate for more complex searches.

In some applications (e.g. the 15-puzzle, which is used as an example here), the time spent accessing the stack is a significant fraction of the node expansion time. In such applications, it is important to eliminate the stack management for as many nodes as possible because it increases the amount of time required to search *each node*. To

```
main()
  initialize stack
  done = FALSE

  if (master)
     create processes on other machines
     input initial state
     send initial state and other data to other processors
     indicate that master has white token
     idle = FALSE
  else
     receive initial state and other data from master
     idle = TRUE

  while (not done)
     if (idle)
        if processor has token, pass it on
        seek work
        if (termination detected) done = TRUE
     else
        set current state to initial state
        make moves in stack to create new current state
        parallel(current_state)

  /* all processors are now idle */

  if (master)
     send termination message to other processors
     receive any messages still in transit
  else
     send termination message to other processors
     receive any messages still in transit

  print results
end
```

Figure 5: Main program (all processors)

contain this overhead, nodes shallower than *MAXDEPTH* are expanded by the `parallel` routine (Figure 6) that includes extra operations, such as checking and altering the bit-mask for every node to ensure that subtrees are never searched more than once. For deep nodes `sequential` (Figure 6) executes pure IDA* (the normal sequential algorithm), except that it checks for messages from other processors. These messages could indicate that processors are looking for work or that the problem has been solved.

2.5. Major Overheads

The four main overheads of PDFS are communication, synchronization (work starvation or other idle time), stack maintenance and, in some domains, information deficiency. Processors must communicate to transfer work and to detect termination. Idle processors send messages over a network and wait for replies. Busy processors, when

```
parallel(state)    /* for depth ≤ MAXDEPTH */

  generate bit-mask of valid successors
  store bit-mask at top of stack
  clear bit corresponding to reversal of last move
  for each legal move m
    if (bit m of bit-mask is 1) and (problem not solved)
      clear bit m of bit-mask
      make move m
      increment node count
      if (not at frontier)
        if (depth ≤ MAXDEPTH)
          push m and 0   /* space for bit-mask */
          if (goal_node or parallel(new_state) ≡ SUCCESSFUL)
            pop()
            return (SUCCESSFUL)
        else
          if (goal_node or sequential(new_state) ≡ SUCCESSFUL)
            pop()
            return (SUCCESSFUL)
      else
        if there are any messages waiting
          clear message_flag
          poll()

      undo last move (m)
  pop()
  return (UNSUCCESSFUL)
end

sequential(state)    /* for depth > MAXDEPTH */

  for each legal move m (not including reversing last move)
    make move m
    increment node count
    if (not at frontier)
      if (goal_node or sequential(new_state) ≡ SUCCESSFUL)
        store last move
        return (SUCCESSFUL)
    else
      if there are any messages waiting
        clear message_flag
        poll()

    undo last move (m)
  return (UNSUCCESSFUL)
end
```

Figure 6: Searching functions (all processors)

they receive a message, achieve reasonable response times by interrupting their searching at the next frontier node (limiting depth of search where the distance to the goal state is estimated). If a request for work (the most frequent type of message) is received, the interrupted processor identifies some work that remains to be done and allocates it to the idle processor. Both processors then update their work stacks and proceed.

Communication overhead can be reduced by decreasing the total number of messages, especially requests for work. Increasing the size of transferred work loads also results in fewer total requests, but can cause processors to remain idle because work becomes harder to find. Another way to reduce the number of requests is to restrict the connectivity, thus reducing the number of processors from which a processor can request work or receive a request for work. However, such a limitation would increase idle time when indirectly accessible processors have work to assign, but adjacent processors do not.

When a processor finishes searching its subtree, it seeks work from others, and continues to do so until it either finds work or receives a termination message. This synchronization delay can be reduced by allowing transfer of smaller (deeper-rooted) subtrees, since the busy processor would have more choices of subtrees to re-assign. Unfortunately, sending out smaller subtrees may also result in increased communication, because there will be more requests as processors finish their searching work sooner. For good speedups, a suitable tradeoff between synchronization and communication overheads must be found.

Most processors are idle at the beginning because only one of them has nodes to search. This disadvantage is small, however, because slave processors receive work quickly (if suitable *MAXDEPTH* values are used). Because only the master processor initially has work available, each slave processor sends its first request to the master. Similarly, near the end many processors may be idle, although the finding of a goal node forces the immediate termination of all processors. The work available from the few busy processors is too small to send out profitably because the overheads incurred in handling requests and sending work exceed the savings provided by parallelization. Our experiments show that the idle times at the beginning and the end are short.

In many parallel algorithms, processors lack information that would be available to a single processor executing a sequential algorithm. This lack of information may cause processors to do extra work that the sequential algorithm would have avoided. A common example is the alpha-beta pruning algorithm. When sibling nodes are searched sequentially, the alpha/beta bounds from older siblings increase the number of cutoffs in younger siblings. On the other hand, when several sibling nodes are searched concurrently, the older ones may not be able to supply better bounds for the younger ones early enough to be beneficial. The lack of information (tighter alpha and beta bounds) usually causes the parallel algorithm to miss cutoffs and thus search more nodes, resulting in lower speedups.

2.5.1. Termination Detection

To avoid wasting potential computing power, all processors should stop at the same time. However, in many applications there is no single event that signals completion (see for example, branch-and-bound algorithms and non-final iterations of IDA*); termination occurs only when no more nodes need to be searched. It is therefore necessary to determine when all processors are idle (i.e. no more work is available). Unfortunately, termination detection is not as trivial as it might seem. Simply polling all processors, asking them if they are busy, does not work.

The termination detection method used here is the one by Dijkstra *et al.* [3], in which a ‘‘token’’ is sent around a ring of processors, numbered 0 to $N-1$, in descending order, starting with processor 0. The ‘‘ring’’ is simply a predetermined path that allows the token to visit all the processors. The token and each processor is considered to have two states, called ‘‘white’’ and ‘‘black’’. When the token is first sent, it is considered to be white, as are all processors. However, the token is not passed on until the processor that holds it is idle, and the processor becomes white immediately after. Meanwhile, a processor that sends work to a higher-numbered processor becomes black. A black processor always turns the token black before passing it on; a white processor leaves the token color unchanged. When a white token returns to processor 0, all other processors have finished searching. If a black token arrives at processor 0, the outgoing token is made white. The token may traverse the ring several times before termination is achieved.

Passing the token constitutes an additional overhead: N extra messages, one for each processor, for each traversal of the ring of N processors, plus an interruption and some extra instructions that decide what to do with the token. However, this overhead is small because, although the token may be received at any time, it is not passed on until the processor that holds it becomes idle, which is usually a long time. In practice, our experiments show that the token makes only a few traversals before termination is detected.

2.5.2. Deadlock Prevention

In a parallel program where work requests can be sent to any processor at any time, there is a danger of deadlock. Deadlock occurs if there is a subset of processors in which every processor waits for a reply from another processor in the subset. The simplest example is when two idle processors request work from each other simultaneously and wait for replies from each other. To prevent deadlock, all processors (including idle ones) must respond to requests for work, thus allowing idle processors to seek work elsewhere.

2.5.3. Analysis

For this analysis, we assume that the tree is uniform with constant depth D and fixed width w. We also assume that both the sequential and parallel algorithms search every node in the tree. The analysis concentrates on the overheads of stack management. The analysis for the interrupt flag is similar, but the latter overhead is smaller and depends on the number of leaf nodes.

Let $M = MAXDEPTH$.

Let $n = \sum_{d=0}^{D} w^d = \frac{w^{D+1}-1}{w-1}$ be the total number of nodes searched.

Let $k = \sum_{d=0}^{M} w^d = \frac{w^{M+1}-1}{w-1}$ be the number of nodes for which stack management is necessary (nodes whose distance from the root is not greater than *MAXDEPTH*).

Let N be the number of processors.

Let α be the cost of searching a node.

Let ε be the extra cost (for each node expansion) of managing the stack.

The sequential time is αn. The parallel time is at least $\frac{\alpha n + \varepsilon k}{N}$, resulting in a maximum speedup of $N \frac{\alpha n}{\alpha n + \varepsilon k} = \frac{N}{1 + (\varepsilon k)/(\alpha n)}$. If $k = 0$, corresponding to the case of splitting only at the root node, the possible speedup is N. At the other extreme, if $k = n$, the speedup is limited to $\frac{N}{1 + \varepsilon/\alpha}$. Usually $\varepsilon << \alpha$, but for problems in which the node expansion cost (α) is low, ε might not be significantly less than α. Note , when $\varepsilon = \alpha$ (in addition to $k = n$), then the maximum possible speedup is only $\frac{N}{2}$.

Unless *MAXDEPTH* is close to D, k is much smaller than n and therefore stack management overhead is not important. If w is large and ε is much smaller than α, stack management may be insignificant even if *MAXDEPTH* = $D - 1$. Thus the method should be effective in domains like computer chess where the cost of node expansion is much higher than for stack management.

There are additional considerations to take into account for some problem domains. For instance, in the example discussed in Section 3, there are cutoffs in the deeper parts of the tree, which cause the number of nodes per level to stop increasing exponentially and eventually start decreasing. In this case, increasing *MAXDEPTH* beyond a certain point has little effect.

3. Experiments

To measure the efficiency of the decentralized parallel depth-first search algorithm described in the previous section, PDFS was implemented in a distributed environment. PDFS is written in C and was tested using Sun-4 workstations, all of which use the UNIX operating system and are connected via an Ethernet. PDFS has one process running on each machine, and any pair of machines can communicate with each other.

Processes communicate using the Network Multi-processor Package (NMP) [12][2]. Developed at the University of Alberta, NMP consists of a set of subroutines that provide a simple interface to the UNIX socket handling routines, as well as a server that creates processes on remote machines. NMP can simulate systems with arbitrary connection topologies, although complete connectivity was used in these experiments.

In a sequential algorithm, successor ordering is deterministic; the node count changes only if the successor ordering algorithm is changed. Parallel searches can affect the successor ordering, hence an unpredictable speedup anomaly is possible which influences the results. This possible anomaly was eliminated from this study, because its benefits might overwhelm the losses from communication and synchronization overheads. The resulting confusion over explanations for the timing differences would complicate the analysis of PDFS's efficiency. The search domain chosen is a simple heuristic search that attempts to find a solution to a problem, given its depth and an admissible heuristic. PDFS searches all nodes whose cost plus some heuristic evaluation is less than or equal to a fixed bound. To avoid the possibility of speedup anomaly, a bound that is less than the known solution depth was used, thus forcing the parallel program to search exactly the same set of nodes as the sequential one. Thus we carried out our tests on the next to last iteration of Korf's IDA* [8]. The biggest such bound was used to obtain results for the largest trees.

3.1. Problem Domain

The results reported here were obtained by solving the four most difficult 15-puzzles (see Appendix A) from the 100 random puzzles presented by Korf [8]. The 15-puzzle is not necessarily ideal for testing search algorithms: the search tree has a low branching factor (width = 3) and the cost of expanding nodes is small. Also, the heuristic value of a node is always one more or one less than that of its parent (thus IDA*'s next threshold is always two more than the current one). The low branching factor should not be a problem for PDFS because subtrees that are sent out can be rooted at various depths. In a shallow, bushy tree, there are many nodes in a small range of depths. In a deep, narrow tree, where there are proportionately fewer nodes at individual depths, large differences between *MAXDEPTH* and *MINDEPTH* allows nodes in a greater range of depths to be given away. In both cases, there is plenty of work to send out. Small differences in the depths of reassigned subtrees are overshadowed by variations in the number of cutoffs (threshold being exceeded) among subtrees.

The small node expansion cost is also not a problem because *MAXDEPTH* is used to eliminate the overheads for most of the nodes. However, PDFS would be even more efficient for problems with higher node expansion costs, because any extra overheads would constitute a smaller proportion of the node expansion cost (i.e. the ratio of communication to node expansion cost would decrease).

3.2. Expectations

Kumar and Rao's results [9] suggest that PDFS's biggest overhead should be communication (the times required to send requests for work and receive replies, and the time required for interrupted processors to look for work to send out). In both Kumar and Rao's program and PDFS, there is plenty of work available to transfer except near the end of the search. Unless unnecessary restrictions are placed on the choice of subtrees to transfer, or processors are slow to respond to requests for work, no processor should be idle for long periods. The startup overhead, as well as the idle time at the end, depends only on the number of processors, not on the size of the problem, and therefore becomes insignificant as the problem size increases.

In PDFS, stack management occurs only in the shallow levels of the tree, where only a small fraction of the total nodes reside. Passing the token for termination detection (see Section 2.5.1) is not expensive. In the experiments described here the token never made more than four circuits around the ordered set of processors, provided suitable values of *MAXDEPTH* and *MINDEPTH* were chosen. Also processors only spend time looking for work when they would otherwise be idle, and so the extra synchronization overhead is also small since it occurs only when processors wait for replies to messages. In more complicated problem domains, however, reducing synchronization overhead by reducing the restrictions on which nodes to give away may increase the unnecessary searches caused by information deficiency.

3.3. Timing Difficulties

There are several difficulties that complicate timing and reduce its accuracy. The main problem is the presence of processes not belonging to PDFS, since other users may run applications on any machine at any time. System processes, which may consume more CPU time on some machines than on others, are also present. Because of the way that NMP calculates times, high idle time figures often indicate the presence of such unrelated processes, rather than the amount of time that a machine is truly idle. Experiments whose measurements were corrupted significantly by such processes were redone. Corruption is assumed if the figures for individual processors differ substantially and repeat solutions eliminate the differences. Competing processes also affect the user time figures. Clearly two CPU-intensive processes on the same machine can only receive about half the available user time each, and system time would also increase because of context switching. True idle time is low when reasonable values for *MINDEPTH* and *MAXDEPTH* are chosen.

In addition, the UNIX timing routines are not completely accurate. Differences in the real and user time measurements can be as much as 4% apart, on identical machines running the same (sequential) program with the same input data, and without interference from other users' applications. The differences between consecutive runs on the same machine were much smaller: usually less that 0.1%, but sometimes close to 0.5%.

These difficulties do not reduce PDFS's efficiency. Machines without other processes search more nodes while busier machines, executing unrelated processes, are effectively slower. Thus faster machines search more nodes than slower ones and the transfer of work achieves load balancing. However, because these differences complicate timing only identical machines were used in our experiments, and measurements obviously corrupted by the existence of unrelated processes were redone.

Because of inherent timing differences, it is difficult to obtain precise speedup figures. The choice of machine to use for the sequential algorithm can affect the speedup significantly. For this reason, the sequential program was run several times on different machines. Separate speedup calculations were made, using the fastest and slowest sequential timings from among those not corrupted by the presence of unrelated processes, to obtain upper and lower bounds for the speedup.

3.4. Results

As described earlier, there is a tradeoff between overheads: idle time vs. communication and stack management. To find optimal values for *MINDEPTH* and *MAXDEPTH*, PDFS was run many times, using different values for these parameters. The results of these experiments are presented in the following graphs and in Appendices B and C. PDFS was run on a system of sixteen Sun-4 workstations of identical speed. Machines running other users' programs were avoided, although there may have been some interference from short programs that started during the execution of PDFS. In the first experiment, two of the problems in Korf's IDA* test set [8] were used as input: the biggest (#88) and the fourth-biggest (#66). Here a ‘‘bigger’’ problem is one that requires more nodes to be searched before a solution is found. These problems were chosen because the purpose of parallelism is to solve *large* problems: in small problems, startup and termination overheads constitute a large fraction of the running time. Also, since there is a large difference in the number of nodes (924,074,079 nodes for problem 66 and 5,156,184,395 nodes for problem 88 — a ratio of more than 5.5 to 1), the effects of problem size on efficiency could be investigated. Note, the node count given here is for the penultimate iteration of IDA*.

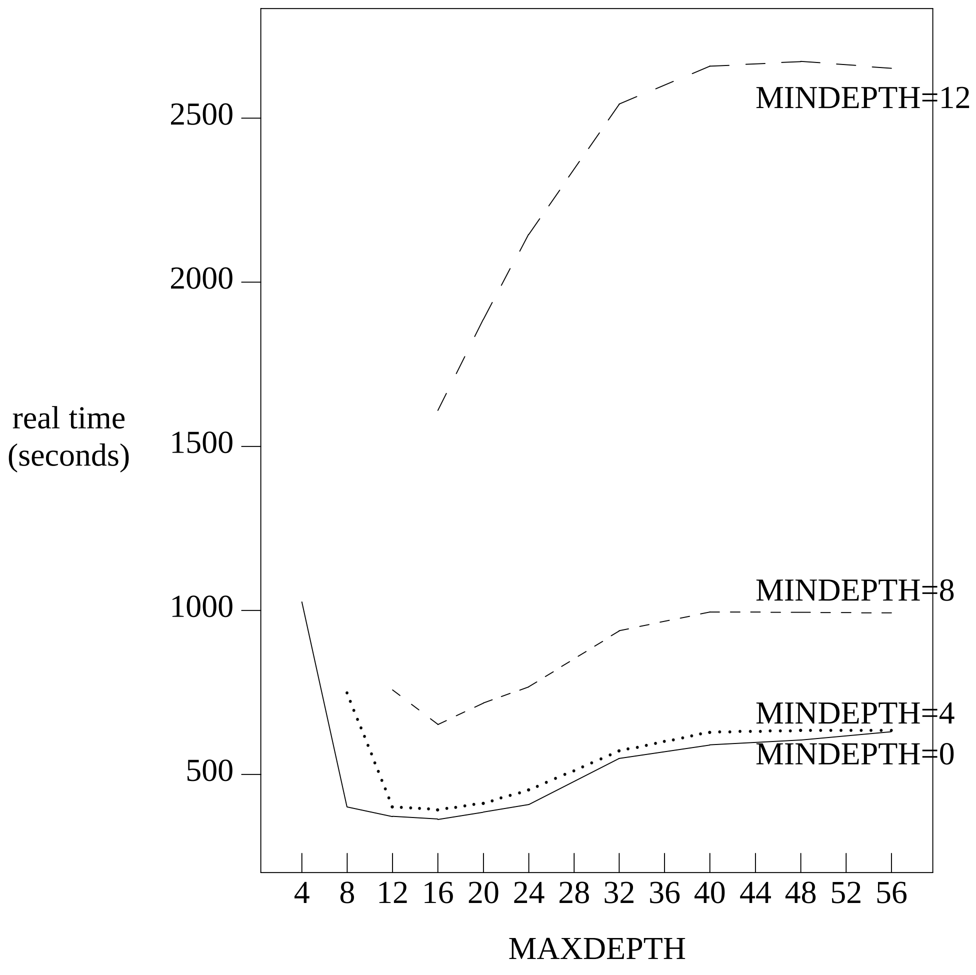


Figure 7: Time as a function of MINDEPTH and MAXDEPTH (problem 66)

Each row in Tables B1 and B2 (Appendix B) displays the statistics for one run with the specified values of *MINDEPTH* and *MAXDEPTH.* The timing figures are averages over all processors. The figures for the node count and the numbers of successful and failed requests for work transfer are totals over all processors. Real time means the elapsed clock time, from start to finish, including wasted time. The user time of a process is the actual CPU time it received. System time is the time spent during system calls, for example to the UNIX `send` and `recv` functions. Idle time, which technically means the time spent doing nothing, also includes the time spent by the CPU on unrelated processes on the same machine; it is calculated by subtracting the total of user and system times from the real time. Wait time measures at least the total time spent looking for work, both sending requests and waiting for replies (the only case of true idle time). Wait time has some user- and system-time components because system calls and other instructions must be executed while looking for work. The figures for real, user, system and idle times include the corresponding components of wait time. The total overhead is at least as large as the total wait time. The number of successful requests is the total times that work was transferred. The last two columns of Table B1 and B2 of Appendix B count how many requests for work were made. Of these some were accepted (success) by the receiving processor, resulting in new work for the sending processor, and the others were rejected (failed) because the receiving processor was either idle itself or because it had no acceptable work to give away. High reject values are specially undesirable. They represent unproductive messages, and occur in great number only for small values of *MAXDEPTH*.

One observation from Tables B1 and B2 is that the system and wait times increase almost linearly with the total work transfer requests. Clearly, each request and each reply requires a certain amount of system time. Wait time also accumulates as more requests are issued. Figures 7 and 8 plot the real (elapsed) times from Tables B1 and B2 and show that there is a range of values for *MAXDEPTH* for which elapsed time is minimal. However, they also show that setting *MINDEPTH* = 0 is optimal for IDA*.

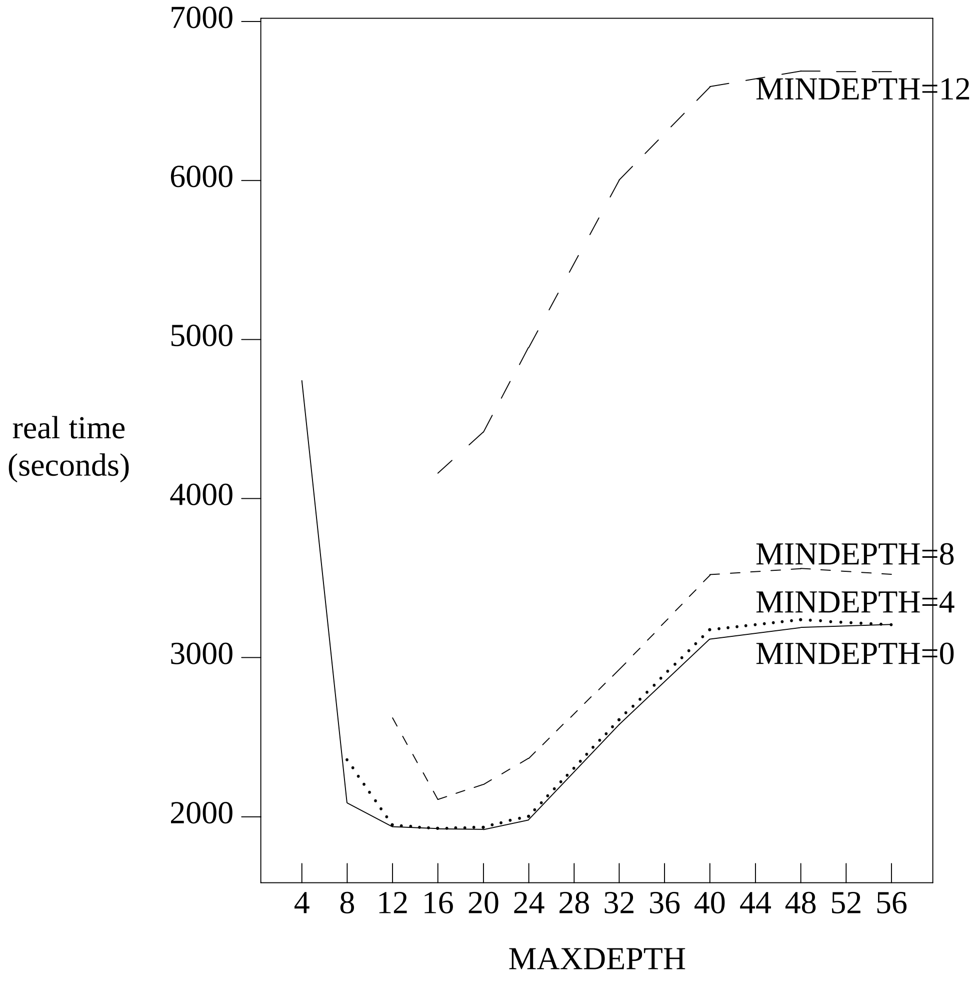


Figure 8: Time as a function of MINDEPTH and MAXDEPTH (problem 88)

3.4.1. Effect of MAXDEPTH

When *MAXDEPTH* falls below its best operating range, processors have trouble finding work, as Tables B1 and B2 show by the frequent work transfer rejections. The reason for this difficulty is that, since there are few nodes at shallow depths, there are few choices of subtrees to distribute. After several requests have been filled by various processors, all subtrees rooted at depths less than *MAXDEPTH* have been or are being searched, thus leaving no further work to transfer. At this point, any processor that completes its subtree remains idle, doing nothing but looking for work for the remainder of the program's execution. The idle time can be substantial, especially when work loads are highly unbalanced. The high system time figures support the conclusion that much of the time is spent sending and replying to requests for work. This overhead increases substantially as *MAXDEPTH* tends to *MINDEPTH* (see Tables B1 and B2 in Appendix B).

However, even small increases in *MAXDEPTH* away from *MINDEPTH* leads to a significant drop in work transfer request rejections. Even as *MAXDEPTH* increases from 12 to 16 to 20, the number of unsuccessful requests drops significantly, indicating that work was still becoming easier to find. Wait time decreased along with the total number of work transfer requests. However, the increase in the number of successful requests suggests that processors had to look for work more often because they received smaller work loads.

Elapsed time also increases as *MAXDEPTH* increases above the operating range. This is perhaps a measure of the stack management cost. The number of nodes for which stack management is necessary increases exponentially as *MAXDEPTH* increases, until cutoffs start occurring. The corresponding increase in user time supports this conclusion, since more work is required to search individual nodes, whereas lack of work would be indicated by an increase in wait time.

As Figures 7 and 8 show, this increase (in real time) levels off as *MAXDEPTH* increases past 40 because the cutoffs at deeper levels reduce the number of additional nodes for which stack management is required. It follows that in these cases relatively few nodes at depths near the threshold value are being searched.

Tables B1 and B2 show that, for $MINDEPTH = 0$, there is little change in the numbers of successful and failed work transfer requests as *MAXDEPTH* increases beyond 20. The timing differences were caused by the extra stack management required for deeper nodes, because the continued increase in user time and the lack of significant change in wait time shows that work was being done.

Figure 9 presents normalized data relating efficiency (minimum user time as a percentage of real time) to depth ratio (*MAXDEPTH* as a percentage of the solution depth). The solution depth for problem #66 is 61 ply and for #88 is 65 ply. We can see that for the larger problem *MAXDEPTH* can fall within a broader range of values and still operate near maximum efficiency. This is a general result applying to a wide spectrum of 15-puzzles, and it also follows that small differences in *MAXDEPTH* are less important for problems with deeper solutions. We can also see that problem #88 is always solved more efficiently than problem #66 over the full range of *MAXDEPTH* values. The main reasons are that idle time at startup and termination is a smaller faction of the total time, and that trees rooted in the distribution range are larger, leading to proportionately fewer requests. In this latter case, since the cost of distributing work is fixed, communication

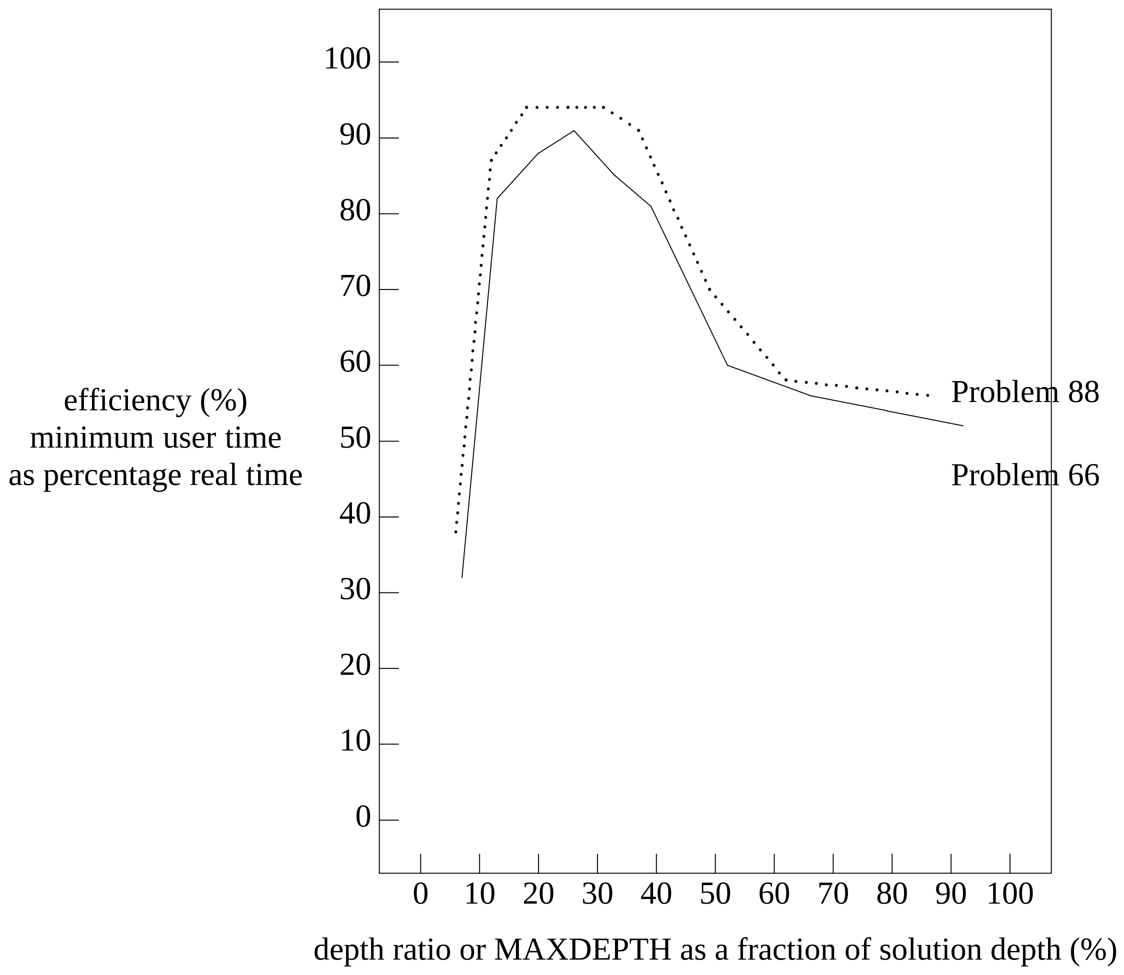


Figure 9: Efficiency as a function of MAXDEPTH and problem size

overheads will occupy a greater proportion of the total time for problem #66. Figure 9 also shows that the optimal value of *MAXDEPTH* for problem 66 is about 25% of the solution depth. For problem 88, the operating range is from 18% to 31%. The range is wider for problem 88 because the optimal solution is deeper: small differences in *MAXDEPTH* are less significant for problems with deeper solutions.

3.4.2. Effect of MINDEPTH

Figure 9 is for the optimal case of *MINDEPTH* = 0, but the graphs for *MINDEPTH* values of 4 and 8 are similar in shape, although the total elapsed times are greater as *MINDEPTH* increases. Figures 7 and 8 also support the view that the elapsed time increases because more messages are sent: since all transferred subtrees are forced to be smaller, processors must seek work more often. The wait time (see Tables B1 and B2) accumulates as more requests are issued, even though processors can find work easily. The degradation in performance is slight when *MINDEPTH* increases from 0 to 4, indicating that the exact choice of value for *MINDEPTH* is unimportant if it is chosen to be between 0 and 4 (and perhaps slightly higher). As *MINDEPTH* increases to 8, the curve shifts upward by a larger amount: close to double for problem 66, although only 15% for problem 88. The difference is smaller for the largest problem, #88, because all subtrees for #88 are proportionally bigger than a similarly rooted subtree for #66, hence lower overheads. The performance degrades rapidly as *MINDEPTH* increases to 12 and beyond.

From these observations, one can conclude that the many failed transfer requests, resulting from the receipt of smaller units of work, are much more of a burden than the risk of processors becoming idle by sending away too much work. These results show that, at least for this simple problem domain, there is no reason to set a minimum depth below which work cannot be sent out.

3.4.3. Significance of the MAXDEPTH–MINDEPTH Difference

Since the range *MAXDEPTH* – *MINDEPTH* controls the area of the tree where nodes can be transferred, Farrage's thesis [4] considers this factor and presents some insights, including the hypothesis that the number of available nodes is linear with the difference *MAXDEPTH* – *MINDEPTH.* To summarize, when the difference is small the wait times are large showing that the processors are having difficulty finding work. Also the system time figures are high, since `send` and `recv` UNIX system calls are issued with each message. Keeping the difference constant and increasing *MINDEPTH* helps keep more processors busy, but doing smaller pieces of work. There can be do doubt that the difference between *MAXDEPTH* and *MINDEPTH* should be as large as possible, but consistent with the best operating range for *MAXDEPTH.*

3.4.4. Summary

In general, requests for work are more likely to be successful as *MAXDEPTH* increases, since the work that can be sent out for a smaller *MAXDEPTH* is a subset of the work that can be sent out for a larger *MAXDEPTH.* The decrease in wait time (with increasing *MAXDEPTH*) also supports this conclusion. Unsuccessful requests result in rejections and repeated requests, which interrupt busy processors and waste their time when they do not have work to send. In addition, each unsuccessful attempt to obtain work increases the amount of time that a processor remains idle. However, larger values of *MAXDEPTH* also increase the stack management overhead. These two conflicting overheads need to be balanced. In the 15-puzzle, this balance occurs when *MINDEPTH* is zero and *MAXDEPTH* is within its optimum range of 15 to 40% of the maximum search depth.

3.4.5. Other Results

Kumar and Rao [9, page 13] found that, using their tree-splitting strategy in a shared-memory system (Sequent Balance 21000), there was little difference in performance for cutoff depths between 25% and 75% of the search depth. They also hypothesized that tree-splitting at the root node is superior to transferring single subtrees [9, page 8].

The experiments described here support the work of Kumar and Rao, who argued that $MINDEPTH = 0$ is the best value, but we show that *MAXDEPTH* is an equally important parameter with best values ranging between 12 and 20 ply, with 8 to 24 also giving reasonably good results. The optimal solution depths for problems 66 and 88 are 61 ply and 65 ply, respectively; and so 8 to 24 translates to about 12% to 36% of the solution depth. The main differences between Kumar and Rao's program and PDFS are that:

- communication costs were lower for their program.
- they used slower processors.
- they split the tree at the root; PDFS can split at any node along the current path.
- their results were obtained on smaller test problems.
- they implemented IDA*, which has extra overheads.

These differences must be considered when comparing their program to PDFS. The first two, and likely also the next two, differences make their results look better; the last two favor PDFS.

Finkel and Manber [6, page 245] also had good speedup results for the 11-queens problem, another simple search domain that PDFS can also be easily tested on.

3.5. Different-Sized Configurations and Problems

Tables C1 to C4 (Appendix C) show the statistics for individual processors in 4, 8, 12, and 16-processor configurations, using Korf's four largest problems (#66, #60, #82 and #88, see Appendix A) as input. As before, the largest bound short of the solution depth was used, so that anomalous speedup gains did not swamp the overhead losses. Any set of *MINDEPTH* and *MAXDEPTH* values in the optimal range would give similar results; 0 and 16 were used here. Problems 60 and 82 are bigger than #66 and smaller than #88. Thus values that work well for both #66 and #88 should work well for these problems also.

Each row in Tables C1 to C4 represents the data for one processor; the first row in each table represents the master processor. Since machines were selected only because they are available (no other applications running on them), corresponding rows of different tables do not necessarily represent the same physical machine (i.e. the machine represented in row k for an n_1-processor configuration is usually not the same as that corresponding to row k in an n_2-processor configuration). Also, the set of machines used in a larger configuration does not necessarily contain all the machines used in a smaller configuration.

Real, user, system, idle and wait times are accounted for the same way as in Tables B1 and B2, except that they now refer to individual processors, instead of averages over all processors. The ‘‘requests sent’’ heading in Tables C1 to C4 of Appendix C is equivalent to the ‘‘work transfers’’ heading in Tables B1 and B2. The accepted requests count is the number of times that a processor sent work in response to a request; the number of rejected requests is the number of times it did not. The total count of the accepted requests is equal to the total successful ones, because each work transfer requires a sender and a receiver. The total numbers of rejected and failed requests differ slightly because processors did not reply to messages that were received after, but sent before, termination was detected.

To summarize, Table 1 contains the elapsed (real) times for the sequential and

Table 1: Summary of times

Times (in seconds) for sequential and parallel programs						
Problem	sequential		parallel			
	fastest	slowest	4 processors	8 processors	12 processors	16 processors
66	5074	5325	1323	674	463	362
60	9873	10299	2573	1306	884	680
82	14913	15655	3875	1952	1331	1012
88	28386	29491	7380	3703	2493	1889

Table 2: Speedups

Speedups								
Problem	4 processors		8 processors		12 processors		16 processors	
	best	worst	best	worst	best	worst	best	worst
66	4.02	3.83	7.90	7.53	11.49	10.95	14.69	14.00
60	4.00	3.84	7.88	7.56	11.64	11.16	15.14	14.51
82	4.04	3.85	8.02	7.64	11.76	11.20	15.47	14.73
88	4.00	3.85	7.96	7.67	11.83	11.39	15.61	15.03

parallel programs for all four problems, and Table 2 shows the corresponding speedups. Because of the differences in the timing measurements among machines, upper and lower bounds for the sequential algorithm's time are included. The lower speedup bound uses the *user* time of the machine with the *shortest* sequential measurement; the upper speedup bound uses the *real* time of the machine with the *longest* sequential measurement. Corrupted experiments were not used for any of these timings. The upper bound can be as much as five percent more than the lower bound. In Table 2, the slower sequential measurements seem to suggest that super-linear speedup is possible; however, these high speedup figures are caused only by the differences in machine speeds. Super-linear speedup cannot occur here because the sequential and parallel algorithms search exactly the same nodes. The true speedup is somewhere between the best and worst shown in Table 2. Perhaps a more accurate estimate of the true speedup can be obtained by running the sequential algorithm on all the machines, then using the average of these times as the sequential measurement.

As before, all machines were of the same type and speed, and were initially idle; any sequential or parallel measurements that were obviously corrupted by other processes that started later were redone. Since this set of experiments is intended to demonstrate the efficiency of PDFS using near-optimal values of *MINDEPTH* and *MAXDEPTH*, rather than to determine these values, stricter criteria were used to determine whether the results of experiments affected by non-PDFS processes should be discarded. The main basis for these criteria is to ensure that the effects of unrelated processes are small compared to the experimental errors in timing.

3.6. Observations

Tables C1 to C4 in Appendix C show that each processor spent negligible (less that 2%) time looking for work. This shows that further improvements caused by changing *MINDEPTH* or *MAXDEPTH* to reduce message frequency will be slight. Processors are busy most of the time because the search tree is large: after receiving a subtree, a processor can continue searching for a long time without having to look for more work. For most of the search, there is ample work for idle processors to find. If the search space is so small that the idle times at startup and termination are significant, there is little benefit in adding more processors, because the application is already over-parallelized. Another observation (see Tables C1 to C4) is that, in each run, each processor searched about the same number of nodes. For the larger configurations, processor 0 usually searches a few more nodes than the others because it starts earlier than the slaves it creates.

On the other hand, Farrage's thesis [4] considers cases when *MINDEPTH* and *MAXDEPTH* are chosen poorly, especially when they are close together. One processor, usually processor 0 because it initially has the entire tree, issues fewer requests and searches many more nodes than the others (the supporting figures are in the thesis [4]). When *MAXDEPTH* is too small one processor will have a large workload that cannot be subdivided. Conversely, for large *MINDEPTH* values, the difference in node counts shows that processor 0 retains most of the tree and only sends small parts of it to other processors, which soon become idle again. There are two solutions to this problem. In one case processors seek work from processor 0 first whenever becoming idle. This solution should work well for centralized algorithms, but the master would become a bottleneck in large configurations. In the other, and more appropriate for decentralized algorithms, larger subtrees are sent out (e.g. by reducing *MINDEPTH*). This distributes work more evenly and almost eliminates any differences between the master and the slaves.

Finally, gaining access to enough idle processors was a problem for us. The main advantage of decentralized algorithms (over centralized ones) is that they are more effective in systems with many processors. Ideally, PDFS should have been tested in a system with hundreds of processors; unfortunately, no more than sixteen identical processors were available. In practice, the processors need not be identical, but in these experiments accurate timings were sought. Even so unexpected difficulties arose. For example, in the 16-processor configurations only, the system time for the machine corresponding to the fifth row is consistently higher than for other machines. Since this machine always displays higher system time figures, even for the sequential program, it follows that either the operating system there was slightly different, or perhaps that some of the memory was disabled, leading to some paging. We retained this machine in our results because the effect of the increased system time is minor.

4. Conclusions

Tables 1 and 2 show that good speedups can be obtained despite the simple work distribution strategy, the low node expansion cost, and the high communication costs. However, near-linear speedups (as Table 2 shows) should not be considered unusual for a simple problem domain like the 15-puzzle. The same set of nodes is expanded regardless of the order in which successors are searched.

In problems with higher node expansion costs, stack management and flag checking become insignificant compared to the other operations involved in node expansion (i.e. the rise in real time shown in Figures 7 and 8 as *MAXDEPTH* increases should be smaller). If the overheads remain constant, speedups should increase because more useful work is done. Thus, *MAXDEPTH* can be increased if the node expansion cost is higher, since the problem is affected less by stack management, and can benefit from the extra freedom in assigning work.

The choice of values to use for *MINDEPTH* and *MAXDEPTH* depends on the type of problem. However, based on the two examples of the 15-puzzle, which differ in size by a factor of more than 5.5, setting *MINDEPTH* to 0 and *MAXDEPTH* to 25% of the maximum search depth (the maximum search depth is usually more easily predetermined than the solution depth) gives good results, at least for exhaustive problems with low node expansion cost. Setting *MINDEPTH* = 0 and *MAXDEPTH* = 16 (reasonably close to 25% of the search depth) gives good performance for both problems, as well as the two intermediate sized ones, as Figure 9 and Tables C1 to C4 show. Different values might be more appropriate for different tree structures (e.g. different width or branching factor, different depth, or no narrowing at deep levels). The range of optimal *MINDEPTH* and *MAXDEPTH* values should be similar for other simple search domains.

More complex work distribution strategies are necessary in domains in which information deficiency can be a problem (e.g. branch-and-bound, including alpha-beta). The extra search effort resulting from information deficiency must be balanced against the other overheads, and thus the speedups should be lower. For shallow, bushy trees (high branching factors), it would be better to transfer multiple subtrees from one node at a time, to ensure that the work transferred is significant compared to that retained by the sending processor. For more complex search domains, such as alpha-beta, information deficiency must also be overcome, by communicating improved solution bounds as they are found.

References

1. G. M. Baudet, On the Branching Factor of the Alpha-beta Pruning Algorithm, *Artificial Intelligence 10(2)*, (1978), 173-199.
2. T. Breitkreutz, S. Sutphen and T. A. Marsland, Developing NMP Applications, Tech. Rep. 89-11, University of Alberta, Department of Computing Science, March 1989.
3. E. W. Dijkstra, W. H. J. Feijen and A. J. M. Gasteren, Derivation of a Termination Detection Algorithm for Distributed Computations, *Information Processing Letters 16*, (1983), 217-219.
4. S. Farrage, Parallel Depth-First Search, M.Sc. Thesis, Computing Science, University of Alberta, Fall, 1991.
5. R. Feldmann, B. Monien, P. Mysliwietz and O. Vornberger, Distributed Game Tree Search, in *Parallel Algorithms for Machine Intelligence and Vision*, V. Kumar, P. S. Gopalakrishnan and L. Kanal (ed.), Springer-Verlag, 1990, 66-101.
6. R. Finkel and U. Manber, DIB—A Distributed Implementation of Backtracking, *ACM Transactions on Programming Languages and Systems 9*, (1987), 235-256.
7. R. M. Hyatt, B. W. Suter and H. L. Nelson, A Parallel Alpha/Beta Tree Searching Algorithm, *Parallel Computing 10(3)*, (1989), 299-308.
8. R. E. Korf, Depth-First Iterative-Deepening: An Optimal Admissible Tree Search, *Artificial Intelligence 27*, (1985), 97-109.
9. V. Kumar and V. N. Rao, Scalable Parallel Formulations of Depth-First Search, in *Parallel Algorithms for Machine Intelligence and Vision*, V. Kumar, P. S. Gopalakrishnan and L. Kanal (ed.), Springer-Verlag, 1990, 1-41.
10. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys, eds., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, 1985.
11. T. A. Marsland and F. Popowich, Parallel Game-Tree Search, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, (1985), 442-452.
12. T. A. Marsland, T. Breitkreutz and S. Sutphen, A Network Multi-Processor for Experiments in Parallelism, *Concurrency: Practice and Experience 3(3)*, (1991), 203-219.
13. V. N. Rao, V. Kumar and K. Ramesh, A Parallel Implementation of Iterative-Deepening A*, *Proc. Sixth National Conference on Artificial Intelligence*, 1987, 178-182.
14. J. Schaeffer, Distributed Game-Tree Searching, *Journal of Parallel and Distributed Computing 6(2)*, (1989), 90-114.

Appendix A: The 15-Puzzle

The problem domain in these experiments is the 15-puzzle, a special case of the $m \times n$ puzzle. It consists of a 4×4 board with tiles numbered from 1 through 15 occupying fifteen of the sixteen squares. A *move* consists of sliding an adjacent tile vertically or horizontally onto the blank square. The aim is to reach the goal state in as few moves as possible. There are initial positions from which the problem cannot be solved (i.e. the graph that represents the 15-puzzle is disconnected); these positions are not used. Figure A1 shows the goal state and the initial states of the four problems used for the experiments in this research, listed here in increasing order of difficulty. A problem's number is the order in which it appears in Korf's paper [8]. Problems 66, 60, 82 and 88 are the four most difficult (most nodes searched by IDA*) of Korf's examples.

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Goal State

11	6	14	12
3	5	1	15
8		10	13
9	7	4	2

Problem 66

11	14	13	1
2	3	12	4
15	7	9	5
10	6	8	

Problem 60

14	10	2	1
13	9	8	11
7	3	6	12
15	5	4	

Problem 82

15	2	12	11
14	13	9	5
1	3	8	7
	10	6	4

Problem 88

Figure A1: The 15-puzzle

IDA* is an effective algorithm for solving the 15-puzzle (A* requires too much memory). The most commonly used heuristic, which is also used in this report, is the *Manhattan distance*. Mathematically, the Manhattan distance between two coordinates (x_1, y_1) and (x_2, y_2) is $|x_2 - x_1| + |y_2 - y_1|$ In the 15-puzzle, this distance is equal to the number of moves that would be required to move a tile from one square to the other, assuming there are no intervening tiles. The heuristic value h is the sum of the Manhattan distances of the current position of each tile to its goal square. Clearly, this heuristic is admissible because the presence of intervening tiles can only increase the number of moves required to transfer a tile into the correct position. The base cost, g, is the number of moves already made, and the expected cost is $f = g + h$.

Appendix B

Table B1: Measured statistics for problem 66

Problem 66, 16 processors, 924074079 nodes searched								
depth limits		average times (seconds)					work transfers	
MIN	MAX	real	user	system	idle	wait	success	failed
0	4	1023	410	576.8	35.3	649.6	32	352010
0	8	397	336	34.8	25.4	39.4	118	23167
0	12	369	333	10.9	24.2	13.1	176	7256
0	16	360	334	2.0	24.1	4.4	394	1100
0	20	382	344	1.2	34.8	4.2	444	523
0	24	405	381	1.6	21.2	5.0	501	480
0	32	545	512	1.8	29.8	5.3	509	539
0	40	587	561	2.2	22.5	6.2	549	737
0	48	597	565	2.0	28.9	6.6	649	501
0	56	628	566	5.6	55.8	6.1	501	509
4	8	744	378	333.8	31.5	375.4	559	223082
4	12	398	336	21.8	39.1	36.8	1837	13654
4	16	387	336	8.3	42.2	29.3	2824	2712
4	20	407	346	6.7	53.2	30.1	3167	784
4	24	443	384	7.1	50.8	32.7	3389	830
4	32	568	516	5.2	46.1	27.6	3087	478
4	40	625	564	7.1	51.9	32.9	3476	674
4	48	630	568	6.2	54.3	30.5	3360	513
4	56	630	568	7.8	52.6	30.6	3240	685
8	12	754	377	296.0	80.2	387.1	8215	169679
8	16	649	352	113.7	182.8	280.4	20074	23014
8	20	715	364	118.1	232.5	334.8	26345	15635
8	24	764	401	118.2	243.7	352.0	28441	12211
8	32	934	534	120.2	278.5	380.9	31871	11573
8	40	992	582	121.6	287.1	384.9	32483	10678
8	48	991	587	122.0	280.4	381.3	31988	11949
8	56	989	587	120.2	280.6	386.1	32696	12093
12	16	1605	444	801.2	359.0	1227.8	48272	318682
12	20	1883	444	695.4	742.6	1488.7	92515	149932
12	24	2142	490	741.4	909.7	1705.9	111803	129379
12	32	2539	625	816.9	1095.9	1971.6	138220	127022
12	40	2655	684	830.2	1139.8	2041.8	142336	125114
12	48	2670	689	821.9	1157.6	2042.3	143485	127527
12	56	2649	687	831.8	1129.4	2025.3	140395	128846

Appendix B

Table B2: Measured statistics for problem 88

Problem 88, 16 processors, 5156184395 nodes searched								
depth limits		average times (seconds)					work transfers	
MIN	MAX	real	user	system	idle	wait	success	failed
0	4	4738	2181	2481.0	74.9	2761.9	22	2062536
0	8	2085	1870	181.2	33.1	199.2	121	169456
0	12	1932	1856	41.0	34.2	46.3	229	32669
0	16	1921	1847	6.0	66.7	5.2	351	2229
0	20	1913	1861	2.3	48.6	2.8	329	559
0	24	1977	1939	2.3	34.9	3.4	410	446
0	32	2577	2536	2.6	36.8	4.5	515	626
0	40	3113	3070	2.7	38.9	5.7	634	500
0	48	3186	3128	3.8	53.4	5.6	613	617
0	56	3204	3144	2.5	56.8	4.4	485	405
4	8	2352	1901	400.4	49.6	435.5	338	394239
4	12	1940	1850	30.3	58.7	39.2	1292	22716
4	16	1920	1847	6.2	65.5	16.4	1938	1941
4	20	1929	1863	4.8	60.9	16.3	2046	1131
4	24	1997	1941	3.9	51.2	18.2	2417	633
4	32	2603	2544	3.6	54.2	15.1	2012	215
4	40	3170	3081	6.8	81.2	16.4	2101	309
4	48	3234	3136	7.1	89.5	17.1	2194	473
4	56	3201	3150	3.6	45.8	18.1	2198	296
8	12	2618	1931	597.1	89.2	685.9	6335	579796
8	16	2105	1859	58.6	186.0	202.4	20569	21157
8	20	2200	1881	72.2	245.6	280.9	27443	6195
8	24	2363	1977	79.2	305.8	312.9	30384	3945
8	32	2923	2569	66.4	286.6	304.5	31415	2980
8	40	3516	3120	75.1	319.8	321.8	33052	2872
8	48	3556	3176	70.5	308.4	314.6	32484	3015
8	56	3518	3175	65.8	276.5	308.0	32106	3036
12	16	4155	2068	1443.5	642.6	2155.3	81008	916403
12	20	4415	2015	951.6	1446.7	2446.1	185591	218963
12	24	4943	2104	964.1	1874.0	2879.9	236223	146042
12	32	6000	2715	1055.2	2228.5	3303.9	273739	131438
12	40	6588	3266	1013.2	2308.1	3344.7	284967	123129
12	48	6685	3320	1038.6	2325.1	3376.9	284578	122250
12	56	6681	3332	1013.6	2334.7	3369.8	286817	123533

Appendix C

Table C1: Problem 66, work done per processor

Problem 66, 924074079 nodes searched with MINDEPTH = 0 and MAXDEPTH = 16									
4 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
229988978	1323	1321	0.5	1.5	0.3	2	5	10	15
230011834	1322	1318	0.6	3.6	1.5	9	9	8	12
232373440	1322	1317	0.9	4.1	2.0	10	18	8	8
231699827	1323	1317	0.7	5.5	1.3	11	12	6	11
8 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
117578361	674	668	0.8	5.2	2.5	16	13	17	35
114032033	671	655	1.1	15.1	2.2	12	16	18	31
115664283	672	663	1.0	8.4	2.7	15	34	18	27
114069190	672	660	1.0	10.5	2.8	18	22	15	31
115368674	672	661	0.9	9.4	2.0	13	48	15	28
114976903	672	660	1.0	10.7	2.7	15	75	11	28
115656337	672	658	1.1	12.7	2.7	16	27	12	34
116728298	672	658	0.8	13.0	2.2	12	10	11	36
12 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
79262897	463	459	1.3	3.4	1.6	11	26	24	96
77265462	462	444	2.4	15.7	5.3	25	81	22	91
73232852	462	431	2.6	28.7	4.8	21	119	19	81
76580538	461	436	1.9	23.9	3.1	13	61	16	90
77974676	462	448	1.6	12.7	3.8	22	38	21	96
77440762	463	446	2.1	15.1	4.6	20	317	22	69
76942967	462	443	2.4	16.6	5.0	23	127	19	89
77441043	463	442	1.9	18.6	4.9	23	83	19	92
76802343	462	440	2.2	20.8	4.7	20	70	19	93
77410724	463	440	2.1	20.4	4.1	20	42	19	94
76825026	461	440	2.3	18.9	3.9	13	82	18	93
76894789	462	437	0.8	24.2	2.7	24	20	17	91
16 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
61352185	362	350	3.5	8.6	6.1	25	105	28	94
59553372	361	343	2.5	15.2	4.7	20	68	26	92
54831234	361	325	2.4	33.2	3.8	14	83	19	97
56580250	357	326	2.7	28.9	5.1	18	109	18	95
55047180	361	310	13.8	36.4	4.5	22	83	15	98
55650858	360	318	2.8	39.6	5.5	23	80	15	98
55748694	360	318	3.0	39.7	5.4	20	130	17	92
55659127	360	318	3.1	39.2	5.0	18	114	15	93
60345892	361	347	2.7	11.0	4.7	20	160	26	86
59234603	361	343	2.1	16.0	5.5	28	76	24	93
59856536	361	342	1.6	16.7	3.7	24	35	24	96
58468162	361	337	2.5	20.9	4.4	18	71	24	94
58009657	361	332	2.4	26.0	4.3	22	68	20	98
57863677	361	332	2.8	26.3	5.3	25	95	19	95
57900530	360	332	2.2	26.3	4.1	16	44	22	96
57972122	359	332	2.5	25.0	4.7	19	176	20	89

Appendix C

Table C2: Problem 60, work done per processor

Problem 60, 1784841519 nodes searched with MINDEPTH = 0 and MAXDEPTH = 16									
4 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
442883040	2573	2563	3.9	6.4	4.7	9	454	8	453
449306582	2572	2563	4.3	4.4	5.1	6	484	7	444
445188201	2572	2563	1.4	7.4	1.2	10	2	6	603
447463696	2572	2557	5.2	9.6	5.8	4	871	8	313
8 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
226230804	1306	1293	5.2	7.9	6.3	10	762	13	258
221699184	1305	1273	4.5	27.4	5.2	9	376	15	310
221305742	1306	1267	5.5	33.2	1.1	7	13	13	363
224045931	1305	1286	4.7	13.9	6.3	12	445	11	302
223479428	1305	1286	4.8	14.7	7.5	21	271	9	327
222299041	1305	1286	4.0	15.5	4.5	8	201	11	337
223380879	1305	1285	3.8	15.8	5.5	15	369	9	313
222400510	1306	1285	5.2	15.5	5.8	8	118	9	347
12 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
151875234	884	875	1.7	7.3	2.0	7	43	17	146
148601032	883	858	1.4	24.4	1.8	9	29	17	144
144980597	877	839	2.9	35.5	4.8	20	57	7	142
147704534	882	847	2.8	31.6	3.9	10	121	7	138
152206076	884	865	2.1	17.0	3.3	9	79	17	139
149806809	884	862	3.1	19.4	4.9	16	140	15	135
149500642	884	860	2.1	21.7	4.3	16	77	15	140
150184519	883	859	3.0	20.4	4.8	13	328	13	120
147845406	884	858	3.3	22.3	4.9	13	235	16	121
149381487	884	856	1.9	25.8	3.5	15	82	14	135
146867793	884	852	2.3	29.7	4.3	22	94	14	134
145887390	883	847	2.9	33.6	4.8	14	322	12	116
16 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
115740209	680	673	2.5	4.5	3.3	8	101	26	71
112811463	677	655	2.6	19.6	4.6	18	74	26	71
107179733	679	629	3.5	46.6	3.7	13	86	16	67
110781483	678	635	2.6	40.7	4.4	21	84	17	65
106777941	679	610	25.7	42.8	5.5	25	74	15	68
111238788	678	633	1.9	42.5	3.2	13	42	12	74
109049356	678	629	2.4	46.6	4.2	13	87	14	69
108286271	678	628	2.4	47.3	4.8	21	79	13	70
115469283	679	658	2.1	18.3	4.4	17	75	24	73
113303119	679	652	2.8	24.4	5.5	23	136	23	67
113371517	679	650	1.6	27.0	4.3	30	24	24	73
113618497	679	651	1.6	25.9	2.9	20	34	21	75
111762369	679	648	2.4	28.2	4.7	21	63	20	74
112516638	679	646	2.6	30.3	5.0	23	76	18	75
112315998	678	642	1.5	33.8	3.6	19	34	17	79
110618854	678	639	2.3	36.4	3.9	19	74	18	74

Appendix C

Table C3: Problem 82, work done per processor

Problem 82, 2790393007 nodes searched with MINDEPTH = 0 and MAXDEPTH = 16									
4 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
697243011	3875	3864	4.2	7.3	5.0	10	477	12	393
697186103	3875	3863	4.7	7.2	5.7	10	453	10	401
698396047	3875	3864	1.4	8.9	1.1	9	3	10	550
697567846	3874	3858	4.6	12.4	5.5	10	722	7	313
8 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
349846012	1952	1944	3.5	4.2	4.7	11	326	17	223
344356770	1951	1915	3.6	32.6	4.9	8	427	17	203
350080394	1950	1940	3.1	6.4	4.4	15	238	15	231
349866477	1950	1935	3.9	10.5	5.3	13	309	12	223
349692527	1951	1936	2.4	11.7	3.1	9	159	12	245
350247713	1951	1936	2.3	12.3	3.7	14	217	8	240
347883455	1950	1933	3.0	14.6	4.7	14	183	10	243
348419659	1951	1934	1.1	15.6	2.5	15	13	8	269
12 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
234622347	1331	1310	17.4	4.2	21.0	12	1998	23	709
232578320	1330	1292	11.0	26.8	12.7	13	504	21	843
227016937	1330	1277	11.3	42.1	12.3	15	401	7	861
233722664	1330	1296	2.4	31.5	2.4	17	27	10	889
233293116	1330	1300	17.5	12.5	21.2	17	1127	16	791
232820234	1330	1297	17.9	15.1	21.9	16	1403	18	764
233635224	1330	1304	7.7	17.9	9.2	17	334	18	861
232258525	1330	1295	14.7	20.0	18.0	17	770	16	823
233291201	1330	1288	16.1	24.9	20.3	16	905	15	811
234830439	1329	1298	6.9	24.5	7.9	15	430	13	856
230039526	1329	1282	17.4	30.0	21.4	16	993	13	804
232284474	1330	1285	13.1	31.9	15.9	12	922	13	808
16 processors									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
181179827	1012	998	5.9	7.8	7.7	12	351	38	232
175687686	1009	980	5.1	23.0	8.7	32	206	28	248
169673908	1010	959	4.6	46.5	6.5	32	103	25	247
174089878	1006	969	1.7	35.3	4.8	31	17	22	256
166253612	1009	918	41.5	48.7	9.6	25	402	22	231
172785932	1011	958	5.1	47.2	8.7	28	234	21	243
171943012	1010	959	5.8	46.1	8.3	19	243	20	243
173267844	1010	962	2.0	45.9	4.4	25	58	15	260
177854459	1011	983	6.2	22.2	9.0	32	346	30	236
177245471	1009	986	6.1	17.4	10.1	32	418	28	233
176150086	1010	985	3.7	21.2	6.4	24	174	28	247
175336264	1010	978	5.7	26.1	8.8	26	252	29	240
175859933	1010	978	6.2	26.4	8.8	17	351	31	232
174834688	1011	974	6.2	30.9	10.2	31	344	26	234
172871789	1010	972	5.2	33.4	8.5	31	178	27	244
175358618	1010	974	4.9	31.0	6.8	17	187	24	244

Appendix C

Table C4: Problem 88, work done per processor

Problem 88, 5156184395 nodes searched with MINDEPTH = 0 and MAXDEPTH = 16

4 processors

nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
1289096513	7380	7365	0.9	14.1	0.5	5	9	12	44
1282460225	7379	7370	1.0	8.3	1.6	14	16	10	38
1290031930	7379	7365	1.0	13.1	1.7	10	41	12	29
1294595727	7379	7357	1.3	20.1	2.2	13	65	8	23

8 processors

nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
646609767	3703	3697	1.5	5.0	2.5	12	28	21	57
632161532	3702	3644	1.7	56.3	3.3	21	175	18	33
648533361	3702	3694	1.6	6.3	3.1	18	52	19	50
645781060	3702	3687	1.5	13.1	3.7	21	35	19	52
652229477	3702	3691	1.3	9.6	3.4	20	13	17	56
646067820	3702	3689	1.2	11.3	3.1	18	45	18	51
643880079	3702	3689	1.4	11.5	3.3	21	35	15	53
640921299	3702	3690	1.0	11.0	2.1	13	19	17	55

12 processors

nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
435528005	2493	2482	3.4	7.9	4.8	14	74	18	159
426908277	2489	2446	1.7	41.0	2.6	15	50	20	153
421396853	2491	2442	4.5	44.6	3.2	6	236	15	138
431442757	2488	2460	3.3	25.3	5.3	17	128	14	148
436670810	2491	2474	3.3	14.0	4.9	15	172	20	141
432480342	2492	2468	3.2	20.6	4.4	13	163	14	148
431507382	2492	2466	2.7	22.8	5.4	25	96	14	153
431667275	2490	2468	2.3	20.3	4.2	20	53	16	155
422671042	2492	2442	3.3	45.8	4.4	13	103	12	155
430383472	2492	2465	3.6	23.4	4.9	10	107	13	153
426182680	2491	2461	2.9	27.0	4.4	18	191	16	140
429345500	2491	2458	3.4	30.1	5.6	19	388	13	125

16 processors

nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	success	failed	accepted	rejected
332498619	1889	1884	1.9	3.2	1.6	8	18	28	282
317182150	1888	1844	6.5	37.2	10.7	33	251	28	262
314610991	1888	1829	7.5	51.7	8.9	26	510	19	248
321118612	1887	1841	6.0	40.2	7.9	10	315	20	262
314519229	1888	1770	73.5	43.6	8.0	17	415	20	252
321720711	1888	1834	6.2	47.6	10.0	29	270	19	260
320860966	1887	1834	5.7	48.0	8.3	18	220	14	270
320190711	1888	1835	5.6	47.3	7.5	12	213	14	269
326843640	1888	1868	6.1	13.8	9.3	31	239	24	267
325599396	1887	1864	5.1	18.5	8.2	31	194	26	268
324350428	1888	1862	6.3	19.4	9.9	28	316	27	259
327569104	1887	1859	6.2	22.0	9.5	29	278	27	261
323383616	1888	1852	6.3	29.2	10.0	27	218	23	268
324626818	1887	1854	6.0	27.2	8.7	18	332	27	257
321821185	1888	1847	4.9	36.2	8.1	29	160	26	266
319288219	1888	1843	6.0	38.4	8.5	16	262	20	267