Parallel Depth First Search, Part I: Implementation*

V. Nageshwara Rao and Vipin Kumar†

Department of Computer Sciences,
University of Texas at Austin,
Austin, Texas 78712

Abstract

This paper presents a parallel formulation of depth-first search which retains the storage efficiency of sequential depth-first search and can be mapped on to any MIMD architecture. To study its effectiveness it has been implemented to solve the 15-puzzle problem on three commercially available multiprocessors — Sequent Balance 21000, the Intel Hypercube and BBN Butterfly. We have been able to achieve fairly linear speedup on Sequent up to 30 processors (the maximum configuration available) and on the Intel Hypercube and BBN Butterfly up to 128 processors (the maximum configurations available). Many researchers considered the ring architecture to be quite suitable for parallel depth-first search. Our experimental results show that hypercube and shared-memory architectures are significantly better.

At the heart of our parallel formulation is a dynamic work distribution scheme that divides the work between different processors. The effectiveness of the parallel formulation is strongly influenced by the work distribution scheme and architectural features such as presence/absence of shared memory, the diameter of the network, relative speed of the communication network, etc. In a companion paper[16], we analyze

*This work was supported by Army Research Office grant # DAAG29-84-K-0060 to the Artificial Intelligence Laboratory, and Office of Naval Research Grant N00014-86-K-0763 to the computer science department at the University of Texas at Austin.

†Arpanet: kumar@sally.utexas.edu

the effectiveness of different load-balancing schemes and architectures, and also present new improved work distribution schemes.

Key words: parallel formulation, depth-first search, work distribution schemes, state-space trees

1 Introduction

Depth-first search (DFS) is a general technique used in Artificial Intelligence for solving a variety of problems in planning, decision making, theorem proving, expert systems, etc. [14, 24]. It is also used under the name of backtracking to solve various combinatorial problems[6] and constraint satisfaction problems[22]. Execution of a Prolog program can be viewed as depth-first search of a proof tree [31]. Iterative-Deepening DFS algorithms are used to solve discrete optimization problems[10, 11] and for theorem proving[29]. A major advantage of the depth-first search strategy is that it requires very little memory. Since many of the problems solved by DFS are highly computation intensive, there has been a great interest in developing parallel versions of depth-first search [7, 32, 13, 4, 21, 8].

We have developed a parallel formulation of depth-first search which retains the storage efficiency of DFS. To study its effectiveness we have incorporated it in IDA* (a DFS algorithm with iterative-deepening [10]) to solve the 15-puzzle problem[23] on three commercially available multiprocessors – Sequent Balance[1] 21000, the Intel iPSC [2] Hypercube and BBN Butterfly[3]. We also tested the effectiveness of parallel depth-first search on a ring embedded in the Intel Hypercube. We have been able to achieve linear speedup on Sequent Balance up to 30 processors (the maximum configuration available) and on the Intel Hypercube and BBN Butterfly up to 128 processors (the maximum configurations available). Contrary to the expectation of many researchers[32, 4, 21], the performance on the ring architecture is not very good.

At the heart of our parallel formulation is a dynamic work distribution scheme that divides the work between different processors. The effectiveness of the parallel formulation

[1]Balance is a trade mark of the Sequent Computer Corp.

[2]iPSC is a trademark of Intel Scientific Computers.

[3]Butterfly is a trade mark of the BBN Advanced Computers, Inc.

is strongly influenced by the work distribution scheme and architectural features such as presence/absence of shared memory, the diameter of the network, relative speed of the communication network, etc. Our experiments suggest (and theoretical analysis of [16] predicts) that on suitable architectures, it is feasible to speed up depth-first search by several orders of magnitude. Although, the paper only deals with (sequential and parallel) depth-first search of state-space trees, the discussion with some modifications is also applicable to depth-first search of AND/OR graphs and trees (e.g., execution of Prolog programs, game tree search). Also, the work distribution schemes used in our implementation can be used in the parallel implementations of other tree traversal algorithms such as divide-and-conquer[6].

Section 2 gives a brief review of sequential depth-first search, the IDA* algorithm, and depth-first branch-and-bound. Section 3 presents a parallel formulation of DFS, and discusses its applicability to IDA* and depth-first branch-and-bound. Section 4 presents performance results of solving 15-puzzle by parallel IDA* on various parallel processors. Section 5 reviews previous work on parallel depth-first search. Section 6 contains concluding remarks.

2 Review of Depth-First Search

2.1 Simple Depth-First Search

Search methods are useful when a problem can be formulated in terms of finding a solution path in an (implicit) directed graph from an initial node to a goal node. The search begins by expanding the initial node; i.e., by generating its successors. At each later step, one of the previously generated nodes is expanded until a goal node is found. (Now the solution path can be constructed by following backward pointers from the goal node to the initial node.) There are many ways in which a generated node can be chosen for expansion, each having its own advantages and disadvantages. In depth-first search, one of the most recently generated nodes is expanded first. Fig. 1 shows a tree structured graph generated by a depth-first search procedure. Numbers on the left corners of the nodes show the order in which the nodes are generated. Numbers on the right corners of the nodes show the order in which the nodes are expanded. Since deeper nodes are expanded first, the search is called depth-first. A more detailed treatment is provided in [14] and [24].

The unit of computation in a search algorithm is the time taken for one node expansion. The total time taken by a sequential search algorithm is roughly proportional to the total

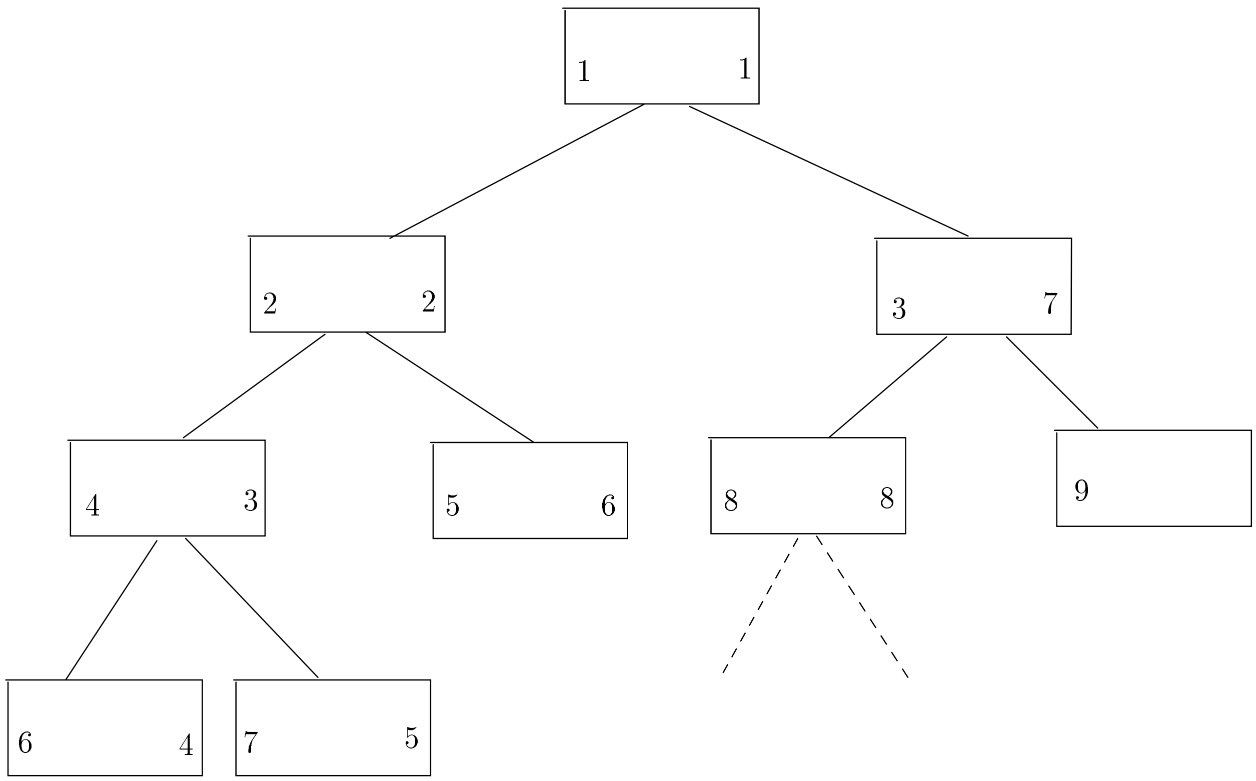


Figure 1: Search Tree generated by a depth-first search procedure. For each node, number in the left corner indicates the order of generation and number in the right corner shows the order of expansion.

number of nodes it expands. Total number of nodes expanded by a search algorithm for a particular instance is called the **problem size** W of the instance. The **effective branching factor** b is defined as the average number of successors of the nodes of the search tree. If the depth of the search tree is d, then the effective branching factor b is approximately $W^{\frac{1}{d}}$.

The main advantage of depth-first search over other search techniques is its low storage requirements. For most other search techniques (such as breadth-first and best-first) the storage requirement is exponential in the length of solution path, whereas for depth-first search, the storage requirement is linear in the depth of the space searched [14, 10]. But simple depth-first search has two major drawbacks.

1. If the search space to the left of the first goal node is infinite (or very large) then search would never terminate (or take a very long time).

2. It finds the left-most solution path, whereas best-first search finds an optimal (i.e., a least cost) solution path.

One possible way of handling the first problem is to have a bound on the depth of the space searched. But if all the goal nodes are beyond the selected bound, then bounded DFS will fail to find any solution path. Iterative-Deepening-A* (IDA*)[10] is a variation of depth-first search that takes care of both these drawbacks.

2.2 Iterative-Deepening A*(IDA*)

IDA* performs repeated cost-bounded depth-first search (DFS) over the search space. Like other heuristic search procedures (such as the A* algorithm[23]), it makes use of two functions h and g. For a node n, g(n) is the cost of reaching n from the initial node, and h(n) is an estimate of the cost of reaching a nearest goal node from n. In each iteration, IDA* keeps on expanding nodes in depth-first fashion until the total cost (f(n) = g(n) + h(n)) of the selected node n exceeds a given threshold. For the first iteration, this threshold is the cost (f-value) of the initial node. 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 node is selected for expansion. If the cost function is admissible (i.e., if for any node n, h(n) is a lower bound on the cost of all paths from n to a goal), then IDA* (like A*) is guaranteed to find an optimal solution path. Iterative-Deepening-A* is an important admissible[4] state-space search algorithm, as it runs in asymptotically optimal time for a wide class of search problems. Furthermore, it requires only linear storage. In contrast, A*, the most widely known admissible state-space-search algorithm, requires exponential storage for most practical problems [24]. For a detailed description of IDA* and its properties, the reader is referred to [10, 11].

2.3 Depth-First Branch-and-Bound

It is also possible to find an optimal solution path in a finite search space using DFS (even without iterative deepening). In this case, DFS is used to search the whole search space exhaustively; i.e., the search continues even after finding the first solution path. (Recall that in simple DFS and IDA*, the search stops after the first solution path is found.) Whenever a new solution path is found, the current best solution path is updated. Whenever an inferior partial solution path (i.e., a partial solution path whose extensions are guaranteed to be worse than the current best solution path) is generated, it is eliminated. This kind of search is called **depth-first branch-and-bound** (depth-first B&B)[14, 19]. Although depth-first B&B would usually perform much more work than best-first B&B, it is (like any other depth-first search strategy) highly space efficient. Note that the alpha-beta game tree search algorithm can be viewed as a depth-first B&B algorithm (see [12, 15]).

3 Parallel Depth-First Search

3.1 A Parallel Formulation of Depth-First Search)

We parallelize DFS by sharing the work to be done among a number of processors. Each processor searches a disjoint part of the search space in a depth-first fashion. When a processor has finished searching its part of the search space, it tries to get an unsearched part of the search space from the other processors. When a goal node is found, all of them quit. If the search space is finite and has no solutions, then eventually all the processors would run out of work, and the (parallel) search will terminate.

Since each processor searches the space in a depth-first manner, the (part of) state space to be searched is efficiently represented by a stack. The depth of the stack is the depth of the

[4] A search algorithm is admissible if it always finds an optimal solution path[23].

node being currently explored; and each level of the stack keeps track of untried alternatives. Each processor maintains its own local stack on which it executes DFS. When the local stack is empty, it takes some of the untried alternatives of another processor's stack. See Fig. 2 for an illustration. In our implementation, at the start of each iteration, all the search space is given to one processor, and other processors are given null spaces (i.e., null stacks). From then on, the search space is divided and distributed among various processors.

The basic driver routine in each of the processors is given below. P_i denotes the ith processor, and stack[i] denotes the stack of the ith processor.

Parallel DFS: Processor P_i

```
while (not terminated) do
        if (stack[i] = empty) then GETWORK() ;
        while (stack[i] ≠ empty) do
                DFS(stack[i]) ;
                GETWORK() ;
        od
        TERMINATION-TEST() ;
od
```

Once available space is fully searched, GETWORK() is called to get more work. If new work is not received (from the targets tried in GETWORK()), then a termination-test routine is called to see if all other processors have finished. If the termination test fails then GETWORK() is called again to get some work. Procedure GETWORK is architecture dependent. The following version is good for shared-memory multiprocessors.

GETWORK()

```
for (j = 0... NUMRETRY-1) do
    target = (target + 1) mod N;
    if work is available at the stack of processor P_target above the cutoff depth
    then
              lock stack[target] ;
              pick work from target.
              unlock stack[target] ;
              return;
    endif
od
return;
```

The procedures GETWORK and TERMINATION-TEST involve communication with other processors. By restricting communication with immediate neighbors only, we can implement parallel DFS on any MIMD architecture.

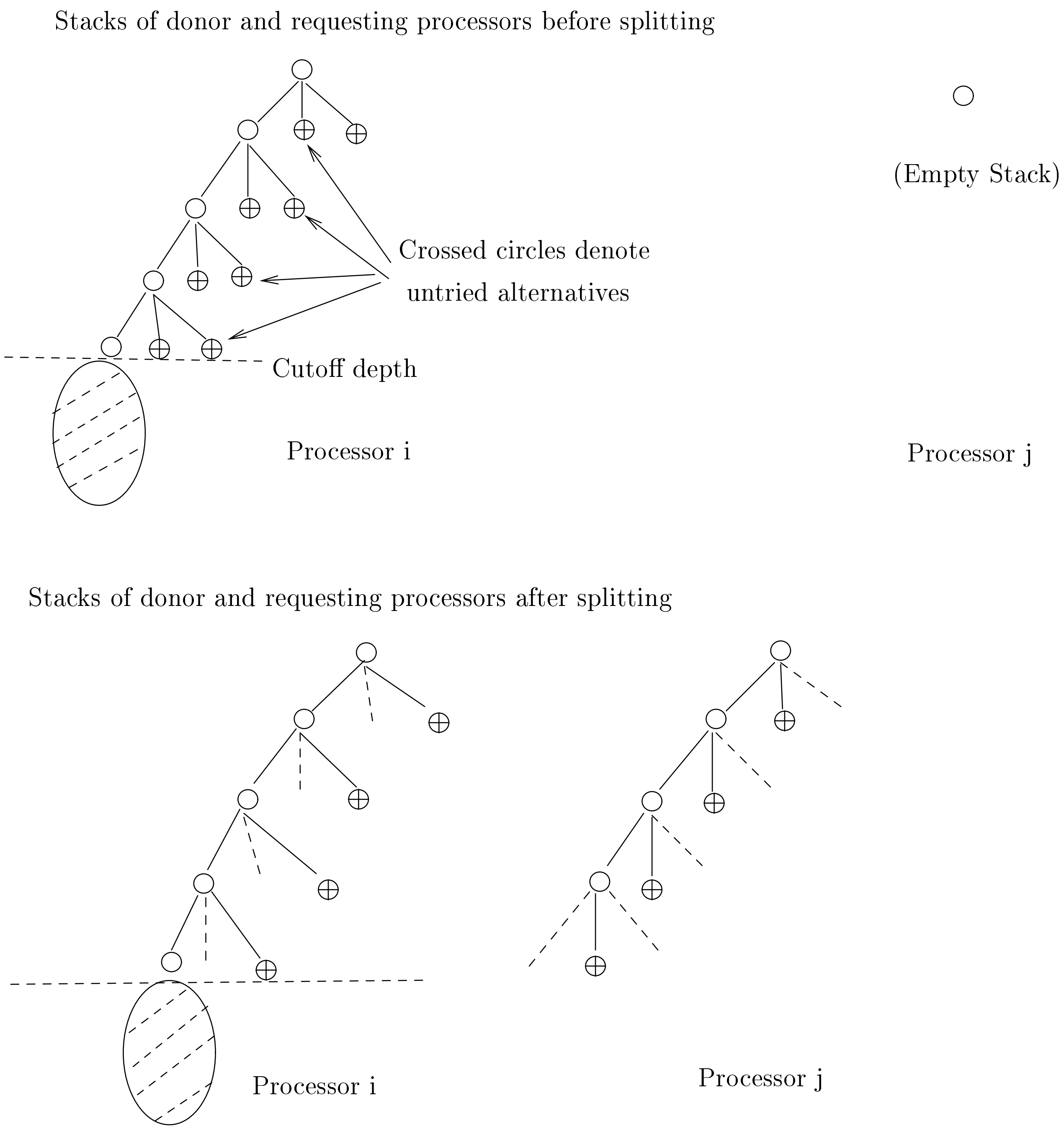


Figure 2: Splitting work in a stack between two processors in parallel DFS. Processor j is requesting work from Processor i. Stacks are assumed to grow downward.

On a distributed-memory system (such as the Intel Hypercube), whenever a processor needs work, it sends a request for work to one of its neighbors. (Each immediate neighbor is polled for work in a round-robin fashion.) If the neighbor has work (i.e., unsearched alternatives) available above the cutoff depth, it sends it to the requesting processor; otherwise it sends a reject message. If the requesting processor receives a reject, then it tries to get work from another neighbor. All processors service request from their neighbor(s) periodically. Whenever a stack is transferred we have two options.

1. Copy the relevant node information from every level of stack and transfer it.
2. Copy the operator sequence applied and regenerate the stack in the requesting processor using the initial node and the operator sequence.

The choice between the two is determined by the ratio $\frac{r_{comm}}{r_{calc}}$, where r_{comm} is the rate of transfer of bytes between processors and r_{calc} is the rate of node expansion.

3.2 Important parameters of the Parallel Formulation

Two parameters of the algorithm are important for performance on a given architecture and a given problem.

1. The splitting strategy
2. The cutoff depth

3.2.1 The Splitting strategy.

When a work transfer is made, work in the donor's stack is split into two stacks one of which is given to the requester. In other words, some of the nodes (i.e., alternatives) from the donor's stack are removed and added to the requester's stack. See Fig. 2 for an illustration. Intuitively it is ideal to split the stack into two equal pieces (called $\frac{1}{2}$-split). If the work given out is too small, then the requester would become idle too soon. If the work given out is too large, then the donor will become idle too soon. From the analysis in [16], it is clear that a $\frac{1}{2}$-split leads to an overall high efficiency for the shared-memory and hypercube architectures.

Many strategies for removing nodes are possible; e.g.,

1. pick up some nodes near the root;
2. pick up some nodes near the cutoff depth;
3. pick up half of all the available nodes above the cutoff depth (this strategy is used in Fig. 2).

The suitability of a splitting strategy is dependent upon the nature of the search space. If the search tree is uniform, then both strategies 1 and 3 would result in a good splitting. If the search space is highly irregular, then only Strategy 3 results in a good splitting. If a strong heuristic is available (that can be used to order the successors so that (some of the) goal nodes move to the left of the state-space tree), then Strategy 2 performs better, as it tries to equitably share the useful[5] part of the search space. The cost of splitting is also important especially if the stacks are deep. For deep search spaces, strategies 1 and 2 would have smaller cost than Strategy 3.

3.2.2 The Cutoff Depth

For both shared-memory and distributed-memory architectures, having a cutoff (= *delta*) ensures that the amount of work transferred in any transfer is at least $\epsilon(\simeq b^{delta})$. This ensures progress and avoids thrashing effects. On a shared-memory system, this also provides a lower bound on the efficiency[6] (assuming that the cost of locating a processor with work is negligible). If U_{comm} were the time for one work transfer and if the load imbalance[7] is negligible then

$$A\ lower\ bound\ on\ efficiency = \frac{\epsilon}{\epsilon + U_{comm}}$$

From the above formula, increasing ϵ leads to a higher lower bound on efficiency. However, increasing ϵ also leads to an increased load imbalance. The value of ϵ should be chosen to balance these opposing effects. On a distributed-memory system such as the Intel Hypercube, checking the availability of work at other nodes usually costs more than getting work. Furthermore, some part of the system may have work while processors in the other part are just

[5] the part that is likely to contain a solution

[6] Efficiency = $\frac{Speedup}{Number\ of\ processors}$

[7] On a shared-memory machine, load imbalance is negligible if ϵ is small compared to $\frac{W}{N}$, the average amount of work done by an individual processor

looking around for work. Hence, the lower bound is not applicable to distributed-memory architectures.

On a shared-memory system, the cutoff depth serves the additional purpose of transferring work without interrupting the target processor, as the target processor can continue working on nodes below the cutoff depth while a requester is picking alternatives above the cutoff depth. The cutoff depth should be chosen so that the work done below cutoff takes much longer time when compared to the stack transfer time, so that a work transfer can occur without significantly slowing down the target processor.

3.3 Speedup Anomalies

In parallel DFS all the processors abort when the first goal node is detected by any processor. Due to this it is possible for parallel DFS to expand fewer or more nodes than DFS, depending upon when a goal node is detected by a processor. Even on different runs for solving the same problem, parallel DFS can expand different number of nodes, as the processors run asynchronously. If parallel DFS expands fewer nodes than DFS, 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 the acceleration anomaly [18, 20].

But there can be no detrimental anomaly (i.e., speedup of less than 1 on N processors) in parallel DFS, if we assume that all the processors have roughly equal speed. In parallel DFS at least one processor at any time is working on a node n such that everything to the left of n in the (cost bounded) tree has been searched. Suppose DFS and parallel DFS start to search at the same time. Let us assume that DFS is exploring a node n at a certain time t. Clearly all the nodes to the left of n (and none of the nodes to the right of n) in the tree must have been searched by DFS until t. It is easily seen that if overheads due to parallel processing (such as locking, work transfer, termination detection) are ignored, then parallel DFS should have also searched all the nodes to the left of n (plus more to the right of n) at time t. This guarantees that parallel DFS running on N processors would never be slower than DFS for any problem instance.

3.4 Applicability to IDA*

Since each iteration of IDA* is a cost-bounded depth-first search, a parallel formulation of IDA* is obtained by executing each iteration via parallel DFS. In IDA*, all but the last iteration terminate without finding a goal node. After termination of each iteration of parallel IDA*, one specifically assigned processor determines the cost bound for the next iteration and restarts parallel depth-first search with the new cost bound. Search stops in the final iteration when one of the processors finds a goal node and informs all others about it. The termination of an iteration can be detected in many ways. On shared-memory architectures (e.g., Sequent, BBN Butterfly), we use a globally shared variable to keep track of the number of idle processors. On distributed memory architectures (e.g., the Intel Hypercube), we use Dijkstra's token termination detection algorithm[1]. See [25] for more details.

3.5 Applicability to Depth-First B&B

Our parallel formulation is applicable to depth-first B&B with one minor modification. Now we need to keep all the processors informed of the current best solution path. On a shared-memory architecture, this can be done by maintaining a global best solution path. On a distributed-memory architecture, this can be done by allowing each processor to maintain the current best solution path known to it. Whenever a processor finds a solution path better than the current best known, it broadcasts it to all the other processors, which update (if necessary) their current best solution path. Note that if a processor's current best solution path is worse than the global best solution path, then it only affects the efficiency of the search but not the correctness.

The acceleration anomaly discussed in Section 3.3 can happen even in parallel depth-first B&B (even though the search does not stop after finding the first solution path) because the search space can be pruned differently in sequential and parallel search. In the rest of the paper, we will only deal with the parallelization of simple DFS and IDA*, although most of the discussion is applicable to the parallelization of depth-first B&B as well.

4 Performance of Parallel DFS on various architectures

4.1 Parallel Architectures used in Experiments

We have studied the performance of parallel DFS on shared-memory/common-bus, shared-memory/Ω–switch, hypercube[27], 1-ring and 2-ring architectures . The Sequent Balance

Interconnection	*Machine*	*CPU speed*	U_{comm}	U_{calc}	*Diameter*
Shared Bus	Sequent Balance	0.60 mips	3.5 msec	1.8 msec	1
Shared Ω–switch	BBN Butterfly	0.50 mips	5.0 msec	2.0 msec	1
Hypercube	IPSC–d7	1.00 mips	27.0 msec	1.0 msec	$\log N$
1-ring	- do -	,,	,,	,,	N
2-ring	- do -	,,	,,	,,	$\frac{N}{2}$

Table 1: Characteristics of the parallel processors used in experiments

21000 was used as a shared-memory/common-bus multiprocessor. BBN Butterfly served as a shared-memory multiprocessor with Ω–network. In both of these architectures the diameter of the network is 1. The Intel Hypercube was used to study hypercube, 1-ring and 2-ring architectures. In a 1-ring, a processor is allowed to get work from only one (left) neighbor, whereas in a 2-ring it is allowed to get work from both neighbors.

Two parameters of an architecture affect the performance of parallel DFS.

1. The Ratio of Communication rate to Computation Rate

   Our unit of communication is the time for one stack transfer U_{comm} and our unit of computation is the time for one node expansion U_{calc}. If N is the total number of nodes expanded and C is the total number of stack transfers made then

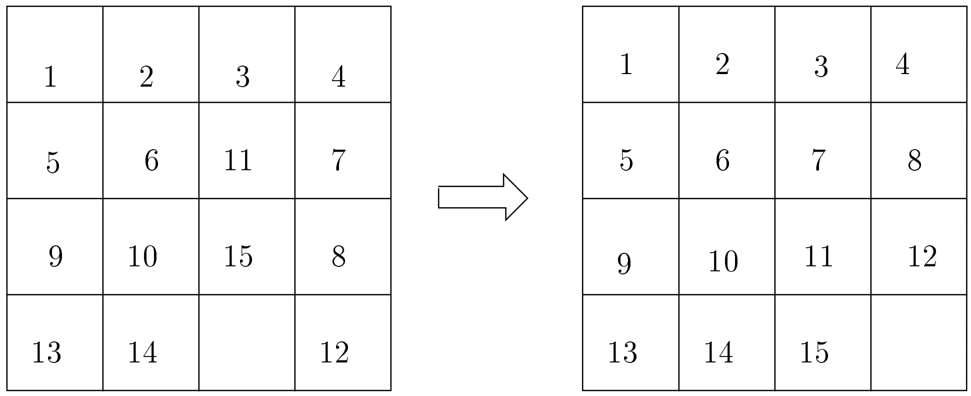
$$Efficiency \leq \frac{U_{calc} * N}{U_{calc} * N + U_{comm} * C}$$

2. Diameter of the network

   Diameter of the network dictates the amount of communication that needs to be made in order to achieve good load balancing. The effect of diameter is analyzed in [16].

Table 1 presents the values of these parameters for the machines under consideration. (U_{comm} and U_{calc} are specific to the implementation of parallel DFS for 15-puzzle[8].)

[8] In our implementations for distributed-memory systems, processors check arrival of requests periodically every 50 node expansions. Hence U_{comm} is approximately $25 * U_{calc} +$ *misc. overhead*



A starting configuration Desired goal configuration

Figure 3: The 15-puzzle

4.2 Experiments for Evaluating Parallel DFS

To test the effectiveness of parallel DFS, we have used it to solve the 15-puzzle problem [23]. 15-puzzle is a 4x4 square tray containing 15 square tiles. The remaining sixteenth square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space can be slid into that space. An instance of the problem consists of a initial position and a specified goal position. The goal is to transform the initial position into the goal position by sliding the tiles around.(See Fig. 3). The 15-puzzle problem is particularly suited for testing the effectiveness of parallel DFS, as it is possible to create search spaces of different sizes (W) by choosing appropriate initial positions. IDA* is the best known sequential algorithm to find optimal solution paths for the 15-puzzle problem[10]. It is significantly faster[9] than simple DFS, as it can use a heuristic function to focus the search (we use the Manhattan distance heuristic [23]). We have parallelized IDA* to test the effectiveness of our parallel formulation of depth-first search.

[9] Note that due to the use of heuristic function h, the effective branching factor of a search tree in IDA* could be much smaller than the average number of successors of a node. For example, in 15-puzzle, the average number of successors of a node is 2 (not counting the parent). Hence, the average branching factor of simple DFS on this problem is 2, whereas due to the use of Manhattan distance heuristic, the effective branching factor of IDA* is approximately 1.3.

4.3 Performance of Parallel IDA*.

We implemented parallel IDA* to solve the 15-puzzle problem on Sequent Balance 21000, a shared-memory parallel processor. To test the effect of the cutoff depth, we experimented with a range of cutoff depths. Depths of the trees generated in our experiments varied between 40 and 60. The speedup performance remained unchanged for a wide range of cutoff values (between $.25 \star depth$ and $.75 \star depth$). Outside this range, performance tends to degrade either due to thrashing (when cutoff is too small) or due to load imbalance (when cutoff is too large). We also experimented with the three splitting strategies described in Section 3.2.1. In our experiments, the third strategy consistently outperformed the other two strategies. The reason is that, in 15-puzzle, the cost-bounded search space (generated by IDA*) tend to be highly imbalanced, and the heuristic ordering of immediate successors does not help to move goal nodes to the left of the state-space tree. In all the results reported in this paper, we use the third splitting strategy, and keep the cutoff depth between $.25 \star depth$ and $.75 \star depth$.

We ran our algorithm on a number of problem instances given in Korf's paper [10]. Each problem was solved using IDA* on one processor, and using parallel IDA* on 9, 6 and 3 processors. As explained earlier (Section 3.3), for the same problem instance, parallel IDA* can expand different number of nodes in the last iteration on different runs. Hence parallel IDA* was run 20 times in each case and the speedup [10] 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 thirteen (randomly chosen) 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. 4). Superlinear speedup indicates that parallel DFS is able to find a goal node by searching a smaller space than (sequential) DFS. Many other researchers have encountered superlinear speedup [7, 21] in parallel depth-first search. In [26], we present an analysis of the phenomenon of superlinear speedup in depth-first search and show that it is possible to obtain superlinear speedup on average if the search space has certain characteristics.

[10] We compute speedup by

$$\frac{The\ time\ taken\ by\ IDA* \ \ (the\ best\ known\ sequential\ algorithm\ for\ 15-puzzle)}{The\ time\ taken\ by\ the\ parallel\ version\ of\ IDA*}$$

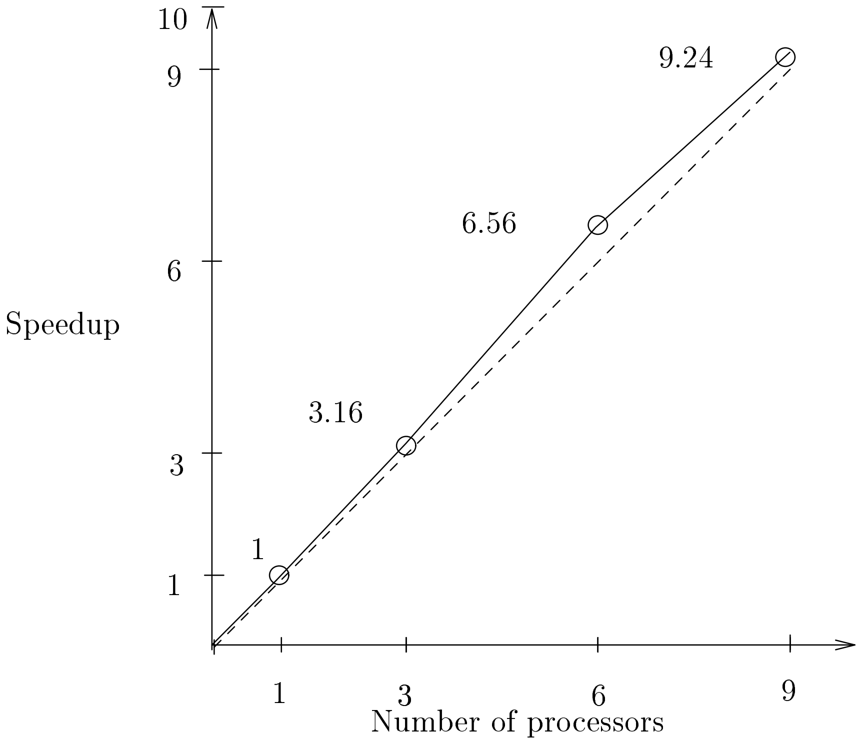


Figure 4: The speedup curve for Parallel IDA* on Sequent Balance

To study the speedup of parallel approach in the absence of anomaly, we modified IDA* and parallel IDA* to find all optimal solution paths. This ensures that both IDA* and parallel IDA* search all the space within the cost bound of the final iteration; hence both IDA* and parallel IDA* explore exactly the same number of nodes. In this case the speedup of parallel IDA* is quite consistently close to but less than N for N processors for every problem instance (Fig. 5). The speedup is slightly less than N because of overheads introduced by distribution of work, termination detection, etc. As shown in Fig. 5, the speedup grows almost linearly even up to 30 processors. This shows that our scheme of splitting work among different processors is quite effective.

To test the effectiveness of our scheme for larger number of processors, we implemented it on BBN Butterfly (120 processors). To test its suitability for distributed-memory architectures (as opposed to shared-memory architectures), we implemented it on the Intel Hypercube (128 processors). We embedded 1-ring and 2-ring on the Intel Hypercube to study the effect of connectivity and diameter on speedup.

As shown in figures 6 and 7, we are able to get linear speedup even for 100+ processors on BBN Butterfly and the Intel Hypercube. On a 1-ring and a 2-ring we are able to get good speedup up to 16 processors. Beyond that the maximum speedup obtained on 2-ring is 24 on 128 processors. The speedup on 1-ring is even smaller. In general, increasing problem size improves speedup for a given number of processors and architecture. The problem sizes for which these speedups were obtained are different for each architecture and are indicated in the respective figures. On the Intel Hypercube, the problem size has to be much bigger than for BBN Butterfly to get similar performance. One reason is that the ratio $\frac{U_{comm}}{U_{calc}}$ is much higher for the Intel Hypercube than for BBN Butterfly. Also, the work distribution schemes used for the two architectures are different. In a companion paper[16], we analyze the effect of work distribution schemes, architectural features and problem size on speedup.

4.4 Performance of Parallel Cost-Bounded DFS.

Parallel version of IDA* incurs two kinds of overheads — the overhead due to work distribution, and the overhead due to termination detection. The first is common to all depth-first search schemes, whereas the second is specific to IDA*. To isolate the overhead due to work distribution, we further modified IDA* and its parallel version to execute only the last

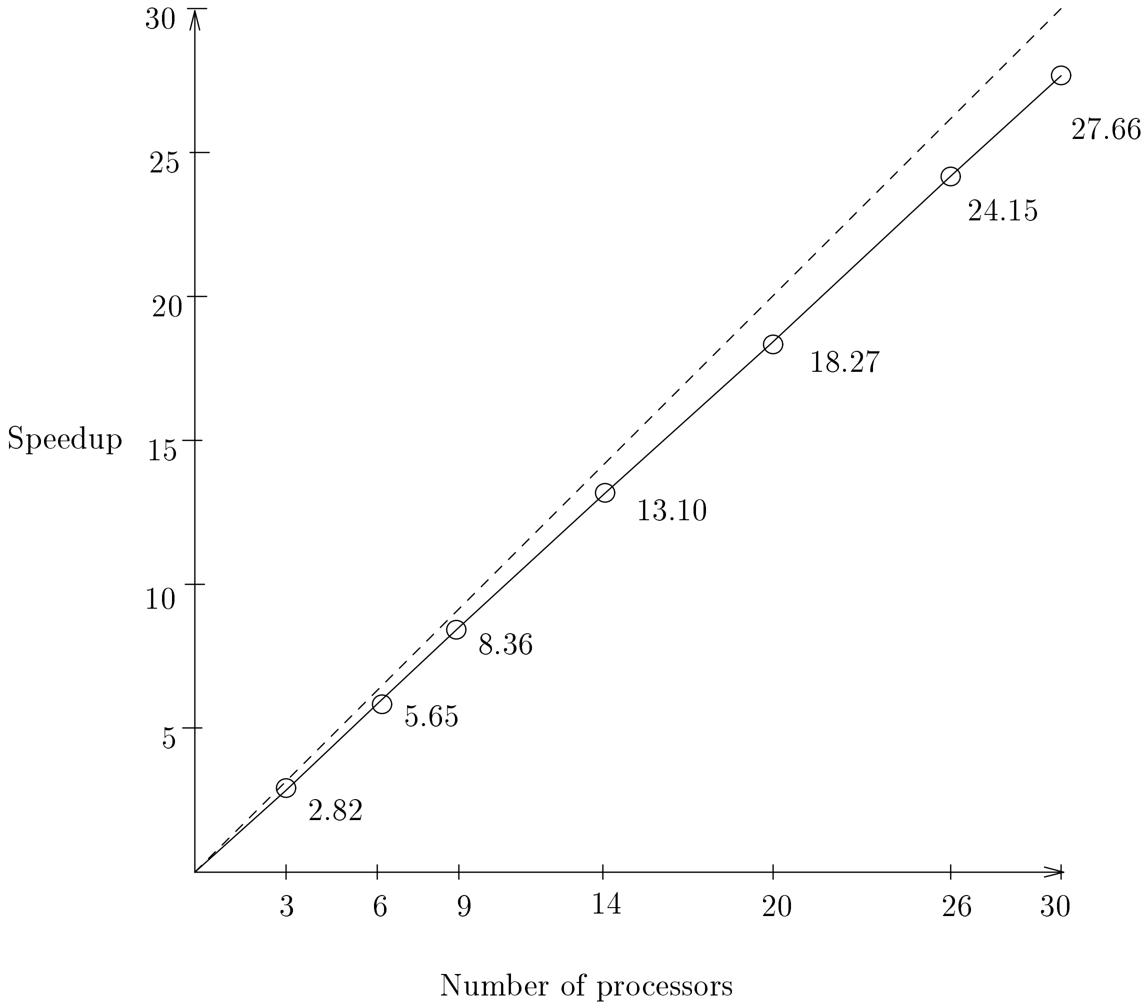


Figure 5: The speedup curve for Parallel IDA* (all-optimal-solution-paths case) on Sequent Balance 21000. Mean sequential execution time $\simeq$ 900secs.

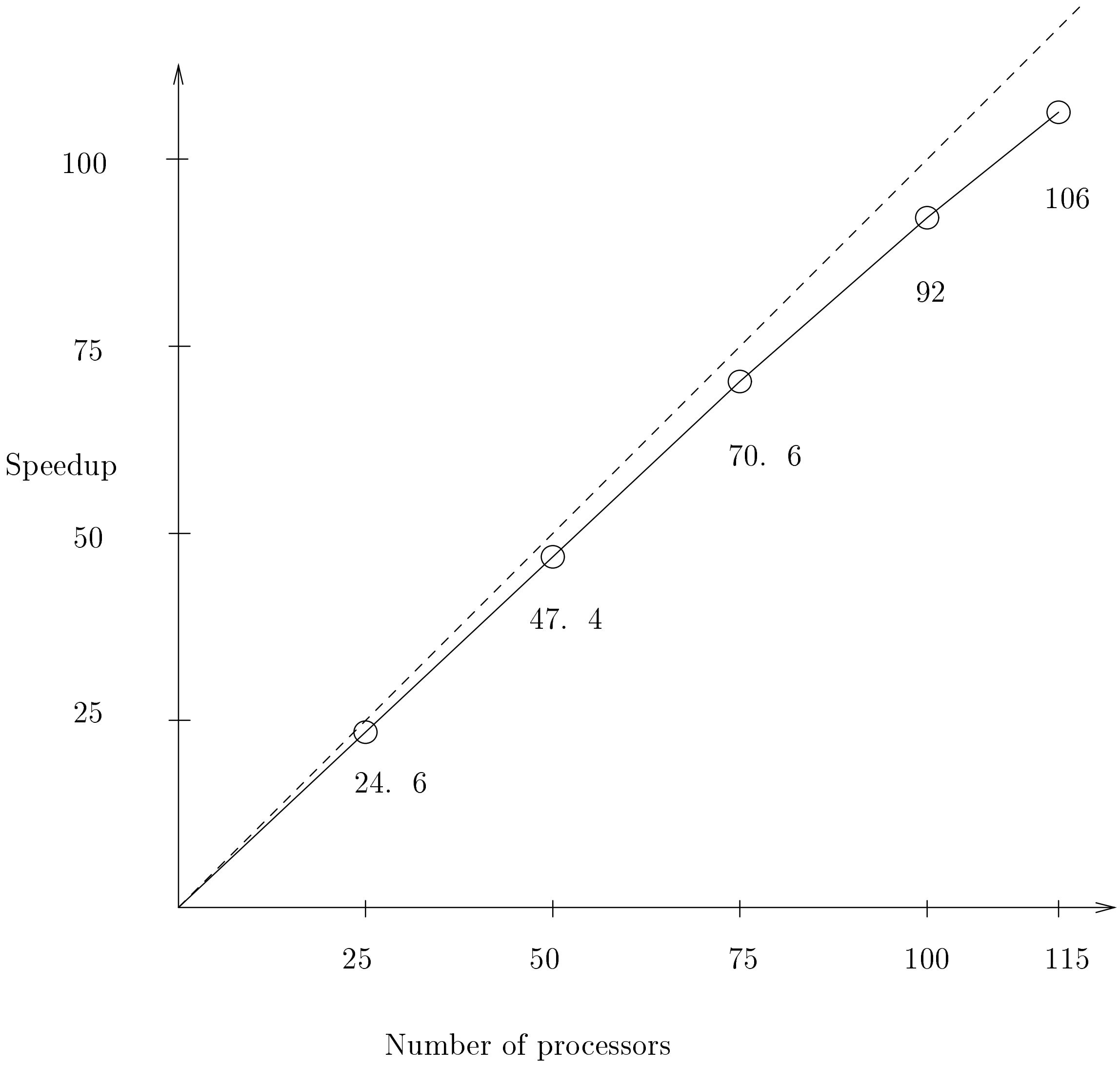


Figure 6: The speedup curve for Parallel IDA*(all-optimal-solution-paths case) on BBN Butterfly. Average sequential Execution time $\simeq$ 2000 secs, average problem size $\simeq$ 700,000 nodes

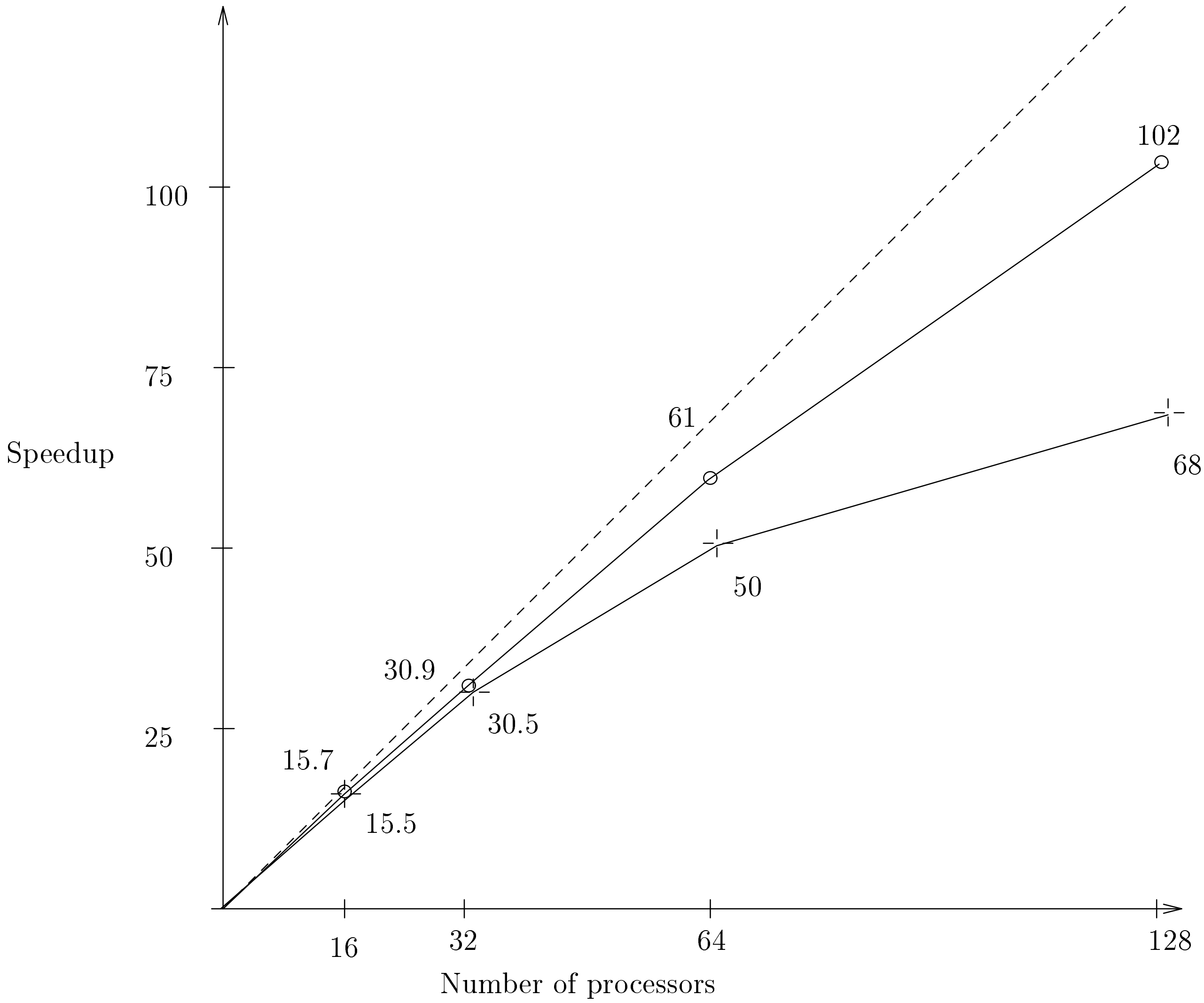


Figure 7: Speedup curves for Parallel IDA*(all-optimal-solution-paths case) on the Intel Hypercube. Upper curve: Average prob. size = 38.2 million nodes, average sequential Exec. time $\simeq$ 36000 secs. Lower curve: Average prob. size = 2.5 million nodes, average sequential Exec. time $\simeq$ 3600 secs.

iteration (by starting IDA* with the final cost bound)[11]. Note that any single iteration of IDA* is a simple cost-bounded depth-first search. Fig. 8 gives details of speedup achieved on various architectures. These speedup figures are somewhat better because there is no overhead of recoordinating processors after every iteration.

5 Related Research.

Dynamic division of work has been used by many researchers for parallelizing depth-first search[4, 32, 21, 3]. Many of these researchers[4, 32, 21] have implemented parallel DFS on the ring architecture and studied its performance for around 16–20 processors. Monien and Vornberger[21] and Wah and Ma[32] present parallel depth-first search procedures on a ring network. The work distribution schemes in these formulations is very similar to the scheme presented in this paper. From our experiments as well as the analysis in [16] it is clear that this work distribution scheme is not able to provide good speedup on large rings. The initialization part in Monien's[21] and Wah's[32] scheme is slightly different than the one discussed in this paper. Before starting parallel search they divide the search space into N parts, and give each part to a processor. If the initial distribution is quite good, then good speedup can be obtained even with the simple work distribution scheme. But good distribution can be difficult to obtain especially for large problems and large number of processors.

Finkel and Manber's work[4] on distributed backtracking has many similarities to the work reported in this paper. They have experimented with several work distribution strategies for the ring architecture. These strategies are different than the one used in this paper. In a companion paper[16], we analyze the effectiveness of different work distribution strategies, and present a new improved work distribution strategy for the ring architecture. The main thrust of Finkel's work is on developing a package called DIB which allows a variety of applications requiring tree traversal to be implemented on a multicomputer. They report speedup results for many problems (n-queens, minimax evaluation of game trees, the traveling salesman problem) on the Crystal multicomputer (which is a collection of 20 Vax 11/750s connected via a token ring). They also investigate several extensions of the work distribution schemes to incorporate fault tolerance in DIB.

[11] We still search for all optimal solution paths to eliminate speedup anomalies

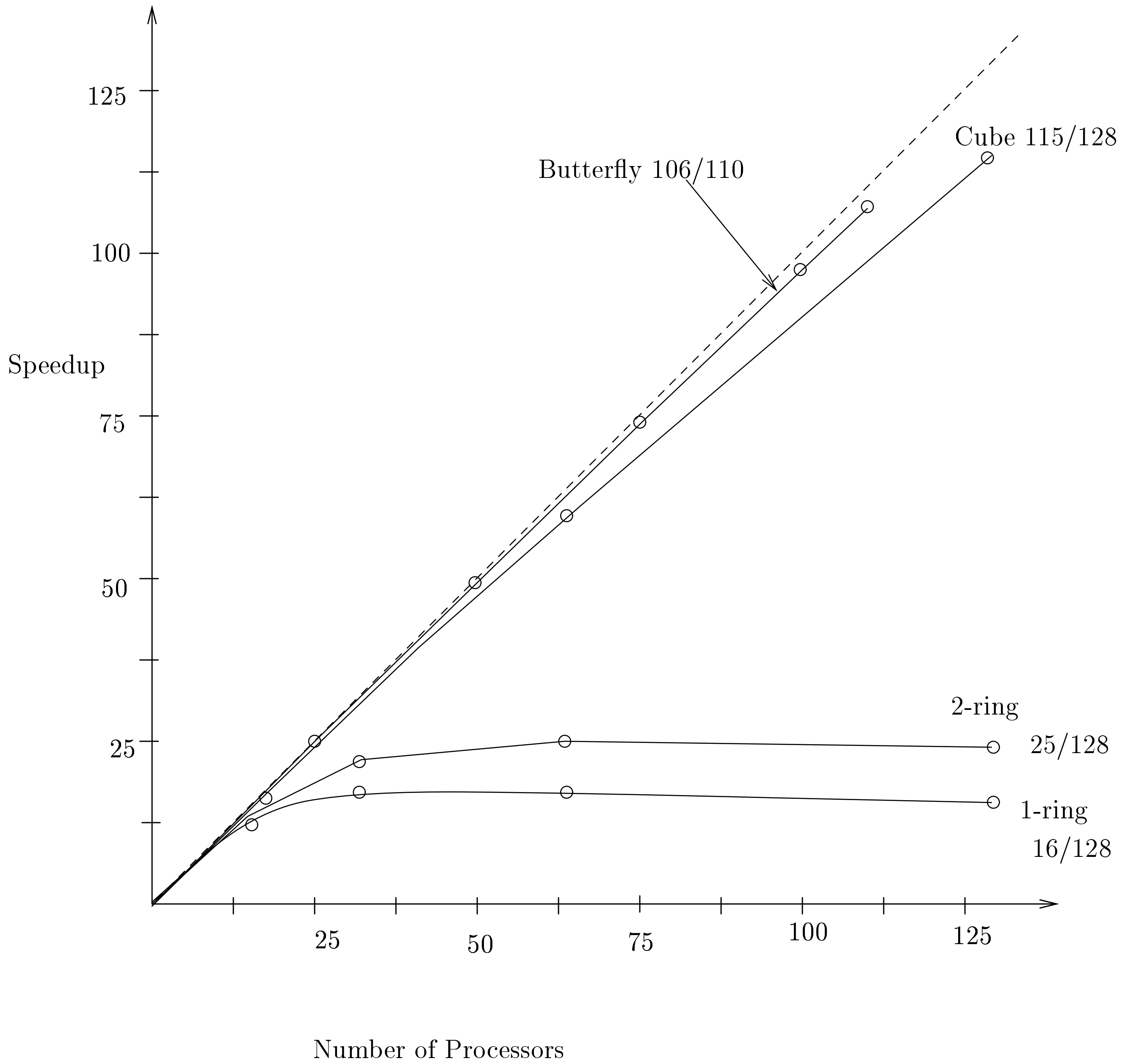


Figure 8: Speedup curves on various architectures for parallel cost-bounded DFS (i.e., only the last iteration of IDA*). Average problem size $\simeq$ 6.68 million nodes

While our experiments agree with Monien, Wah and Finkel in that good efficiency is achievable for 16 processor rings, our experimental results show and our analysis in [16] predicts that it is unrealistic to achieve the same for rings with larger (64-128) number of processors. Clearly a shared-memory machine or a distributed-memory machine with low diameter like the Intel Hypercube is far superior to a ring for a large number of processors.

Janakiram et. al. [8] present a parallel formulation in which different processors search the space in different (random) orders. The speedup in this scheme is dependent upon the probability distribution of goal nodes in the search tree. A major feature of this scheme is that it is fault tolerant, as any single processor is guaranteed to find a goal node. The work done by each processor in this scheme can be executed in parallel using our work distribution scheme to give additional speedup.

A number of parallel formulations have been proposed for depth-first B&B. One of the first such formulation was proposed by El-Dessoki and Huen [3]. Their formulation uses dynamic work sharing much like our formulation. Imai et al [7] proposed another parallel formulation of depth-first B&B. In this scheme, the search tree is maintained as a shared data structure, and different processors remove and expand one node at a time in depth-first fashion. A major drawback of this approach is that the upper bound on speedup, irrespective of N and problem size, is $\frac{U_{calc}}{U_{comm}}$. Hence the approach is suited only for those problems for which U_{calc} is very large compared to U_{comm}. (For Imai's algorithm U_{comm} is the time taken for picking up 1 node from the global data structure, and U_{calc} is the time to generate successors of a node.) Another major drawback is that it requires a shared memory and hence is not suited for a distributed-memory machine such as the Intel Hypercube. Imai's approach more or less follows a left to right scan of the tree and hence is less prone to anomalies than our parallel DFS.

Kumar and Kanal [13] present a parallel formulation of depth-first B&B in which different processors search the space with different expectations (cost bounds). At any time, at least one processor has the property that if it terminates, it returns an optimal solution path; the other processors conduct a look-ahead search. The scheme is in principle similar to executing different iterations of IDA* in parallel with dynamically changing cost bounds. This approach requires very little communication between processors, but the maximum speedup obtained is problem dependent. As with Janakiram's scheme[8], the work done by each processor in this scheme can be executed in parallel using our work distribution method to give additional speedup.

Most systems for exploiting OR-parallelism in logic programs are essentially implementations of parallel depth-first search [9, 17, 28, 5, 30, 2]. These systems also use dynamic work sharing to divide the work evenly among processors. A major problem in such systems is that the size of the stack grows very rapidly for many logic programs, which makes stack splitting rather expansive. Hence much of the current research in such systems is on developing techniques that allow parts of the stack to be used by many processors[28].

6 Conclusions.

We have presented performance results of a parallel formulation of depth-first search on various parallel architectures. Our parallel DFS retains the storage efficiency of sequential DFS, and it can be mapped on to any MIMD architecture. In our experiments, we are able to achieve a speedup of over 100 on 128 processors on commercial multiprocessors such as BBN Butterfly and the Intel Hypercube. These results (as well as our analysis in [16]) clearly show that depth-first search can be speeded up by several orders of magnitude. The sequential DFS running on the CRAY-XMP (which has one of the fastest scalar CPUs) is only about 16 times faster than a single node of the Intel Hypercube. This shows that for problems that are not amenable to vector processing, multiprocessors can provide highly cost-effective computing power at low cost.

The architecture of the multiprocessor and the work distribution algorithm have been found to have significant impact on the performance of the parallel depth-first search algorithm. Other researchers[32, 21] considered the ring architecture to be quite suitable for parallel depth-first search. Our experimental results show that hypercube and shared-memory architectures are significantly better. In a companion paper[16], we analyze the effectiveness of different load-balancing schemes and architectures, and also present new improved work distribution schemes for ring and shared-memory architectures.

Acknowledgements: We would like to thank Sequent Computer Corp. for providing access to 30-processor Sequent Balance 21000, Intel Scientific Computers for access to a 128-processor Intel Hypercube, and Center for Automation Research, University of Maryland for access to 120-processor BBN Butterfly. Mohamed Gouda and Nathan Netanyahu provided useful comments on an earlier draft of the paper. K. Ramesh helped in the implementation of the parallel algorithm on Sequent Balance and BBN Butterfly.

References

[1] E. W. Dijkstra, W. H. Seijen, and A. J. M. Van Gasteren. Derivation of a termination detection algorithm for a distributed computation. *Information Processing Letters*, 16-5:217–219, 1983.

[2] Terry Disz, Ewing Lusk, and Ross Overbeek. Experiments with OR-parallel logic programs. In *Proceedings of the Fourth International Conference on Logic Programming*, volume 2, pages 576–600, 1987.

[3] O. I. El-Dessouki and W. H. Huen. Distributed enumeration on network computers. *IEEE Transactions on Computers*, C-29:818–825, September 1980.

[4] Raphael A. Finkel and Udi Manber. DIB - a distributed implementation of backtracking. *ACM Transactions of Programming Languages and Systems*, 9 No. 2:235–256, April 1987.

[5] Bogumil Hausman, Andrzej Ciepielewski, and Seif Haridi. OR-parallel PROLOG made efficient on shared memory multiprocessors. In *Proceedings of the Fourth Symposium on Logic Programming*, pages 69–79, 1987.

[6] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, MD, 1978.

[7] M. Imai, Y. Yoshida, and T. Fukumura. A parallel searching scheme for multiprocessor systems and its application to combinatorial problems. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 416–418, 1979.

[8] Virendra K. Janakiram, Dharma P. Agrawal, and Ram Mehrotra. Randomized parallel algorithms for prolog programs and backtracking applications. In *Proceedings of International Conference on Parallel Processing*, pages 278–281, 1987.

[9] S. Kasif, M. Kohli, and J. Minker. PRISM: A parallel inference system for problem solving. Technical report, Computer Science Department, University of Maryland, College Park, MD, 1983.

[10] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[11] Richard Korf. Optimal path finding algorithms. In L. N. Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence*. Springer-Verlag, New York, NY, 1988.

[12] V. Kumar and L. N. Kanal. A general branch-and-bound formulations for understanding and synthesizing and/or tree search procedures. *Artificial Intelligence*, 21:179–198, 1983.

[13] V. Kumar and L. N. Kanal. Parallel branch-and-bound formulations for and/or tree search. *IEEE Transactions Pattern Analysis and Machine Intelligence*, PAMI–6:768–778, 1984.

[14] Vipin Kumar. Depth-first search. In Stuart C. Shapiro, editor, *Encyclopaedia of Artificial Intelligence: Vol 2*, pages 1004–1005. John Wiley and Sons, New York, NY, 1987. Revised version appears in the second edition of the encyclopedia to be published in 1992.

[15] Vipin Kumar, Dana Nau, and L. N. Kanal. General branch-and-bound formulation for and/or graph and game tree search. In L. N. Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence*. Springer-Verlag, New York, NY, 1988.

[16] Vipin Kumar and V. Nageshwara Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16 (6):501–519, December 1987.

[17] K. Kumon, H. Masuzawa, A. Itashiki, K. Satoh, and Y. Sohma. Kabu-wake: A new parallel inference method and its evaluation. In *Proceedings of COMPCON 86*, March 1986.

[18] T. H. Lai and Sartaj Sahni. Anomalies in parallel branch and bound algorithms. In *Proceedings of International Conference on Parallel Processing*, pages 183–190, 1983.

[19] E. L. Lawler and D. Woods. Branch-and-bound methods: A survey. *Operations Research*, 14, 1966.

[20] Guo-Jie Li and Benjamin W. Wah. Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Transactions on Computers*, C–35, June 1986.

[21] B. Monien and O. Vornberger. The ring machine. Technical report, University of Paderborn, FRG, 1985. Also in *Computers and Artificial Intelligence*, 3(1987).

[22] Bernard Nadel. Tree search and arc consistency in constraint satisfaction algorithms. In L. N. Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, New York, NY, 1988.

[23] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

[24] Judea Pearl. *Heuristics-Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, 1984.

[25] V. Nageshwara Rao, V. Kumar, and K. Ramesh. A parallel implementation of iterative-deepening-a*. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, pages 878–882, 1987.

[26] V. Nageshwara Rao and Vipin Kumar. Superlinear speedup in state-space search. In *Proceedings of the 1988 Foundation of Software Technology and Theoretical Computer Science*, number 338 in Lecture Notes in Computer Science, pages 161–174. Springer-Verlag, 1988.

[27] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28–1:22–33, 1985.

[28] Kish Shen and David H.D. Warren. A simulation study of the Argonne model for OR-parallel execution of PROLOG. In *Proceedings of the Fourth Symposium on Logic Programming*, pages 54–68, September 1987. San Francisco, CA.

[29] M. E. Stickel and W. M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 1073–1075, 1985.

[30] Peter Tinker. Performance and pragmatics of an OR-parallel logic programming system. *International Journal of Parallel Programming*, ?, 1988.

[31] M. H. van Emden. An interpreting algorithm for prolog programs. In J.A. Campbell, editor, *Implementations of Prolog.* Ellis Horwood, West Sussex, UK, 1984.

[32] Benjamin W. Wah and Y. W. Eva Ma. Manip—a multicomputer architecture for solving combinatorial extremum-search problems. *IEEE Transactions on Computers*, c–33, May 1984.