On the Efficiency of Parallel Backtracking*

V. Nageshwara Rao[‡] and Vipin Kumar[§]

[‡] Department of Computer Sciences,
University of Central Florida,
Orlando, Florida 32816

[§]Computer Science Department
University of Minnesota,
Minneapolis, MN 55455

January 1992

Abstract

It is known that isolated executions of parallel backtrack search exhibit speedup anomalies. In this paper we present analytical models and experimental results on the average case behavior of parallel backtracking. We consider two types of backtrack search algorithms: (i) simple backtracking (which does not use any heuristic information); (ii) heuristic backtracking (which uses heuristics to order and prune search). We present analytical models to compare the average number of nodes visited in sequential and parallel search for each case. For simple backtracking, we show that the average speedup obtained is (i) linear when distribution of solutions is uniform and (ii) superlinear when distribution of solutions is non-uniform. For heuristic backtracking, the average speedup obtained is at least linear (i.e., either linear or superlinear), and the speedup obtained on a subset of instances (called difficult instances) is superlinear. We also present experimental results over many synthetic and practical problems on various parallel machines, that validate our theoretical analysis.

*This work was supported by Army Research Office grant # DAAG29-84-K-0060 to the Artificial Intelligence Laboratory, University of Texas at Austin and by Army Research Office grant # 28408-MA-SDI to the University of Minnesota and by the Army High Performance Computing Research Center at the University of Minnesota This paper will also appear in IEEE Transaction on Parallel and Distributed Systems.

1 Introduction

Consider the problem of finding a solution in a state-space tree containing one or more solutions[10, 28, 26, 6]. Backtracking, also called Depth-first Search, is a widely used technique for solving such problems because of its storage efficiency [13, 28]. Throughout the paper, we use the two names interchangeably. We use the acronym DFS to denote backtracking or depth-first search on state-space trees. There are many variants of DFS algorithms, each of which is tuned to certain types of problems. In this paper we deal with two important ones : (i) simple backtracking (which does not use any heuristic information); (ii) heuristic backtracking (which uses ordering and/or pruning heuristics to reduce search complexity).

A number of parallel formulations of DFS have been developed by various researchers[12, 7, 22, 2, 25, 23]. In one such formulation[23], N processors concurrently perform backtracking in disjoint parts of a state-space tree. The parts of the state-space searched by different processors are roughly of equal sizes. But the actual parts of the search space searched by different processors and the sequence in which nodes of these subspaces are visited are determined dynamically; and these can be different for different executions. As a result, for some execution sequences, the parallel version may find a solution by visiting fewer nodes than the sequential version thus giving superlinear speedup. (The speedup is defined as the ratio of the times taken by sequential and parallel DFS.) And for other execution sequences, it may find a solution only after visiting more nodes thus giving sublinear speedup). This type of behavior is common for a variety of parallel search algorithms, and is referred to as 'speedup anomaly' [18, 19]. The superlinear speedup in isolated executions of parallel DFS has been reported by many researchers [12, 25, 22, 7, 33, 20]. It may appear that on the average the speedup would be either linear or sublinear; otherwise, even parallel DFS executed on sequential processor via time-slicing would perform better than sequential DFS.

This paper considers the average case speedup anomalies in parallel DFS algorithms that are based on the techniques developed in [23, 17]. Though simple backtracking and heuristic backtracking algorithms we analyze here use a DFS strategy, their behavior is very different, and they are analyzed separately. We develop abstract models for the search spaces that are traversed by these two types of DFS algorithms. We analyze and compare the average number of nodes visited by sequential search and parallel search in each case. For simple backtracking, we show that the average speedup obtained is (i) linear when the distribution of solutions is uniform and (ii) superlinear when the distribution of solutions is non-uniform. For heuristic backtracking, the average speedup obtained is at least linear (i.e., either linear or superlinear), and the speedup obtained on a subset of instances (that are "difficult" instances) is superlinear. The theoretical analysis is validated by experimental analysis on example problems such as the problem of generating test-patterns for digital circuits[3], $N-$queens, 15-puzzle[26], and the hackers problem[31].

The result that "parallel backtrack search gives at least a linear speedup on the average" is important since DFS is currently the best known and practically useful algorithm to solve a number of important problems. The occurrence of consistent superlinear speedup on certain problems implies that the sequential DFS algorithm is suboptimal for these problems and that parallel DFS time-sliced on one processor dominates sequential DFS. This is highly significant because no other known search technique dominates sequential DFS for some of these problems. We have restricted our attention in this paper to state-space search on trees, as DFS algorithms are most effective for searching trees.

The overall speedup obtained in parallel DFS depends upon two factors: search overhead (defined as the ratio of nodes expanded by parallel and sequential search), and communication overhead (amount of time wasted by different processors in communication, synchronization, etc.). They are orthogonal in the sense that their causes are completely different. Search overhead is caused because sequential and parallel DFS search the nodes in a different order. Communication overhead is dependent upon the target architecture and the load balancing technique. The communication overhead in parallel DFS was analyzed in our previously published papers[17, 16, 4, 14], and was experimentally validated on a variety of problems and architectures.[1] In this paper, we only analyze search overhead. However, in the experiments, which were run only on real multiprocessors, both overheads are incurred. Hence the overall speedup observed in experiments may be less than linear (i.e., less than N on N processors) even if the model predicts that parallel search expands fewer nodes than sequential search. In parallel DFS, the effect of communication overhead is less significant for larger instances (i.e., for instances that take longer time to execute). Hence, the larger instances for each problem obey the analyses more accurately than the smaller instances. The reader should keep this in mind when interpreting the experimental results presented in this paper.

In Section 2, we briefly describe the two different kinds of DFS algorithms that are being analyzed in this paper. In Section 3, we review parallel DFS. Simple backtrack search algorithms are analyzed in Section 4 and ordered backtrack search algorithms are analyzed in 5. Section 6 contains related research and section 7 contains concluding remarks.

[1] In these experiments, parallel DFS was modified to find all optimal solutions, so that the number of nodes searched by sequential DFS and parallel DFS become equal, making search overhead to be 1.

2 Types of DFS algorithms

Consider problems that can be formulated in terms of finding a solution path in an implicit directed state-space tree from an initial node to a goal node. The tree is generated on the fly with the aid of a successor-generator function; given a node of the tree, this function generates its successors. Backtracking (i.e., DFS) can be used to solve these problems as follows. The search begins by expanding the initial node; i.e., by generating its successors. At each later step, one of the most recently generated nodes is expanded. (In some problems, heuristic information is used to order the successors of an expanded node. This determines the order in which these successors will be visited by the DFS method. Heuristic information is also used to prune some unpromising parts of a search tree. Pruned nodes are discarded from further searching.) If this most recently generated node does not have any successors or if it can be determined that the node will not lead to any solutions, then backtracking is done, and a most recently generated node from the remaining (as yet unexpanded) nodes is selected for expansion. A major advantage of DFS is that its storage requirement is linear in the depth of the search space being searched. The following are two search methods that use a backtrack search strategy.

1. **Simple Backtracking** is a depth-first search method that is used to find any one solution and that uses no heuristics for ordering the successors of an expanded node. Heuristics may be used to prune nodes of the search space so that search can be avoided under these nodes.

2. **Ordered Backtracking** is a depth-first search method that is used to find any one solution. It may use heuristics for ordering the successors of an expanded node. It may also use heuristics to prune nodes of the search space so that search can be avoided under these nodes. This method is also referred to as ordered DFS[13].

3 Parallel DFS

There are many different parallel formulations of DFS[7, 15, 19, 34, 2, 8, 23] that are suitable for execution on asynchronous MIMD multiprocessors. The formulation discussed here is used quite commonly [22, 23, 2, 4, 14]. In this formulation, each processor searches a disjoint part of the search space. Whenever a processor completely searches its assigned part, it requests a busy processor for work. The busy processor splits its remaining search space into two pieces and gives one piece to the requesting processor. When a solution is found by any processor, it notifies all the other processors. If the search space is finite and has no solutions, then eventually all the processors would run out of work, and the search (sequential or parallel) will terminate without finding any solution. In backtrack search algorithms, the search terminates after the whole search space is exhausted (i.e., either searched or pruned).

The parallel DFS algorithms we analyze theoretically differ slightly from the above description as follows. For simplicity, the analyses of the models assumes that an initial static partitioning of the search space is sufficient for good load balancing. But in the parallel DFS used in all of our experimental results, the work is partitioned dynamically. The reader will see that there is a close agreement between our experiments and analyses.

4 Analysis of Speedup in Simple Backtracking with No Heuristic Information

4.1 Assumptions and Definitions

The state-space tree has M leaf nodes, with solutions occurring only among the leaf nodes. The amount of computation needed to visit each leaf node is the same. The execution time of a search is proportional to the number of leaf nodes visited. This is not an unreasonable assumption, as on search trees with branching factor greater than one, the number of nodes visited by DFS is roughly proportional to the number of leaves visited. Also, in this section we don't model the effect of pruning heuristic explicitly. We assume that M is the number leaf nodes in the state-space tree that has already been pruned using the pruning function.

Both sequential and parallel DFS stop after finding one solution. In parallel DFS, the state-space tree is equally partitioned among N processors, thus each processor gets a subtree of with $\frac{M}{N}$ leaf nodes. There is at least one solution in the entire tree (otherwise both parallel search and sequential search would visit the entire tree without finding a solution, resulting in linear speedup). There is no information to order the search of the state-space tree; hence the density of solutions across the search frontier is independent of the order of the search. Solution density ρ of a leaf node is the probability of the leaf node being a solution. We assume a Bernoulli distribution of solutions; i.e., the event of a leaf node being a solution is independent of any other leaf node being a solution. We also assume that $\rho << 1$.

W_N denotes the average of the total number of nodes visited by N processors before one of the processors finds a solution. W_1 is the average number of leaf nodes visited by sequential DFS before a solution is found. Clearly, both W_1 and W_N are less than or equal to M.

Since the execution time of a search (in the sequential as well as parallel case) is proportional to the number of nodes expanded,

$$Speedup\ S \simeq \frac{W_1}{W_N} \times N$$

Efficiency E, is the speedup divided by N. E denotes the effective utilization of computing resources.

$$E = \frac{S}{N} \simeq \frac{W_1}{W_N}$$

4.2 Efficiency Analysis

Consider the search frontier of M leaf nodes being statically divided into N regions, each with $K = \frac{M}{N}$ leaves. Let the density of solutions among the leaves in the i^{th} region be ρ_i. In the parallel case, processor i searches region i, independently until one of the processors finds a solution. In the sequential case, the regions are arranged in a random sequence and searched in that order.

Theorem 4.1 *If $\rho(> 0)$ is the density in a region and number of leaves K in the region is large , then mean number of leaves visited by a single processor searching the region is $\frac{1}{\rho}$.*

Proof: Since we have a Bernoulli distribution,

$$\begin{aligned} Mean\ number\ of\ trials\ &= 1 \times \rho + 2 \times (1-\rho) \times \rho + 3 \times (1-\rho)^2 \times \rho + \cdots + K \times (1-\rho)^K \times \rho \\ &= \frac{1-(1-\rho)^{K+1}}{1-(1-\rho)} - K(1-\rho)^{K+1} \\ &= \frac{1}{\rho} - (1-\rho)^{K+1}(\frac{1}{\rho} - K) \end{aligned}$$

For large enough K, the second term in the above becomes less than 1; hence,

$$Mean\ number\ of\ trials\ \simeq \frac{1}{\rho}$$

□

Sequential DFS selects any one of the N regions with probability $1/N$, and searches it to find a solution. Hence the average number of leaf nodes expanded by sequential DFS is[2]

$$W_1 \simeq \frac{1}{N}(\frac{1}{\rho_1} + \frac{1}{\rho_2} + \cdots + \frac{1}{\rho_N})$$

[2]The given expression assumes that a solution is always found in the selected region, and thus only one region has to be searched. But with probability $(1-\rho_i)^K$, region i does not have any solution, and another region would need to be searched. Taking this into account would make the expression for W_1 more precise, and and increase the average value of W_1 somewhat. But the reader can verify that the overall results of the analysis will not change.

In each step of parallel DFS, one node from each of the N regions is explored simultaneously. Hence the probability of success in a step of the parallel algorithm is $1-\prod_{i=1}^{i=N}(1-\rho_i)$. This is approximately $\rho_1+\rho_2+\cdots+\rho_N$ (neglecting the second order terms since ρ_i's are assumed to be small). Hence

$$W_N \simeq \frac{N}{\rho_1+\rho_2+\cdots+\rho_N}$$

Inspecting the above equations, we see that $W_1 = \frac{1}{HM}$ and $W_N = \frac{1}{AM}$, where HM is the harmonic mean of the $\rho' s$; AM is their arithmetic mean. Since we know that arithmetic mean (AM) and harmonic mean (HM) satisfy the relation : $AM \geq HM$, we have $W_1 \geq W_N$. In particular,

- when ρ_i's are equal, $AM = HM$, therefore $W_1 \simeq W_N$. When solutions are uniformly distributed, the average speedup for parallel DFS is linear.
- when they are different, $AM > HM$, therefore $W_1 > W_N$. When solution densities in different regions are non-uniform, the average speedup for parallel DFS is superlinear.

The assumption that "the event of each node being a solution is independent of the event of other nodes being a solution" is unlikely to be true for practical problems. Still, the above analysis suggests that parallel DFS can obtain higher efficiency than sequential DFS provided that solutions are not distributed uniformly in the search space, and no information about densities in different regions is available. This characteristic happens to be true for a variety of problem spaces searched by simple backtracking.

4.3 Experimental Results

We present experimental results of the performance of parallel DFS on three problems: (i) the hacker's problem [31] (ii) the 15-puzzle problem[26] and (iii) the $N-$queens problem[6].

In all the experiments discussed in this section, both sequential and parallel DFS visit the newly generated successors of a node in a random order. This is different from "conventional" DFS in which successors are visited in some statically defined "left to right" order or in "ordered" DFS in which successors are visited in some heuristic order. We visit the successors in a random order (rather than left-to-right or heuristic order) because we are trying to validate our model which assumes that no heuristic information is available to order the nodes - hence any random order is as good as any other.[3] To get the average run time, each

[3]As an aside, the reader should note that the ordering information does not always improve the effectiveness of DFS. For example, experience with the IDA* algorithm on the 15-puzzle problem [11] indicates that

experiment is repeated many times. Note that, besides the random ordering of successors, there is another source of variability in execution time of parallel DFS. This is because the parts of the state-space tree searched by different processors are determined dynamically, and are highly dependent on run time events beyond programmers control. Hence, for parallel DFS, each experiment is repeated even more frequently than it is for sequential DFS.

The hacker's problem involves searching a complete binary tree in which some of the leaf nodes are solution nodes. The path to a solution node represents a correct password among the various binary sequences of a fixed length. There may be more than one solution, due to wild card notation. We implemented a program on Sequent Balance 21000 multiprocessor and experimented with different cases for up to 16 processors. Experiments were done with two different kinds of trees. In one case, one or more solutions are distributed uniformly in the whole search space. This corresponds to the case in which the branching points due to wild cards are up in the tree (more than 1 solutions) or there are no wild cards (exactly one solution). In this case, as predicted by our analysis, sequential search and parallel search do approximately the same amount of work, hence the speed up in Parallel DFS is linear. This case corresponds to the curve labeled as 1 in Figure 1. The efficiency starts decreasing beyond 8 processors because communication overhead is higher for larger number of processors.

In the second case, four solutions were distributed uniformly in a small subspace of the total space and this subspace was randomly located in the whole space. This corresponds to the case in which branching point due to characters denoted as wild cards are low in the tree. In this case, as expected, the efficiency of parallel DFS is greater than 1, as the regions searched by different processors tend to have different solution densities. The results are shown in Figure 1. The fractions indicated next to each curve denotes the size of the subspace in which solutions are located. For example, $\frac{1}{8}$ means that the curve is for the case in which solutions are located (randomly) only in $\frac{1}{8}$ of the space . In Figure 1, the reader would notice that there is a peak in efficiency at r processors for the case in which solutions are distributed in $\frac{1}{r}$ fraction of the search space (see the curves labeled 1/8 and 1/16). This happens because it is possible for one of the r processors to receive the region containing all or most of the solutions, thus giving it a substantially higher density region compared with other processors. Of course, since the search space in the experiments is distributed dynamically, this above best case happens only some of the times, and the probability of its occurrence becomes smaller as the number of processors increases. This is why the heights of the peaks in the successive curves for decreasing values of $1/r$ are also decreasing.

---

the use of the Manhattan distance heuristic for ordering (which is the best known admissible heuristic for 15-puzzle) does not make DFS any better. On the other hand, in problems such as $N-$queens, the ordering information improves the performance of DFS substantially[9, 32].

The experiments for 15-puzzle were performed on the BBN Butterfly parallel processor for up to 9 processors. The experiments involved instances of the 15-puzzle with uniform distribution and non-uniform distribution of solutions. Depth bounded DFS was used to limit the search space for each instance. Sequential DFS and parallel DFS were both executed with a depth-bound equal to the depth of the shallowest solution nodes. The average timings for sequential and parallel search were obtained by running each experiment 100 times (for every 15-puzzle instance). Figure 2 shows the average speedups obtained. The instances with uniform distribution of solutions show a near linear speedup. The maximum deviation of speedup is indicated by the banded region. The width of the banded region is expected to reduce if a lot more repetitions (say a 1000 for every instance) were tried. The instances with non-uniform distribution of solutions give superlinear speedups.

Figure 3 show the efficiency versus the number of processors for the $N-$queens problem. The problem is naturally known to exhibit non uniformity in solution density[8]. Each data point shown was obtained by averaging over 100 trials. As we can see, parallel DFS exhibits better efficiency than sequential DFS. As the number of processors is increased for any fixed problem size, the efficiency goes down because the overhead for parallel execution masks the gains due to parallel execution. We expect that on larger instances of such problems, parallel DFS will exhibit superlinear speedup even for larger number of processors.

All of these experiments confirm the predictions of the model. Superlinear speedup occurs in parallel DFS if the density of solutions in the regions searched by different processors are different. Linear speedup occurs when solutions are uniformly distributed in the whole search space, which means that solution density in regions searched by different processors in parallel is the same.

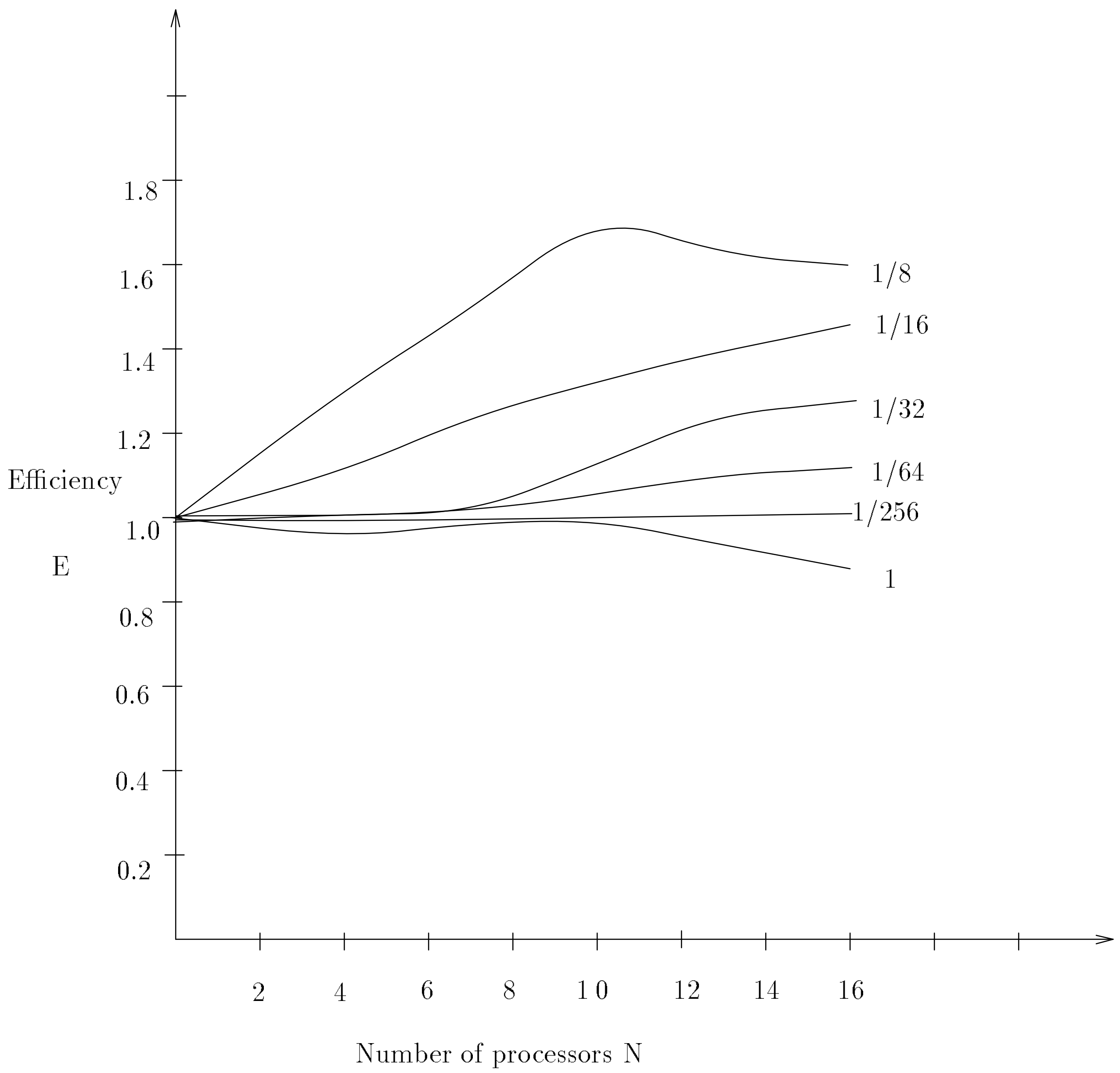

Figure 1: Efficiency curves for the hackers problem. An efficiency greater than 1 indicates superlinear speedup.

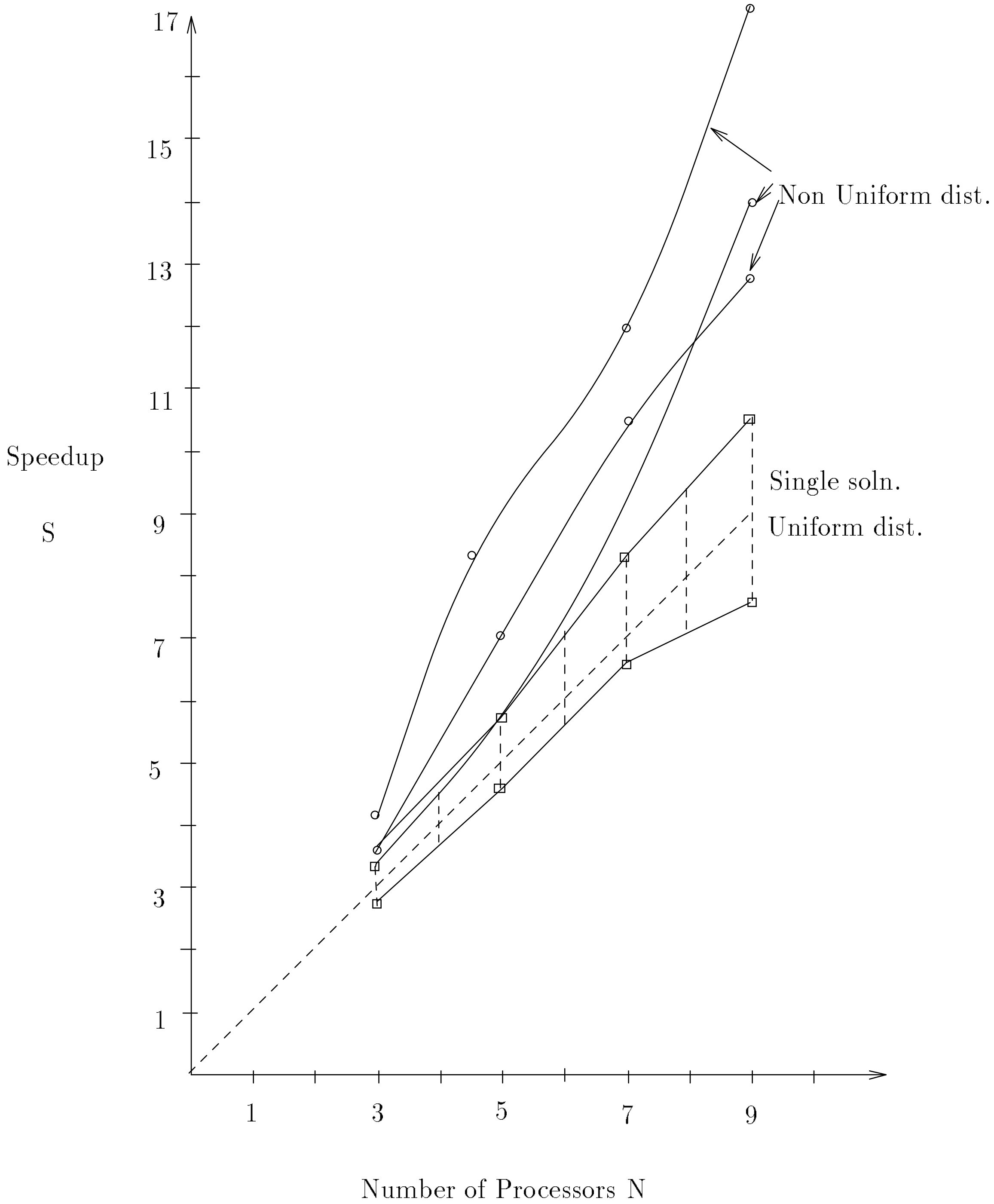

Figure 2: Speedup curves for the 15-puzzle problem.

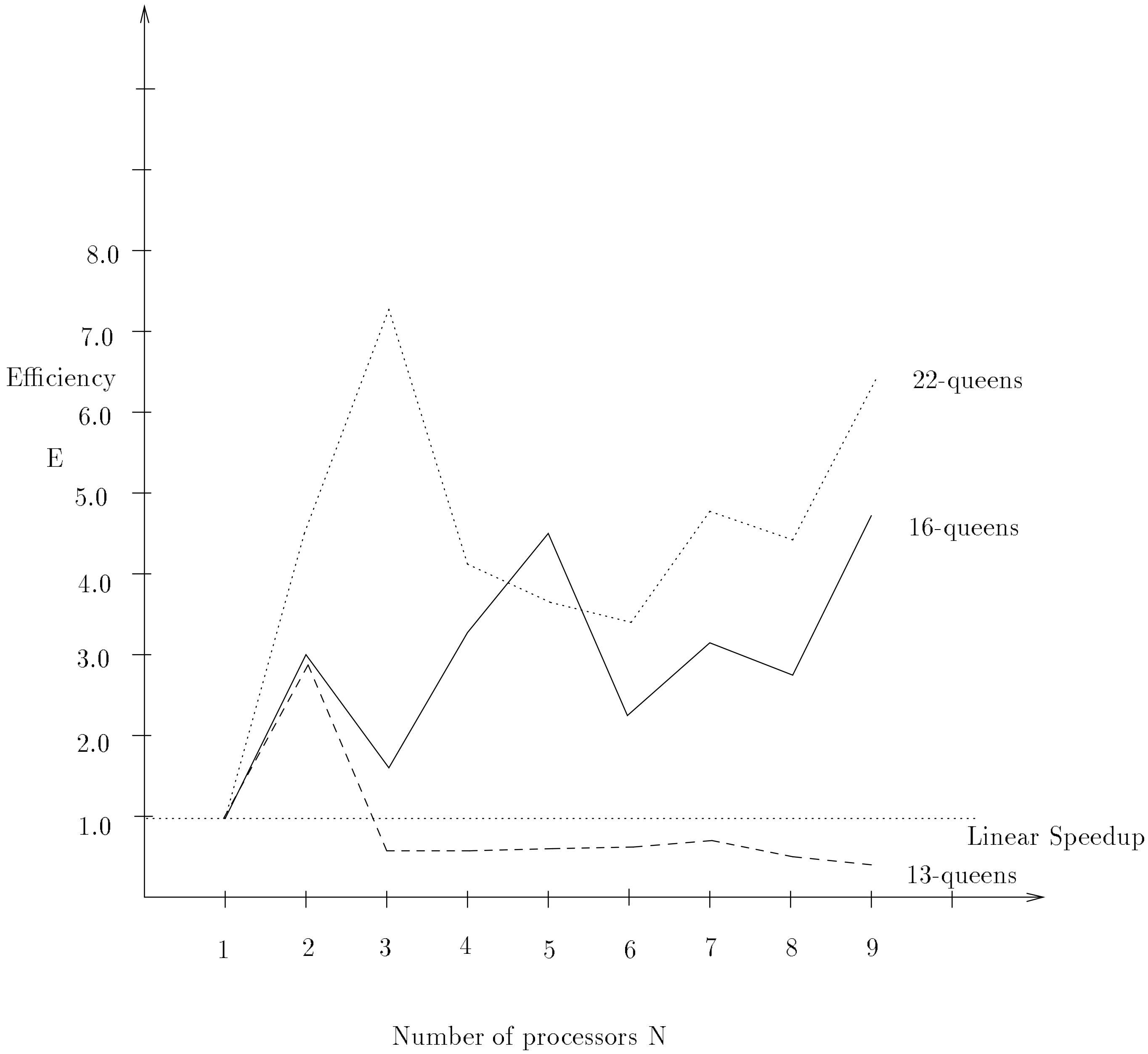

Figure 3: Efficiency Curves for the N-Queens problem.

5 Speedup in Parallel Ordered Depth-First Search

5.1 Assumptions and Definitions

We are given a balanced binary tree of depth d. The tree contains $2^{d+1} - 1$ nodes of which 2^d are leaf nodes. Some of the $2^{d+1} - 1$ nodes are solution nodes. We find one of these solutions by traversing the tree using (sequential or parallel) DFS. A bounding heuristic is available that makes it unnecessary to search below a non-leaf node in either of the following two cases:

1. It identifies a solution that can be reached from that node or even identifies the node to be a solution.

2. It identifies that no solution exists in the subtree rooted at that node, and thus makes it unnecessary to search below the node.

When a bounding heuristic succeeds in pruning a non-leaf node, there is no need to search further from that node. (If a node is a non-leaf node and if the bounding heuristic does not succeed, then the search proceeds as usual under the node.) We characterize a bounding heuristic by its success rate $(1 - \nu)$; i.e. the probability with which the procedure succeeds in pruning a node. For the purposes of our discussion we shall assume $0.5 < \nu < 1.0$. This ensures that the effective branching factor is greater than 1. For $\nu \leq 0.5$, the search complexity becomes insignificant.

Consider a balanced binary tree of depth k that has been pruned using the $(1 - \nu)$–bounding heuristic. Let $F(k)$ be the number of leaf nodes in this tree. Clearly, $F(k)$ would be no more than than 2^k.

If our given tree has no solutions, then DFS will visit $F(d)$ leaf nodes. If the tree has one or more solutions, then DFS will find a solution by visiting fewer leaf nodes than $F(d)$. The actual number of leaf nodes visited will depend upon the location of the left most solution in the tree, which in turn will depend upon the order in which successors of nodes are visited. In an extreme case, if the "correct" successor of each node is visited first by DFS, then a solution will be found by visiting exactly one leaf node (as the left most node of the search tree will be a solution). In practice, an ordering heuristic is available that aids us in visiting the more promising node first and postponing the visit to the inferior one (if necessary) later. We characterize an ordering heuristic by a parameter γ. The heuristic makes the correct choice in ordering with a probability γ; i.e., γ-fraction of the time, subtree containing a solution is visited, and the remaining $(1 - \gamma)$-fraction of the time, subtree containing no solution is visited. Obviously, it only makes sense to consider $0.5 < \gamma < 1.0$. $\gamma \leq 0.5$ means that the heuristic provides worse information than a random coin toss. When $\gamma = 1.0$, the ordering is perfect and the solution is found after visiting only 1 leaf node.

We shall refer to these trees as *OB-trees* (ordered-bounded trees), as both ordering and bounding information is available to reduce the search. To summarize, *OB-trees* model search problems where bounding and/or ordering heuristics are available to guide the search and their error probability is constant. The reader should be cautioned that for some problems, this may not necessarily be true.

5.2 Efficiency Analysis

We now analyze the average number of leaf nodes visited by the sequential and parallel DFS algorithms on OB-trees. Let $S(d)$ be the average number of leaf nodes (pruned nodes or terminal nodes) visited by sequential search and $P(d)$ be the sum of the average number of leaf nodes visited by each processor in parallel DFS.

Theorem 5.1 $F(k) = (2\nu)^k + (1-\nu)\frac{(2\nu)^k - 1}{2\nu - 1}$

Proof: See the proof of Theorem A.1 in Appendix A.
□

Theorem 5.2 $S(d) \approx (1-\gamma)F(d) + 1$.

Proof :

See the proof of Theorem A.2 in Appendix A.
□

Thus the use of ordering and bounding heuristics in sequential DFS cuts down the effective size of the original search tree by a large factor. The bounding heuristic reduces the effective branching factor from 2 to approximately 2ν. The ordering heuristic reduces the overall search effort by a factor of $(1-\gamma)$. Even though OB-trees are complete binary trees, a backtrack algorithm that uses the pruning heuristic reduces the branching factor to less than 2.

Now let's consider the number of nodes visited by parallel DFS. Clearly, the bounding heuristic can be used by parallel DFS just as effectively (as it is used in sequential DFS). However, it might appear that parallel DFS cannot make a good use of the ordering heuristic, as only one of the processors will work on the most promising part of the space whereas other processors will work on less promising parts. But the following theorem says that this intuition is wrong.

Theorem 5.3 *For OB-trees, parallel DFS expands no more nodes on the average than sequential DFS.*

Proof: We first consider a two processor parallel DFS and later generalize the result. In two–processor parallel DFS, the tree is statically partitioned at the root and the processors search two trees of depth $d-1$ independently until at least one of them succeeds. Each of them individually performs the sequential DFS with the help of bounding and pruning heuristics. Note that though the second processor violates the advice of the ordering heuristic at the root node, it follows its advice everywhere else. Consider the case in which the root node is not pruned by the bounding heuristic. Now there are two possible cases:

Case 1: Solution exists in the left subtree. This case happens γ-fraction of the time. In this case, sequential DFS visits $S(d-1)$ leaf nodes on the average, whereas parallel DFS visits at most $2S(d-1)$ leaf nodes. If the left subtree also has a solution, then parallel DFS visits exactly $2S(d-1)$ leaf nodes on the average. Otherwise (if both subtrees have a solution), then the average work done in parallel DFS will be smaller.

Case 2: Solution does not exist in the left subtree (i.e., it exists in the right subtree). This case happens $(1-\gamma)$-fraction of the time. In this case, sequential DFS visits $F(d-1)+S(d-1)$ leaf nodes on the average, whereas parallel DFS visits exactly $2S(d-1)$ leaf nodes.

Thus γ-fraction of the time parallel DFS visits at most $S(d-1)$ extra nodes, and $(1-\gamma)$-fraction of the time it visits $F(d-1)-S(d-1)$ fewer nodes than sequential DFS.

Hence, on the average (ignoring the case in which solution is found at the root itself[4]),

$$P(d) - S(d) \leq \gamma S(d-1) - (1-\gamma)(F(d-1) - S(d-1))$$

$$= S(d-1) - (1-\gamma)F(d-1)$$

$$\approx 0 \text{ by Theorem A.2}$$

This result is extended to the case where we have 2^a processors performing the parallel search, in the following theorem.

Theorem 5.4 *If P^a is the number of nodes expanded in parallel search with 2^a processors, where $a > 0$, then $P^a \leq P^{a-1}$.*

Proof: This theorem compares the search efficiency when 2^{a-1} processors are being used to that when 2^a processors are being used. In the first case, the entire search tree is being split into 2^{a-1} equal parts near the root and each such part is searched by one processor. In the

[4]When the root is pruned we can ignore the difference between $P(d)$ and $S(d)$, as the tree has only one node.

second case, each of these is again being split into two equal parts and two processors share the work that one processor used to do. Let us compare the number of nodes expanded by one processor in the first case with the corresponding pair of processors in the second case. We know that the subtree we are dealing with is an OB-tree. There fore, theorem 5.3 shows that the pair of processors do at most as much work as the single processor in the first case. By summing over all 2^{a-1} parts in the whole tree, the theorem follows. An induction on a with this theorem shows that $P(d) \leq S(d)$ holds for the case of 2^k processors performing the parallel search.

□

5.2.1 Superlinear Speedup on Hard to Solve Instances

Theorem 5.3 has the following important consequence. If we partition a randomly selected set of problem instances into two subsets such that on one subset the average speedup is sublinear, then the average speedup on the other one will be superlinear. One such partition is according to the correctness of ordering near the root. Let us call those instances on which the ordering heuristic makes correct decision near the root, **easy-to-solve instances** and the others, **hard-to-solve instances**. For sequential DFS, easy-to-solve instances take smaller time to solve than hard-to-solve instances.

For the 2-processor case, easy-to-solve instances are those γ-fraction of the total instances in which the ordering heuristic makes correct decisions at the root. On these, parallel version obtains an average speedup of 1 (i.e., no speedup). On the remaining instances, the average speedup is roughly $\frac{2-\gamma}{1-\gamma}$, which can be arbitrarily high depending upon how close γ is to 1. On 2^a processors, the easiest to solve instances are those γ^a-fraction of the total instances on which sequential search makes correct decision on the first a branches starting at the root. The maximum superlinearity is available on the hardest to solve instances which are a fraction $((1-\gamma)^a)$ of the total instances.

5.3 Experimental Results

The problem we chose to experiment with is the test generation problem that arises in computer aided-design (CAD) for VLSI. The problem of Automatic Test Pattern Generation (ATPG) is to obtain a set of logical assignments to the inputs of an integrated circuit that will distinguish between a faulty and fault-free circuit in the presence of a set of faults. An input pattern is said to be a test for a given fault if, in the presence of the fault, it produces an output that is different for the faulty and fault-free circuits. We studied sequential and parallel implementations of an algorithm called PODEM (Path-Oriented Decision Making

[3]) used for combinational circuits (and for sequential circuits based on the level-sensitive scan design approach). This is one of the most successful algorithms for the problem and it is widely used. The number of faults possible in a circuit is proportional to the number of signal lines in it. It is known that the sequential algorithm is able to generate tests for more than 90% of the faults in reasonable time but spends an enormous amount of time (much more than 90% of execution time) trying to generate tests for the remaining faults. As a result, the execution of the algorithm is terminated when it fails to generate a test after a predefined number of node expansions or **backtracks**. Those faults that cannot be solved in reasonable time by the serial algorithm are called *hard-to-detect* (HTD) faults[27]. In practice, it is very important to generate tests for as many faults as possible. Higher fault coverage results in more reliable chips. The ATPG problem fits the model we have analyzed very well for the following reasons: (i) the search tree generated is binary; (ii) For a non-redundant fault, the problem typically has one or a small number of solutions; (iii) a good but imperfect ordering heuristic is available; (iv) a bounding heuristic is available which prunes the search below a node when either the pruned node itself is a solution or it can no longer lead to solutions.

Our experiments with the ATPG problem support our analysis that on hard-to-solve instances, the parallel algorithm shows a superlinear speedup. We implemented sequential and parallel versions of PODEM on a 128 processor Symult 2010 multiprocessor. We performed an experiment using the ISCAS-85 benchmark files as test data. More details on our implementation and experimental results can be found in [1]. Our experiments were conducted as follows. The HTD faults were first filtered out by picking those faults from the seven files whose test patterns could not be found within 25 backtracks using the sequential algorithm. The serial and parallel PODEM algorithms were both used to find test patterns for these HTD faults. Since some of these HTD faults may not be solvable (by the sequential and/or parallel PODEM algorithm) in a reasonable time, an upper limit was imposed on the total number of backtracks that a sequential or parallel algorithm could make. If the sequential or parallel algorithm exceeded this limit (the sum of backtracks made by all processors being counted in the parallel case), then the algorithm was aborted, and the fault was classified as undetectable (for that backtrack limit). Time taken by purely sequential PODEM and the parallel PODEM were used to compute the speedup. These results are shown in Figure 4 for each circuit in the ISCAS-85 benchmark. In these experiments, 25600 was used as the upperlimit for backtracks.

To test the variation of superlinearity with hardness of faults, we selected two sets of faults. The first set consisted of those faults that the serial algorithm was able to solve after executing a total number of backtracks (node expansions) in the range 1600-6400. Similarly,

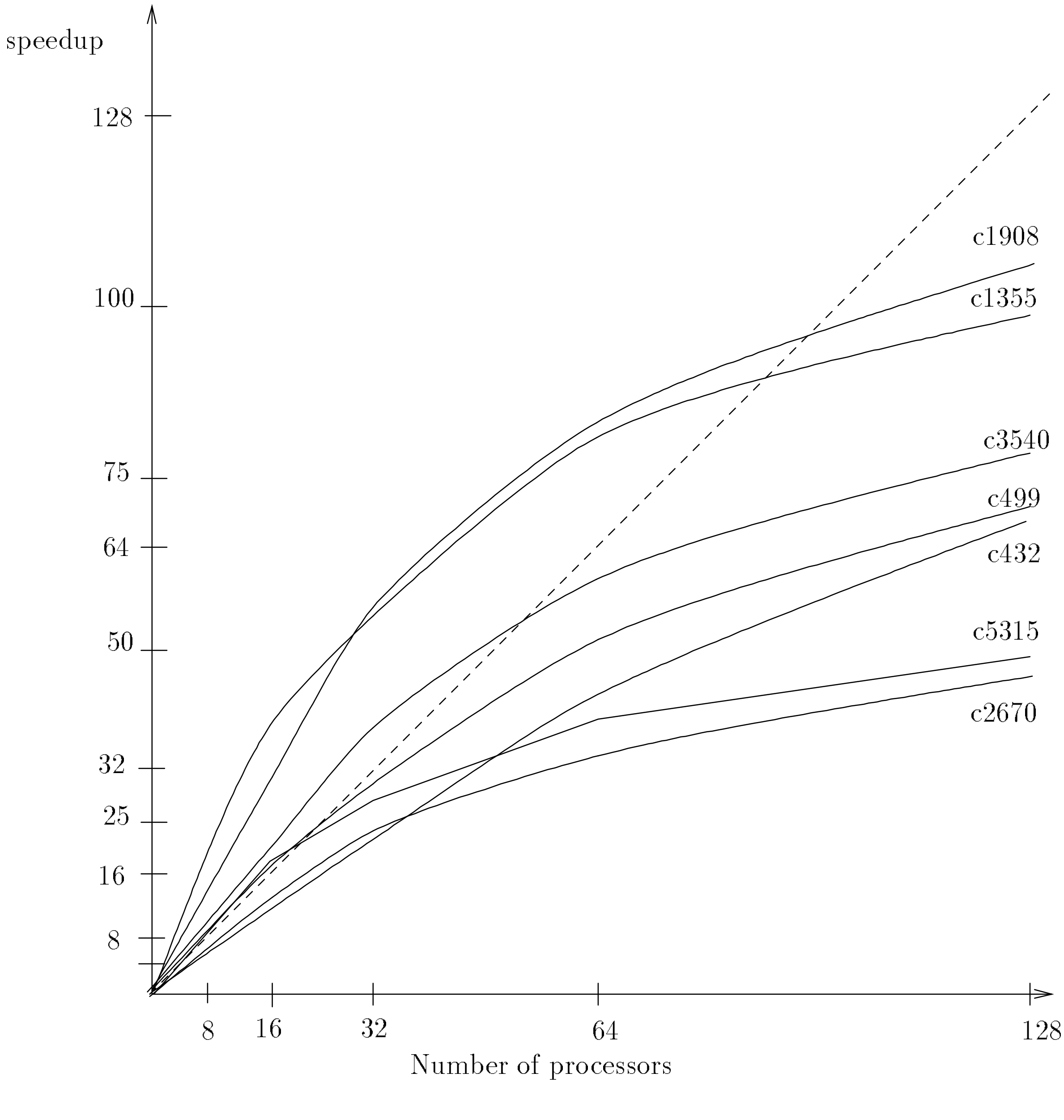

Figure 4: Speedup Curves for the ATPG problem

the second set of faults was solved by the serial algorithm in the backtrack range 6400-25600. The faults in the second set were thus harder to solve for the serial algorithm. Two of the seven files, namely c499 and c1355, did not yield any faults for either of the two sets. We executed the parallel algorithm for 16 to 128 processors and averaged the speedups obtained for a given number of processors separately for the two sets of faults. The run-time for each fault was itself the average obtained over 10 runs. These results are shown in Figure 5. From all these results, it is clear that superlinearity increases with the increasing hardness of instances. The degree of superlinearity decreases with the increasing number of processors because the efficiency of parallel DFS decreases if the problem size is fixed and the number of processors is increased[16].

Note that the above experimental results validate only the discussion in Section 5.2.1. To validate Theorem 5.3, it would be necessary to find the number of nodes expanded by parallel DFS even for easy to detect faults. For such faults, experimental run time will not be roughly proportional to the number of nodes searched by parallel DFS because for small trees communication overhead becomes significant. Superlinearity for hard-to-detect faults was experimentally observed for other ATPG heuristics by Patil and Banerjee in [27].

6 Related Research

The occurrence of speedup anomalies in simple backtracking was studied in [22] and [8]. Monien, et. al.[22] studied a parallel formulation of DFS for solving the satisfiability problem. In this formulation, each processor tries to prove the satisfiability of a different subformula of the input formula. Due to the nature of the satisfiability problem, each of these subformulas leads to a search space with a different average density of solutions. These differing solution densities are responsible for the average superlinear speedup. In the context of a model, Monien et al. showed that it is possible to obtain (average) superlinear speedup for the SAT problem. Our analysis of simple backtracking (in Section 4) is done for a similar model, but our results are general and stronger. We had also analyzed the average case behavior of parallel (simple) backtracking in [24]. The theoretical results we present here are much stronger than those in [24]. In [24], we showed that if the regions searched by a few of the processors had all the solutions uniformly distributed and the regions searched by all the rest of the processors had no solutions at all, the average speedup in parallel backtracking would be superlinear. Our analysis in Section 4 shows that any non-uniformity in solution densities among the regions searched by different processors leads to a superlinear speedup on the average. The other two types of heuristic DFS algorithms we discuss are outside the scope of both [22] and [24].

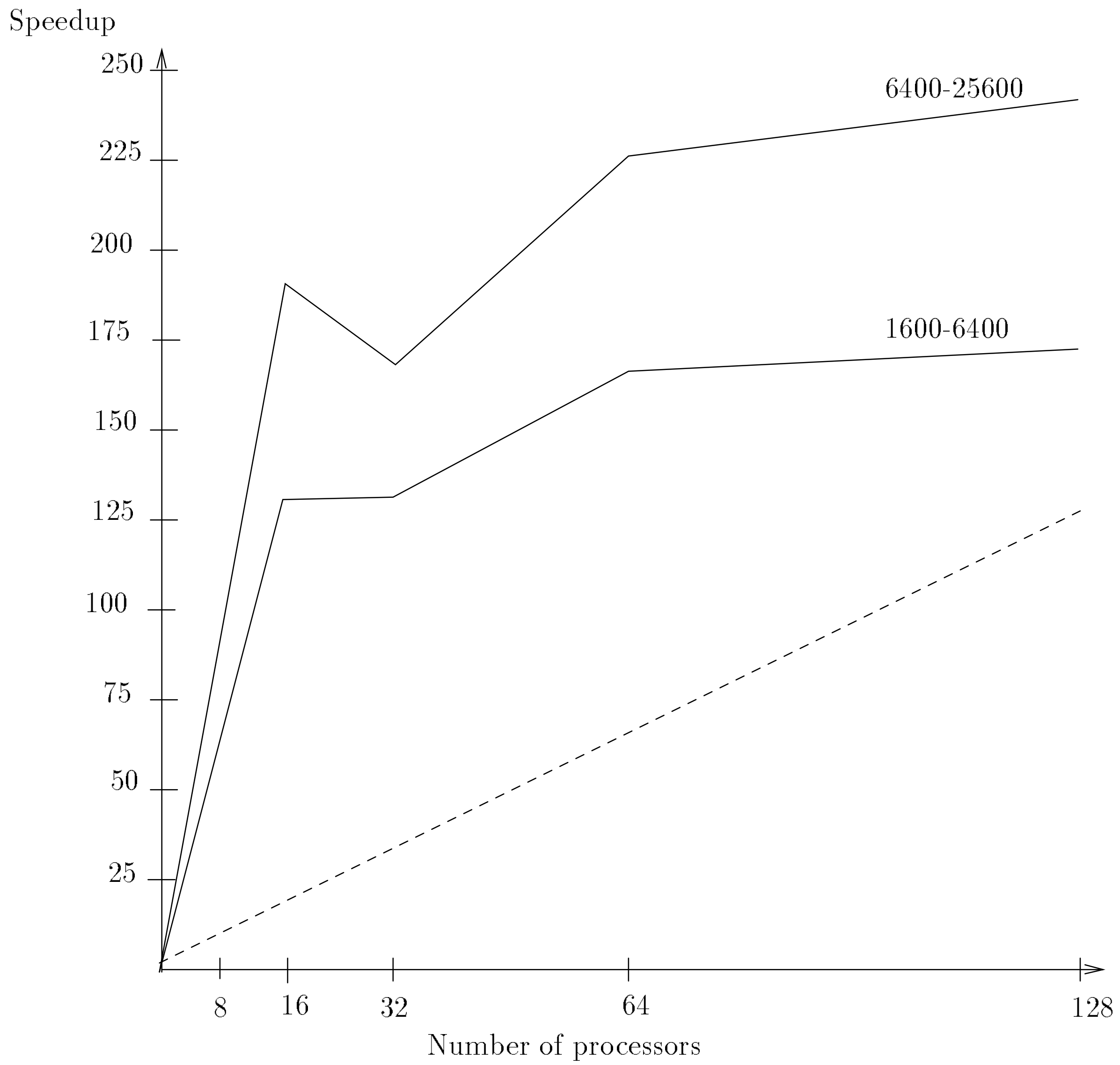

Figure 5: Speedup curves for hard-to-solve instances of the ATPG problem

If the search space is searched in a random fashion (i.e., if newly generated successors of a node are ordered randomly) , then the number of nodes expanded before a solution is found is a random variable (let's call it T(1)). One very simple parallel formulation of DFS presented in [21, 8] is to let the same search space be searched by many processors in an independent random order until one of the processors finds a solution. The total number of nodes expanded by a processor in this formulation is again a random variable (let's call it T(N)). Clearly, T(N) = $\min\{V_1,...,V_N\}$, where each V_i is a random variable. If the average value of T(N) is less than $\frac{1}{N}$ times T(1), then also we can expect superlinear speedup[21, 8]. For certain distributions of T(1), this happens to be the case. For example, if the probability of finding a solution at any level of the state-space tree is the same, then T(1) has this property[21]. Note that our parallel formulation of DFS dominates the one in [21, 8] in terms of efficiency, as in our parallel formulation there is no duplication of work. Hence, our parallel formulation will exhibit superlinear speedup on any search space for which the formulation in [21, 8] exhibits superlinear speedup, but the converse is not true.

For certain problems, probabilistic algorithms[29][5] can perform substantially better than simple backtracking. For example, this happens for problems in which the state-space tree is a balanced binary tree (like in the Hacker's Problem discussed in Section 4.3), and if the overall density of solutions among the leaf nodes is relatively high but solutions are distributed nonuniformly. Probabilistic search performs better than simple backtracking for these problems because it can make the density of solutions at the leaf nodes look virtually uniform. The reader can infer from the analysis in Section 4 that for these kinds of search spaces, parallel DFS also obtains similar homogenization of solution density, even though each processor still performs enumeration of (a part of) the search space. The domain of applicability of the two techniques (probabilistic algorithms vs sequential or parallel DFS) however is different. When the depth of leaf nodes in a tree varies, a probabilistic search algorithm visits shallow nodes much more frequently than deep nodes. In the state-space tree of many problems (such as the $N-$queens problem), shallow nodes correspond to failure nodes, and solution nodes are located deep in the tree. For such problems, probabilistic algorithms do not perform as well as simple backtracking, as they visit failure nodes more frequently. (Note that the simple backtracking will visit a failure node exactly once.) When the density of solutions among leaf nodes is low, the expected running time of a probabilistic algorithm can also be very high. (In the extreme case, when there is no solution, the probabilistic search will never terminate, whereas simple backtracking and our parallel DFS will.) For these cases also, an enumerative search algorithm such as simple backtracking

[5]A probabilistic algorithm for state space search can be obtained by generating random walks from root node to leaf nodes until a solution is found

is superior to a probabilistic algorithm, and the parallel variant retains the advantages of homogenization.

For ordered DFS, randomized parallel DFS algorithms such as those given in [21, 8] will perform poorly, they are not able to benefit from the ordering heuristics. Probabilistic algorithms have the same weakness. For decision problems, our analysis in Section 5 shows that the utilization of ordering heuristic cuts the search down by a large factor for both the sequential DFS and our parallel DFS. In the case of optimization problems (i.e., when we are interested in finding a least-cost solution), randomized parallel DFS algorithms in [21, 8] as well as probabilistic algorithms are not useful. One cannot guarantee the optimality of a solution unless an exhaustive search is performed.

Saletore and Kale [30] present a parallel formulation of DFS which is quite different than the ones in [22, 23, 2]. Their formulation explicitly ensures that the number of nodes searched by sequential and parallel formulations are nearly equal. The results of our paper do not apply to their parallel DFS.

In [5], a general model for explaining the occurrence of superlinear speedups in a variety of search problems is presented. It is shown that if the parallel algorithm performs less work than the corresponding sequential algorithm, superlinear speedup is possible. In this paper, we identified and analyzed problems for which this is indeed the case.

7 Conclusions

We presented analytical models and theoretical results characterizing the average case behavior of parallel backtrack search (DFS) algorithms. We showed that on average, parallel DFS does not show deceleration anomalies for two types of problems. We also presented experimental results validating our claims on multiprocessors. Further, we identified certain problem characteristics which lead to superlinear speedups in parallel DFS. For problems with these characteristics, the parallel DFS algorithm is better than the sequential DFS algorithm even when it is time-sliced on one processor. While isolated occurrences of speedup anomalies in parallel DFS had been reported earlier by various researchers, no experimental or analytical results showing possibility of superlinear speedup on the average (with the exception of results in [22, 24]) were available for parallel DFS.

A number of questions need to be addressed by further research. On problems for which sequential DFS is dominated by parallel DFS but not by any other search technique, what is best possible sequential search algorithm? Is it the one derived by running parallel DFS on one processor in a time slicing mode? If yes, then what is the optimum number of processors to emulate in this mode? In the case of ordered backtrack search, we showed that parallel search is more efficient on **hard-to-solve** instances while sequential search is more efficient on **easy-to-solve** instances. In practice, one should therefore use a combination of sequential and parallel search. What is the optimal combination? This paper analyzed efficiency of parallel DFS for certain models. It would be interesting to perform similar analysis for other models and also for other parallel formulations of backtrack search such as those given in [7] and in [30].

Acknowledgements: We would like to thank Sunil Arvindam and Hang He Ng for helping us with some of the experiments. We would also like to thank Dr. James C. Browne and Dr Vineet Singh for many helpful discussions.

APPENDICES

A Details of Analysis for Ordered DFS Algorithms.

Theorem A.1 $F(k) = (2\nu)^k + (1-\nu)\frac{(2\nu)^k - 1}{2\nu - 1}$

Proof. It is clear that $F(0) = 1$. Now consider the case when $k \geq 0$. With $(1-\nu)$ probability, the root node is pruned, and thus $F(k) = 1$. With the remaining ν probability, the root node is not pruned, and has 2 successors. In this case, $F(k) = 2F(k-1)$.

Hence, we have

$$F(k) = (1-\nu)1 + \nu 2F(k-1)$$

$$= (1-\nu)\sum_{i=0}^{i=d-1}(2\nu)^i + (2\nu)^d F(0)$$

$$= (1-\nu)\frac{(2\nu)^k - 1}{2\nu - 1} + (2\nu)^k$$

□

Theorem A.2 $S(d) \approx (1-\gamma)F(d) + 1$.

Proof : When bounding heuristic succeeds at the root, DFS visits only one leaf; otherwise it visits only the left subtree at the root γ fraction of the time or it visits the left subtree unsuccessfully and then visits the right subtree $(1-\gamma)$ fraction of the time. Hence, we have,

$$S(d) = (1-\nu)1 + \nu(\gamma S(d-1) + (1-\gamma)(F(d-1) + S(d-1)))$$

$$= (1-\nu) + \nu(S(d-1) + (1-\gamma)F(d-1))$$

$$S(d) = (1-\nu) + \nu(1-\gamma)F(d-1) + \nu S(d-1)$$

$$= (1-\nu)\sum_{i=0}^{i=d-1}\nu^i + (1-\gamma)\sum_{i=1}^{i=d}\nu^i F(d-i)$$

For $d >> 1$ (a moderate d suffices) we have

$$\sum_{i=0}^{i=d-1}\nu^i \approx \frac{1}{1-\nu} \ as\ \nu < 1.0$$

Hence

$$S(d) = 1 + (1-\gamma)\sum_{i=1}^{i=d}\nu^i F(d-i)$$

Using theorem A.1 we can simplify this to

$$S(d) = 1 + (1-\gamma)\sum_{i=1}^{i=d}(\nu^d 2^{d-i} + \frac{1-\nu}{2\nu-1}(\nu^d 2^{d-i} - \nu^i))$$

$$S(d) = 1 + (1-\gamma)(\nu^d \sum_{i=1}^{i=d} 2^{d-i} + \frac{1-\nu}{2\nu-1}(\nu^d \sum_{i=1}^{i=d} 2^{d-i} - \sum_{i=1}^{i=d} \nu^i))$$

$$= 1 + (1-\gamma)(\nu^d(2^d-1) + \frac{1-\nu}{2\nu-1}(\nu^d(2^d-1) - \frac{1}{1-\nu}))$$

$$= 1 + (1-\gamma)(\nu^d 2^d - \nu^d + \frac{1-\nu}{2\nu-1}(\nu^d 2^d - \nu^d) - \frac{1}{2\nu-1})$$

$$= 1 + (1-\gamma)(\nu^d 2^d + \frac{1-\nu}{2\nu-1}(\nu^d 2^d - 1) - \nu^d(1 + \frac{1-\nu}{2\nu-1}) + \frac{1-\nu}{2\nu-1} - \frac{1}{2\nu-1})$$

$$= 1 + (1-\gamma)(F(d) - \frac{1-\nu}{2\nu-1}\nu^{d+1} - \frac{\nu}{2\nu-1})$$

Under the previous assumption $d >> 1$, the ν^{d+1} term can be ignored. The term $\frac{\nu}{2\nu-1}$ becomes negligible when ν is larger than 0.5, say for $\nu \geq 0.55$. Thus

Theorem A.3

$$P(d) \leq 1 + S(d)$$

The error ignored is small.

□

In two–processor parallel depth-first search, the tree is statically partitioned at the root and the processors search two trees of depth d-1 independently until at least one of them succeeds. Each of them individually performs the sequential DFS consulting the three wise oracles. Note that though the second processor violates the advice of the ordering heuristic at the root node, it follows its advice everywhere else. Hence

$$P(d) \leq (1-\nu)2 + \nu 2S(d-1)$$

$(1-\nu)\%$ of the time the root is pruned by **prune** and as a liberal convention we have two nodes expanded in the parallel search. Otherwise, two OB-trees of depth d-1 are searched, two nodes being expanded in each step, until one of the processors succeed.

The inequality arises because the average of the minimum of two trials is not more than the minimum of the two averages. From the formula for S(d) we have

$$P(d) \leq (1-\gamma)2 + 2\nu(1 + (1-\gamma)F(d-1))$$

$$\leq 2 + 2\nu(1-\gamma)F(d-1)$$

using theorem A.1,

$$P(d) \leq 2 + (1-\gamma)((2\nu)^d + \frac{1-\nu}{2\nu-1}((2\nu)^d - 2\nu - 1 + 1))$$

$$\leq 2 + (1-\gamma)((2\nu)^d + \frac{1-\nu}{2\nu-1}((2\nu)^d - 1) - (1-\nu))$$

$$\leq 2 + (1-\gamma)F(d) - 1 + \gamma + \nu - \gamma\nu$$

$$P(d) \leq 2 + (1-\gamma)F(d)$$

where the error ignored is less than 1.

From Theorem A.1 we have

$$P(d) \leq 1 + S(d)$$

The inequality becomes an equality if we restrict that there be only one solution in the entire search tree.

Theorem A.4 *On the average parallel search visits same number of nodes as sequential search on a large randomly selected set of problem instances, with single solutions.*

Proof: See above arguments.

References

[1] S. Arvindam, Vipin Kumar, V. Nageshwara Rao, and Vineet Singh. Automatic test pattern generation on multiprocessors. *Parallel Computing*, 17, number 12:1323–1342, December 1991.

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

[3] Prabhakar Goel. An implicit enumeration algorithm to generate tests for combinatorial logic circuits. *IEEE Transactions on Computers*, C-30,No. 3:215–222, 1981.

[4] Ananth Grama, Vipin Kumar, and V. Nageshwara Rao. Experimental evaluation of load balancing techniques for the hypercube. In *Proceedings of the Parallel Computing 91 Conference*, 1991.

[5] D. P. Helmbold and C. E. McDowell. Modeling speedup (n) greater than n. In *Proceedings of International conference on Parallel Processing*, pages 8–12, 1988.

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

[7] M. Imai, Y. Yoshida, and T. Fukumura. A parallel searching scheme for multiprocessor systems and its application to combinatorial problems. In *IJCAI*, 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] L. V. Kale. A perfect heuristic for the n non-attacking queens problem. *Information Processing Letters*, 34:173–178, April 1990.

[10] Laveen Kanal and Vipin Kumar. *Search in Artificial Intelligence*. Springer-Verlag, New York, 1988.

[11] Richard Korf. *Personal Communication*. 1988.

[12] W. Kornfeld. The use of parallelism to implement a heuristic search. In *IJCAI*, pages 575–580, 1981.

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

[14] Vipin Kumar, Ananth Grama, and V. Nageshwara Rao. Scalable load balancing techniques for parallel computers. Technical report, Tech Report 91-55, Computer Science Department, University of Minnesota, 1991.

[15] Vipin Kumar and Laveen Kanal. Parallel branch-and-bound formulations for and/or tree search. *IEEE Trans. Pattern. Anal. and Machine Intell.*, PAMI–6:768–778, 1984.

[16] Vipin Kumar and V. N. Rao. Scalable parallel formulations of depth-first search. In Vipin Kumar, P. S. Gopalakrishnan, and Laveen Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, 1990.

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

[18] T. H. Lai and Sartaj Sahni. Anomalies in parallel branch and bound algorithms. *Communications of the ACM*, pages 594–602, 1984.

[19] Guo-Jie Li and Benjamin W. Wah. Computational efficiency of parallel approximate branch-and-bound algorithms. In *International Conf on Paralle Processing*, pages 473–480, 1984.

[20] Kai Li. Ivy: A shared virtual memory system for parallel computing. In *Proceedings of International conference on Parallel Processing: Vol II*, pages 94–101, 1988.

[21] R. Mehrotra and E. Gehringer. Superlinear speedup through randomized algorithms. In *Proceedings of International conference on Parallel Processing*, pages 291–300, 1985.

[22] B. Monien, O. Vornberger, and E. Spekenmeyer. Superlinear speedup for parallel backtracking. Technical Report 30, Univ. of Paderborn, FRG, 1986.

[23] V. Nageshwara Rao and Vipin Kumar. Parallel depth-first search, part I: Implementation. *International Journal of Parallel Programming*, 16 (6):479–499, 1987.

[24] V. Nageshwara Rao and Vipin Kumar. Superlinear speedup in state-space search. In *Proceedings of the 1988 Foundation of Software Technology and Theoretcal Computer Science*, December 1988. Lecture Notes in Computer Science number 338, Springer Verlag.

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

[26] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Press, 1980.

[27] S. Patil and P. Banerjee. A parallel branch-and-bound algorithm for test generation. *IEEE Transactions on Computer Aided Design*, 9-3:313–322, 1990.

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

[29] Michael O. Rabin. Probabilistic algorithms. In J. Traub, editor, *Algorithms and Complexity: New Directions and Results*, pages 21–39. Academic Press, London, 1988.

[30] Vikram Saletore and L. V. Kale. Consistent linear speedup to a first solution in parallel state-space search. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 227–233, August 1990.

[31] H. Stone and P. Sipala. The average complexity of depth-first search with backtracking and cutoff. *IBM Journal of Research and Development*, May 1986.

[32] H. S. Stone and J. Stone. Efficient search techniques - an empirical study of the n-queens problem. Technical Report RC12057, IBM TJ-Watson Research Center, NY, 1986.

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

[34] 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.