International Journal of Parallel Programming, Vol. 17, No. 3, 1988

A Randomized Parallel Branch-and-Bound Algorithm[1]

Virendra K. Janakiram,[2] Edward F. Gehringer,[3] Dharma P. Agrawal,[3] and Ravi Mehrotra[4]

Received July 1988; revised February 1989

Randomized algorithms are algorithms that employ randomness in their solution method. We show that the performance of randomized algorithms is less affected by factors that prevent most parallel deterministic algorithms from attaining their theoretical speedup bounds. A major reason is that the mapping of randomized algorithms onto multiprocessors involves very little scheduling or communication overhead. Furthermore, reliability is enhanced because the failure of a single processor leads only to degradation, not failure, of the algorithm. We present results of an extensive simulation done on a multiprocessor simulator, running a randomized branch-and-bound algorithm. The particular case we consider is the knapsack problem, due to its ease of formulation. We observe the largest speedups in precisely those problems that take large amounts of time to solve.

KEY WORDS: Branch-and-bound; randomized algorithms; parallel algorithms; speedup.

1. INTRODUCTION

Much of the interest in parallel processing derives from a desire to solve large problems. Algorithms that perform well on small problems do not always exhibit acceptable performance as the problem size grows. Algorithms whose complexity increases rather slowly with problem

[1] This work has been supported by the U.S. Army Research Office under Contract No. DAAG 29-85-K-0236.

[2] Author's current address: AT&T Bell Labs, Crawford's Corner Road, Holmdel, NJ 07733.

[3] Computer Systems Laboratory, Department of Electrical and Computer Engineering, Box 7911, North Carolina State University, Raleigh, North Carolina 27695.

[4] Arthur Andersen and Company, 33 West Monroe, Chicago, IL 60603.

0885-7458/88/0600-0277$06.00/0

size—less than linearly, for example—can effectively be "scaled up" to handle large problems. This is true for serial processing, and usually also for parallel processing, provided that the algorithm can be effectively decomposed for multiple processors.

Some algorithms decompose more effectively than others. An algorithm that requires frequent interprocessor communication will perform more poorly as processors are added. If the processors can work more or less independently, however, the total amount of work tends to remain nearly constant, and the benefit of adding processors is more apparent. A computation using n processors then finishes in about $1/n$th the time that a single processor would need; so that *linear speedup* has been achieved. Thus, to perform parallel processing efficiently, we need to concentrate on both computational complexity and speedup. In this paper, we show that a particular class of algorithms known as *randomized* algorithms offer benefits related to both speedup and complexity: the calculations require less elapsed time, and the code for decomposing the problem into concurrent processes is much simpler.

2. RANDOMIZED ALGORITHMS

A randomized algorithm,[1] is an algorithm obtained by introducing randomness into the solution procedure. The use of randomized algorithms can yield a significant reduction in the average time to perform certain problem-solving searches. A randomized algorithm operating on a fixed input may take any one of several possible computation paths, depending on some internally controlled random process derived from the problem structure. As an illustration, consider the simple problem P of finding a path from a root node r of a tree T of v nodes to a node e such that e is a leaf node of T. It is perhaps convenient to think of the problem as follows. Node r represents a starting node and leaf nodes are the target or goal nodes. All goal nodes are equally desirable and we need to find a path to any one of them.

A deterministic algorithm to solve P will start from node r and (deterministically) select an edge (e.g., the leftmost) by which to proceed to an adjacent node x. If x is not a leaf node, this process will be repeated, and continued until ultimately a leaf node is reached. The algorithm will find the same leaf node e each time it is executed (e.g., node a in Fig. 1).

In contrast, a randomized algorithm will start from node r and randomly select an edge to some adjacent node x. If x is not a leaf node, the step is repeated by randomly selecting one of the remaining edges connected to node x. This process is continued until ultimately a leaf node is reached. The algorithm follows a completely random solution path,

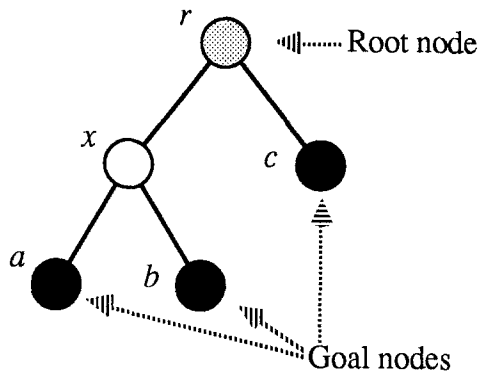

Fig. 1. A search tree.

and so may not find the same leaf node e each time it is executed. (In Fig. 1, it may find either a, b, or c.) Researchers have applied randomized algorithms to problems in coding theory, symbol manipulation, number theory, combinatorial analysis, discrete optimization, and various artificial-intelligence oriented applications, e.g., see Refs. 2–5. Rabin[6] cites several problems for which a (sequential) randomized algorithm outperforms deterministic algorithms.

3. SPEEDUP

Speedup is a measure of how much faster a computation finishes on an n-processor multiprocessor than on a uniprocessor. For a deterministic algorithm, we will define it as $T(1)/T(n)$, the ratio of the elapsed time for solving the problem on one processor divided by the elapsed time for solving it on n processors. For a randomized algorithm, the execution time is a random variable. Let $T(k)$ be the random variable which represents the time to solve the problem when k processors work independently. Let $E[T(n)]$ represent the expected value of the random variable $T(n)$. Then the average speedup $S(n)$ can be defined as

$$S(n) = \frac{E[T(1)]}{E[T(n)]} \tag{1}$$

With no communication (other than to report final results), $T(n)$ is given by the $\min_{1 \leqslant i \leqslant n} T(i)$. Speedup is possible even without interprocess communication during the running of this algorithm: the first processor to finish simply reports to the others that it has found the solution. The speedup depends only on the structure and statistical properties of the

solution space and the use of a randomized algorithm. Increasing the amount of communication, up to a point, enables the processors to converge on the solution more quickly. As the amount of communication per process is held constant, increasing the number of processors decreases the solution time.

3.1. Conditions that Limit Speedup

Regardless of the algorithm, it is never easy to approach the maximum "theoretical" speedup. On any practical multiprocessor, contention for shared memory and synchronization of processes induce significant overheads. These are pitfalls on which many promising algorithms falter. Contention occurs when processors request data more quickly than remote memory can deliver it. Clearly, when the rate at which an individual processor accesses data is independent of the number of processors in the system, if enough processors are added, the memory will eventually saturate and impose a maximum bound on speedup.[7]

A *global synchronization point* is a stage in a parallel computation which all processors must reach before any processor may proceed past it. The presence of global synchronization points obviously limits the performance of a parallel algorithm. This is true for two reasons. First, deterministic algorithms tend to spawn processes which are identical, except for the data on which they operate. Running in lock-step, the processes tend to reference global data at almost exactly the same time, effecting maximal contention. Second, processes run at different speeds. In addition to processor-speed variations, different processes may take different branches between synchronization points. Finally, the overhead imposed by background tasks may differ from processor to processor: clocks need to be updated and accounting information must be collected, for example. These factors introduce randomness into process execution times. Synchronization means that all processes are limited to the speed of the slowest.

Randomized algorithms ease both contention and synchronization. Contention is less serious because randomness is introduced into memory-reference patterns; instead of accessing memory units in the same order, for example, randomized processes may "choose" to access them in different orders. Randomized algorithms like the ones we have described are also asynchronous, devoid of global synchronization points. Instead of waiting for each other, they exchange information asynchronously, by reading or writing shared memory at arbitrary times.[8] Thus, there is reason to believe that these algorithms may approach their theoretical speedup more closely than deterministic algorithms, whose performance is usually distinctly sublinear even when linear speedup is theoretically possible.[9]

4. RANDOMIZATION OF BRANCH-AND-BOUND PROBLEMS

Search algorithms are among the most difficult to implement efficiently on a vector or array processor. The memory-reference patterns of an SIMD algorithm can be optimized to avoid contention for memory, but this is less of an advantage for randomized algorithms, which inherently exhibit less contention than their deterministic counterparts. On these algorithms, multiprocessors may offer their most dramatic performance improvement over other architectures.

Janakiram *et al.*[10] have shown the efficacy of a randomized algorithm for parallel backtracking. By using a technique similar to the one described here, they show that with no inter-processor communication, speedups of k (for k processors) can be obtained on some problems, while on other kinds of problems speedups of $(k+1)/2$ are possible. On the kinds of problems discussed by Stone and and Sipala,[11] which on average take times that are but polynomial functions of problem size, in order to achieve comparable speedups, a small amount of interprocessor communication is introduced. The paper also describes the results of using randomization in a PROLOG interpreter for parallel execution.

5. ASYNCHRONOUS BRANCH-AND-BOUND ALFORITHMS

Branch-and-bound is an optimization technique that has been used extensively to solve a wide variety of problems on uniprocessor systems.[12] It has been applied to problems in scheduling,[13] knapsack,[14,15] traveling salesman,[16] facility allocation,[17] and integer programming.[12,18] Branch-and-bound algorithms have also been subjected to extensive theoretical analysis, and researchers have begun to investigate the effects of parallelizing branch-and-bound programs (e.g., see Refs. 19 and 20).

For the sake of completeness we shall give a qualitative adumbration of the branch-and-bound algorithm.

The first step is to split the given problem into several (usually 2) smaller sub-problems. This may be achieved in many ways: In the case of *relaxation guided* procedures, these subproblems may be obtained by considering all solutions with some constraint on the solution removed. Alternatively, sub-problems may be obtained by imposing an *a priori* value on some parameter of the solution. In any case, the rule that governs this splitting is termed the *branching rule.* From among these subproblems, one is chosen, by means of a *selection rule*, for another application of the branching rule. The selection rule will determine the ordering of the search for the solution. This process is carried out recursively until the sub-problem itself represents a complete and *feasible solution.* It is desired to

find the feasible solution which optimizes a given *cost criterion.* At each stage it will become apparent that some sub-solutions will not lead to a possible solution. These may be eliminated from further consideration. Of prime importance in the branch-and-bound algorithm is the existence of a *bounding function.* This function, say g, has the following properties: (a) Applied to any sub-problem, it returns a value that is a bound on the best possible value of the cost function obtainable by solving all the sub-problems reachable from this sub-problem. Where the sub-problem cannot be decomposed into a feasible solution, g will return a symbol denoting this fact. (b) Applied to a feasible solution g returns the exact value of the cost.

The entire process may be visualized as searching a *branch-and-bound tree,* with each node representing a sub-problem, and each leaf representing either a feasible solution, or a sub-problem that will not yield a possible solution. Each node will have a value associated with it which is simply the value of the bounding function g. From this point on, we will assume, without loss of generality, that a solution with the minimum cost is being sought. Further, we assume that the selection rule results in the search of the branch-and-bound tree in a *depth-first* fashion. Another strategy that is employed is the *best-first,* in which the node with the lowest value of g is taken first. Best-first search has been shown[21] to result in a smaller number of sub-problem expansions; but on the other hand, considerable time and space expenditure is required to extract the best sub-problem to expand. Our method seems to be applicable to best-first searches also, as will be apparent, but we have not yet applied it to best-first searches, or compared best-first search with the randomized depth-first method.

The success of the branch-and-bound algorithm stems primarily from the fact that the bounding function, g, can be used to remove from consideration, fairly early in the proceedings, sub-problems that will not yield solutions better than the one at hand. As time progresses, successively better solutions will have been found, and with this "experience" gained, the branch-and-bound algorithm is able to eliminate progressively larger sub-trees. Thus, the branch-and-bound algorithm dynamically prunes the search tree. The branch-and-bound algorithm just described is given in Fig. 2a, and is called *lfbb,* (*l*eft *f*irst *b*ranch-and-*b*ound).

A parallel version of the branch-and-bound has been considered by Wah and Ma.[20] MANIP is a special-purpose machine proposed by them solely for the solution of branch-and-bound problems. The selection rule used here is best-first, but depth-first search is used when secondary memory runs out. They have indicated that simulation studies have shown a speedup of k, for k processors. However, the speedup is calculated only with respect to the number of iterations; important factors such as

```
minfeas : real { Cost of the minimum feasible solution }

function lfbb(instance)
    begin
    cost ← g(instance) { bounding function }
    if (cost < minfeas) then
        if (not feasible(instance)) then
            begin
            i ← 1
            repeat
                nextinstance ← i^th child of instance
                lfbb(nextinstance)
                i ← i + 1
            until (i > S) { S = No. of children of instance }
            end
        else
            minfeas ← cost
    end
```

Fig. 2a. The algorithm *lfbb*.

(secondary) memory access times which could have a serious impact on the performance have been neglected.

El-Desouki and Huen[22] have considered a scheme somewhat similar to the one we are about to describe. Here, the workload of each processor is deterministically apportioned at the outset. Suppose that (as is very likely) one processor finishes examining its portion of the solution tree before the others. It then enters into a complicated and lengthy dialogue with each of the other processors to determine which particular portion of the solution tree is most suitable for exploration. Not only is there extra communication cost involved here, but also an unknown amount of extra computation overhead incurred by all the processors. Another serious drawback of this scheme is apparent when considering what would happen should a processor fail. In such a case, a portion of the of the tree will remain unexamined, thereby producing erroneous results. The method to be described avoids these problems.

Consider the conventional branch-and-bound algorithm of Fig. 2a. Instead of choosing the next node to be evaluated in left-first fashion, we will "randomize" the search by making a random choice for the next node from among the unexamined children. The cost of the best feasible solution

```
minfeas : real { Cost of the minimum feasible solution }

function rsbb(instance)
    begin
    cost ← g(instance) { bounding function }
    if (cost < minfeas) then
        if (not feasible(instance)) then
            repeat
                nextinstance ← randomly chosen unevaluated
                               child of instance
                rsbb(nextinstance)
                quit ← (all children of instance have been examined)
            until (quit)
        else
            minfeas ← cost
    end
```

Fig. 2b. The algorithm *rsbb*.

available currently is made available globally to all processors. Other details of the algorithm remain unchanged. This algorithm is called *rsbb* (*r*andom *s*earch *b*ranch-and-*b*ound) and is given in Fig. 2b. It can be shown that the expected solution time on a uniprocessor using *rsbb* is the same as that of *lfbb* for a very general class of problems. However, the advantage of *rsbb* is that, under parallel execution, the decision as to the next node to evaluate can be made locally, without global information. If we use k processors, for example, each will, in the main, examine a different portion of the tree. Of course there is a finite probability that replication of work will occur, but by allowing a very small amount of simple and terse communication between processors, this probability can be kept relatively small, and good speedups can be achieved. Another advantage of the decentralized operation is that fault-tolerance and easy expandability are inherent in such a system. In the next section we present a probabilistic model for the branch-and-bound process, and then use this model to estimate the expected speedups that would be obtained using parallel *rsbb*.

6. MODEL DESCRIPTION

We will first establish that the expected solution time for a branch-and-bound problem solved using *lfbb* is the same as when using *rsbb*.

To do so, we describe a random process for generating branch-and-bound trees. It is based on Smith's[23] procedure. A random branch-and-bound tree may be generated as follows:

(i) Let a root node exist, which is unsprouted ($level = 0$).

(ii) Each unsprouted node n at level i is sprouted as follows: Let n have S children, where S is a random integer whose probability mass function (p.m.f.) is given by $p_S(t; i)$, $(p_S(t; i = 0) > 0)$. Each node will be assigned a cost that depends on the node costs of its ancestors, adding to them a random number Q, whose distribution is given by $p_Q(t)$. $(P_Q(t = 0) = 0.)$

(iii) Repeat (ii) until there are no unsprouted nodes.

This procedure is slightly more general than that given by Smith. The node cost obtained in this way is the simulated equivalent of the value of the bounding function g, described in the last section. The cost of a particular node is at least equal to that of its parent. The increase in the cost of a node (namely, the random variable Q) over the cost of its parent is termed the *incremental node cost*, and is a nonnegative number; the cost of any node is defined as the sum of the incremental node costs of its ancestors.

Theorem. For all the trees generated by this procedure, the probability distribution of the solution times using depth-first search is independent of the order in which successive nodes are chosen.

Outline of Proof. First note that $p_S(t; i)$ and $p_Q(t)$ are *sibling-independent*. Then any selection order can be transformed into another simply by pivoting about a node. The proof is then obvious from this observation. ∎

It is thus clear that *lfbb* and *rsbb* take the same time to solve a problem, on average. The next step is to estimate the expected solution time of *lfbb*. This has been determined by Smith,[23] and by Wah and Yu.[24] The latter paper has given a conceptually satisfying model of the branch-and-bound process which is also very accurate. The extension of this model to the multiprocessor case, however, does not seem to be possible. Instead, a simpler model has been devised, that is adequate for the present purpose, which is to estimate the speedup on a parallel system.

First, regarding the structure of the branch-and-bound tree, let us make some assumptions which are similar to those made by Wah and Yu:

(i) The solution tree is assumed to be finite and of constant degree. As trees of degree two are the most commonly occurring ones,

binary trees are considered here; but this is not a restriction on the analysis.

(ii) It is assumed that the tree is full, and of depth D.

(iii) The incremental node cost is assumed to be *exponentially* distributed. This assumption is made not only because it makes the mathematics tractable, but because this has been the distribution that has been observed in practice. Wah and Yu have shown that it occurs in the knapsack problem,[24] and in integer programming,[21] while Smith[23] reports that for the traveling salesman problem, the distribution is geometric, which is the discrete analogue of the exponential distribution.

Figure 3 shows a branch-and-bound solution tree. The nodes are labeled (i, j), where i is the level and j the serial number of the node within this level, beginning from the left with 0. As *lfbb* proceeds, it will follow the path $(0, 0)$, $(1, 0)$, ..., $(D, 0)$, where D is the depth of the tree. Node $(D, 0)$ is a feasible solution. Now *lfbb* will backtrack and look for other feasible solutions, each time updating the variable *minfeas*, which contains the cost of the best possible solution obtained thus far. When *lfbb* encounters a node whose cost happens to be greater [Recall that *minimization* problems are being dealt with here.] than or equal to *minfeas*, it will simply discard the children of this node.

Let $N_{i,j}$ be the number of nodes examined by *lfbb* in a tree rooted at (i, j). It is desired to find $N_{0,0}$, the number of nodes examined in the entire tree. $N_{i,j}$ can be expressed as a recurrence:

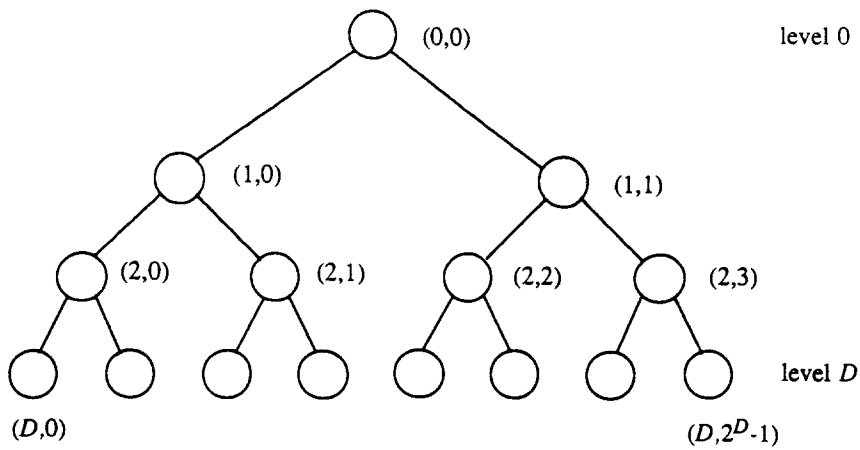

Fig. 3. Numbering conventions for the branch-and bound tree.

$$N_{i,j} = 1 + \rho_{i,j}(N_{i+1,2j} + N_{i+1,2j+1})$$
$$\rho_{0,0} = 1 \tag{2}$$
$$N_{D,j} = 1$$

where $\rho_{i,j}$ is the probability that the node cost of (i, j) is less than *minfeas*.

Equation (2) states that *lfbb must* examine node (i, j), and then with probability $\rho_{i,j}$, examine the subtrees rooted at its children. The root (0, 0) is examined with probability 1 ($\rho_{0,0} = 1$) and a leaf node is terminal ($N_{D,j} = 1$). The computation of $\rho_{i,j}$ is now described.

Consider the scenario in Fig. 4. Here, *lfbb* has encountered the l solutions in subtrees T_2, T_3, and T_4, and is now examining node a, deciding whether to select the subtree rooted at a for searching or to backtrack. The best feasible solution will be the one with minimum cost among those obtained so far. The cost of each solution is the sum of D (not necessarily independent) random variables, each random variable being the incremental node cost.

An exact analysis of this cost is intractable. The treatment developed here makes certain simplifying assumptions. The results thus obtained are approximate, but are of sufficient accuracy for the present purpose.

Suppose that the algorithm has found l solutions by the time it encounters node a at level m (see Fig. 4). Now the cost of a particular solu-

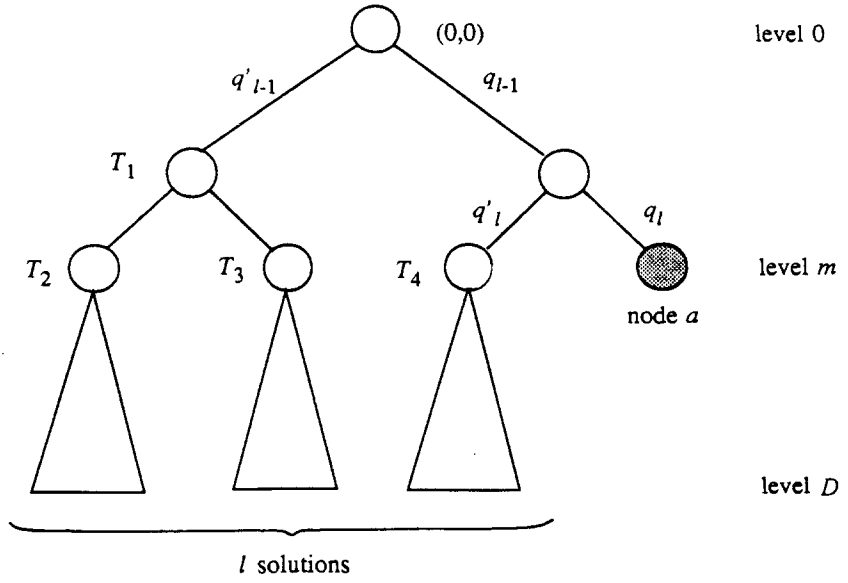

Fig. 4. Branch-and-bound solution in progress.

tion is the sum of D incremental costs. This sum may be considered in two parts: (a) The sum of the incremental cost of the node at level $m-1$ and the cost of its parent, and (b) the sum of the remaining incremental costs. Although, in general, node a and all the solution nodes before node a is encountered may not have all their ancestors in common, it is assumed that this is indeed the case, and thus the contribution of these ancestors to the variation in node costs among the solution nodes, and also node a, is ignored. The contribution to the solution cost under part (b) earlier will be due to the nodes below level $(m-1)$. Many of these ancestors are common to the l solution nodes. However, it is assumed instead that these ancestors are not common, which implies that the events that lead to the generation of the solutions are *independent*. The approximations made in computing the sums (a) and (b) are "opposite," in a sense. In (a), complete *dependence* is assumed, when independent events exist, while in (b) *independence* is assumed when dependence exists. Hence, the errors caused by these approximations could be expected to compensate for each other, to some extent. Later, we present experimental evidence that shows that the cumulative error is not serious.

A chain of nodes, then, from level $m-1$ to level D will have on it some n $(n=D-m+1)$, nodes. The contribution of these n nodes to the solution cost will be the sum of n random variables, each of which is the incremental cost of a node. This sum is termed the *length* of the chain. The situation is depicted in Fig. 5. There are l such chains, corresponding to the l solutions. By the assumptions made earlier, the length of each chain is an independent random variable. If the minimum length of these l chains is less than or equal to the incremental cost of node a, then it may be concluded that there is no solution in the subtree rooted at a that is any better than the best one found so far. Further exploration of this subtree is useless, and hence node a will be cut off.

Let p_c be the probability of cutoff, and Q a random variable that describes the incremental node cost. Then

$$dp_c = Pr\{\min(RR \text{ cost of } l \text{ } RR \text{ chains}) \leqslant t\} \times RRPr\{t \leqslant Q \leqslant t+dt\}$$

$$= \{1-[1-F_{Q,n}(t)]^l\} f_Q(t)\, dt \tag{3}$$

$F_{Q,n}(t)$ is the probability distribution of length of a chain. A chain is made up of n nodes, each of which contributes a random amount Q to the length. Q has a density function $f_Q(t)$. So,

$$F_{Q,n}(t) = \int_0^t f_Q^{(n)}(x)\, dx \tag{4}$$

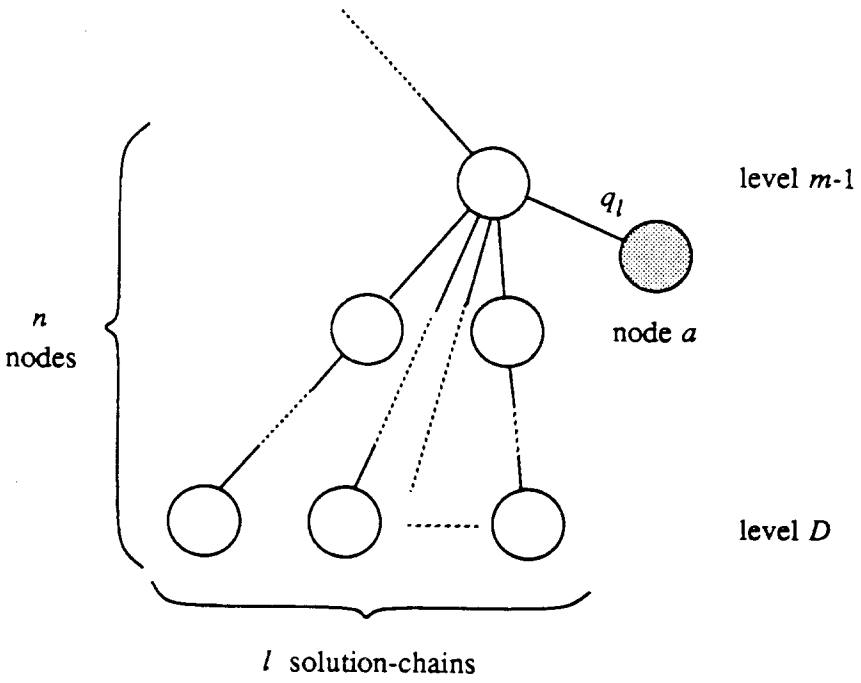

Fig. 5. Cost of a chain of nodes.

where the power is in the sense of convolution. The cutoff probability can then be written:

$$p_c(n, l) = \int_0^\infty \left\{1 - \left[1 - \int_0^t f_Q^{(n)}(x)\, dx\right]^l\right\} f_Q(t)\, dt \tag{5}$$

Because of the reasons adduced earlier, $f_Q(t)$ is set to be the exponential distribution:

$$f_Q(t) = \lambda e^{-\lambda t}$$

Then Eq. (5) becomes

$$p_c(n, l) = 1 - \int_0^\infty \left[\sum_{k=0}^{n} \frac{t^k}{k!}\right]^l e^{-(l+1)t}\, dt \tag{6}$$

Although the integral in Eq. (6) can be obtained for small values of l and n by applying the multinomial expansion, this technique is impractical for large l. (l may take values in the order of 10^5.) An alternative is to use the Guass-Laguerre quadrature.[25] But for large l, this method gives rise to

significant inaccuracies. An asymptotic expression may, however, be derived for large l, using the Laplace method[26]:

$$p_c(n, l) \sim 1 - \frac{\mu}{n+1} \sum_{i=0}^{\infty} \frac{(-1)^i \mu^i}{i!} \cdot \frac{\Gamma[(i+1)/(n+1)]}{l^{(i+1)/(n+1)}} \tag{7}$$

where

$$\mu = [(n+1)!]^{1/(n+1)}$$

The convergence of the infinite sum in Eq. (7) is quite rapid: when $l \approx 10n$, about four terms are required for a 1% error. For larger l, an even smaller number of terms is sufficient.

The values of (i, j) in $\rho_{i,j}$, and (n, l) in $p_c(n, l)$ are unique to each node, and a little algebraic manipulation shows that the quantity $\rho_{i,j}$ is related to $p_c(n, l)$ by:

$$\rho_{i,j} = 1 - p_c(d - i + 1, j2^{d-i}) \tag{8}$$

At this point all the elements are in place to solve the original problem given in Eq. (2), using Eqs. (6)–(8). The expected number of nodes examined by the branch-and-bound for trees of various depths can be calculated. The results are shown in Table I, which compares the model with a simulation using 1000–2000 trials on random trees. For trees of depth 10 and below, the discrepancy between the model and the simulation is about 5%, increasing to about 14% for a depth of 12.

Suppose there are two processors working on two different subtrees, each exchanging information as to the cost of the best solution. Then the

Table I. Expected Number of Nodes Expanded

Tree depth	Simulation	Model
2	6.34	6.58
3	11.50	12.59
4	19.83	21.85
5	32.36	34.43
6	49.32	50.45
7	76.76	73.57
8	110.72	107.21
9	160.19	156.64
10	229.83	230.03
11	321.37	340.06
12	443.40	506.45

decision whether to cut off at some node will have the benefit of both these searches: twice the number of independent chains have been generated. The effect of this mutual assistance can be represented in our model by simply replacing l by kl in Eq. (6), where k is the number of processors. The result is the speedup, *given* that each processor has chosen a different subtree to work upon. In the next section, an expression for the unconditional expected speedup for a k-processor system with global memory is derived.

7. SPEEDUP CALCULATIONS

The results obtained from our model can now be utilized to estimate the expected speedups. In the derivation, queueing and communication delays have been neglected. In the next section, it is demonstrated that these are quite small.

Suppose some k processors are deployed to search the tree using *rsbb*. At each level each processor makes a random decision regarding the next child to examine. Given a sufficiently large tree, there is but an infinitesimal probability that two processors follow the exact same path while searching the tree. Of more concern is the likelihood that a processor will take a path already trodden earlier by another processor, resulting in replication of work. To avoid this, the algorithm maintains a global list that keeps the status of the subtrees at each level. It has been shown in the case of the backtracking problem[10] that the level to which this list must be maintained does not need to be large. Similar reasoning may be applied here. In the next section, it will be shown that a list maintained for the first five levels is adequate for up to 10 processors.

The following terms are defined. Let $\mathbf{S}_k$ be the expected speedup using k processors; $T(j, n)$ be the expected time taken by j processors to search the left subtree, which has n nodes; and $T'(j, n)$ be the expected time to search the right subtree *after* the left subtree has been searched. The time to examine one node is given as 1 unit. Define $\gamma \equiv T'(j, n)/T(j, n)$. It is clear that

$$T(j, 2n) = T(j, n) + T'(j, n) + 1$$

$$\approx T(j, n) + T'(j, n) \text{ (for large trees)}$$

So, ignoring the error due to this approximation,

$$\frac{T(j, 2n)}{T(j, n)} = (1 + \gamma) \tag{9}$$

Equation (9) expresses the ratio of the time that j processors take to search a tree of $2n$ nodes to the time they take for a tree of n nodes. Now,

$$\frac{T(1, 2n)}{T(j, 2n)} = \mathbf{S}_j = \frac{T(1, n)}{T(j, n)} \tag{10}$$

assuming that the speedup is independent of problem size, which will be the case when the number of processors is relatively small. Transposing terms gives

$$\frac{T(j, 2n)}{T(j, n)} = \frac{T(1, 2n)}{T(1, n)} \tag{11}$$

Equation (11) implies that Eq. (9) may be used to estimate γ from Table I, which gives an average value of $\gamma \approx 0.4$. Extending Eq. (9),

$$\frac{T(j, 2^{i+1})}{T(j, 1)} = (1+\gamma)^{i+1} \tag{12}$$

Now, $2^{D+1} \approx N$ is the total number of nodes. Let $2^{i+1} \approx n$ be the size of the subtree searched during the first t time units. Then,

$$T(j, n) = T(j, N)(1+\gamma)^{\log_2 n - \log_2 N} \tag{13}$$

which gives

$$\beta(t) \equiv \frac{n}{N} = \left(\frac{t}{T_N}\right)^{\alpha} \tag{14}$$

where

$\beta(t) \equiv n/N =$ fraction of nodes examined during the first t time units

$T_N =$ time to search the entire tree of N nodes, and

$\alpha = 1/\log_2(1+\gamma)$

Suppose there are some k processors working on one part of the tree, with l processors working elsewhere in another subtree, assisting these k processors. Let $S_{k,l}$ be the expected speedup obtained in the first subtree. ($\mathbf{S}_k \equiv S_{k,0}$). At the node rooted at this subtree, the k processors will randomly choose their next subtree to examine. Of the k, some k_1 will choose the left subtree, and k_2 the right. Let p_i be the probability of this partition, $\{k_1, k_2\}$. Without loss of generality, it can be assumed that $k_1 \geq k_2$. Let σ_i be the speedup, given this partition.

$$S_{k,l} = \sum_i p_i \sigma_i \tag{15}$$

where the sum is over all the possible partitions. Consider the partition $\{k_1, k_2\}$; $k_1 \geqslant k_2$, $k_2 \neq 0$. The k_1 processors, now assisted by $l+k_2$ processors, will finish searching the left subtree consisting of $N/2$ nodes in an expected time $t_1 = T(k_1, N/2)$.

$$t_1 = T(k_1, N/2) = \frac{T(1, N/2)}{S_{k_1, l+k_2}}$$
$$= \frac{1}{1+\gamma} \cdot \frac{T(1, N)}{S_{k_1, l+k_2}} \tag{16}$$

In time t_1, the k_2 processors have finished some fraction $\beta(t_1)$ of their work. From Eq. (13),

$$\beta(t_1) = \left[\frac{t_1}{T(k_2, N/2)}\right]^{\alpha} \tag{17}$$

Now, $T(k_2, N/2)$ can be written as:

$$T(k_2, N/2) = \frac{T(1, N/2)}{S_{k_2, l+k_1}}$$
$$= \frac{T(1, N)}{1+\gamma} \cdot \frac{1}{S_{k_2, l+k_1}} \tag{18}$$

whence,

$$\beta(t_1) = \left[\frac{S_{k_2, l+k_1}}{S_{k_1, l+k_2}}\right]^{\alpha} \tag{19}$$

After time t_1, all the processors will combine to work upon the unfinished portion of the tree. Let t_2 be the expected time to finish the remaining work. Up to time t_1, the growth of β is governed by k_2 processors. At time t_1, k_1 processors are added to assist these k_2 processors. At this point, the growth of β is accelerated in accordance with Eq. (14). The amount of work done in time t_1 is $\beta(t_1)$, and the time to finish the remaining work is then found by shifting the β-curve for $k_1 + k_2$ processors appropriately. Figure 6 shows the two β-curves, with the curve for $k_1 + k_2$ processors shifted so that at time t_1, $\beta(t_1)$ fraction of work would have been completed. The β-curve for t_2 can then be calculated:

$$t_2 = T(k, N/2) - T(k, N/2)\,\beta(t_1)^{1/\alpha}$$
$$= \frac{T(1, N)}{S_{k,l}} \cdot \frac{1}{1+\gamma} \cdot (1 - \beta(t_1)^{1/\alpha}) \tag{20}$$

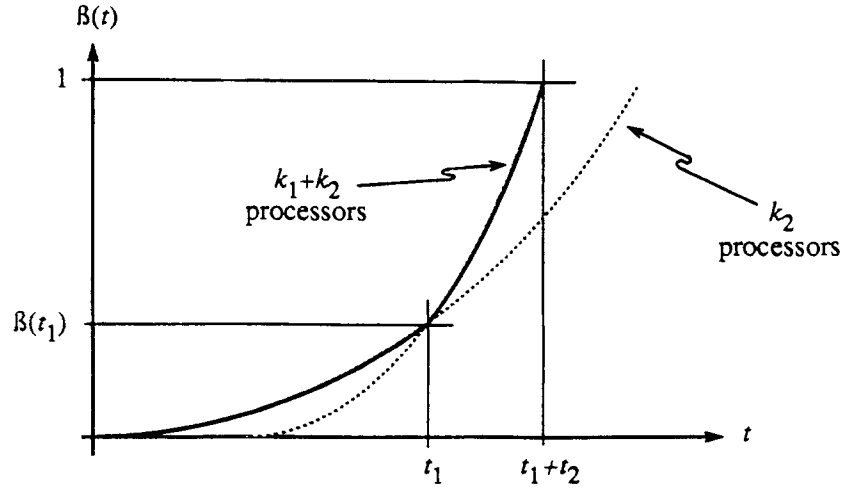

Fig. 6. Calculation of solution times.

The total time to search the tree, $T(k, N)$ is $t_1 + t_2$ and is given by:

$$T(k, N) = \frac{T(1, N)}{1+\gamma}\left(\frac{1}{S_{k_1, l+k_2}} + \frac{1-\beta(t_1)^{1/\alpha}}{S_{k,l}}\right)$$

Transposing terms, the speedup σ_i for a given partition can then be written:

$$\sigma_i = (1+\gamma)\left(\frac{1}{S_{k_1, l+k_2}} + \frac{1-\beta(t_1)^{1/\alpha}}{S_{k,l}}\right)^{-1} \tag{21}$$

Now, assuming, as before, that speedup is independent of problem size, Eq. (15) can be expanded as

$$S_{k,l} = p_0 S_{k,l} + \sum_i p_i \sigma_i \tag{22}$$

where p_0 is the probability of a k,0 partition, and the summation is over all partitions.

Equation (22) expresses a recurrence relation for the expected speedup. The calculation of the probability p_i of a particular partition $\{k_1, k_2\}$ is a straightforward matter of considering the probability of the occupancy numbers[27]:

$$p_i = \begin{cases} \dfrac{k!}{k_1!\, k_2!} \cdot \dfrac{1}{2^{k-1}}; & k_1 \neq k_2 \\ \dfrac{k!}{[(k/2)!]^2} \cdot \dfrac{1}{2^k}; & k_1 = k_2 \end{cases} \tag{23}$$

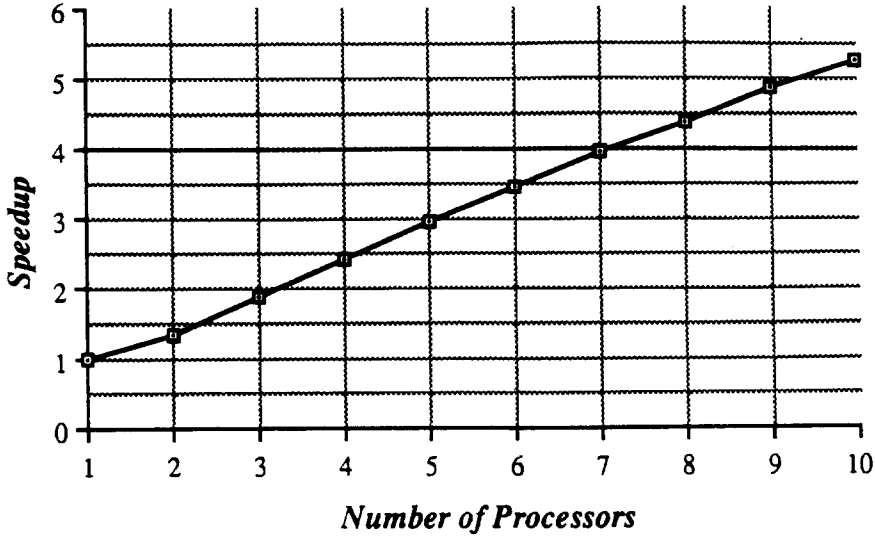

Fig. 7. Estimated speedup.

The terminating condition for the recurrence is the set $S_{1,l}$ which can be obtained from the model as explained in the previous section. Figure 7 shows a plot of these values. Setting $S_{1,l} = 1$, (all l), gives the case when only local versions of the best solution are kept by each processor.

The next section describes the simulations that were used to verify the model and presents the results.

8. SIMULATION RESULTS

The performance of the randomized branch-and-bound algorithm was measured on a multiprocessor simulator MPSIM,[28] (implementing the

Table II. Details of Data-sets for the Knapsack Problem[a]

Data-set No.	Profits (P_i)	Weights (W_i)	Capacity (M)
1	Random	Random	$\sum W_i/2$
2	Random	Random	$2 \times \max\{W_i\}$
3	$W_i + 10$	Random	$\sum W_i/2$
4	$W_i + 10$	Random	$2 \times \max\{W_i\}$
5	Random	$P_i + 10$	$\sum W_i/2$
6	Random	$P_i + 10$	$2 \times \max\{W_i\}$

[a] Note: Random ≡ Random integer in the range [1, 100].

Table III. Average Speedups Obtained from Simulation (in Terms of the Number of Nodes Examined)[a]

Data Set No.	Processors	Speedup for Global Memory Size 31	15	None	Range of Uniprocessor Solution Times
	2	1.72	1.78	1.50	
	3	2.34	2.61	2.00	
	4	2.21	2.00	1.77	
	5	3.06	2.57	2.98	
1	6	4.23	4.09	2.26	120–880
	7	3.71	4.44	4.31	
	8	4.37	4.53	2.83	
	9	4.67	4.99	2.54	
	10	4.79	4.39	2.90	
	2	1.57	1.51	1.37	
	3	2.06	2.19	2.22	
	4	2.11	2.19	2.04	
	5	2.85	2.40	3.16	
2	6	4.16	4.67	2.62	580–2600
	7	4.11	4.53	7.39	
	8	4.44	4.19	3.32	
	9	6.06	7.29	3.37	
	10	6.01	5.64	3.98	
	2	1.67	1.64	1.54	
	3	2.92	2.88	1.88	
	4	2.51	2.34	1.97	
	5	3.06	2.60	4.96	
3	6	3.67	3.92	2.69	280–3200
	7	7.07	6.19	6.05	
	8	5.90	6.08	3.33	
	9	7.31	6.48	2.75	
	10	5.47	6.21	3.23	
	2	1.51	1.46	1.52	
	3	2.36	2.02	2.17	
	4	2.23	2.12	2.25	
	5	2.74	2.54	2.66	
4	6	4.15	4.84	2.68	2500–24100
	7	3.52	4.84	11.88	
	8	3.79	3.43	3.79	
	9	8.29	7.03	3.50	
	10	4.91	5.29	4.14	

[a] Notes: (i) For a description of the data sets see Table II; (ii) Time to examine a node is taken as 1 time unit.

Table continued

Table III. (*Continued*)

Data Set No.	Processors	Speedup for Global Memory Size 31	15	None	Range of Uniprocessor Solution Times
	2	1.40	1.27	1.44	
	3	1.64	1.60	1.68	
	4	1.65	1.70	1.64	
	5	1.69	1.70	1.64	
5	6	1.74	1.89	1.70	62–235
	7	1.80	1.87	1.75	
	8	1.73	1.90	1.78	
	9	1.94	1.92	1.65	
	10	1.86	1.74	1.70	
	2	1.48	1.42	1.23	
	3	4.29	4.30	2.85	
	4	2.34	1.78	1.50	
	5	5.60	4.44	3.78	
6	6	5.08	4.79	3.55	120–2240
	7	5.38	4.80	3.73	
	8	5.34	4.97	3.88	
	9	5.42	4.57	1.87	
	10	6.30	4.94	3.65	

PRAM-CREW model), running under ULTRIX on a MICROVAX. The randomized algorithm *rsbb* is implemented in C. The software can be divided into three parts: (i) The multiprocessor simulator, (ii) the skeletal algorithm *rsbb*, and (iii) problem-specific procedures and data-structures. Mutual exclusion is enforced by the use of a monitor.

The problem that was chosen was the 30-element 0/1 knapsack problem.[29] The knapsack problem was chosen because of its relative ease of formulation.

In order to fully explore the possible range of the problem space, a suggestion of Horowitz and Sahni[29] has been adopted. They suggested dividing the problem space into several areas represented by different data sets. Six sets of problems were used, with each set consisting of 50 randomly generated problem instances. The sets are described in Table II. To determine the effect of keeping a global list of the nodes visited, the same problems were solved using three different sizes of lists: (i) the first 5 levels (31 nodes in all), (ii) the first 4 levels (15 nodes), and (iii) no global list. The results are shown in Table III.

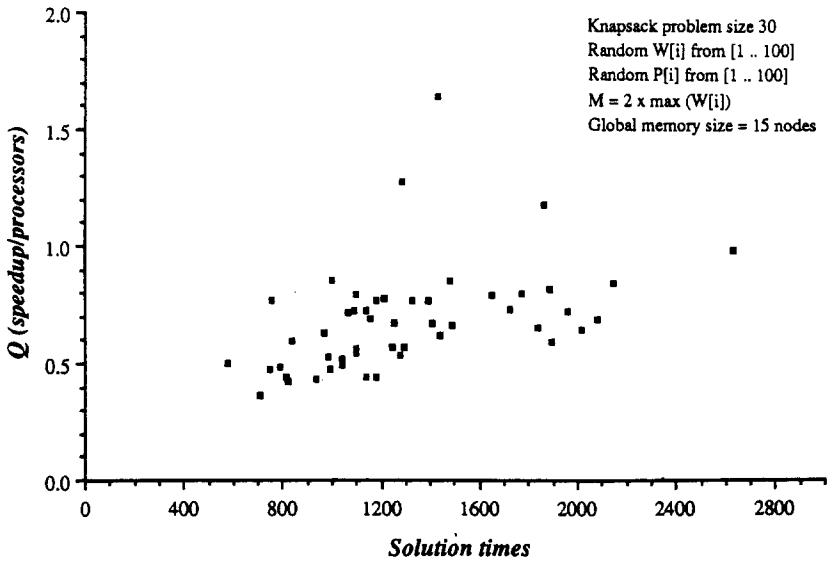

Fig. 8. Scatter plot of solution times for data set 2.

Table IV. Average Speedups Obtained from Simulation (in Terms of VAX Instructions)[a]

Data Set No.	Processors	Speedup (Global Memory Size = 15 nodes)	Range of Uniprocessor Solution Times
1	2	1.63	3.0×10^4–1.70×10^5
	3	2.13	
	4	1.88	
	5	2.73	
2	2	1.51	1.88×10^5–5.20×10^5
	3	2.26	
	4	2.16	
	5	2.91	
3	2	1.62	7.0×10^4–5.1×10^5
	3	2.80	
	4	2.26	
	5	2.55	

[a] Notes: (i) For a description of the data sets see Table II; (ii) Speedups are real-time ratios; (iii) Unit of time is 1 VAX Instruction.

An important observation about the behavior of the algorithm is that large speedups occur in precisely those problem instances that take large times to solve. Fig. 8 shows a scatter-plot of the speedup obtained versus the solution time for a single processor, where this trend is readily discernible. Indeed, some of the problem instances are solved with super-linear speedups. Precisely the same behavior is observed in the randomized backtracking algorithm.[10]

A detailed analysis of the trace files produced by the simulator was done in order to obtain real-time information. Table IV shows the speedups obtained as ratios of actual times. Unfortunately, the size of the trace file thus obtained grows enormously as the solution time or the number of processors increases, and is beyond the capacity of the machine. Hence, it has been possible to obtain results only for a few of the problems, and only for up to five processors. A comparison with Table III shows that the degradation because of global memory accesses is quite small. The analysis for these instances showed that, on average, processors spent about 20% of the total time in accessing the global location containing the cost of the best solution; and about 8% of the time accessing the global list of searched subtrees (for a list size of 15 nodes). It should be noted in this connection that for the knapsack problem relatively simple bounding functions and branching rules can be formulated which incur little computation cost, as compared to, say, the traveling salesman problem. The global memory access percentages would be even better in the latter case.

9. CONCLUSIONS

Traditionally, parallel algorithms have been designed by partitioning a given workload among several processors, which must collaborate to solve the problem. We have proposed a methodologically different approach to algorithm design: using randomized algorithms, each processor is capable of finding the solution by itself. The processors may in part duplicate each other's work, but together they will reach the solution faster than any one of them working alone.

This paper has reported on a study of a randomized branch-and-bound algorithm. A model has been presented to predict the performance that can be realized from solving large branch-and-bound problems on multiprocessors. When this algorithm was actually run on a multiprocessor simulator, the number of nodes it examined was always within 15% of that predicted. The analysis was then extended to predict the speedup for different numbers of processors. The analysis assumes a global memory of unlimited size to record the nodes that have been visited, and thus avoid redundant work. In practice, such a memory is of finite size. This analysis

was again validated by simulation, which showed that a global memory of 31 nodes was large enough so that most problem instances approached their theoretical speedup. Six different instances of the 0/1 knapsack problem were studied. In five of six cases, the measured speedup for 10 processors was within 20% of that predicted. These results indicate that randomized algorithms can achieve impressive performance, and that, despite certain simplifying assumptions, the model is successful in predicting their behavior.

It is significant that this behavior has been obtained with only a small amount of global memory. By contrast, deterministic algorithms may require frequent communication to avoid duplication of work. Randomized algorithms, with much more limited communication, may duplicate work, but our results show that this causes little performance degradation, at least for up to 10 processors.

Finally, randomized algorithms offer significant *reliability* advantages. Because no processor is relied upon to explore a particular portion of the solution space, the failure of a single processor does not prevent a successful solution. Indeed the failure of several processors will at worst cause a graceful degradation in speedup, as long as at least one processor continues to function. Some care must be taken with respect to shared data. Since the randomized algorithms we have described share a rather small amount of information, which needs to be updated relatively infrequently, the shared data can inexpensively be replicated in several memories. When the failure of one of these memories is detected, another copy of the data can be made somewhere else in the multiprocessor. Thus the computation can survive failures of either processors or memory.

REFERENCES

1. A. C. Yao, Probabilistic Computations: Toward a Unified Measure of Complexity, in *Proc. 18th Annual IEEE Symp. on Fundamentals of Computer Science*, New York, pp. 222–237 (1977).
2. W. Dobosiewicz, Sorting by Distributive Partitioning, *Inf. Proc. Lett.*, Vol. 7, No. 1, (January 1978).
3. M. O. Ruban, Probabilistic Algorithm in Finite Fields, Technical Report, Massachusetts Institute of Technology (1979).
4. P. G. Spirakis, Probabilistic Algorithms, Algorithms with Random Inputs and Random Combinatorial Structures, Ph.D. thesis, Harvard University, Department of Mathematics (December 1981).
5. G. L. Thompson and S. Singhal, A Successful Algorithm for Solving Directed Hamiltonian Path Problems, *Operations Research Letters*, 3(12):35–42 (April 1984).
6. M. O. Rabin, Probabilistic Algorithms, in *Algorithms and Complexity*, J. F. Traub (ed.), Academic Press, pp. 21–39 (1976).
7. D. Vrsalovic, D. P. Siewiorek, Z. Z. Segall, and E. F. Gehringer, Performance Prediction

for Multiprocessor Systems, in *Proc. 13th Intl. Conf. on Parallel Processing*, pp. 139–146 (August 1984).
8. G. M. Baudet, The Design and Analysis of Algorithms for Asynchronous Multiprocessors, Ph.D. thesis, Technical Report CMU-CS-78-116, Carnegie-Mellon University Department of Computer Science (April 1978).
9. E. F. Gehringer, A. K. Jones, and Z. Z. Segall, The Cm* Testbed, *IEEE Computer*, pp. 40–53 (October 1982).
10. V. K. Janakiram, D. P. Agrawal, and R. Mehrotra, Randomized Parallel Algorithms for PROLOG Programs and Backtracking Applications, in *Proc. 1987 Intl. Conf. on Parallel Processing*, Chicago, Illinois, pp. 278–282 (1987).
11. H. S. Stone and P. Sipala, The Average Complexity of Depth-First Search with Backtracking and Cutoff, *IBM J. of Res. and Development*, **30**(3):242–258 (May 1986).
12. R. S. Garfinkel and A. L. Nemhauser, *Integer Programming*, New York, John Wiley & Sons, Inc. (1972).
13. B. Lagewes, J. Lenston, and A. Rennooy Kan, Job-shop Scheduling by Implicit Enumeration, *Management Science*, **24**(4) (1977).
14. G. Ingangiole and J. Korsh, A Reduction Algorithm for Zero-one Single Knapsack Problems, *Management Science*, **20**(4):460–663 (1973).
15. G. Ingangiole and J. Korsh, A General Algorithm for One-dimensional Knapsack Problems, *Operations Research*, **25**(5):752–759 (1977).
16. R. Garfinkel, On Partitioning the Feasible Set in a Branch-and-bound Algorithm for the Asymmetric Traveling Salesman Problem, *Operations Research*, **21**(9):340–342 (1973).
17. A. Efroymson and T. C. Ray, A Branch-and-bound Algorithm for Plant Location, *Operations Research*, **14**:361–368 (1966).
18. A. M. Geoffrion and R. E. Marston, Integer Programming Algorithms: A Framework and State-of-the-art Survey, *Management Science*, **18**(9):465–491 (May 1972).
19. T. Lai and S. Sahni, Anomalies in Parallel Branch-and-bound Algorithms, in *Proc. 1983 Intl. Conf. Parallel Processing*, Bellaire, Michigan, pp. 183–190 (1983).
20. B. W. Wah and Y. W. Eva Ma, MANIP—A Multicomputer Architecture for Solving Combinatorial Extremum-search Problems, *IEEE Trans. on Computers*, **C-33**(5):377–390 (May 1984).
21. B. W. Wah and C. F. Yu, Probabilistic Modelling of Branch and Bound Algorithms, in *Proc. of the COMPSAC*, pp. 647–653 (1982).
22. O. I. El-Dessouki and W. H. Huen, Distributed Enumeration on Network Computers, *IEEE Transactions on Computers*, **C-29**(9):818–825 (Sept. 1980).
23. D. R. Smith, Random Trees and the Analysis of Branch-and-bound Procedures, *J. ACM*, **31**(1):163–188 (January 1984).
24. B. W. Wah and C. F. Yu, Stochastic Modeling of Branch-and-bound Algorithms with Best-first Search, *IEEE Trans. on Software Eng.*, **SE-11**(9):922–933 (September 1985).
25. A. H. Stroud and D. Secrest, *Gaussian Quadrature Formulas*, New Jersey, Prentice-Hall (1966).
26. N. G. De Bruijn, *Asymptotic Methods in Analysis*, Amsterdam, North Holland Publishing Co. (1961).
27. W. Feller, *An Introduction to Probability Theory and Applications*, New York, Wiley (1971).
28. E. D. Brooks, A Multitasking Kernal for the C and Fortran Programming Languages, Tech. Rep. UCID 20167, Lawerence Livermore National Laboratory, Livermore, California (September 1984).
29. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Maryland, Computer Science Press (1984).

Printed in Belgium