DIB—A Distributed Implementation of Backtracking

by

Raphael Finkel
and
Udi Manber

Computer Sciences Technical Report #588

March 1985

DIB - A DISTRIBUTED IMPLEMENTATION OF BACKTRACKING*

(Preliminary Version)

Raphael Finkel and Udi Manber

Department of Computer Science
University of Wisconsin
Madison, WI 53706
(608) 262-1204

ABSTRACT

DIB is a general-purpose package that allows applications that use backtrack or branch-and-bound to be implemented on a multicomputer. It is based on a distributed algorithm, transparent to the user, that divides the problem into subproblems and dynamically allocates them to the available machines. The application program needs only to specify the root of the recursion tree, and for any given node in the recursion tree what is the computation involved in this node and how to generate its children. The package runs on the Crystal multicomputer at the University of Wisconsin-Madison. Any number of machines may be devoted to the application. Our initial experience with DIB has been very promising. Applications such as traveling salesman, knapsack, eight queens, and knights tour, have been coded quite easily. We have found that almost perfect speedup is achievable for certain recursive backtrack problems. Only a small fraction of the time is spent in communication.

1. INTRODUCTION

We expect that in the next few years a significant number of professionals from many areas will have access to advanced workstations connected through networks. Many of these workstations will serve as personal computers and will be frequently idle. The ability to distribute computationally intensive jobs will greatly enhance the utilization of such systems.

Although several distributed systems and programming languages have been developed and many more are now being developed, it is apparent from the experience we have so far that writing distributed programs is significantly more difficult than writing sequential programs. Programmers have to deal with a variety of new issues, including synchronization, concurrency, communication protocols, and fault tolerance. Even seemingly simple parts (for example, termination) become complicated and error-prone in a distributed environment. In the current stage distributed programming is almost always left to experts.

* This research was supported in part by the National Science Foundation under grants MCS-8303134 and MCS-8105904, and by DARPA contract N00014-82-C-2087.

There are two complementary approaches to make distributed programming easier. The first approach is to develop better programming languages. Extensive efforts are directed towards this goal and significant advances have been made, but we are still quite far from an "easy-to-use" system. It seems that distributed programs are inherently complex. The second approach, which is the one we are taking, is to develop library packages in specific areas. These packages will be suitable only to specific applications and will lack the generality of a programming language. They will also be probably less efficient than direct implementations of specific algorithms. However, one can hope to make these packages relatively easy to use. In particular, the distributed part of a program can be transparent to the user. Such tools will enable a novice programmer to execute distributed programs that are written in a basically sequential manner. As a result, programmers who are unable or unwilling to master the techniques of distributed programming can still use the full power of networks.

In this paper we present a package called DIB that allows applications that use backtrack or branch-and-bound algorithms to be implemented on a multicomputer. (A *multicomputer* is a collection of *machines*, each with its own local store, that co-operate by exchange of messages.) These algorithms are useful for many classes of problems. The design of DIB emphasizes flexibility and simplicity. The distributed part of the algorithm is hidden from the user. An application using DIB is written as a sequential program.

DIB's requirements from the distributed operating system are minimal. The machines are assumed to be connected by a network that supports a message-passing mechanism. Each machine can send a message to every other machine. The order of message delivery is unpredictable. Messages are put in a buffer in the receiving machine. Once in a while a machine checks its buffer and reads its messages. We do not assume any interrupt capability, so the sending machine has no control over when its message is read. In other words, the machines are assumed to be independent. They may be part of a multicomputer or workstations connected through a local area network. Although DIB is implemented on Crystal, an experimental multicomputer where each machine runs

To be presented at *The Fifth International Conference on Distributed Computing Systems*, Denver, May 1985.

only one task, it can easily be adapted to an environment where each machine is running a timesharing operating system. Each machine may run at a different speed, which may depend on its current load. A machine can stop executing the algorithm for a while and continue later. It can also decide to stop (as a result of a high load, for example). We discuss fault tolerance in section 5.

The distribution of work in DIB is dynamic. When a machine M_1 finishes the work it was given it sends a "request for work" to another machine M_2. If M_2 is currently working it divides its work and sends part of it to M_1. There is no need to know in advance the time it takes to perform a piece of work or the relative speeds of the machines. If M_1 receives what turns out to be the bulk of the work or if M_1 suddenly becomes slow, other machines will finish their work and get M_1's part. Thus, DIB contains an "automatic" load-balancing mechanism. The algorithms for distribution of work, which are described in section 5, are efficient in terms of the amount of communication.

The organization of the paper is as follows. In section 2 we discuss related work. In section 3 we define the class of problems for which DIB is useful. In section 4 we describe the user interface of DIB. In section 5 we discuss several distributed algorithms that are used in DIB. In section 6 we describe the experiments we performed and our experience with DIB. Section 7 contains suggestions for further research, and final remarks and conclusions appear in section 8.

2. RELATION TO OTHER WORK

The field of distributed algorithms is growing. Several approaches to tree-search algorithms have been proposed in the past. In particular, decompositions of alpha-beta search have been the subject of much effort. Baudet[1] gives the entire problem to each machine, with each one constrained to find a solution within a different window. Narrow windows speed up the search. He reports a speedup limited by about 6, no matter how many machines are used. Finkel and Fishburn[2] describe a "tree-splitting" algorithm that maps subtrees of the lookahead tree to machines. The machines are arranged in a static tree, and the mapping assigns subproblems of a given problem to the children of the machine with that problem. The speedup ranges from $n^{1/2}$ on optimally-sorted trees to n on pessimally-sorted trees, where n is the number of machines. Akl and Barnard[3] suggest a method that Finkel and Fishburn[4] call "mandatory work first", in which subtrees can be evaluated in two modes, full and partial. Only one subtree (the one that appears most likely to succeed) is fully evaluated; the others are partially evaluated and later fully evaluated if it turns out that they have a chance to be the best subtree. Allocation of work to machines is identical to the "tree-splitting" method described above. Although Akl and Barnard report poor speedup, later analysis shows that their method results in speedup approximately $n^{0.8}$ for optimally-sorted trees and $n^{0.9}$ for pessimally-sorted trees, given a degree about 40 for the lookahead tree and 2 for the machine tree.

These tree-splitting methods require a fixed machine tree. They suffer from machines sitting idle while work is still pending in other parts of the tree. They also allocate identical amounts of processing power to large regions of the tree, even though the left-most region is most likely to yield results quickly.

Alpha-beta search seems to be the most heavily studied tree-search algorithm. Branch-and-bound algorithms have also been studied. One approach[5] is to have each machine compute one node of the search tree and then re-assign work based on the current bounds and cost functions. The amount of communication needed is quite high. Anomalies have been noticed, both by Finkel and Fishburn and by others[6], in which super-linear speedups can be obtained if a high-quality solution is found early in the search. Li and Wah[7] try to maximize the number of anomalies by suggesting an evaluation order on the tree. Moller-Nielsen and Staunstrup[8] have experimented with many different multicomputer algorithms, including branch-and-bound, and found that good speedups are possible when most of the tree has to be traversed.

All of these approaches fit into a more general class of "quotient-network" algorithms[9], in which a logical problem structure (here, a tree structure) is mapped in some way onto the physical machines. (Under tree-splitting, the map is dynamic, but the machines must be arranged as a tree and the map respects tree level.) The DIB program that we will describe allows a highly dynamic mapping between logical subproblems and machines.

3. THE GENERAL CLASS OF PROBLEMS

In this section we give a brief description of the class of problems that can be supported by DIB. DIB is still in development stage; we expect to extend this class in the future. The problems we consider are such that the computation is performed by traversing a tree. The tree is usually built dynamically during the traversal. Each node of the tree contains data that was received from its parent. According to this data the node may decide to "generate" several children and pass more data to them, or it may decide that it is leaf, in which case it performs some computation and passes the outcome to its parent.

The kind of outcome the computation yields depends on the application. Branch-and-bound applications are described by Lawler[10]. Here, each node v computes a function, called the *objective function*, which depends on values computed along the path from the root to v. We are usually interested in finding the leaf with the minimum value of this function. In many cases, it is possible to determine a lower bound on the value of the objective function in a subtree rooted at a given internal node v. As a result, if this lower bound exceeds the a value already attained by a leaf then there is no need to explore further and v can "decide" that it is a leaf. This is the "bound" part of the branch-and-bound method.

In some applications the outcomes of all the leaves have to be collected. The outcomes of the computation of all the children of a particular node may have to be combined in some way to produce the outcome for this node. The outcome of the computation is then defined as the outcome of the root. Examples of such algorithms are general recursive procedures; the tree corresponds in this case to the recursion tree.

The only major requirement of DIB is that the computation of each subtree (a subtree is defined as a node with all its descendants) can be correctly performed without any knowledge of outcome of any node from another subtree. It may be the case that the outcome of another subtree may affect the efficiency of the computation, but it must not affect the correctness. This requirement is essential in order to be able to divide the computation among the machines. It gives us complete freedom to distribute the work in any way we choose. While it rules out many applications it still leaves a rather large class of programs.

4. USER INTERFACE

To write an application using DIB the user has to supply three procedures, Generate, FirstProb, and PrintAnswer. All these procedures are very similar to the procedures one would expect to have in a sequential program for this application. The main procedure is Generate. Its purpose is to generate children of a given node, thus building the computation tree, and to determine whether a node is a leaf and what to do in this case. FirstProb defines the root of the tree, and PrintAnswer generates the output. The formal definitions of these procedures are given below.

```
procedure Generate(
    var Done : Boolean; (* set by Generate to true if no
        more children can be generated *)
    First : Boolean; (* if true, generate first child of
        Parent; otherwise next sibling of Child *)
    var Parent : ProblemType; (* this is variable since
        some applications need to change the parent after
        learning something about the children *)
    var Child : ProblemType; (* the output of this pro-
        cedure, unless Done = true *)
    var Report : Boolean; (* set to true if an output is
        required after completing the computation in this
        node *)
    var Ans : AnswerType (* what to report, if Report is
        true *) );

procedure FirstProb( var P : ProblemType; Size :
integer);

procedure PrintAnswer(P : AnswerType);
```

The application may supply several additional procedures.

Printing procedures
: used for debugging.

NonTrivial
: returns true if the given subproblem is expected to require substantial amount of time. DIB will only attempt to divide and distribute nontrivial problems, even though there may be idle machines. If the application does not supply this procedure, the default value is true.

UseNewInfo
: accepts new information that has been broadcast by another node. (DIB provides BroadcastInfo, which allows a node to broadcast information, such as a new bound in branch-and-bound, to other nodes.)

ApplicInit
: Initializes data and distributes it to all machines at the beginning of the computation.

Combine
: given the outcomes of all children of a particular node, computes the resulting outcome of that node.

An example of an application is given in Appendix I.

DIB also provides a test procedure which, given a DIB application, produces a sequential program. The test procedure is used in the initial debugging phase. Most errors can thus be eliminated before resorting to distributed debugging which is much harder.

5. THE UNDERLYING DISTRIBUTED ALGORITHM

We begin with some notation. Let T denote the computation tree. We associate with each node v of T a *problem* P_v, which corresponds to the computation of the subtree rooted at v. We assume that a description of P_v includes all the data required to perform the computation. P_{root} is the problem we want to solve. Let P_v be a problem such that v is an internal node and let $u_1, u_2, \ldots, u_k$ be the children of v. To solve P_v one has to solve P_{u_i}, $i = 1,2,\ldots,k$ and then combine the outcomes.

The algorithm is based on a depth-first search of the tree. If there is only one machine then the computation is straightforward. Assume that there are n active machines $M_1, M_2, \ldots, M_n$, $n > 1$. Each machine M_i maintains two tables, *WorkGotten* and *WorkGiven*. *WorkGotten* contains a set of problems that were received from other machines. M_i is said to be *responsible* for these problems. A problem is kept in *WorkGotten* until it is solved (by M_i or other machines), in which case the outcome is reported to its parent problem. *WorkGiven* contains a set of problems that M_i has given to other machines. The purpose of these tables is twofold. First, they are necessary for applications in which the outcomes of the children are collected and combined at the parent (that is, the computation proceeds up the tree as well as down the tree). Second, if several machines fail, then the remaining machines can determine which problems have not been solved and solve them. A subset of the DIB package handles the simpler case of computations that do not require these tables.

The user may associate priorities with subproblems according to their likely chances to lead to best solutions. Such priorities are commonly used in different heuristics for branch-and-bound algorithms. Each machine M_i maintains a *heap* that contains all the outstanding problems M_i has. (There is some redundancy in the purposes of the heap and WorkGotten; it turns out to be easier both conceptually and in terms of the implementation.) The ordering of problems in the heap is based on their priority and their depth. M_i takes problems from its heap and performs the necessary computation. If the heap is empty, M_i sends *requests for work* to other machines, which are selected by an algorithm that will be described later.

A machine M_j that receives a request for work can *grant* this request by sending away some problems. M_j selects the problems it sends away by first checking its heap. If the heap is not empty then a portion of it is sent. Otherwise, if the current problem M_j is working on, P_v, is not trivial (see the previous section), P_v is divided, a portion of it is given away, and the rest is put in the *WorkGotten* table (as being "gotten" from M_j) and in the heap. Hence, a request is not granted only if it is not worthwhile to divide the current problem.

Subproblems with higher priorities will be given away sooner and will be worked on sooner. Obviously, we cannot guarantee any particular order of execution because of the nondeterministic nature of the distribution of work. We only make particular orders more likely.

We are currently experimenting with several algorithms for determining which machine to ask for work, how much work should be given away, and what to do if a request cannot be granted. Several tradeoffs are involved. We obviously want to minimize the number of request and work messages. On the other hand, we want to minimize the idle time of machines that are waiting for work and make the algorithm robust so that it is not dependent on the cooperation of all machines. In addition, the algorithm should detect termination efficiently.

A similar general problem was studied by Manber[11], who suggested several algorithms and proved lower bounds. Only worst-case behavior was considered. One feature of these algorithms that we use here is sending a constant portion of the available work (usually about a half) instead of one unit. This policy minimizes the number of messages but it increases their sizes. In many cases, the overhead in preparing, sending, and receiving one message outweighs the extra cost of larger messages. Also, since the tree is generally not random, one can describe several subproblems with a short description (for example, "all the children from i to j"). We found that communication costs (which are measured by the time spent on communication-related activities, including waiting) can increase by a factor of up to 3 when the portion of work given is small (less then 5%).

The first algorithm for sending request messages that we tried is similar to the simpler algorithm given by Manber. Each machine M_i keeps a variable called *Helper*. Helper is initially set to $(i \bmod n)+1$, which is called the *successor* of M_i. M_i sends a request for work to Helper. If Helper cannot grant the request then it forwards the request to its successor. Once the request is granted, M_i sets the new value of Helper to be the successor of the machine that granted the request. This way requests are distributed fairly evenly around the ring. The machine responsible for the problem associated with the root sends out termination messages to all other machines when the results of all the children of the root come in. There are two main drawbacks in this algorithm. Near termination, most machines have no work, causing most request messages to be forwarded many times. If there are many machines they can flood the network before they detect termination. In addition, the algorithm is not robust. If one machine fails, all requests may stop there.

In the second algorithm, each machine sends requests for work to k other machines that are either selected in the same way as the first algorithm or at random. $k>1$ is a constant which may depend on n and on the application. We used $k=2$, since n was small. Requests are never forwarded: if there is no work to give then the requests are ignored. This way the number of messages is always small even close to termination. The drawback of this algorithm is that a machine might send requests to k other machines that have also asked for work at the same time. None will respond, so the first machine will "give up". We have found several ways to avoid this possibility, but we also found that in general it is unlikely except close to termination, when it makes no difference. In all the applications we tried the amount of time spent in communication related activities was very small (see section 6.2).

We are currently in the middle of implementing fault tolerance in DIB. When a machine M_i finishes its work and its *WorkGiven* table is not empty, then it is likely that the work that M_i gave away went to a failing machine. While M_i is looking for more work it also starts to redo the outstanding work in its *WorkGiven* table. The implementation of the recovery part has not yet been completed, and we omit the details. We are also looking at the more complicated tree algorithms in[11].

6. EXPERIMENTS

6.1. The Crystal Multicomputer

The Crystal multicomputer is discussed in detail elsewhere[12, 13]. It is a collection of VAX-11/750 computers (currently there are 20) with 2MB of memory each, connected by a 10 Mb/sec token ring. Crystal is a vehicle that serves a variety of research projects involving distributed computation. It can be used simultaneously by multiple research projects by partitioning the available machines according to the requirements of each project.

Users can employ the Crystal multicomputer in a number of ways. Projects that need direct control of machine resources can be implemented using a reliable

communication service (the "nugget") that resides on each machine. DIB is implemented with the assistance of a library package (the "simple-application package") that interacts with the nugget. Projects that prefer a higher-level interface can be implemented using the Charlotte distributed operating system. Development, debugging, and execution of projects takes place remotely through any of several VAX-11/780 hosts running Berkeley Unix 4.2.* Acquiring a partition of machines, resetting each machine of the partition, and then loading an application onto each machine may be performed interactively from any host machine.

6.2. Preliminary experimental results

So far we have tried the following applications, which use typical backtracking algorithms.

The eight queens problem
: Find all possible arrangements of N (chess) queens on an $N \times N$ board such that no queen can attack another queen.

knights tour
: Find all possible tours a (chess) knight can take covering the whole board except k squares without repeating a square. (This is basically an Hamiltonian path problem.)

The traveling-salesman problem
: Find a minimal-cost tour of cities. Inter-city costs are initialized from a uniform distribution in [0,1]. We used the straightforward implementation trying all possible tours; at every node of the tree the cost of the partial tour is compared to the current minimum; a new minimum is broadcast to all machines.

Petri nets reachability problems
: Given a Petri net and a state find all states reachable from this state. Only simple algorithms have been implemented so far.

knapsack
: Given a set of items, each having a size and value, find a subset of items such that the sum of the sizes is bounded by a given constant and the sum of the values is maximized.

Figure 1 displays total running times for the traveling salesman problem on 11 cities for 1 to 16 machines. Figure 2 shows similar results for a version of traveling salesman that sorts the children, always visiting the closest city first. The times are given in seconds; they were measured by the host machine. Distribution of work was handled by the first algorithm described in the previous section.

The distances among the cities were selected at random. The underlying algorithm is the straightforward naive algorithm that tries all possibilities except when a bound cuts off a subtree. Its performance is far from that of more sophisticated algorithms. It is used here to demonstrate the characteristics of DIB, which, as a result of its generality, cannot compete with very well tuned implementations.

Figure 1: Traveling Salesman Problem

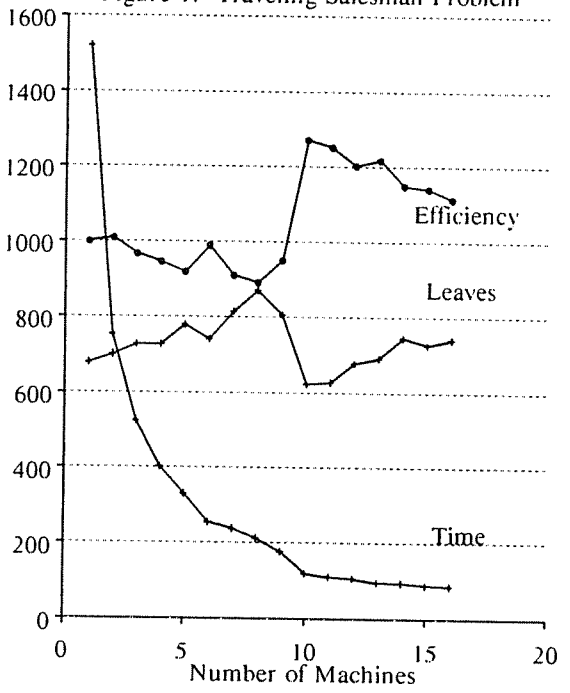
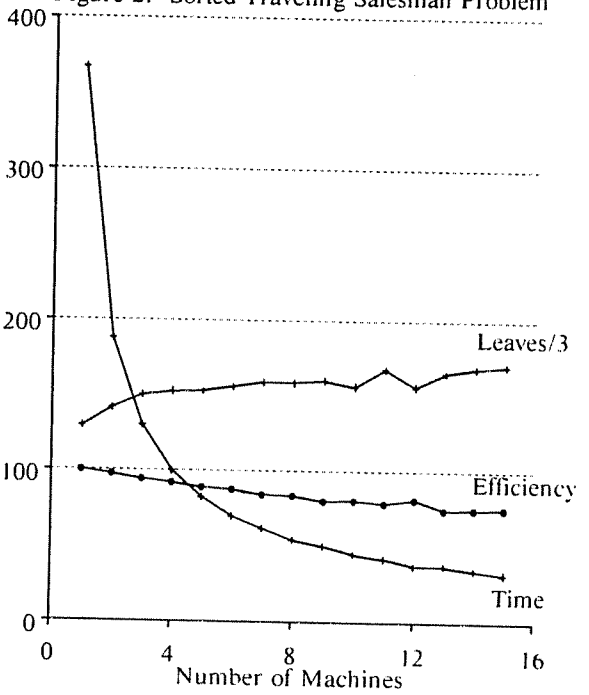


Figure 2: Sorted Traveling Salesman Problem



* Unix is a trademark of Bell Labs

The figures also show the **efficiency**, which is defined as the speedup divided by the number of machines. A score of 1000 (Figure 1) or 100 (figure 2) represents perfect efficiency. This figure demonstrates that the efficiency and the amount of work (as shown by the number of leaves examined) mirror each other.

The algorithm reaches a leaf only if the cost of a partial tour without the leaf is still smaller than the current lower bound. The sooner good lower bounds are found the less work the algorithm has to perform. Sometimes, by dividing the problem differently among the machines, the best tour is found earlier and the performance of the algorithm is substantially improved. As a result, it is possible to get better than a linear speedup on the one hand, and "slowdown" when the number of machines increases on the other hand. The number of leaves the algorithm reaches is thus a measure of the amount of work performed. Both figures show the number of leaves; Figure two divides the number by 3 to fit on the same graph. The efficiency and the number of leaves are roughly mirror images.

Table 1: Distribution of work

Machine	computation time	communication time
1	549.606	2.780
2	549.316	2.881
3	549.585	2.688
4	549.292	2.909
5	549.338	2.912
6	549.001	3.227
7	549.425	2.894
8	549.019	3.299
9	548.648	3.627
10	549.592	2.755
11	549.544	3.125
12	549.325	3.291
13	549.100	3.533
14	549.513	3.155
15	549.289	3.321
16	549.672	3.024

Table 1 presents the individual machine running times for the 14-city travelling salesman problem executing on 16 machines. The most striking feature of the algorithm is its success in dynamically distributing work evenly among the machines. The running times of any application were divided into three parts: computation time, inter-machine communication time, and communication to the host. Communication to the host included basically only I/O, and since we only output the number of leaves and the best tour, this part was negligible. Inter-machine communication included the time it takes for a machine to perform all the activities connected with communication. We measured all the overhead resulting from the communication instead of just measuring the delay time directly in the network. We included the time it takes to generate a message, to wait for an answer, and to receive, relay, and process a message.

7. FURTHER RESEARCH

There are several directions for advancing this research. We are currently looking at the following areas.

- Adapting DIB to more applications by supplying more routines for synchronization, broadcast, and distribution of work.
- Implementing robust fault tolerant facilities. In particular, we are studying recovery mechanisms that do not require timeouts. In an environment of personal workstations failures can be caused by the "owner" of the machine who decides to stop all the "foreign" processes. Hence, the likelyhood of failures is independent on the hardware.
- Analyzing the performance of the various algorithms for work distribution, distributed termination, and fault tolerance.
- Tuning the algorithms to particular applications.
- Adapting DIB to be used as an instructional tool in networking and distributed computing courses.

8. CONCLUSIONS

DIB is a first attempt at providing an easily used facility for writing distributed programs involving backtracking. The distributed part of DIB is completely hidden from the user, making it especially suitable for programmers without expertise in distributed or parallel programming. It requires minimal support from the distributed operating system and, as a result, it should be relatively portable. Our results so far indicate that for simple programs DIB is an efficient and very easy to use tool to automatically distribute work in a distributed environment.

REFERENCES

1. G. M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. Department of Computer Science, Carnegie-Mellon University (April 1978).
2. R. A. Finkel and J. P. Fishburn, "Parallelism in Alpha-Beta Search." *Journal of Artificial Intelligence* **19**(1)(September 1982).
3. S. G. Akl, D. T. Barnard, and R. J. Doran, "Simulation and analysis in deriving time and storage requirements for a parallel alpha-beta algorithm." *Proc. 1980 International Conference on Parallel Processing*, pp. 231-234 (August 1980).
4. R. A. Finkel and J. P. Fishburn, "Improved speedup bounds for parallel alpha-beta search." *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-5**(1)(January 1983).
5. G. B. Adams III and H. J. Siegel, M. Imai, T. Fukumura, and Y. Yoshida, "A parallelized branch-and-bound algorithm: implementation and efficiency." *Systems, Computers, Controls* **10**(3) pp. 270-277 (1979).

6. T-H Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," *CACM* **27**(6) pp. 594-602 (June 1984).

7. G-J Li and B. W. Wah, "Computational efficiency of parallel approximate branch-and-bound algorithms," *Proceedings of the International Conference on Parallel Processing*, (August 1984.).

8. P. Moller-Nielsen and J. Staunstrup, "Experiments with a Multiprocessor," *Technical Report PB-185*, Computer Science Department, AArhus University, (November 1984).

9. J. A. Fishburn and R. A. Finkel, "Quotient networks," *IEEE Transactions on Computers* **C-31**(4)(April 1981).

10. E. L. Lawler and D. Wood, "Branch and Bound methods: a survey," *Operations Research* **14**(4) pp. 699-719 (1966).

11. U. Manber, "On Maintaining Dynamic Information in a Concurrent Environment," *Sixteenth Annual ACM Symposium on Theory of Computing*, pp. 273-278 (April 1984). revised January 1985

12. D. DeWitt, R. Finkel, and M. Solomon, "The Crystal multicomputer: Design and implementation experience," Technical Report 553, University of Wisconsin-Madison Computer Sciences (September 1984).

13. R. Cook, R. Finkel, D. DeWitt, L. Landweber, and T. Virgilio, "The crystal nugget: Part I of the first report on the crystal project," Technical Report 499, Computer Sciences Department, University of Wisconsin (April 1983).

14. Raphael Finkel, Robert Cook, David DeWitt, Nancy Hall, and Lawrence Landweber, "Wisconsin Modula: Part III of the First Report on the crystal project," Technical Report 501, University of Wisconsin-Madison Computer Sciences (April 1983).

Appendix I: a sample application program

This example is presented in Modula, the language in which we have done our experiments[14]. For conciseness, we omit the formal-argument declarations for Generate, since we have already presented them earlier in the paper.

```
module queens;

const
  MaxProbSize = 20;
type
  QueenArray = array 1:MaxProbSize of integer;
  ProblemType =
    record
      Size : integer;
      Length : integer;
      Queens : QueenArray
    end;
  AnswerType = ProblemType;

procedure PrintAnswer(const P : ProblemType);
var
  Seq : integer;
begin
  with P do
    Seq := 1;
    while Seq <= Length do
      printf(" %d",Queens[Seq]);
      inc(Seq);
    end;
    printf("0);
  end;
end PrintAnswer;

procedure Generate : (* parameters were defined in section 4 *)
var
  Seq : integer;
  ThisQueen, ThatQueen : integer;
  Bad : Boolean;
begin
  if First and (Parent.Length = Parent.Size) then
    Report := true;
    Done := true;
    Ans := Parent;
  else
    (* not a leaf *)
    Report := false;
    if First then
      Child := Parent;
      with Child do
        inc(Length);
        Queens[Length] := 0;
      end;
      First := false;
    end;
    with Child do
      loop (* search for a reasonable place for current queen *)
        inc(Queens[Length]);
        ThisQueen := Queens[Length];
        when ThisQueen > Size do
          Done := true;
          exit;
        Seq := 1;
        Bad := false;
        while Seq < Length do
          (* check if Queens[Seq] conflicts with Queens[Length] *)
          ThatQueen := Queens[Seq];
          when (ThatQueen = ThisQueen) or
            (ThatQueen + Seq - Length = ThisQueen) or
            (ThatQueen - Seq + Length = ThisQueen) do
            Bad := true;
            exit; (* not a solution *)
          inc(Seq);
        end; (* while Seq < Length *)
        when not Bad do
          Done := false;
          exit;
      end; (* search for reasonable place for current queen *)
    end; (* with Child *)
  end; (* not a leaf *)
end Generate;

procedure NonTrivial(const P : ProblemType);
begin
  NonTrivial := P.Length < P.Size-2;
end NonTrivial;

procedure FirstProb(var P : ProblemType);
begin
  P.Size := Size;
  P.Length := 0;
end FirstProb;

end queens.
```