PROCEEDINGS 1981 NATIONAL COMPUTER CONFERENCE, AFIPS PRESS.

The architecture of MANIP—a parallel computer system for solving NP-complete problems

by BENJAMIN W. WAH
Purdue University
West Lafayette, Indiana

and

Y. W. MA
University of California, Berkeley
Berkeley, California

ABSTRACT

In this paper, we study the network architecture of MANIP, a parallel Machine for processing Non-determInistic Polynomial complete problems. The most general technique that can be used to solve a wide variety of NP-complete problems on a uni-processor system, optimally or sub-optimally, is the branch and bound algorithm. We have adapted and extended the branch and bound algorithm for parallel processing. The parallel branch and bound algorithm requires a combination of sorting and merging. A common memory to sort for a large number of processors can become a bottleneck in the system. We have proposed a system with distributed intelligence so that sorting can be carried out in a distributed fashion. A uni-directional ring network is proved to be the optimal and most cost-effective inter-processor communication network when sorting is done by a hardware priority queue in each processor.

I. INTRODUCTION

A class of common, deterministic problems defined in computer science, operations research, and other application areas is the *NP-complete problems*.[35] This class of problems is characterized by a deterministic algorithm that computes a function from a countable domain into a countable range, and it generally involves the optimization of an objective function. The computation time for all known optimal algorithms for this class of problems increases exponentially with the problem size, i.e., if n represents the size of the problem, then the computation time goes up as k^n where $k > 1$. There is a subclass of NP-complete problems called *strong NP-complete problems*[16] such that there is no "pseudopolynomial" algorithm which solves the problem in a time bounded by a polynomial in the input length and the magnitude of the largest number in the given problem instance. The implication of a problem being strongly NP-complete is that there is no *fully polynomial time approximation scheme* which solves the problem in a time bounded by a polynomial in the input length and the reciprocal of the prescribed degree of accuracy. Many problems in areas like deterministic scheduling, graph theory, routing, database, mathematical programming, automata and language theory, image processing, microprogram optimization, etc., have been proved to be either NP-complete or strongly NP-complete.[17] The set of NP-complete problems therefore spans a wide spectrum of application areas.

We are presenting in this paper the architectural design of a parallel computer system that can be used to solve *NP-complete problems without fully polynomial time approximation schemes*. Since the time complexity to solve these problems optimally is exponential, the common approach is to solve optimally for small problems and to solve sub-optimally using heuristics for large problems. The most general technique that can be used to solve a wide variety of these problems, optimally or sub-optimally, is the branch and bound algorithm.[44] The branch and bound algorithm will be discussed in detail in Section II. Conventionally, the branch and bound algorithm has generally been studied with respect to limited memory space, the selection and bounding criteria, the theoretical behavior, and the adaptation to a single computer system. What little work that has gone on from the viewpoint of parallelism has been directed toward a general purpose computer network.. The problem of the necessary parallel computer architecture and its associated operating system to provide an execution environment for a branch and bound algorithm has been little studied or less understood. The significance of this study therefore lies in two aspects. First, it can result in the design of a special purpose VLSI parallel computer system to execute the parallel branch and bound algorithm. The number of computers can be designed to fit the need of the applications. Second, with a better understanding of the parallel branch and bound algorithm, it can be designed into existing computer networks and distributed computer systems.

The feasibility of this study has greatly increased with re-

cent changes in the state of the art in memory; VLSI, and communication technologies. The cost per unit of memory is decreasing and a wide variety of new storage devices, such as CCD memory, bubble memory are available. A number of other technologies are undergoing intensive study, including holographic, laser and other optical, and magneto-optical, and have the potential for commercial development within the next decade. At the same time, the number of components per chip is doubling each year and there is a trend of increased specialization in the functions of the VLSI chip[15, 40] and the design of a single chip computer system.[55] A conference was held in Caltech in 1979 to investigate the potential of VLSI technology.[62] Lastly, the improvement in wideband communication technology allows local or remote computers to be interconnected together using optical fibers and satellites.

With the economic feasibility and consequent existence of these new technologies, more powerful search strategies can be used in the branch and bound algorithm. Traditional implementation of branch and bound algorithm is faced with the problem of limited memory size. With larger and inexpensive secondary storage, the branch and bound algorithm can be designed with a virtual backing store. Candidate problems unlikely to lead to the optimal solution can be stored in the secondary storage. The conventional virtual memory system does not work very efficiently here because the access characteristics of a branch and bound algorithm are significantly different from the access characteristics of a program. The complexity of the problem is compounded as parallel computers are used. Another problem faced in the efficient implementation of branch and bound algorithm is sorting. In order for the execution time to be minimum, single processor implementation sorts the intermediate sub-problems by the lower bounds in ascending order and the sub-problem with the minimum lower bound is picked up for expansion (best first search). Other heuristic search strategies may also involve searching through a set of values generated by heuristic functions.[29] In a parallel computer system, the requirement that the global set of subproblems are completely sorted by lower bounds can be relaxed. Suppose there are n processors, it is sufficient to place one of the n sub-problems with minimum lower bounds in each processor and not important which one of these n sub-problems is evaluated in a particular computer. This relaxed sorting requirement can be incorporated into the design of a more efficient architecture than conventional architectures that perform complete sorting.

In this paper, we study the network architecture of MANIP, an architecture using VLSI technology to implement a parallel branch and bound algorithm. We want to design special purpose processors for evaluating the bounds and simple interconnection network for interconnecting the processors. The system is designed with the following design objectives: First, the system should be modularly expandable to include a very large number of processors. Second, the design must have high performance and the cost should be low by replicating simple cells. Third, the system should use distributed control so that there would not be a controller that becomes the bottleneck in future system expansion. Fourth, efficient load balancing strategies should be implemented so that the processors can be kept busy most of the time. Lastly, the system should be recoverable from hardware failures.

This paper is divided into five sections. Section II presents the branch and bound algorithm and the parallel version of the branch and bound algorithm and discusses the previous work on parallel computer architecture for branch and bound algorithm. Section III identifies the architectural alternatives in implementing the parallel branch and bound algorithm. Section IV presents the network architecture and its optimality. Lastly, section V provides some discussions on the problem of implementation, the performance of the network, and gives some concluding remarks.

II. PREVIOUS WORK

A. *Parallel Branch and Bound Algorithm*

An NP-complete problem is usually put into the form of a constrained optimization problem*:

$$\begin{array}{ll} \text{minimize} & C_0(x) \\ \text{subject to} & g_1(x) \geq 0 \\ & g_2(x) \geq 0 \\ & \vdots \\ & g_m(x) \geq 0 \\ \text{and} & x \in X \end{array}$$

where X represents the domain of optimization defined by the m constraints, normally an Euclidean n-space, and x denotes a vector $(x_1, x_2, \ldots, x_n)$. A solution vector that lies in X is said to be a *feasible solution* and a feasible solution for which $C_0(x)$ is minimal is said to be an *optimal solution*.

Many methods exist to solve for the optimal solution in the aforementioned optimization problem. Some of these are specially designed techniques like Gomory's cutting plane method for solving integer programming problems. However, the most general algorithm, although sometimes not the most efficient, is the branch and bound algorithm. In this section, we describe the branch and bound algorithm and expand the algorithm into a parallel version so that it can be implemented on a parallel computer system.

1. Previous work on branch and bound algorithm

The branch and bound algorithm is an organized and intelligently structured search of the space of all feasible solutions. It has been extensively studied in areas such as artificial intelligence and operations research.[27, 44, 48, 53] It has been applied extensively to solve problems in scheduling,[41, 46] knapsack,[33, 34] traveling salesman,[18, 24, 25] facility allocation,[11, 59] integer programming,[19,20] and many others. Dominance relation similar to that used in dynamic programming has been used to prune search tree nodes.

Theoretical properties of the branch and bound algorithm have been developed in several studies.[28–32, 37, 53] One study[29] shows that depth-first search, breadth-first search and best-

* There are also problems which are not NP-complete and are put into this form.

first search are special cases of heuristic search. In heuristic search, an evaluation function $f(n)$ for a sub-problem n is computed as the sum of cost of an optimal path from a given start node to n and cost of an optimal path from n to a goal. An ordered search algorithm picks up a sub-problem with the minimum value of f for expansion each time. Any general heuristic functions can be included in the computation and the choice of a heuristic function depends on the application.

2. Essential features of branch and bound algorithm

In branch and bound algorithm, the space of all feasible solutions is repeatedly partitioned into smaller and smaller subsets and both the lower and upper bounds are calculated for the cost of solutions within each subset. After each partitioning, these subsets with a lower bound (in the case of minimization) that exceeds either the cost of a known feasible solution or the least upper bound of all the subsets, are excluded from all further partitioning. The partitioning process continues until a feasible solution is found such that the cost is no greater than the lower bound for any subset. The state of the partitioning process at any time can be represented as a partial tree (Figure 1). Each node in the tree represents a partition and is termed *sub-problem.* The partitioning process selects a partition and breaks up this partition into smaller partitions which in essence extends the node in the partial tree representing this partition by one level and using the sons to denote the smaller partitions. In Figure 1, node j is expanded in the partitioning process into k other partitions which are represented as sons of node j in the partial tree.

There are two essential features of a branch and bound algorithm, namely, the branching rule and the bounding rule. Let us discuss these with respect to the tree in Figure 1. Each node in the partial tree has two numbers associated with it—the upper bound and the lower bound of the sub-problem. The leaf nodes in the partial tree are candidates for partitioning. We say that a leaf node of the partial tree whose lower bound is less than both the value of a known feasible solution and the greatest upper bound of all leaf nodes is *active;* otherwise it is designated as *terminated,* and need not be considered in any further computation.

The branching algorithm examines the set of active leaf nodes and selects one for expansion based on some pre-defined criterion. If the set of active nodes is maintained in a first-in-first-out (FIFO) list, the algorithm is called a *breadth-first search.* If the set is maintained in a last-in-first-out list, then the algorithm is termed *depth-first search.* Lastly, if the node selected for expansion is one with the minimum lower bound, then the search algorithm is called a *best-first search.* In a breadth-first search, the nodes of the tree will always be examined in levels; that is, a node at a lower level will always be examined before a node at a higher level. This search will always find a goal node nearest to the root. However, the sequence of nodes examined is always predetermined and therefore the search is "blind." The depth-first search has a similar behavior except that a sub-tree is generated completely before the other sub-trees are examined. In both of these alogrithms, since the next node to be examined is known, the state of the parent node leading to the next node

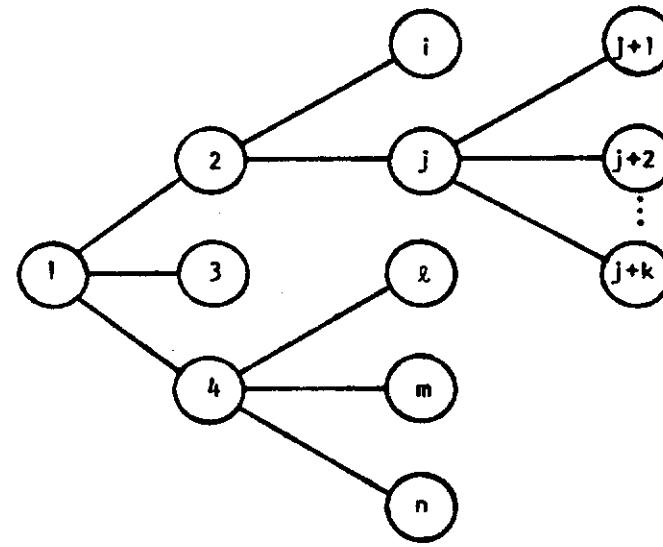


Figure 1—A branch and bound tree

does not have to be kept because the path to the next node from the root node is easily found and unique. These two algorithms are therefore somewhat space-economical. On the other hand, the best-first search is space consuming because all the active sub-problems must be stored as intermediate data in the computer. However, the total number of nodes expanded is minimized in the sense that any branching operation performed under this policy must also be performed under other policies, provided that all the bounds are unique.[44] Since time is a critical factor in evaluating large NP-complete problems, we will implement the best-first branching algorithm in MANIP. The large intermediate storage problem can be solved by moving sub-problems with large lower bounds to the secondary storage.

Once the sub-problem has been selected for partitioning, the next task is to select some undetermined parameters in the sub-problem in order to define alternatives for these parameters and create multiple sub-problems. For example, in the traveling salesman problem, the undetermined alternatives are the set of untraversed edges. In expanding a sub-problem, an untraversed edge (i,j) is selected and two alternatives can be created, namely, the edge is traversed and that the salesman goes directly from city i to city j and vice versa. The parameter chosen to be expanded is usually done in a rather ad hoc fashion.

After new sub-problems are created, the bounding algorithm is applied to evaluate the upper and lower bounds of a sub-problem. In general, only the lower bound is evaluated because the merit of using the upper bound is very small. The bounding algorithm designed is highly dependent on the problem. For example, in an integer programming problem, a linear program with the integer constraints relaxed can be used as a lower bound;[43] in a traveling salesman problem, an assignment algorithm[10] or a spanning tree algorithm can be used as the bounding algorithm. We present an example of an NP-complete problem, the vertex covering problem[17] in order to illustrate the parallel branch and bound algorithm.

In the *vertex covering problem,* the problem is to find, in an

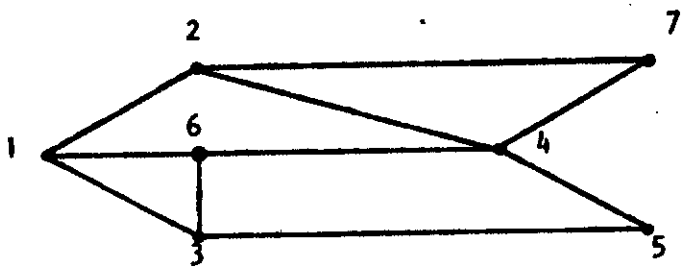


Figure 2a—An example graph

undirected graph, the minimum number of vertices that are needed to "cover" all the edges in the graph. (*Cover* means that all the edges in the graph emanate from at least one of the included vertices.) The branching rule uses the best-first search and branches on an unselected vertex with the largest out-degree. Two sub-problems can be created, one including this vertex in the set and one excluding it. The lower bound in the bounding rule is chosen to be the minimum number of unselected vertices such that the total out-degree is greater than or equal to the number of uncovered edges. Notice that edges emanating from different vertices in the lower bound calculation may overlap and therefore this vertex does not necessarily cover all the uncovered edges. Further, if a vertex has been excluded in a previous stage and there are uncovered edges emanating from this excluded vertex in the current sub-problem, the unselected vertex covering these edges must be included in the minimal set first. As an example, the branch and bound tree for the graph in Figure 2a is shown in Figure 2b.

3. The parallel branch and bound algorithm

We identify three sources of parallelism in the branch and bound algorithm.

a. *Parallel evaluation of subproblems.* Since multiple sub-problems are available, they can be evaluated simultaneously. Due to overheads in inter-processor communications and sorting, and because some sub-problem evaluations are unnecessary, the improvement in execution time is usually less than n times (n is the number of processors). For example, Figure 2c shows the parallel evaluation of branch and bound algorithm on the graph in Figure 2a using two processors. It is seen that the parallel evaluation of node 2 in Figure 2c is not useful, since the corresponding node 3 in Figure 2b is not evaluated. When the problem size is large, the parallelism will contribute to better improvement in execution time.

b. *Parallel sorting of subproblems.* In the best-first search, the list of sub-problems must be maintained in a sorted order by the lower bounds. This sorting can be done by parallel architecture such as Batcher's sorting network.[2] In Section III we give a discussion on the type of interconnection network required for parallel sorting.

c. *Parallel execution of the bounding algorithm.* Specially designed architecture can be used to implement a bounding algorithm. For example, if the simplex algorithm is used, then matrix manipulation hardware is helpful. However, the architecture is designed for solving general NP-complete problems. therefore the bounding algorithm has to be changed for different problems. In this case, software implementation of the algorithm is more cost effective.

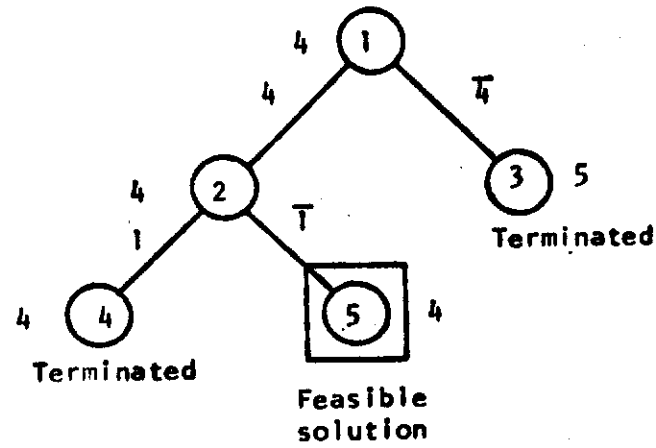


Figure 2b—The branch and bound tree for Figure 2a. (The number in the node indicates the order of evaluation; the number outside the node indicates the lower bound, the number on the edge indicates the included or excluded node.)

4. Efficiency considerations

Many results have been proved for the non-parallel version of the branch and bound algorithm.[28, 31, 37] It has been shown that the best-first search is the best branching rule and minimizes the number of sub-problems expanded.[44] Furthermore the branch and bound algorithm can be used as a genera purpose heuristic to compute solutions that differ from th optimum by no more than a prescribed amount.[44] Suppose i was decided at the outset that a deviation of 10% from th optimum is tolerable. If a feasible solution of 150 is obtained then all sub-problems with lower bounds of 136.4 or mor (= 150/1.1) will be terminated. This technique significantl reduces the amount of intermediate storage and the time t

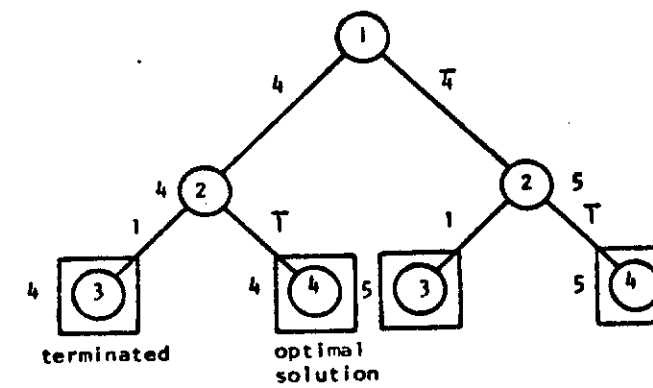


Figure 2c—The parallel branch and bound tree for Figure 2a with two processors

arrive at a sub-optimal solution. Technique is also available to find the best solution in a given length of time.[44] It consists basically of searching for an optimal solution for a length of time equal to T/2. If one is not found, then search is continued for a sub-optimal solution that differs from the optimal by no more than 5% in time of length T/4. The time for searching is halved each time while the precision of the solution is reduced until a solution is found. All these can be incorporated into the parallel branch and bound algorithm.

Unfortunately, very little can be said about the efficiency of the parallel branch and bound algorithm. It was found in one study[44] that only those sub-problems with lower bounds smaller than the optimal solution will be evaluated in a branch and bound algorithm. For a parallel branch and bound algorithm, the improvement in execution time will be n times (n is the number of processors) if the number of sub-problems in the intermediate list with lower bounds smaller than or equal to the optimal solution is always greater than or equal to n. However, this number is highly dependent on the problem and the partitioning being carried out earlier. Simulations are used to find the speed improvement using n processors.

B. Parallelism in NP-Complete Problem Evaulation

Many studies have been made to design multiple computers to speed up problems in searching. Kuck[39] has provided a survey on using parallelism to evaluate arithmetic expressions and linear recurrences, and execute programs. Tree structured architectures are proposed to solve problems in searching[6] and database.[63] One of the tree architectures proposed to solve a wide variety of problems is the X-tree.[7] Alpha-beta algorithm has been proposed to be evaluated on a tree architecture[1] and a general purpose network computer.[14] Decision tree evaluation is also speeded up by using associative processors.[47] A variety of SIMD and MIMD interconnection networks have been proposed for processor-processor communication or processor-memory communication. Examples of these include Benes,[5] indirect binary n-cube,[56] banyan,[21] STARAN's flip network,[3] Omega,[45] data manipulator,[13] ILLIAC IV's mesh,[38] perfect shuffle,[64] PM21,[62] delta,[54] reverse exchange network,[67] etc. However, these networks are usually designed for general purpose applications and therefore the necessary features for processing NP-complete problems are not identified. Our study identifies the necessary architectural features and therefore would provide insights to evaluate NP-complete problems on these computers.

Harris and Smith[22] proposed a tree architecture to solve the traveling salesman problem. Basically, the system dedicates one subproblem to each processor and this processor reports to its parent processor when the evaluation is complete. Because of the limited degree of communication, some processors may be working on tasks that can otherwise be eliminated if a better interconnection network is designed. Desai[8, 9] also proposed a staged MIMD system to solve an 0-1 integer program using implicit enumeration. Nevertheless, implicit enumeration is time consuming and wasteful, and for NP-complete problems, the critical issues of exponential space or exponential time must be addressed in the algorithm.

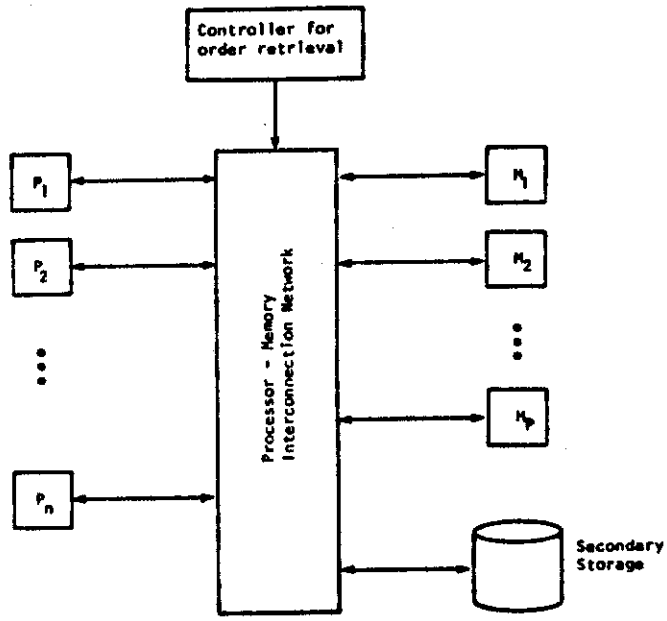


Figure 3a—Common memory

The only published work on applying branch and bound algorithm to solve NP-complete problems is by El-Dessouki and Huen.[12] A general purpose network computer is assumed. Due to memory space limitation, depth first search is used to evaluate sub-problems. Because the network is assumed to be slow and possibly distributed geographically, extensive inter-processor communication cannot be done. No performance results are given on the evaluation of example NP-complete problems. However, depth first search has been shown to be sub-optimal in minimizing the execution time of the branch and bound algorithm.[44] In the light of VLSI technology, larger and inexpensive memories, and faster communication media, the consideration of reducing the execution time (at the expense of larger memory space requirement) is a more critical problem.

In the next two sections, we present the architecture required to support the parallel branch and bound algorithm. We first compare two architectural alternatives and prove that the uni-directional ring network is the optimal interconnection network.

III. ARCHITECTURAL ALTERNATIVES SUPPORTING THE PARALLEL BRANCH AND BOUND ALGORITHM

There are basically two architectural alternatives to implement a parallel branch and bound algorithm, that is, the parallel processors can be interconnected either through a common memory or directly to each other.

A. Common Memory (Figure 3a)

In this implementation, the memory to store the sub-problems is separated from the processors. Because a single

memory would become a bottleneck in the accesses, multiple memory modules would have to be used. There is a processor-memory interconnection network that connects the processors and memories together. The number of memory modules used depends on the frequency of accesses from the processors which in turn depends on the complexity of the bounding algorithm. Sub-problems generated by the processors are stored through the interconnection network in the memories. A secondary storage is also connected to the network for extended storage.

Since it is required to order the sub-problem in ascending order by their lower bounds, the memory must be capable of order retrieval of the sub-problems. This means that each memory module must be capable of order retrieval of the sub-problems, and an external interconnection network must be capable of merging the extrema obtained from each module. The memory modules can be implemented with associative memory,[58] or they can be implemented as VLSI priority queues.*[40] The processor-memory interconnection network can be designed for merging the sub-problems with minimum lower bounds from each memory module. Suppose there are n processors and p memory modules, then n sub-problems with minimum lower bounds in each memory module are fed to the sorting network. Sorting algorithms have been developed on mesh computers,[52,65] perfect shuffle,[64] Batcher's odd-even merging network,[2,61] and others.[42,49] Optimal sorting networks have been investigated by Muller and Preparata,[51] Baudet and Stevenson,[4] Hirschberg,[26] and Preparata.[57] It was found that sorting of n numbers can be done in time $O(\log n)$ with n^2 intermediate processors. Since sorting networks do not allow intermediate results to be used until the sorting is completed, the maximum speed improvement that we can have is $O(\frac{n}{\log n})$, assuming that the number of iterations is improved by a factor of n.

The above sorting process is carried out in a decentralized fashion in the common memory. On the other hand, it is possible to perform sorting in a distributed fashion by sorting the sub-problems locally in each processor and exchanging messages among the processors. This leads to the second architectural alternative.

B. Private Memory (Figure 3b)

In this alternative, each processor has a private memory and is implemented as a unit. Sorting of sub-problems by lower bounds is carried out locally within each processor. Since it is not sufficient for the processor to work on local minima only, a processor-processor interconnection network is used so that local minima from different processors can be sent over the network and distributed. The objective of the distribution is to disribute the n global minima so that each processor has one

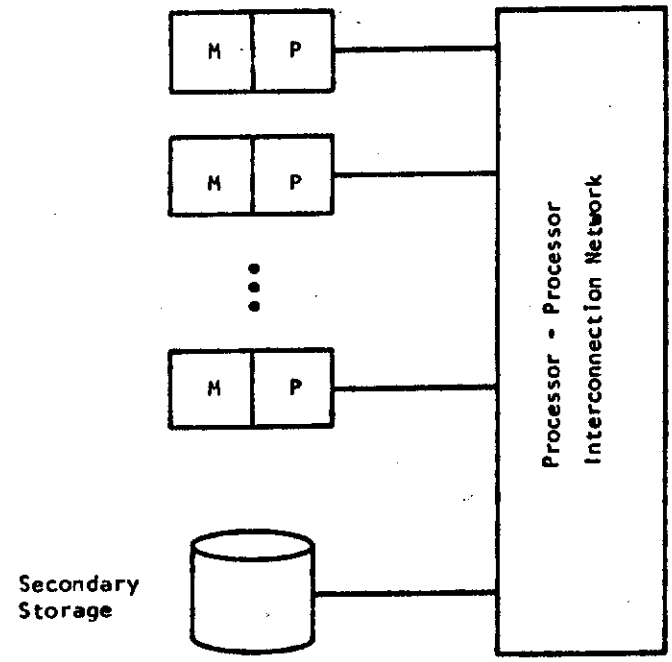


Figure 3b—Private memory

of the global minima. Expansion of the global minima, as mentioned earlier in Section II, is the most effective criterion of the parallel branch and bound algorithm.

The parallel branch and bound algorithm is implemented in this architecture as follows. Each processor keeps a list of sub-problems that are sorted by the lower bounds in ascending order. At the beginning of a cycle, each processor picks up the sub-problem with the minimum lower bounds in its list and expands it into two or more sub-problems. The lower bounds for the expanded sub-problems are evaluated and the sub-problems are inserted back into the list. The local minima from each processor are sent to neighboring processors and inserted into the local lists there. This process repeats until the n global minima are distributed one to each processor. The cycle then starts anew until each processor has one of the n global minima. This procedure can be improved by overlapping the distribution with the expansion of the sub-problems. Two possibilities can occur. First, the distribution time can be smaller than or equal to the lower bound evaluation time. So, although the distribution is completed, the distribution must be carried out again when the lower bounds for the currently expanded sub-problems are available. Second, the distribution time can be greater than the lower bound evaluation time, so the processors remain idle until the distribution is complete. In both of these cases, complete overlap is not attained due to the different processing and distribution times. A compromise can be made by overlapping the sub-problem expansion with the sub-problem distribution. In the case that the distribution is completed first, a local sorting can be performed when the sub-problems are evaluated and the processors expand the local minima without waiting for a complete distribution. In the case that the sub-problems evaluation is completed first, the next sub-problem in the local list can be evaluated immediately without waiting for the distribu-

* A VLSI priority queue is a distributed logic device that maintains the sub-problems in a sorted order. The logical structure is a two input, two output device (deque) such that tags can be input or output from the top or bottom. Comparators are inserted between consecutive elements in the queue. For any two consecutive elements, if the top element is greater than the bottom element, these two elements are exchanged. By this means, larger elements are "dropped" to the bottom of the queue and smaller elements "float" to the top of the queue. Further, elements can be inserted into the queue continuously without waiting for the previous element to be sorted in the queue.

tion to complete. It is shown in Section IV that this strategy is actually very effective.

C. Discussion

There are advantages and disadvantages associated with each of the architectural alternatives presented in this section. The first approach can either be fast (the processor-memory interconnection network is a hardware sorting network such as Batcher's network[36]) and expensive or slow and not quite expensive (an external software sorter is used). Nonetheless, the interconnection network available today generally possesses properties of substantial delays, high cost and is difficult to evolve. Furthermore, the sorting network orderly retrieves the n sub-problems that have the minimum lower bounds. Since this is not required by the system, this may lead to unnecessary degradation in performance. Another characteristic of the sorting network is that sorting has to be completed before the list is available. On the other hand, the second alternative can utilize the current VLSI technology to implement the processor and memory on a single chip. Although the processor-processor interconnection network may be expensive and incurs substantial overhead in sub-problem distributions, it is shown that sorting does not have to be completed before sub-problem expansion can begin and this causes a relatively small degradation in performance. This, together with a few other nice properties, make this a more cost-effective design. We therefore select the second alternative in our design.

IV. NETWORK ARCHITECTURE

The objective of the network is to have a *complete distribution;* that is, to distribute the sub-problems in the local memories of the processors so that the n global minima can be distributed, one to each of the n processors. The locations of these n global minima are not known *a priori;* otherwise the problem is very simple and the processor with more than one global minima can send one of these sub-problems to processors without any. Since predetermined distribution operations are unknown, we can allow all the processors to carry out the same distribution operations (e.g., distribute to the nearest neighbor), or to carry out different distribution operations (e.g., one processor may be distributing to its nearest neighbor while the others are not). The former type of distribution possesses the property that each processor is connected to and from the same number of neighboring processors and has the *state preserving property*. That is, if the global minima have been distributed to the processors, continual redistribution would not disturb the state and the global minima would remain distributed to the processors. On the other hand, each processor may be connected to and from a different number of neighboring processors in the latter case and it is rather difficult to preserve the state. For this reason, we choose to investigate the former case only.

The design of the interconnection network ranges from a simple uni-directional ring network where each processor can communicate with one of its neighbors to a fully connected network where communications can be carried out simultaneously with all the processors. Analysis in this section shows that a simple uni-directional ring network is the optimal interconnection network. In order to do this, an urn model must first be developed.

A. The Urn Model

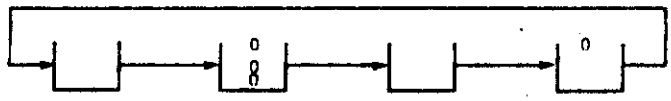
The n processors in the system are represented as n urns that contain n white marbles which stand for the global minima and $S - n$ yellow marbles where S is the total number of active sub-problems. The white marbles are originally distributed randomly to the urns. The distribution process moves the marbles around so that eventually, one white marble is distributed to each urn. The white marbles are always "lighter" than the yellow marbles so that they always "float" to the top of the urn. During the distribution process, one or more marbles are taken from each urn and distributed to one or more urns in the system. If a white marble exists in the urn, it is always distributed first. The ordering of the yellow and white marbles in the urns models the ordering of the sub-problems by lower bounds in ascending order in the processors. If one of the n global minima (white marble) exists in a processor (urn), it is always ordered before the other sub-problems (yellow marbles) and is always distributed first. It should be noted that this model does not take into account the ordering of the white marbles which is important in a conventional sorting and merging problem. It is sufficient for exactly one white marble to be distributed to each urn whereas in a sorting and merging problem, the white marbles are ordered before they are distributed to the urns. It is hoped that the relaxation induced in this problem can help to reduce the amount of marble movements.

We investigate distribution strategies that correspond to different degrees of interconnection. The first strategy shifts a white marble, if there are any, to the urn on the right. This corresponds to a *uni-directional ring network* (Figure 4a). A more general strategy distributes the jth marble ($j < k$) in the ith urn to the $([i + j] \ mod \ n)$th urn in parallel. This corresponds to a *k-connected network* (Figure 5a). When $k = 1$, this becomes the uni-directional ring network. In Figures 4 and 5, we have also shown the state of the system after a number of distributions.

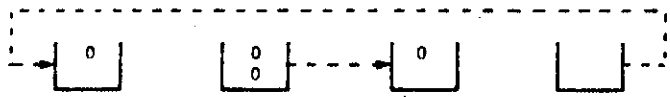
In evaluating the interconnection network, all the overheads must be accounted for in the distribution process. The overheads in a distribution include the time to shift and the time to let the white marbles "float" to the top (which corresponds to merging the newly arrived sub-problems into the original list). The lower bounds for the number of distributions to achieve complete distribution are shown in the next section.

B. Lower Bound for a Complete Distribution

In evaluating the lower bound, the sorting method in each processor must be taken into account. The overhead for sorting depends on the implementation. The complexity is $O(m \log m)$ for sorting m numbers by software (e.g., heap

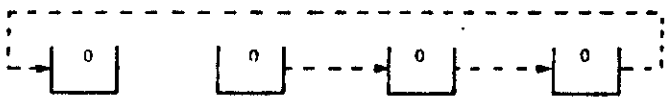


(a) Uni-directional ring network connecting four urns with initial state (0,3,0,1)



(b) State after 1 left shift

(c) State after 2 left shifts



(d) State after 3 left shifts

----➡ distribution of marbles

Figure 4—Uni-directional ring network connecting four urns

sort,[36]), $O(m)$ for sorting by a hardware priority queue, and $O((\log m)^2)$ for sorting by Batcher's odd-even merging network.[36] In the following theorem, we evaluate the lower bound of a complete distribution for the three sorting methods.

Theorem 1

Let K be the communication time to transfer one or more sub-problems in parallel to the other processors, m be the maximum number of sub-problems that can be stored in a processor, and n be the number of processors in the system. Depending on the degree of connection, the lower bounds on the number of operations for a complete distribution is between $O(K + n \log m)$ and $O(nK + n \log m)$ for sorting by software, is between $O(K + n)$ and $O(Kn)$ for sorting by hardware priority queues and is between $O(K + \log^2 m)$ and $O(Kn + n \log^2 m)$ for sorting by Batcher's networks.

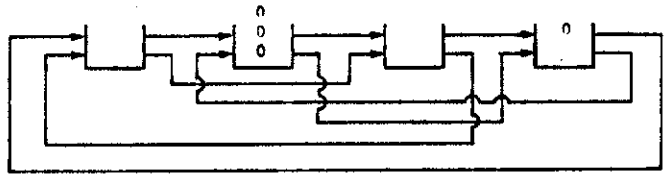
Proof

Suppose each urn is connected to n^x other urns $(0 \leq x \leq 1)$, that is, urn i is connected to urn $(i + k) \bmod n$ $(1 \leq k \leq n^x)$. The maximum delay to transfer a marble from one urn to another is n^{1-x}. Assuming all the n marbles reside in one single urn and transfers can be made in parallel to n^x other urns, it would take n^{1-x} transfers to take all the marbles out from this urn. Since all the transfers are carried out simultaneously, each urn would be receiving n^x marbles in a time interval K.

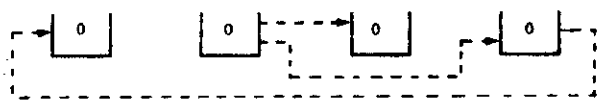
After each transfer, the marbles must be inserted into the local lists before another distribution can take place. For sorting by software, the time needed is the time to insert n^x numbers into a priority tree that may contain as many as m numbers. The total sorting overhead is therefore $O(n^x \log m)$. For sorting by a hardware priority queue, each insertion takes constant time and n^x numbers can be inserted into the priority queue in time $O(n^x)$. For sorting by Batcher's network, all the m numbers in a processor are connected to a network. The sorting overhead is therefore $O(\log^2 m)$.

To summarize, n^{1-x} iterations are needed, and each iteration takes time K for communication and additional overhead for sorting. The lower bound on the total overhead is therefore $O(n^{1-x}[K + n^x \log m])$ for sorting by software; $O(n^{1-x}(K + n^x))$ for sorting by hardware priority queues; and $O(n^{1-x}(K + \log^2 m)$ for sorting by Batcher's network. For an n-connected network, $x = 1$ and the lower bounds are $O(K + n \log m)$, $O(K + n)$, $O(K + \log^2 m)$ for sorting by software, hardware priority queue and Batcher's network respectively. For a uni-directional ring network, $x = 0$ and the corresponding lower bounds are $O(Kn + n \log m)$, $O(Kn)$, and $O(Kn + n \log^2 m)$.

From the above theorem, it is obvious that sorting by hardware priority queues is better than sorting by software and Batcher's network for a uni-directional ring network. However, depending on the relative sizes of K, m, and n, sorting by priority queues may be better than sorting by Batcher's network or vice versa in an n-connected network. In general, K is very small because of the advances in communication technologies; n is usually large because it governs the degree of parallelism; m is also large, and in an n-connected network, $m \geq n$. Taking these into account, the lower bounds for an n-connected network are $O(n \log m)$, $O(n)$ and $O(\log^2 m)$ for sorting by software, hardware priority queues, and Batcher's network. Batcher's network has less overhead if $m \leq O(c^{\sqrt{n}})$, where $c > 1$. On the other hand, Batcher's net-



(a) 2-connected network connecting four urns with initial state (0,3,0,1)



(b) State after 1 shift.

----➡ distribution of marbles

Figure 5—2-connected network connecting four urns

work uses $O(m \log^2 m)$ hardware,[36] as compared to $O(m)$ for a hardware priority queue. For the present time, we favor the use of a hardware priority queue because of its reduced hardware complexity. In the next section, we prove that the uni-directional ring network is the optimal network if hardware priority queues are used.

C. The Optimal Interconnection Network with Hardware Priority Queue

In this section, we show that the uni-directional ring network is the optimal interconnection network by showing that the amount of work needed for a complete distribution is $O(n-1)$ and therefore equals the lower bound evaluated earlier.

Theorem 2

The number of distributions for a complete distribution in a k-connected network ($1 \leq k \leq n$) is at most $n-1$.

Proof

We first prove the case of the uni-directional ring (1-connected) network. The proof is by contradiction. Suppose a white marble cannot get to the top of urn i in $n-1$ distributions and remains in the second position, that is, after $n-1$ distributions, urn i still contains at least two white marbles and the distribution is not complete. An urn that starts with 0 or 1 marbles can never get more than one marble after $n-1$ distributions. Hence urn i must have started with at least two marbles. And in $n-1$ distributions, $n-1$ distinct white marbles must have passed over the top of urn i, because if not, the second white marble in urn i would have a chance to get to the top of urn i. This implies that there are altogether $n-1+2 = n+1$ white marbles in the system, which contradicts the original assumption that there are n white marbles in the urns. Complete distribution can always be achieved in $n-1$ distributions. The proof for a k-connected network with $k > 1$ is similar and will not be repeated here.

The overall amount of work is therefore $(n-1)*$(sorting overhead). Since the sorting overhead is the smallest in a uni-directional ring network, the overall complexity to achieve a complete distribution is therefore $O(n)$. As we have proved in Theorem 1 that the lower bound of distributions using hardware priority queues is $O(n)$, the uni-directional ring network is the optimal interconnection network. Although the number of distributions to achieve a complete distribution in a k-connected network ($k > 1$) may be smaller, as evidenced in the simulation results shown later. The performance can only be improved by a constant factor because the lower bound is also $O(n)$. Furthermore, the number of network links in a k-connected network ($k > 1$) is n^k, as compared to n in a uni-directional ring network. We conclude that the uni-directional ring network is the optimal and most cost-effective way of implementation.

In the remainder of this section, we present some results on the average fraction of urns containing white marbles using the k-connected network and try to answer the question we raise in Section III—namely, what is the degradation in performance if a complete distribution is not attained before the processors pick up sub-problems for expansion. The evaluation results are obtained by generating all the possible combinations of n white marbles in the n urns as initial distributions. It is seen in Figure 6 that the increase in the average fraction of urns containing white marbles due to increasing k is rather small. Furthermore, the sorting overhead is not included in the evaluation. The final performance for $k > 1$ is expected to be less than the performance of the uni-directional ring network. In Figure 7, the fraction of urns containing marbles for the different number of distributions in a uni-directional ring network is shown. It is seen that these curves approach different asymptotic values as the number of urns is increased. The asymptotic average fraction of urns containing white marbles as the number of urns is increased for no distribution ($s = 0$) has been shown to be 0.5. The analyses for cases where $s > 0$ are similar but more difficult. It is also seen that the improvement is significant for the first few distributions, but the improvement is diminishing as the number of distributions is increased. This implies that the fraction of urns containing white marbles is significantly improved by a small number of distributions. In general, less than half of the urns do not contain white marbles for an incomplete distribution.

D. Technology Dependent Considerations

We have assumed in the urn model that each marble represents a sub-problem. Actually, a sub-problem is characterized not only by a lower bound, but also by the state of the problem. For example, let a graph of p nodes be represented in the form of a p by p connectivity matrix, and each sub-problem include a partial assignment of the nodes and edges. In a distribution, the partial assignment must also be transferred with the lower bound of the sub-problem. If p is large, the transfer time can be in the order of milliseconds or seconds. On the other hand, sorting in the processors has a relatively small overhead as compared with the distribution time. If we examine the complexity measure of the uni-directional ring network again, we discover a more serious problem. The overheads for complete distribution is $O(n)$. Suppose the number of cycles in a parallel branch and bound algorithm improves by a factor of n, and in each cycle, there is an overhead for distribution of $O(n)$; this implies that there is no overall improvement in performance as far as complexity measure is concerned. These observations imply that it is necessary to design additional hardware or strategies in order to reduce the distribution overhead so that distribution can be overlapped completely with sub-problem expansion. There are several alternatives.

The first alternative considers sending the tags (each consisting of the urn number and the lower bound) instead of the white marble (the entire sub-problem) in a distribution. After $O(n)$ distributions, complete distribution is obtained. These tags are then gated to an external controller which counts the number of white marbles in each urn and decides on the optimal transfer sequence of white marbles from one urn to another. A k-connected network may be used in order to allow k parallel sub-problem transfers to be made from each

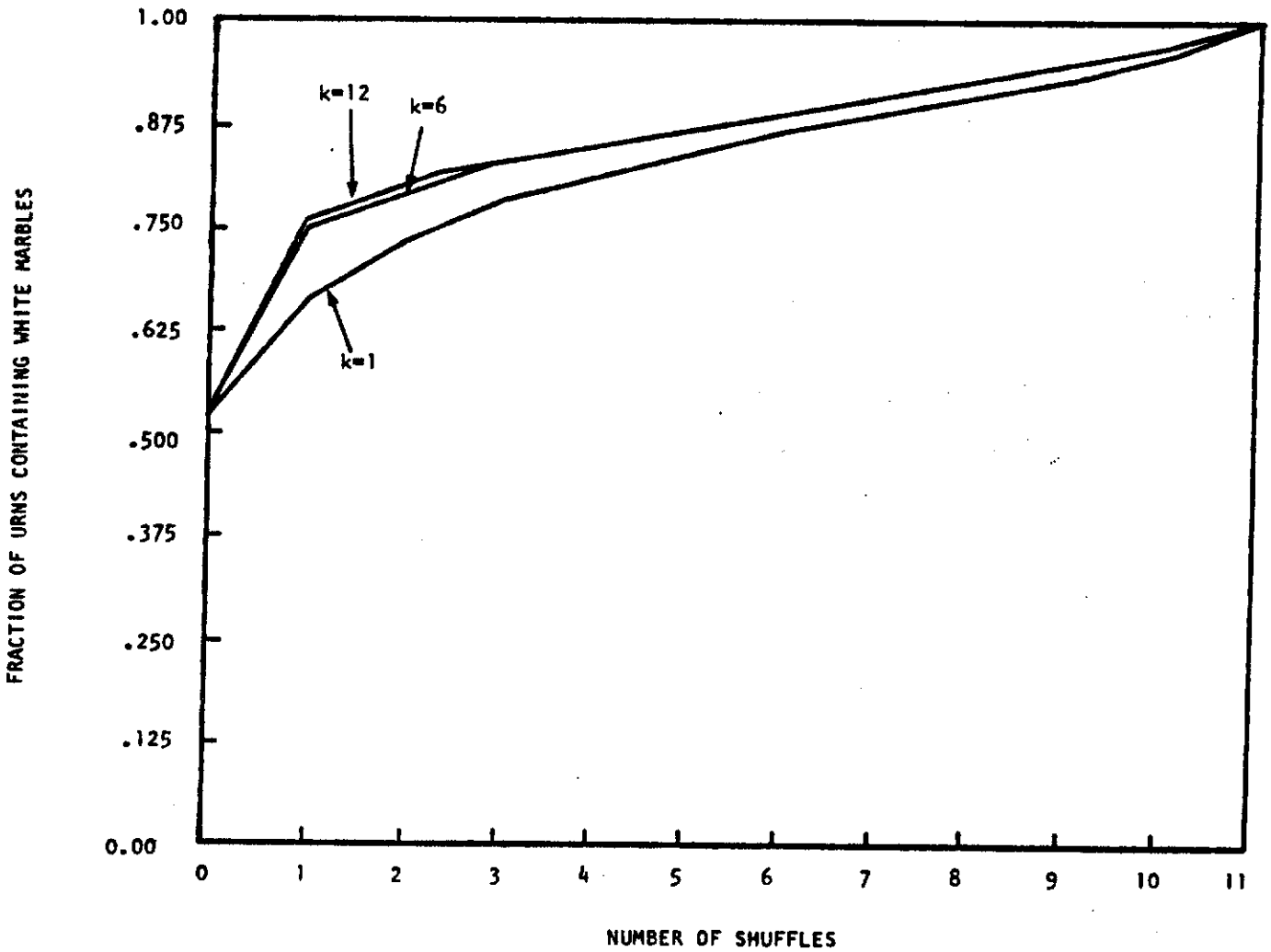


Figure 6—Performance of the k-connected network for 12 urns ($k = 1$ for uni-directional ring network)

urn. Of course, the value of k has to be determined so that the response time requirement is satisfied.

The second alternative considers sending n marbles from each urn and passing them through a sorting network such as Batcher's sorting network. The first n marbles coming out from the sorting network must be white, and they are returned in parallel to the n urns. As we know, the complexity for sorting n^2 numbers using Batcher's network is $O(4 \log^2 n)$. The overall improvement of the parallel branch and bound algorithm is at most

$$O\left(\frac{n}{\log^2 n}\right)$$

Further, this scheme uses extensive hardware and may not be practical when n is large.

None of the above schemes is perfect and requires additional hardware support. However, when hardware becomes sufficiently inexpensive, it may be possible to use more processors and allow them to operate at over 50% efficiency (the average number of urns containing white marbles without any distribution is over 50%). Furthermore, we have assumed so far that the system operates in a coupled fashion; that is, the sub-problems are evaluated while the distributions are made and the evaluation of the next set of problems does not start until part or all of the distributions are done. In practice, the sub-problems have different sizes and different processing times and therefore it would be inefficient for the system to wait until all the processors are finished. Each processor would behave independently and execute the lower bound evaluation function in its local memory. When this evaluation is finished, it picks up a sub-problem with the minimum lower bound from its local list of sub-problems. Since the time when one processor picks up a sub-problem to the time when another processor picks up a new sup-problem can be relatively short, the distribution process may not be completed and the system would be operating at less than optimal performance.

Finally, if an urn does not contain a white marble (one of the first n global minima), it may contain a marble of different color (which may correspond to one of the $jn + 1$th to $(j + 1)n$th global minima, $j > 0$, and this is distributed accord-

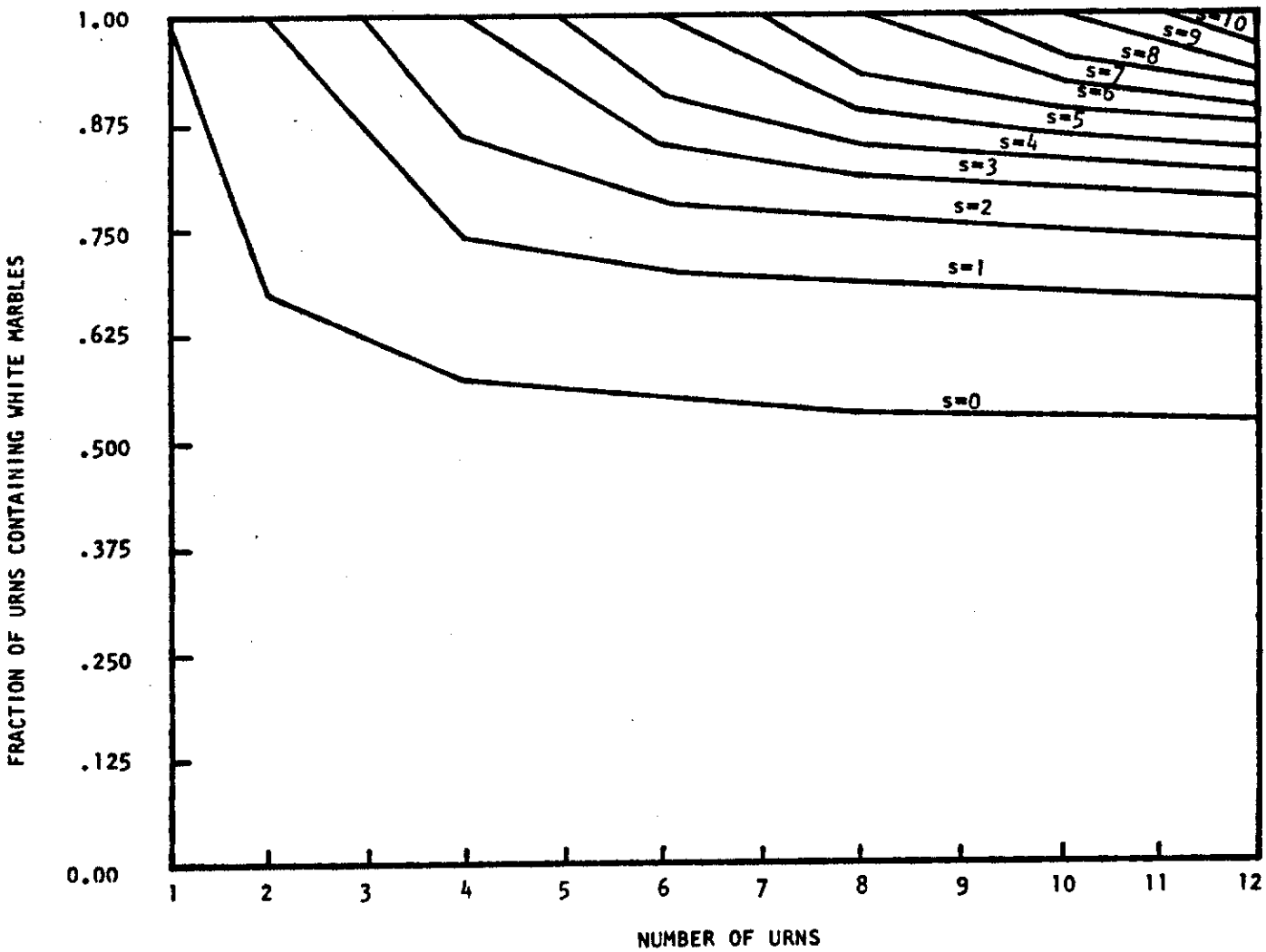


Figure 7—Performance of the uni-directional ring network ($k = 1$) for different number of urns (s = distributions)

ingly. So although a processor may not be working on one of the nth global minima, the expansion of a sub-problem with the minimum lower bound may still contribute to the speedup.

One interesting point to notice is when a small number of distributions are made, the distribution of the first global minima improves; that is, the number of urns containing white marbles increases. However, the distribution of the $n + 1$th to $2n$th global minima which are represented as black marbles may be worse. This distribution is important because it governs the distribution of the white marbles in the next iteration (when the black marbles in this iteration become the white marbles in the next iteration). It is shown that the average number of urns containing black marbles after a complete distribution is actually smaller than the average number of urns if the marbles were distributed randomly. This phenomenon is illustrated in Figure 8. Fortunately, the difference between these two average numbers for large n is insignificant. The simulation results are not included here.

The problem discussed in this section for speeding up the distribution time can be solved by faster technology. The ring network can run up to several hundred mega-bits per second and can be used as a "barrel" as in Control Data Corporation's 6000 and 7000 series computers.[66] It can be realized with sub-nanosecond emitter coupled logic that can operate at rates up to 100 MHz. Off-the-shelf parts, such as Fairchild's F100K, are available to implement the uni-directional ring network.

V. CONCLUSION

In this paper, we have proposed and studied the network architecture of MANIP, a parallel computer system for processing NP-complete problems. NP-complete problems have the unique property that the computation time for all known optimal algorithms increases exponentially with the problem size. Thus a small increase in the problem size may cause a very large increase in the problem space needed for the opti-

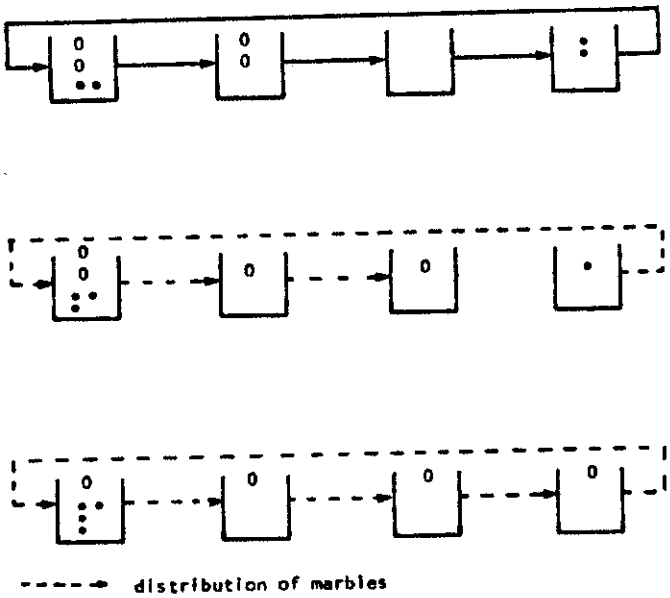


Figure 8—The decrease in the number of urns containing black marbles when the shuffle is complete (the white marbles represent the first n global minima; the black marbles represent the n + 1st to $2n$th global minima)

mal algorithm to complete the examination. Due to the inherent difficulty in solving NP-complete problems, parallelism in processing is proposed to expand the size of solvable problems. The most general technique that can be used to solve a wide variety of NP-complete problems on a uni-processor system, optimally or suboptimally, is the branch and bound algorithm. We have studied in this paper a parallel version of the branch and bound algorithm which can be executed efficiently on a parallel computer system.

The parallel branch and bound algorithm requires a combination of sorting and merging. The sub-problems are evaluated to produce new sub-problems which are inserted into a list of previously created sub-problems. This list is maintained in a sorted order by the lower bounds of the sub-problems so that the minima can be picked up for expansion in the next cycle. The process is terminated when a feasible solution is found with a value smaller than the lower bounds of all the sub-problems in the list. Since it is important to maintain a global sorted list of sub-problems, a common memory shared by all the processors can be used. However, this can become a bottleneck when the number of processors is large. We have proposed an alternative such that each processor has a local memory and the processors communicate with each other through an inter-processor communication network. When the processors have created new sub-problems, they are first inserted into the local list, and then sub-problems with minimum lower bounds from each processor are distributed until a set of n global minima are obtained. These n global minima are distributed to the n processors in the system for processing (complete distribution).

We have proved that the lower bound for the amount of work to achieve a complete distribution is $O(n)$ when sorting is done by a hardware priority queue within each processor. We have also shown that the uni-directional ring network is the optimal and most cost-effective way of implementing the inter-processor communication network. Sorting by other sorting methods gives different performance. Sorting by software, such as heap sort, has a worse performance, while sorting by Batcher's odd-even merging network has a better performance at the expense of increased hardware complexity. The proposed interconnection network is reliable because of its simplicity and it reconfigurability. Faulty processors can be switched off the network without affecting the performance of other processors. Redundant rings can also be used to increase the reliability of the network.

The architecture of the processors and the performance evaluation of the system is given in a different paper. The simulation results there show that with complete distributions, the number of iterations reduce by a factor of n using n processors. With no distribution, the performance is very poor. However, when one or more distributions are applied in each iteration, the total number of iterations is the same as if a complete distribution is used (with very small variations). One major problem encountered in the simulations is the problem of insufficient memory space. The branch and bound algorithm has to switch from best-first search to depth-first search when memory space is exhausted. This significantly degrades the memory performance. A method to increase the virtual space of the branch and bound algorithm is to use a virtual memory management system. This will be presented in a future paper.

REFERENCES

1. S. G. Aki, D. T. Barnard, and R. J. Doran, "Simulation and Analysis in Deriving Time and Storage Requirements for a Parallel Alpha-Beta Algorithm," *Proc. of 1980 Int'l. Conf. on Parallel Processing*, Michigan, pp. 231-234, 1980.
2. K. E. Batcher, "Sorting Networks and their Applications," *Proc. AFIPS Spring Joint Computer Conference*, Vol. 32, pp. 307-314, Apr. 1968.
3. K. E. Batcher, "The Flip Network in STARAN," *Proc. of 1976 Int'l Conf. on Parallel Processing*, Michigan, pp. 65-71, 1976.
4. G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," *IEEE Trans. on Computers*, Vol. C-27, No. 1, pp. 84-87, Jan. 1978.
5. V. E. Benes, "Optimal Rearrangeable Multistate Connecting Networks," *Bell System Technical Journal*, Vol. 48, No. 4, pp. 1641-1656, July 1964.
6. J. L. Bentley and H. T. Kung, "A Tree Machine for Searching Problems," *Proc. of 1979 International Conf. on Parallel Processing*, Michigan, pp. 257-266, 1979.
7. B. C. Desai, "The BPU, A Staged Parallel Processing System to Solve the Zero-One Problem," *Proc. of ICS78*, Taipei, Taiwan, pp. 802-817, Dec. 1978.
8. B. C. Desai, "A Parallel Microcessing System," *Proc. of 1979 Int'l. Conf. on Parallel Processing*, p. 136, 1979.
9. A. M. Despain and D. A. Patterson, "X-tree: A Tree Structured Multiprocessor Computer Architecture," *Proc. of 5th Symp. on Comp. Arch.*, Palo Alto, CA, 1978, pp. 144-151.
10. W. L. Eastman, "A Solution to the Traveling Salesman Problem," presented at the American Summer Meeting of the Econometric Society, Cambridge, Mass., Aug. 1958.
11. A. Efromyson and T. C. Ray, "A Branch and Bound Algorithm for Plant Location," *Operations Research*, Vol. 14, pp. 361-368, 1966.
12. El-Dessouki and W. H. Huen, "Distributed Enumeration on Network Computers," *Proc. of 1979 Int'l. Conf. on Parallel Processing*, Michigan, pp. 137-146, 1979. Also published in *IEEE Trans. on Computers*, Vol. C-29, No. 9, pp. 818-825, Sept. 1980.
13. T. Feng, "Data Manipulating Functions in Parallel Processors and Their Implications," *IEEE Trans. Computers*, Vol. C-23, No. 3, pp. 309-318, Mar. 1974.

14. J. P. Fishburn, R. A. Finkel, and S. A. Lawless, "Parallel Alpha-Beta Search on ARACHNE," *Proc. of 1980 Int'l. Conf. on Parallel Processing*, Michigan, pp. 235-243, 1980.
15. Foster, M. J. and H. T. Kung, "Design of special-purpose VLSI chips," *IEEE Computer*, Vol. 13, No. 1, pp. 26-40, 1980.
16. M. R. Garey and D. S. Johnson, "Strong NP-completeness Results: Motivations, Examples, and Implications," *JACM*, Vol. 25, No. 3, pp. 499-508, July 1978.
17. M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, San Francisco, 1979.
18. R. Garfinkel, "On Partitioning the Feasible Set in a Branch and Bound Algorithm for the Asymmetric Travelling Salesman Problem," *Operations Research*, Vol. 21, No. 1, pp. 340-342, 1973.
19. R. S. Garfinkel and G. L. Nemhauser, *Integer Programming*, John Wiley and Sons, Inc., New York, 1972.
20. A. M. Geoffrion and R. E. Marsten, "Integer Programming Algorithms: A Framework and State-of-the-Art Survey," *Management Science*, Vol. 18, No. 9, pp. 465-491, May 1972.
21. L. R. Goke and G. J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," *Proc. 1st Annual Comp. Architecture Conf.*, pp. 21-28, Dec. 1973.
22. J. A. Harris and D. R. Smith, "Hierarchical Multi-processor Organizations," *Proc. 4th Annual Symp. on Comp. Arch.*, pp. 41-48, 1977.
23. M. Held and R. Karp, "A Dynamic Programming Approach to Sequencing Problems," *Jr. of SIAM*, Vol. 10, pp. 196-210, 1962.
24. M. Held and R. Karp, "The Travelling Salesman Problem and Minimum Spanning Trees," *Operations Research*, Vol. 18, pp. 1138-1162, 1970.
25. M. Held and R. Karp, "The Travelling Salesman Problem and Minimum Spanning Trees, Part II," *Math. Prog.*, Vol. 1, pp. 6-25, 1971.
26. D. S. Hirschberg, "Fast Parallel Sorting Algorithms," *CACM*, Vol. 21, No. 8, pp. 657-601, Aug. 1978.
27. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Maryland, 1978.
28. T. Ibaraki, "Computational Efficiency of Approximate Branch and Bound Algorithms," *Math. of Oper. Reseurch*, Vol. 1, No. 3, pp. 287-298, 1976.
29. T. Ibaraki, "Theoretical Comparisons of Search Strategies in Branch and Bound Algorithms," *Int. Jr. of Comp. and Info. Sci.*, Vol. 5, No. 4, pp. 315-344, 1976.
30. T. Ibaraki, "On the Computational Efficiency of Branch and Bound Algorithms," *J. of Oper. Res. Soc. of Japan*, Vol. 20, No. 1, pp. 16-35, 1977.
31. T. Ibaraki, "The Power of Dominance Relations in Branch and Bound Algorithms," *JACM*, Vol. 24, No. 2, pp. 264-279, 1977.
32. T. Ibaraki, "Depth-m Search in Branch-and-Bound Algorithms," *Int. Jr. of Comp. and Inf. Sci.*, Vol. 7, No. 4, pp. 315-343, 1978.
33. G. Ingargiola and J. Korsh, "A Reduction Algorithm for Zero-one Single Knapsack Problems," *Management Science*, Vol. 20, No. 4, pp. 460-663, 1973.
34. G. Ingargiola and J. Korsh, "A General Algorithm for One Dimensional Knapsack Problems," *Operations Research*, Vol. 25, No. 5, pp. 752-759, 1977.
35. R. M. Karp, "Reducibility Among Combinational Problems," *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, pp. 85-104, 1972.
36. D. E. Knuth, *The Art of Computer Programming, Sorting, and Searching*, Vol. 3, Addison-Wesley, 1973.
37. W. Kohler and K. Steiglitz, "Characterization and Theoretical Comparison of Branch and Bound Algorithms for Permutation Problems," *JACM*, Vol. 21, No. 1, pp. 140-156, 1974.
38. D. J. Kuck, "ILLIAC IV Software and Application Programming," *IEEE Trans. on Comp.*, Vol. C-17, pp. 746-757, Aug. 1968.
39. D. J. Kuck, "A Survey of Parallel Michine Organization and Programming," *Computing Survey*, Vol. 9, No. 1, pp. 29-59, 1977.
40. H. T. Kung, "The Structure of Parallel Algorithms," *Advances in Computers*, Vol. 19, Yovits, M. C. ed., Academic Press, New York, 1980.
41. B. Lageweg, J. Lenstra and A. Rinnooykar, "Job-shop Scheduling by Implicit Enumeration," *Management Science*, Vol. 24, No. 4, pp. 441-400, 1977.
42. E. A. Lamagna, "The Complexity of Monotone Networks for Certain Bilinear Forms, Routing Problems, Sorting and Merging," *IEEE Trans. on Computers*, Vol. C-28, No. 10, pp. 773-782, Oct. 1979.
43. A. H. Land and A. Doig, "An Automatic Method for Solving Discrete Programming Problems," *Econometrica*, Vol. 28, pp. 497-520, 1960.
44. Lawler, E. L. and Wood, D. W., "Branch and Bound Methods: A Survey," *Operations Research*, Vol. 14, pp. 699-719, 1966.
45. D. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Computers*, Vol. C-24, No. 12, pp. 215-255, Dec. 1975.
46. J. Lenstra, "Sequencing by Enumerative Methods," *Math. Centre. Tract 69*, Mathematisch Centrum, Amsterdam, 1976.
47. D. D. Marshall, "A Parallel Processor Approach for Searching Decision Trees", *Proc. of 1977 Int'l. Conf. on Parallel Processing*, Michigan, pp. 199-201, 1977.
48. L. Mitten, "Branch and Bound Methods: General Formulation and Properties," *Operations Research*, Vol. 18, pp. 24-34, 1970.
49. H. P. Moravec, "Fully Connecting Multiple Computers with Pipelined Sorting Nets," *IEEE Trans. on Computers*, Vol. C-28, No. 10, pp. 795-798, Oct. 1979.
50. T. Morin and R. Marsten, "Branch and Bound Strategies for Dynamic Programming," *Operations Research*, Vol. 24, pp. 611-627, 1976.
51. D. E. Muller and F. P. Preparata, "Bounds to Complexities of Networks for Sorting and Switching," *JACM*, Vol. 22, No. 2, pp. 195-201, Apr. 1975.
52. D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh Connected Parallel Computer," *IEEE Trans. on Computers*, Vol. C-27, No. 1, pp. 2-7, Jan. 1979.
53. N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*, McGraw Hill, New York, 1971.
54. J. H. Patel, "Processor-Memory Interconnections for Multiprocessors," *6th Annual Symposium on Computer Architecture*, pp. 168-177, 1979.
55. D. A. Patterson and C. H. Sequin, "Design Considerations for Single Chip Computers of the Future," *IEEE Trans. on Computers*, Vol. C-29, No. 2, pp. 108-116, Feb. 1980.
56. M. C. Pease, "The Indirect Binary n-cube Microprocessor Array," *IEEE Trans. on Computers*, Vol. C-26, No. 5, pp. 458-473, May 1977.
57. F. P. Preparata, "Parallelism in Sorting," *Proc. of 1977 Int'l. Conf. on Parallel Processing*, Michigan, pp. 202-206, Aug. 1977.
58. C. V. Ramamoorthy, J. L. Turner, and B. W. Wah, "A Design of a Fast Cellular Associative Memory for Ordered Retrieval," *IEEE Trans. on Computers*, Vol. C-27, No. 9, pp. 800-814, Sept. 1978.
59. G. Sa, "Branch and Bound and Approximate Solutions to the Capacitated Plant Location Problem," *Operations Research*, Vol. 17, No. 6, pp. 1005-1016, 1969.
60. S. Sahni, "General Techniques for Combinational Approximation," *Operations Research*, Vol. 25, No. 6, pp. 920-936, 1977.
61. R. Sedgewick, "Data Movement in Odd-Even Merging," *SIAM Journal of Computing*, Vol. 7, No. 3, pp. 239-272, Aug. 1978.
62. C. L. Seitz, *Proceedings of the Caltech Conference on Very Large Scale Integration*, California Institute of Technology, Jan. 1979.
63. S. W. Song, "A HIghly Concurrent Tree Machine for Database Applications," *Proc. of 1980 Int'l. Conf. on Parallel Processing*, Michigan, pp. 259-268, 1979.
64. H. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. on Computers*, Vol. C-20, No. 2, pp. 153-161, Feb. 1971.
65. C. D. Thompson and H. T. Kung, "Sorting on a Mesh-Connected Parallel Computer," *CACM*, Vol. 20, No. 4, pp. 263-271, Apr. 1977.
66. J. E. Thornton, *Design of a Computer: The Control Data 6000*, Scott, Foresman and Company, Glenview, Illinois, pp. 141-153, 1970.
67. C. L. Wu and T. Y. Feng, "The Reverse-Exchange Interconnection Network," *IEEE Trans. on Computers*, Vol. C-29, No. 9, pp. 801-811, Sept. 1980.