PROCEEDINGS OF 8th ANNUAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1981.

MANIP-A PARALLEL COMPUTER SYSTEM FOR IMPLEMENTING BRANCH AND BOUND ALGORITHMS

BENJAMIN W. WAH

School of Electrical Engineering
Purdue University
West Lafayette, Indiana

Y. W. MA

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California

ABSTRACT

In this paper, we propose and analyze the design of MANIP, a parallel machine for processing nondeterministic polynomial complete problems. The most general technique that can be used to solve a wide variety of NP-complete problems on a uniprocessor system, optimally or suboptimally, 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 unidirectional ring network is proved to be the optimal and most cost-effective interprocessor communication network when sorting is done by a hardware priority queue in each processor. Lastly, the performance on the proposed system is evaluated using the vertex covering problem.

INTRODUCTION

The branch and bound algorithm[1] is an efficient algorithm to solve for problems that are known to be nondeterministic polynomial (NP)-complete.[2] An important characteristic of NP-complete problems is that there is no known optimal algorithm to solve the problem with a computation time that increases polynomially with the size of the problem. The computation time for all known optimal algorithms for this class of problems increases exponentially with the problem size (if n represents the size of the problem, the computation time goes up as k^n in which $k > 1$).

To illustrate the extent of this complexity, suppose a given computer can solve a problem of size 10 in 0.001 sec. If the problem has a time complexity of 2^n, then the same computer can solve the problem of size 60 in 357 centuries. Moreover, improved technologies appear to have little effect on improving the problem size that can be solved. For example, the computer just described can solve a 2^n problem with $n = 31.8$ in one hour. With a computer 1,000 times faster, the problem size that can be solved is 41.7. With a computer 10,000 times faster, the problem size that can be solved is 45.1.

It is obvious that improving the technology alone is not enough to expand the solvable problem space of NP-complete problems. The common approach is to solve small problems optimally and to apply heuristics to solve large problems suboptimally. Numerous algorithms and heuristics have been designed to solve specific NP-complete problems; however, the most general technique that can be used to solve a wide variety of NP-complete problems, optimally or suboptimally, is the branch and bound algorithm.

In this paper, we present a study of a parallel version of the branch and bound algorithm that can be executed efficiently on a parallel computer system. It has been recognized that the speed of a single computer is limited by technology. To increase the throughput further, multiple computers have to be connected together to form a parallel processor system; previously, computers and interconnection networks were designed using discrete components or a large set of semiconductor chips. The decreased system reliability as the system complexity increased severely constrained the degree of parallelism possible. Systems like C.mmp and ILLIAC IV were limited to less than 100 processors. With the new

0149-7111/81/0000/0239$00.75 © 1981 IEEE

advances in very large scale integration (VLSI)[3] and communication technologies, it is now possible to design one or more computers on one semiconductor chip.[4] The degree of parallelism, therefore, is potentially unlimited.

A large number of parallel processor systems with general-purpose interconnection networks have been proposed. Examples of these include C.mmp,[5] Cm*,[6] ILLIAC IV,[7] X-tree,[8] Data flow processor,[9] and TRAC,[10]. These systems however, are not designed for processing NP-complete problems, and therefore, increasing the number of processors yields rapidly diminishing returns.

Several architectures have been proposed to solve NP-complete problems. Harris and Smith[11] 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 completed. Because of the limited degree of communications, some processors might be working on tasks that could be eliminated if a better interconnection network were designed. Desai[12,13] also proposed a staged MIMD system to solve 0-1 integer programs 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.

The only published work on applying branch and bound algorithms to solve NP-complete problems in a network of computers was done by EL-Dessouki and Huen.[14] The algorithm is based on a general-purpose network architecture with limited memory space and slow interprocessor communication. Depth-first search criteria is used and is not effective in decreasing the execution time. With VLSI technology, larger and inexpensive memories, and faster communication technology, reducing the execution time (at the expense of larger memory space requirements and more interprocessor communications) is a more critical problem. Using VLSI technology, it would be feasible and cost effective to design special-purpose computers for such a task. Our design pertains to the fact that at most, one processor is implemented in each chip, but the design will still be applicable in the future when multiple processors can be designed into one chip.

In this paper, we propose the design of MANIP, an architecture using VLSI technology to implement a parallel branch and bound algorithm. We wanted to design special-purpose processors for evaluating the bounds and a simple interconnection network for interconnecting the processors. The system was designed according to the following design objectives:

1. The system should be modularly expandable to include a very large number of processors.
2. The design must have high performance, and the cost should be kept low by replicating simple cells.
3. The system should use distributed control so that a controller could not become the bottleneck in future system expansion.
4. Efficient load balancing strategies should be implemented so that the processors can be kept busy most of the time.
5. The system should be recoverable from hardware failures.

In the following sections, the branch and bound algorithm and the parallel version of the branch and bound algorithm are presented. The architectural alternatives in implementing the parallel branch and bound algorithm are identified, and the network architecture is described. The architectural features necessary in the processor, as well as a performance evaluation of the system, are given. The conclusion includes an approach to the problem of implementation.

PARALLEL BRANCH AND BOUND ALGORITHM

The type of problems that a branch and bound algorithm can evaluate can be put into a constrained optimization:

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

in which X represents the domain of optimization defined by the m constraints, normally an euclidean n-space, and $\mathbf{x}$ denotes a vector $(x_1, x_2, \ldots, x_n)$. (Problems that are NP-complete can be put into this form. There also exists problems that are not NP-complete, but are put into this form.) A solution

vector that lies in x is called a *feasible solution*, and a feasible solution for which $C_0(x)$ is minimal is called an *optimal solution*. 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.

Essential Features of the Branch and Bound Algorithm

The branch and bound algorithm[1,15] is an organized and intelligently structured search of the space of all feasible solutions. Generally, 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, the 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 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 called a *subproblem*. The partitioning process selects a partition and breaks up this partition into smaller partitions. This extends the node in the partial tree representing this partition by one level and uses 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: the branching rule and the bounding rule. With respect to the partial tree in Figure 1, each node in the tree has two numbers associated with it–the upper bound and the lower bound of the subproblem. The leaf nodes in the partial tree are candidates for partitioning. 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 *terminated* and need not be considered in any further computation.

The branching algorithm examines the set of active leaf nodes and, based on some predefined criterion, selects one for expansion. 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, the algorithm is called a *depth-first search*. Lastly, if the node selected for expansion is one with the minimum lower bound, 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, so the search is "blind." The depth-first search has a similar behavior except that a subtree is generated completely before the other subtrees are examined. In both algorithms, the next node to be examined is known, so the state of the parent node leading to the next node does not have to be kept, because the path to the next node from the root node is easily found and is unique. These two algorithms, therefore, are space saving.

In contrast, the best-first search is space consuming, because all active subproblems must be stored as intermediate data in the computer. The total number of nodes expanded, however, is decreased in the sense that any branching operation performed under this policy must also be performed under other policies, provided that all bounds are unique.[1] Because time is critical in evaluating large NP-complete problems, the best-first branching algorithm is implemented in MANIP. The large intermediate storage problem can be solved by moving subproblems with larger lower bounds to secondary storage.

Once the subproblem has been selected for partitioning, some undetermined parameters in the subproblem must be selected so that alternatives for these parameters can be defined and multiple subproblems created. For example, in the traveling salesman problem, the undetermined alternatives are the set of untraversed edges. In expanding a subproblem, an untraversed edge (i, j) is selected, and two alternatives can be created: (1) the edge is traversed and the salesman goes directly from city i to city j and (2) vice versa. The parameter chosen to be expanded is usually done ad hoc.

After new subproblems are created, the bounding algorithm is applied to evaluate the upper and lower bounds of a subproblem. Generally, only the lower bound is evaluated, because the merits of using the upper bound are small. The bounding algorithm that is designed is highly dependent on the problem. For example, in an integer programming problem, a linear program with relaxed integer constraints can be used as a lower bound[16]; in the traveling salesman problem, an

assignment algorithm[17] or a spanning tree algorithm can be used as the bounding algorithm.

The vertex-covering problem[18] used as the basis of performance evaluation in our study is an example of an NP-complete problem. In this problem, the minimum number of vertices that are needed to "cover" all edges in an undirected graph are to be found. (Cover means that all 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, from which two subproblems can be created: one including the 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. Note that edges emanating from different vertices in the lower bound calculation may overlap, and therefore, this vertex does not necessarily cover all 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 subproblem, 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 2 is shown in Figure 3.

The Parallel Branch and Bound Algorithm

Three sources of parallelism are identified in the branch and bound algorithm: parallel evaluation of subproblems, parallel sorting of subproblems, and parallel execution of the bounding algorithms.

Parallel Evaluation of Subproblems

Because multiple subproblems are available, they can be evaluated simultaneously. Owing to overheads in interprocessor communication and sorting, and because some subproblem evaluations are unnecessary, improvement in execution time is usually less than n times (n is the number of processors). For example, Figure 4 shows the parallel evaluation of the branch and bound algorithm on the graph in Figure 2, using two processors. The parallel evaluation of node 2 in Figure 4 is not useful, because the corresponding node 3 in Figure 3 is not evaluated. When the problem size is large, the parallelism will contribute to better improvement in execution time.

Parallel Sorting of Subproblems

In the best-first search, the list of subproblems 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.[19] (Types of interconnection network required for parallel sorting will be discussed later.)

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, matrix manipulation hardware is helpful. The architecture, however, 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.

Efficiency Considerations

Many results have been proved for the nonparallel version of the branch and bound algorithm.[20-23] It has been shown that the best-first search is the best branching rule and minimizes the number of subproblems expanded.[1] Furthermore, the branch and bound algorithm can be used as a general-purpose heuristic to compute solutions that differ from the optimum by no more than a prescribed amount.[1] Suppose it was decided at the outset that a deviation of 10% from the optimum was tolerable. If a feasible solution of 150 is obtained, all subproblems with lower bounds of 136.4 (or 150/1.1) or more will be terminated. This technique reduces significantly the amount of intermediate storage and the time to arrive at a suboptimal solution. Techniques are also available to find the best solution in a given length of time.[1] All these can be incorporated into the parallel branch and bound algorithm.

Unfortunately, little can be said about the efficiency of the parallel branch and bound algorithm. Lawler and Wood[1] found that only those subproblems 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 the number of iterations will be n times (n is the number of processors) if the number of subproblems in the intermediate list with lower bounds less than or equal to the optimal solution is always greater than or equal to n.

This number is highly dependent on the problem and the partitioning carried out earlier. Simulation results in a following section show that improvement for the vertex-covering problem indeed approaches n.

In the next three sections, the architecture required to support the parallel branch and bound algorithm is presented. Two architectural alternatives are compared, and the unidirectional ring network is proved to be the optimal interconnection network.

ARCHITECTURAL ALTERNATIVES SUPPORTING THE PARALLEL BRANCH AND BOUND ALGORITHM

To implement the parallel branch and bound algorithm, the parallel computers must be interconnected. The functions of the interconnection network are:

1. To connect the multiple processors so that idle processors can access subproblems in other processors

2. In the case of best-first search, to allow the subproblems to be sorted so that subproblems with minimum lower bounds are evaluated first

Two types of networks can be identified: the single-stage distribution network and the single-stage or multistage sorting network. Figures 5 and 6 illustrate some features of these two networks.

Single-Stage Distribution Network

The single-stage distribution network (Figure 5) is designed to distribute the n subproblems with minimum lower bounds to each of the n processors. Subproblems with minimum lower bounds in each processor can be sent in parallel to neighboring processors. The subproblems are then sorted in the local processors, and the distribution process repeats until one of the n subproblems with minimum lower bounds is distributed to a processor. The order in which these n subproblems finally appear in the processors is not important. This interconnection does not require that connections be made to neighboring processors, but as long as each processor is connected to and from k other processors, the processors can be renumbered so that each processor is connected to k neighboring processors.

Two possibilities exist in the process of distribution:

1. The distribution time is smaller than or equal to the lower bound evaluation time. Although the distribution is completed, the distribution must be carried out again when the lower bounds for the currently expanded subproblems are available.

2. The distribution time is greater than the lower bound evaluation time. The processors remain idle until the distribution is complete.

In both cases, complete overlap is not attained because of the different processing and distribution times. A compromise could be made by overlapping the subproblem expansion with the subproblem distribution. If the distribution was completed first, a local sorting could be performed when the subproblems were evaluated, and the processors could expand the local minima without waiting for a complete distribution. If the subproblems evaluation was completed first, the next subproblem in the local list could be evaluated immediately without having to wait for the distribution to complete.

Single-Stage or Multistage Sorting Network

A sorting network such as Batcher's network[19] usually achieves the same objective as the distribution network and is usually faster. Referring to Figure 6, the number of memory modules (m) may be different from the number of processors. To distribute the n subproblems with minimum lower bounds into the n processors, n subproblems with minimum lower bounds are taken from each memory module and sorted through the sorting network. The n subproblems with the smallest lower bounds (out of $m \times n$ subproblems) are sent to the processors. This can be performed in a time $O(\log n)$.[24] Assuming that the number of iterations improve by a factor of n, using n processors, the maximum performance improvement is $O(n/\log n)$. With this approach, each memory module must be able to retrieve subproblems in order. The memory modules can be implemented with associative memory[25] or as VLSI priority queues.[26]

The disadvantages of using a sorting network are the complex hardware complexity and a delay of $O(\log n)$ before the sorted subproblems can be used. Furthermore, the n subproblems selected from the $m \times n$ subproblems are sorted by the lower bounds, which is not required by the system. In a distribution network, subproblem evaluation can start without waiting for the distributions to complete, and the resulting degradation in performance to the overall

parallel branch and bound algorithm is very small. This, together with a few other properties (discussed later), makes the single-stage distribution network the most cost-effective design, and consequently, it was selected for our design.

NETWORK ARCHITECTURE

The objective of the network is to have complete distribution; that is, distribution of the subproblems in 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 beforehand, but otherwise, the problem is simple, and the processor with more than one global minimum can send one of these subproblems to processors that have none.

Because predetermined distribution operations are unknown, all processors can carry out the same distribution operations (for example, distributing to the nearest neighbor) or different distribution operations (for example, one processor may be distributing to its nearest neighbor while the others are not). The first type of distribution operation has not only the property that each processor is connected to and from the same number of neighboring processors but also the *state preserving property*; that is, if the global minima have been distributed to the processors, continued redistribution would not disturb the state, and the global minima would remain distributed to the processors. In the second type of distribution operation, each processor may be connected to and from a different number of neighboring processors, and it is difficult to preserve the state. (For this reason, we chose to investigate only the first type.)

The interconnection network design ranges from a simple unidirectional ring network in which each processor can communicate with one of its neighbors to a fully connected network in which communication can be carried out simultaneously with all processors. An analysis in this section shows that a simple unidirectional ring network is the optimal interconnection network. To do this, an urn model must first be developed.

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 in which S is the total number of active subproblems. The white marbles are originally distributed randomly to the urns. The distribution process moves the marbles around so that one white marble is distributed to each urn. The white marbles are always "lighter" than the yellow marbles, so 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 subproblems 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 subproblems (yellow marbles) and is always distributed first. Note 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. (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.

Distribution strategies correspond to different degrees of interconnection. The first strategy shifts a white marble, if there is one, to the urn on the right. This corresponds to a unidirectional ring network, as in Figure 7(a). A more general strategy distributes the jth marble ($j \leq k$) in the ith urn to the $[(i + j) \bmod n)]$th urn in parallel. This corresponds to a k-connected network as in Figure 8(a). When $k = 1$, this becomes the unidirectional ring network. Figures 7(b), 7(c), 7(d), and 8(b) show the states of the systems after a number of distributions.

Properties of the Distribution Network

In evaluating the distribution network, all overheads must be accounted for in the distribution process. 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 subproblems into the original list). The overhead for sorting in each processor depends on the implementation. The complexity is $O(m \log m)$ for sorting m numbers by software (for example, heap sort[27]), $O(m)$ for sorting by a hardware priority queue, and $O[(\log m)^2]$ for sorting by Batcher's odd-even merging network.[27]

The performance of the distribution network is summarized in the following two theorems. Theorem 1 obtains the lower bounds of the number of distributions, and theorem 2 shows that the unidirectional ring network achieves this lower bound efficiently.

THEOREM 1. *Let K be the communication time to transfer one or more subproblems in parallel to the other processors, m be the maximum number of subproblems 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, between $O(K + n)$ and $O(Kn)$ for sorting by hardware priority queues, and between $O(K + \log^2 m)$ and $O[(Kn + n) \log^2 m]$ for sorting by Batcher's networks.*

Proof. Only a sketch of the proof will be given here. Suppose each urn is connected to n^x other urns ($0 \le x \le 1$). The maximum delay to transfer a marble from one urn to another is $n.^{1-x}$ Assuming all 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 marbles out from this urn.

Because all transfers are carried out simultaneously, each urn would receive n^x marbles in a time interval K. The lower bound for the number of iterations, therefore, is $O(n^{1-x})$. By taking into account the overheads for sorting and communication in each iteration, it is shown that for an n-connected network, x equals 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 unidirectional ring network, x equals 0, and the corresponding lower bounds are $O(Kn + n \log m)$, $O(Kn)$, and $O[(Kn + n) \log^2 m]$.

From theorem 1, it is obvious that for a unidirectional ring network, sorting by hardware priority queues is better than sorting by software and Batcher's network. Depending on the relative sizes of K, m, and n, sorting by priority queues may also be better than sorting by Batcher's network (vice versa in an n-connected network). Generally, K is small because of the advances in communications technologies. Because it governs the degree of parallelism, n is usually large; m is also large, and in an n-connected network, $m \ge n$. Taking these values 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, respectively. Batcher's network has less overhead if $m \le O(c^{\sqrt{m}})$ in which $c > 1$; however, Batcher's network uses $O(m \log^2 m)$ hardware[27] compared with $O(m)$ for a hardware priority queue. For the present time, we favor the use of a hardware priority queue because of its reduced hardware complexity.

THEOREM 2. *The number of distributions for a complete distribution in a k-connected network ($1 \le k \le n$) is at most $n - 1$.*

Proof. The proof for the 1-connected network is outlined here, and the proof for the k-connected network is similar. 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 zero marbles or one marble can never get more than one marble after $n - 1$ distributions. Hence, urn i must have started with at least two marbles. 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 altogether, there are $(n - 1) + 2 = n + 1$ 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 overall amount of work, therefore, is $(n - 1)$ * (sorting overhead). Because the sorting overhead is the smallest in a unidirectional ring network, the overall complexity to achieve a complete distribution is $O(n)$. We proved in theorem 1 that the lower bound of distributions using hardware priority queues is $O(n)$. 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 compared with n in a unidirectional ring network. We conclude that the unidirectional ring network is the best 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

determine the degradation in performance if a complete distribution is not attained before the processors pick up subproblems for expansion. The evaluation results were obtained by generating all possible combinations of n white marbles in the n urns as initial distributions.

Figure 9 shows that the increase in the average fraction of urns containing white marbles because of increasing k was rather small. (The sorting overhead was not included in the evaluation.) The final performance for $k > 1$ is expected to be less than the performance of the unidirectional ring network with sorting overhead included.

Figure 10 shows the fraction of urns containing white marbles for the different number of distributions in a unidirectional ring network. These curves approach different asymptotic values as the number of urns was increased. The asymptotic average fraction of urns containing white marbles as the number of urns is increased for no distribution ($s = 0$) has been found to be 0.5. The analyses for cases in which s is greater than 0 are similar, but more difficult. Figure 10 also shows that the improvement was significant for the first few distributions, but that it diminished as the number of distributions 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 did not contain white marbles, for an incomplete distribution.

Other Considerations

After examining the complexity measures of theorems 1 and 2, we discovered a 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 $O(n)$: there appears to be 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 to reduce the distribution overhead so that distribution can be overlapped completely with subproblem expansion. There are several alternatives.

The first alternative considers sending the tags (which consist of the urn number and the lower bound) instead of the white marble (the entire subproblem) 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 to allow k parallel subproblem transfers to be made from each urn. Of course, the value of k has to be determined so that the response time requirement is satisfied. This solution is not perfect when n is large.

The second alternative allows the system to operate without any distributions. As seen in Figure 10, the average number of urns containing white marbles without any distribution is over 50%, and therefore, the system can operate at 50% efficiency without any distributions. This complexity, which is $O(n/2)$, is still better than $O(n/\log n)$ when an external sorting network is used.[24] Furthermore, we have assumed so far that the subproblems are evaluated while the distributions are made, and the evaluation of the next set of problems does not start until all or part of the distributions are done. In practice, the subproblems have different sizes and different processing times, and it would be inefficient for the system to wait until all the processors were finished. Each processor would behave independently and execute the lower bound evaluation function in its local memory. When this evaluation was finished, it would pick up a subproblem with the minimum lower bound from its local list of subproblems. Because the time when one processor picked up a subproblem to the time when another processor picked up a new subproblem could be relatively short, the distribution process might not be completed, and the system would be operating at less than optimal performance.

The third alternative considers that 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 $(jn + 1)$st to $[(j + 1)n]$th global minima, $j > 0$, and this is distributed accordingly. Although a processor may not be working on one of the nth global minima, the expansion of a subproblem with the minimum lower bound may still contribute to the speedup.

It is interesting that 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 $(n + 1)$st 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). 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 (Figure 11). Fortunately, the difference between these two average numbers for large n is insignificant. The simulation results are not included here. In a subsequent section, the performance of the parallel branch and bound algorithm under incomplete distribution is shown. The performance under a very small number of distributions is as good as the performance under a complete distribution.

PROCESSOR ARCHITECTURE

The processors are designed to evaluate the lower bounds of subproblems, to manage the local lists of subproblems, and to communicate with other processors by a unidirectional ring network. They possess the following four architectural components (Figure 12):

CPU: The central processing unit is a general-purpose computer that can be used to evaluate different types of bounding functions. It has such features as an arithmetic and logic unit (ALU), register files, a control unit, and local memory. The functions of CPU are to evaluate the bounding function for the selected subproblem, to manage the set of subproblems in its local space, and to coordinate with the communication processor to transfer subproblems to and from other processors and secondary storage.

Communication Processors: The communication processors are responsible for transmitting and receiving subproblems to and from other processors. These transfers are carried out in parallel. The communication processor receives a subproblem and writes it to local memory; a tag, consisting of the lower bound and the subproblem address in local memory, is written to the VLSI priority queue. The tags are sorted according to lower bounds in the priority queue. The communication processor picks up a subproblem with the greatest or smallest lower bounds and sends it to secondary storage or other processors. The communication processors operate independently of CPU.

VLSI Priority Queue: A VLSI priority queue is a distributed logic device that maintains the subproblems in a sorted order. The logical structure is a two-input, two-output device (deque), so tags can be input or output from the top or the bottom. The deque is ordered such that the top of the deque contains the tag with minimum lower bound. Each tag in the deque also contains an address that points to the location of the subproblem in the local memory. The hardware implementation[26,28,29] inserts comparators 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. In this way, 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.

In the previous discussion of the interconnection network, the distribution process was applied to subproblems with the minimum lower bound in the local list. As the list grows and exceeds the size of local memory, however, the excess must be moved to other processors where local memory is available or to secondary memory. In the latter case, it is better to move the set of subproblems that are the n global maxima. A similar interconnection network can be designed to distribute the set of subproblems so that the set of n global maxima is obtained before being written into secondary memory.

Global Data Register Containing Minimum Feasible Solution: The last architectural feature in the system is a common data register for all processors that contains the value of the minimum feasible solution. As a processor generates a feasible solution, its value is compared with the global feasible solution and replaces the global value if it is smaller. Contention may occur at this register if multiple processors try to update this register.

A good way to resolve the contention is to use a sequential associative memory.[25] The values of the feasible solutions are shifted out bit-serially and synchronously from all processors into a sequential associative memory that finds the minimum feasible solution. Because the associative memory is sequential, only n associative memory cells have to be used. An associative memory cell capable of minimum searches can be implemented with two flip-flops and four NAND gates. The processors and secondary storage are also constantly examining the updated values in the global data register and removing subproblems that have lower bounds greater than the minimum feasible solution.

We have presented in this section the functional design of the processor architecture. It is intended that this

design be implemented as a special-purpose single-chip or multichip computer.

PERFORMANCE EVALUATION OF THE PARALLEL BRANCH AND BOUND ALGORITHM

Following are some results on the performance of the parallel branch and bound algorithm using the vertex-covering problem. It is assumed in the simulation model that all processors operate synchronously. Each processor picks up a subproblem for expansion from its local list. Newly created subproblems are inserted back, and a complete distribution is attained before the next cycle starts. The two sequential steps just described above are an *iteration*. (Because it was more important to study the feasibility of the branch and bound algorithm at this time rather than to simulate the details of an implementation, only the number of iterations required to solve a problem were computed.)

The simulation model is written in C Language and implemented on a DEC VAX 11/780 computer with virtual memory. The simulation program is complex because complicated data structures and heap sort are used to achieve a fast insertion and retrieval time. The simulation program uses the following parameters: p, number of nodes in the graph; $PROB$, probability that an edge exists between two vertices in the graph; and n, number of processors.

Ten cases were simulated for each combination of p, $PROB$, and n. The number of nodes in the graph varied between 10 and 90. Parameters $PROB$ and n were chosen from the sets {0.1, 0.25, 0.5} and {1, 2, 4, 8, 16, 32, 64}, respectively. Simulation took about 50 hours of computer time on the VAX computer.

The VAX 11/780 computer was chosen for the simulations because of the availability of virtual memory. Large data structures have to be used when n and p are large. For example, for $n = 64$, $p = 90$, and a processor capacity of 110 subproblems, the total memory required for each processor is about 30K bytes, resulting in a total of 2M bytes for 64 processors. Furthermore, we assume that all computations are performed in CPU, and no secondary storage space is available during execution time. For this reason, best-first search cannot be applied when the memory space in a processor is exhausted, so the evaluation is allowed to switch to depth-first search within a processor when the amount of free space is less than the stack size of depth-first search. For comparison, assume that each processor has a storage capacity of 17 subproblems for best-first search, and additional memory is included to implement a stack for depth-first search (the space to store $p + 1$ subproblems). The results obtained are worse than if best-first search is used alone; however, this degradation is not because of the inherent property of the parallel branch and bound algorithm, but because of the unavailability of secondary storage space during computation.

We have performed a large number of simulations, a selected set of which is shown in Figures 13 through 15. In these graphs, the total number of iterations are broken down into the iterations for depth-first search and the iterations for best-first search. Sorting overhead is not included in the results. Figure 13 shows the average number of iterations required by a 64-processor system to solve a vertex-covering problem of different sizes. Insufficient memory space in the VAX computer caused a switch to depth-first search when p was greater than 50, and the effects of depth-first search became predominant when p was greater than 80. Figure 14 shows the performance of using different processors to solve a vertex covering problem of 90 nodes. The effects of depth-first search were still predominant; however, the decrease in the average number of iterations as the number of processors increased was linear. The usual diminishing return effects of parallel processors did not occur here. Figure 15 shows performance in terms of the average number of iterations to solve vertex-covering problems when $PROB$ equaled 0.25 and different numbers of processors were used. The slope of the curves were parallel when p was greater than 60, and this is also an indication that the effects of diminishing return is small.

When no distribution is used, the performance of the system is bad, because a large number of processors are idle. For example, when tests were done with random graphs of 60 to 90 nodes and 12 processors, the number of iterations with no distribution was as high as 10 times the iterations with complete distribution. Twelve processors were used so that the results could be compared with the results of Figure 9 ($k = 1$). Suprisingly, when one or more distributions were applied in each iteration, the total number of iterations was the same as if a complete distribution was used (with very small variations). The growth of the curve is therefore much faster than the curve in Figure 9. These results indicate that the performance degradation under incomplete distribution is extremely small.

The total simulation results indicate that the algorithm performs well when memory is sufficient to perform best-first search. Once it is switched to depth-first

search, the performance of the system drops, and the average number of iterations increases rapidly (exponential with the problem size). Furthermore, these results indicate that the improvement in using n processors is very near to n, and the effects of diminishing return is very small for the graphs tested. The simulators are currently being changed so that subproblems can be moved to secondary storage when there is insufficient space in main memory. It is expected that the performance will improve significantly. This will be reported in a future paper.

CONCLUSIONS

In this paper, we have proposed and studied the design of MANIP, a parallel computer system for processing NP-complete problems. Nondeterministic polynomial-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 optimal algorithm to complete the examination. Owing 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 uniprocessor system, optimally or suboptimally, is the branch and bound algorithm. A parallel version of the branch and bound algorithm that can be executed efficiently on a parallel computer system was described.

The parallel branch and bound algorithm requires a combination of sorting and merging. The subproblems are evaluated to produce new subproblems, which are inserted into a list of subproblems created previously. This list is maintained in a sorted order by the lower bounds of the subproblems 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 subproblems in the list. Because it is important to maintain a global sorted list of subproblems, a common memory shared by all processors can be used; however, this can become a bottleneck when the number of processors is large. We have proposed an alternative design such that each processor has a local memory, and the processors communicate with each other through an interprocessor communication network. When the processors have created new subproblems, they are first inserted into the local list, and then subproblems 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 unidirectional ring network is the optimal and most cost-effective way of implementing the interprocessor 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 an expense of increased hardware complexity. The proposed interconnection network is reliable because it is simple and it can be reconfigured. 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.

Because only limited memory can be implemented in each processor, we have proposed a system in which subproblems with maximum lower bounds are "paged" out onto secondary storage. We have not implemented this in the simulation model and have allowed the processing to switch from best-first search to depth-first search when there is insufficient memory in the simulation. We observed that the effects of depth-first search becomes predominant when the problem size is large. The effects of paging out subproblems with large lower bounds will be reported in a future paper.

REFERENCES

[1]E. L. Lawler and D. W. Wood. Branch and bound methods: A survey, *Operations Research*, 14 (1966), 699-719.

[2]R. M. Karp. Reducibility among combinational problems, *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, eds. (New York: Plenum Press, 1972), 85-104.

[3]C. Mead and L. Conway. *Introduction to VLSI Systems.* (Reading, Mass: Addison-Wesley, 1980).

[4]C. H. Sequin. Single chip computers, the new VLSI building blocks, *Proceedings of the Caltech Conference on Very Large Scale Integration* (1979), 435-445.

[5]W. A. Wulf and C. G. Bell. C.mmp–A multi-mini processor, *Proceedings of the AFIPS 1972 Fall Joint Computer Conference*, 41 (1972), 765-777.

[6]R. J. Swan, S. H. Fuller, and D. P. Siewiorek. The structure and architecture of Cm*: A modular, multi-microprocessor, *Proceedings of the AFIPS 1977 National Computer Conference*, 46 (1977), 637-644.

[7]D. J. Kuck. ILLIAC IV software and application programming, *IEEE Transactions on Computers*, C-17 (Aug. 1968), 746-757.

[8]A. M. Despain and D. A. Patterson. X-tree: A tree structured multiprocessor computer architecture, *Proceedings of the 5th Symposium on Computer Architecture* (1978), 144-151.

[9]J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor, *Proceedings of the Second Annual Symposium on Computer Architecture* (1975), 126-132.

[10]M. C. Sejnowski et al. An overview of the Texas reconfigurable array computer, *Proceedings of NCC* (1980), 631-641.

[11]J. A. Harris and D. R. Smith. Hierarchical multiprocessor organizations, *Proceedings of the 4th Annual Symposium on Computer Architecture* (1977), 41-48.

[12]B. C. Desai. The BPU, a staged parallel processing system to solve the zero-one problem, *Proceedings of ICS'78* (Dec. 1978), 802, 817.

[13]B. C. Desai. A parallel microprocessing system, *Proceedings of the 1979 International Conference on Parallel Processing* (1979), 136.

[14]O. I. El-Dessouki and W. H. Huen. Distributed enumeration on network computers, *IEEE Transactions on Computers*, C-29 (Sept. 1980), 818-825.

[15]L. Mitten. Branch and bound methods: General formulation and properties, *Operations Research*, 18 (1970), 24-34.

[16]A. H. Land and A. Doig. An automatic method for solving discrete programming problems, *Econometrica*, 28 (1960), 497-520.

[17]W. L. Eastman. A solution to the traveling salesman problem, (Presented at the American Summer Meeting of the Econometric Society), Cambridge, Mass., 1958.

[18]M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-completeness* (San Francisco: W. H. Freeman and Co, 1979).

[19]K. E. Batcher. Sorting networks and their applications, *Proceedings of AFIPS SJCC*, 32 (April 1968), 307-314.

[20]W. Kohler and K. Steiglitz. Characterization and theoretical comparison of branch and bound algorithms for permutation problems, *JACM*, 21 (1974), 140-156.

[21]T. Ibaraki. Computational efficiency of approximate branch and bound algorithms, *Mathematics of Operations Research* (1976), 287-298.

[22]_____. On the computational efficiency of branch and bound algorithms, *Journal of Operational Research Society of Japan*, 20 (1977), 16-35.

[23]_____. The power of dominance relations in branch and bound algorithms, *JACM*, 24 (1977), 264-279.

[24]F. P. Preparate. Parallelism in sorting, *Proceedings of the 1977 International Conference on Parallel Processing* (Aug. 1977), 202-206.

[25]C. V. Ramamoorthy, J. L. Turner, and B. W. Wah. A design of a fast cellular associative memory for ordered retrieval, *IEEE Transactions on Computers*, C-27 (Sept. 1978), 800-814.

[26]H. T. Kung. The structure of parallel algorithms (Research report), Carnegie-Mellon University, 1978.

[27]D. E. Knuth. *The Art of Computer Programming, Sorting, and Searching*, vol. 3 (Reading, Mass: Addison-Wesley, 1973).

[28]T. C. Chen and C. Tung. Storage management operations in linked shift register loops, *IBM Journal of Research Development*, 20 (March 1976), 123-131.

[29]W. E. Kluge. Data file management in shift register memories, *Proceedings of the 3rd International Conference on Very Large Data Bases Part II* (Oct. 1977), 1-13.

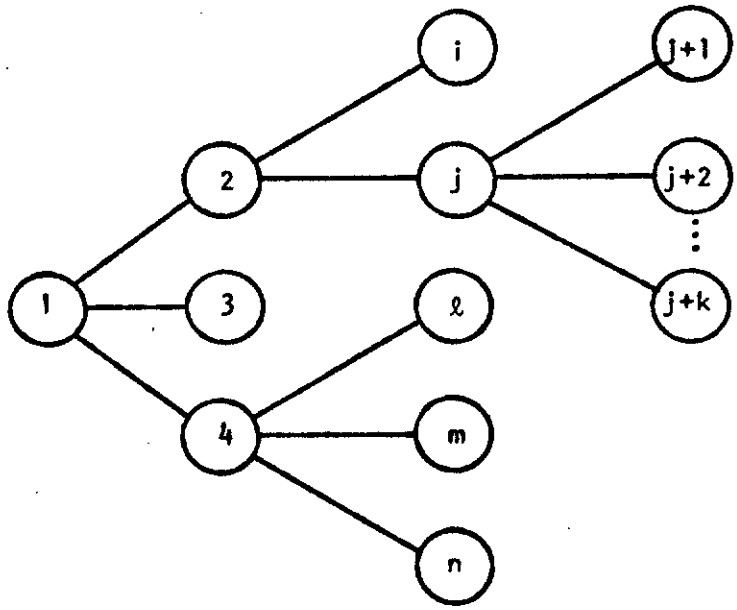


Figure 1. A Branch and Bound Tree

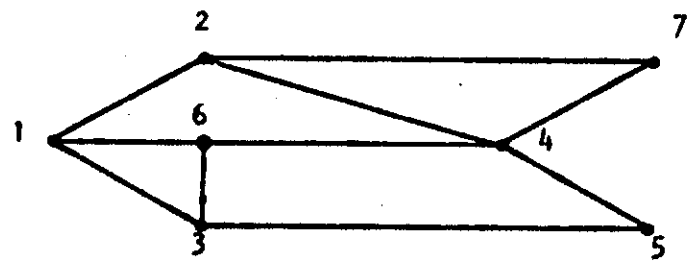


Figure 2. Example Graph

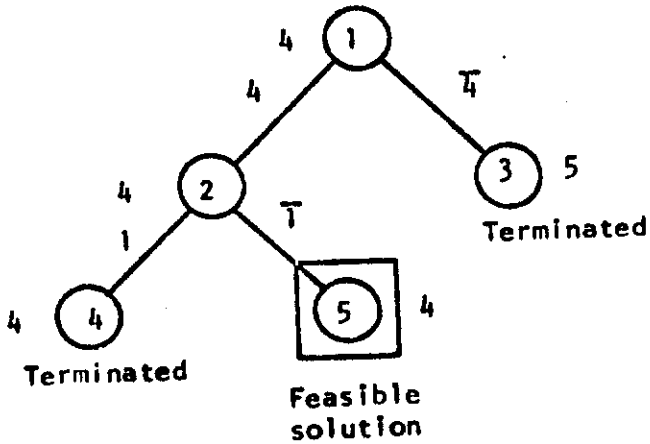


Figure 3. Branch and Bound Tree for Figure 2
- Number in node = evaluation order
- Number outside node = lower bound
- Number on edge = included or excluded node

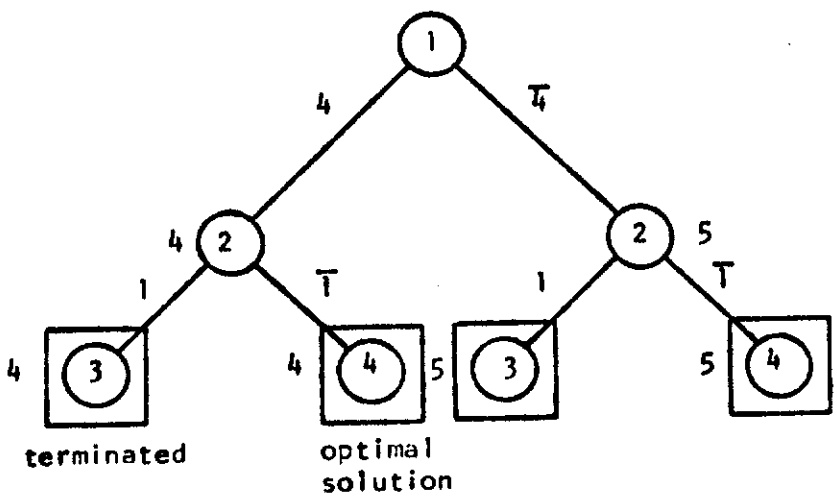


Figure 4. Parallel Branch and Bound Tree for Figure 2, with Two Processors

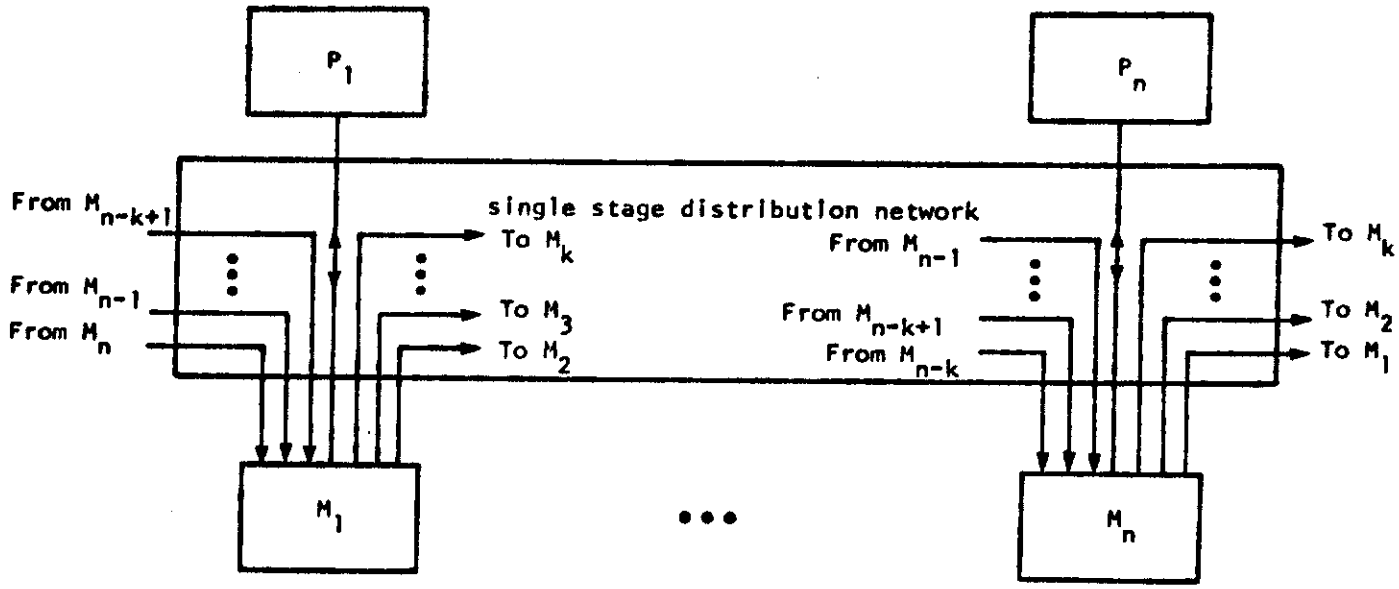


Figure 5. K-Connected Single-Stage Distribution Network
- Number of Memories = Number of Processors.
- k Subproblems are distributed in parallel to neighboring processors.

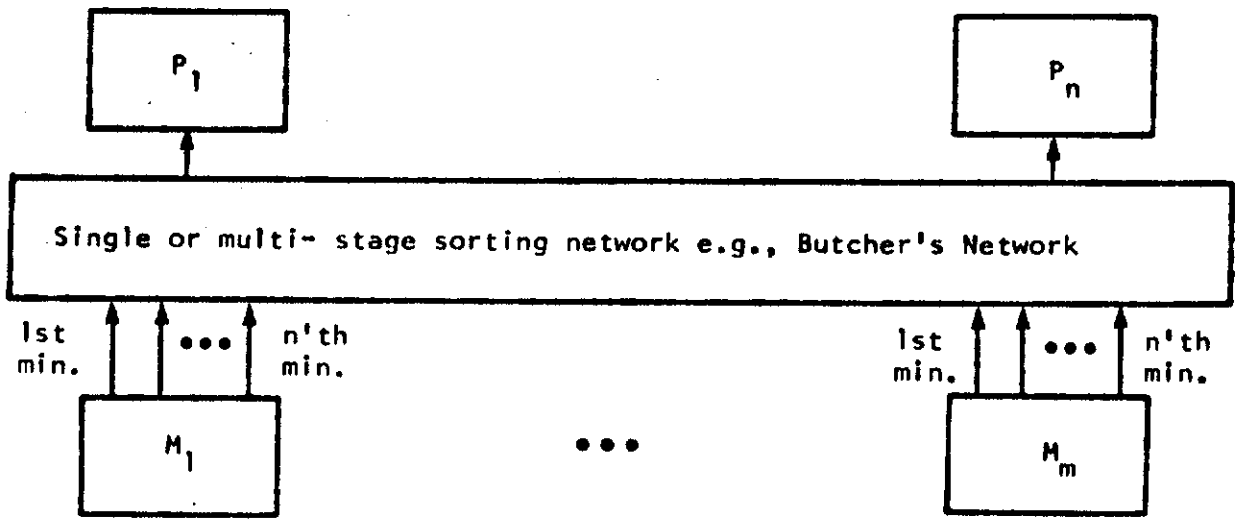
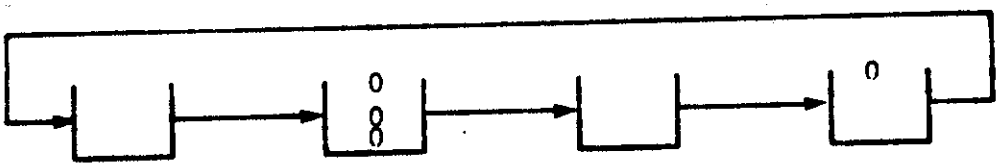
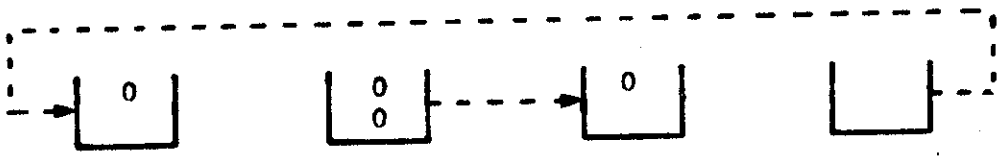


Figure 6. Single-Stage or Multistage Sorting Network
- Number of memories may not equal number of processors.
- n minima are taken from each memory to be sorted.

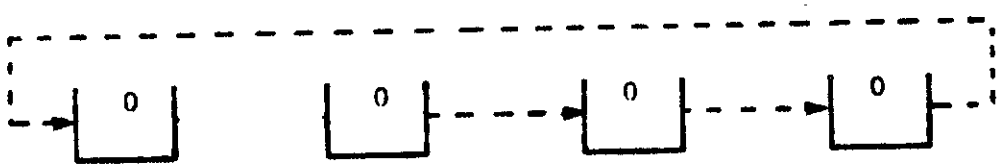


(a) Uni-directional ring network connecting four urns with initial state (0,3,0,1)



(b) State after 1 left shift

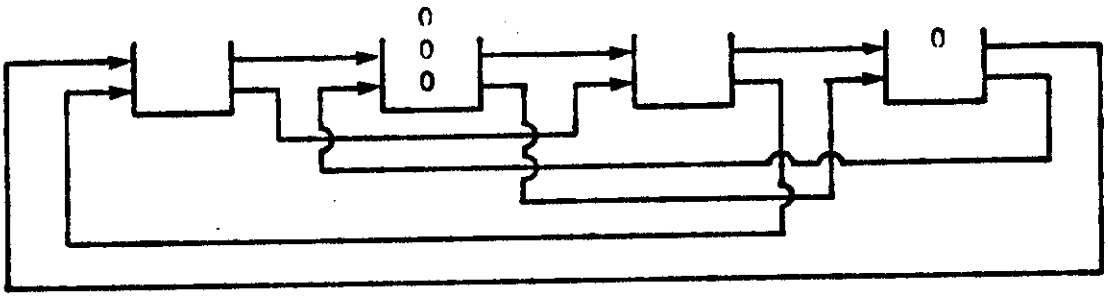
(c) State after 2 left shifts



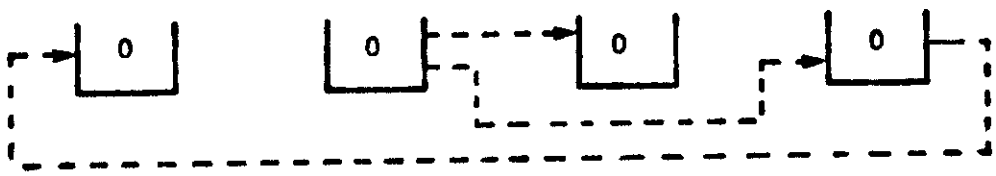
(d) State after 3 left shifts

distribution of marbles

Figure 7. Unidirectional Ring Network Connecting Four Urns



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



(b) State after 1 shift.

----► distribution of marbles

Figure 8. 2-Connected Network Connecting Four Urns

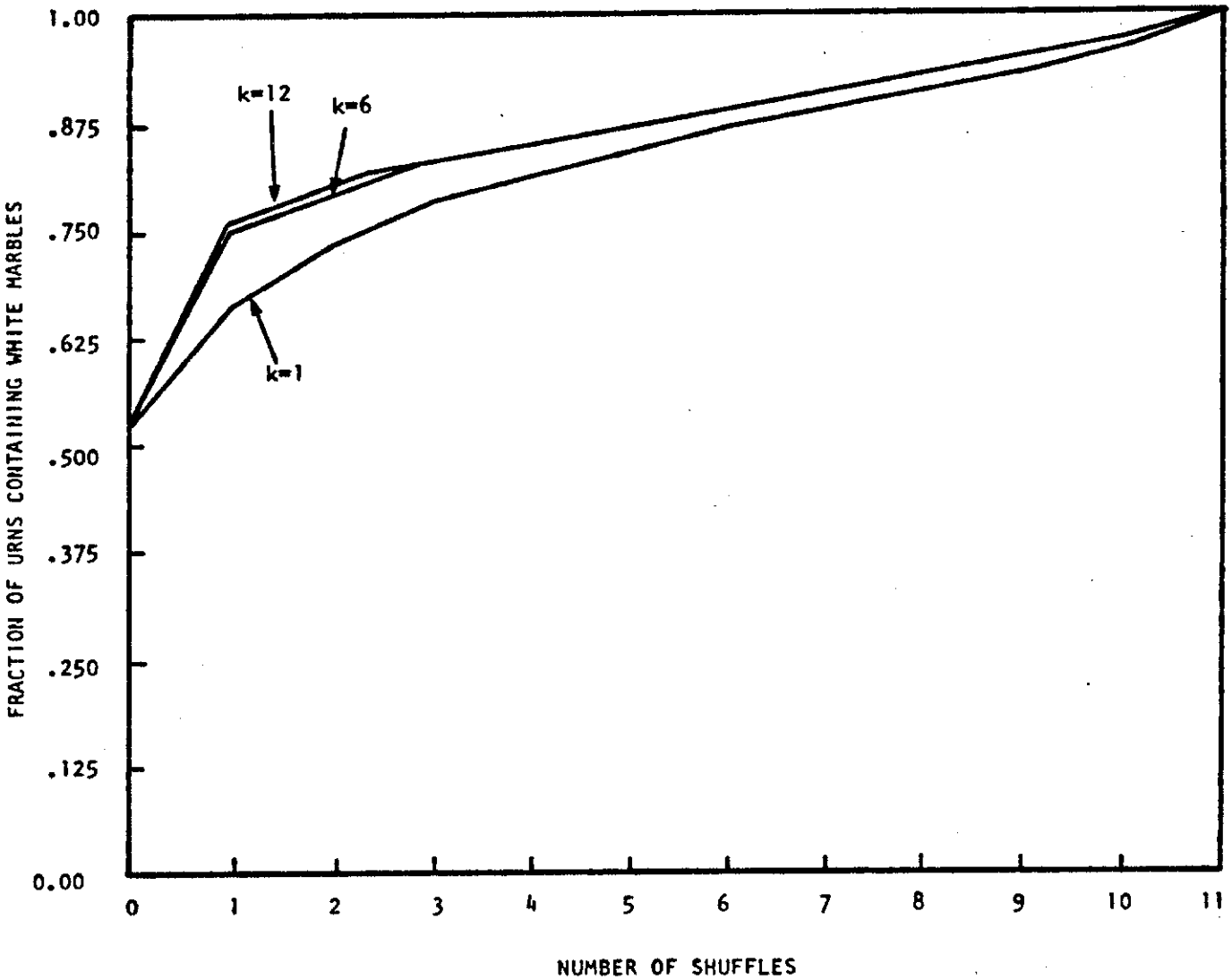


Figure 9. Performance of k-Connected Network for 12 Urns (k = 1 for Unidirectional Ring Network)

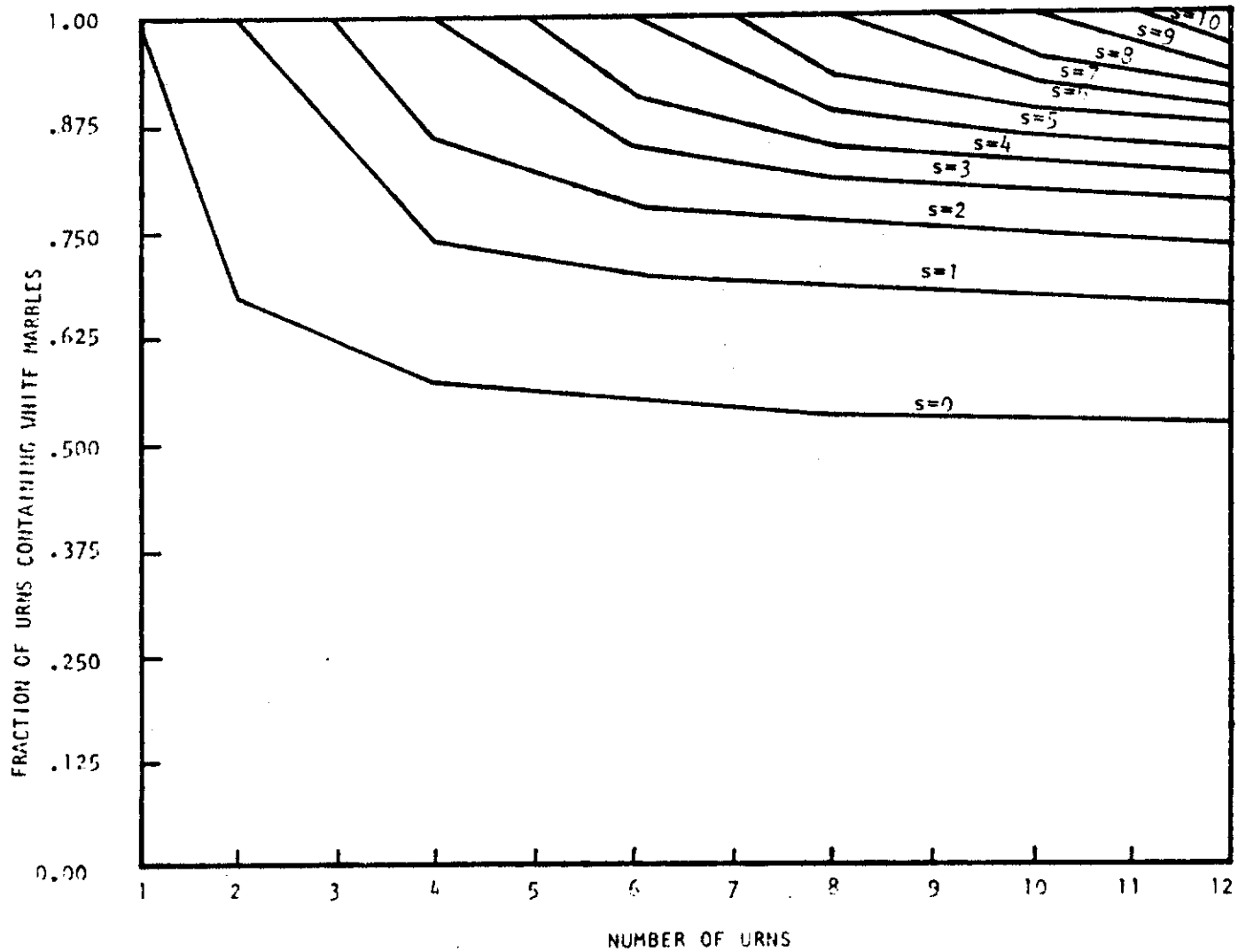


Figure 10. Performance of Unidirectional Ring Network
- k = 1 for Unidirectional Ring Network
- s = Distribution

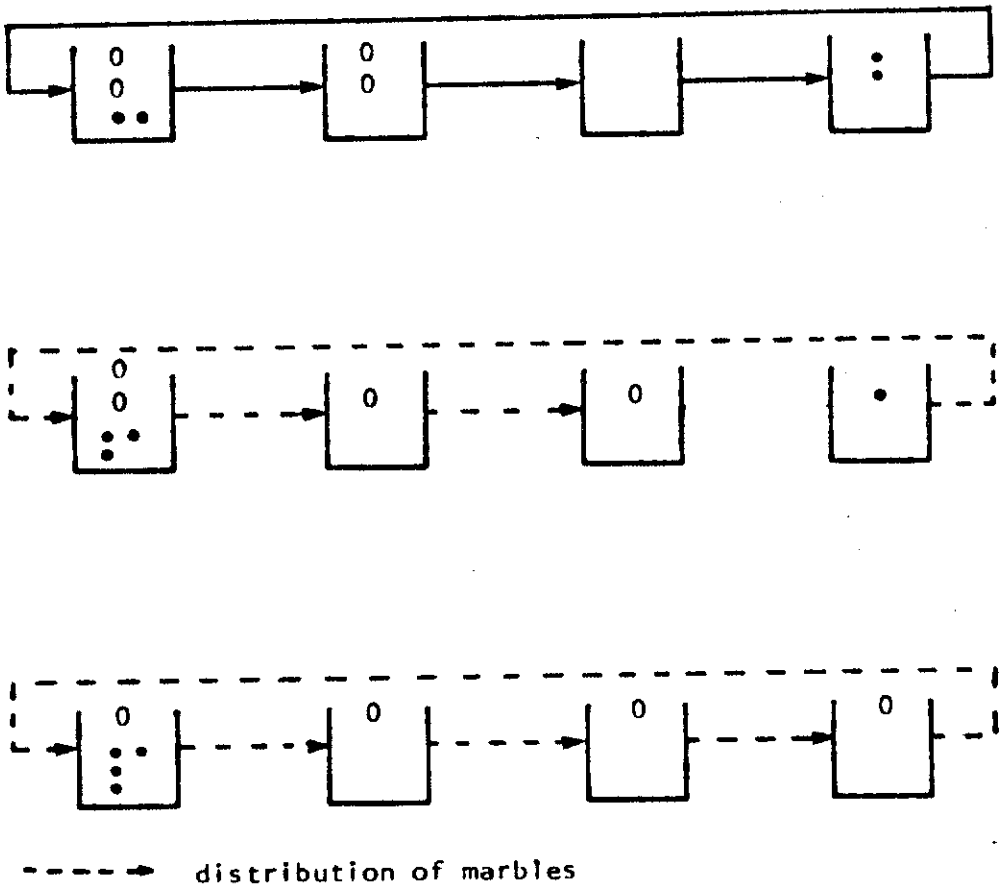


Figure 11. Decrease in Urns Containing Black Marbles when Distribution Is Complete

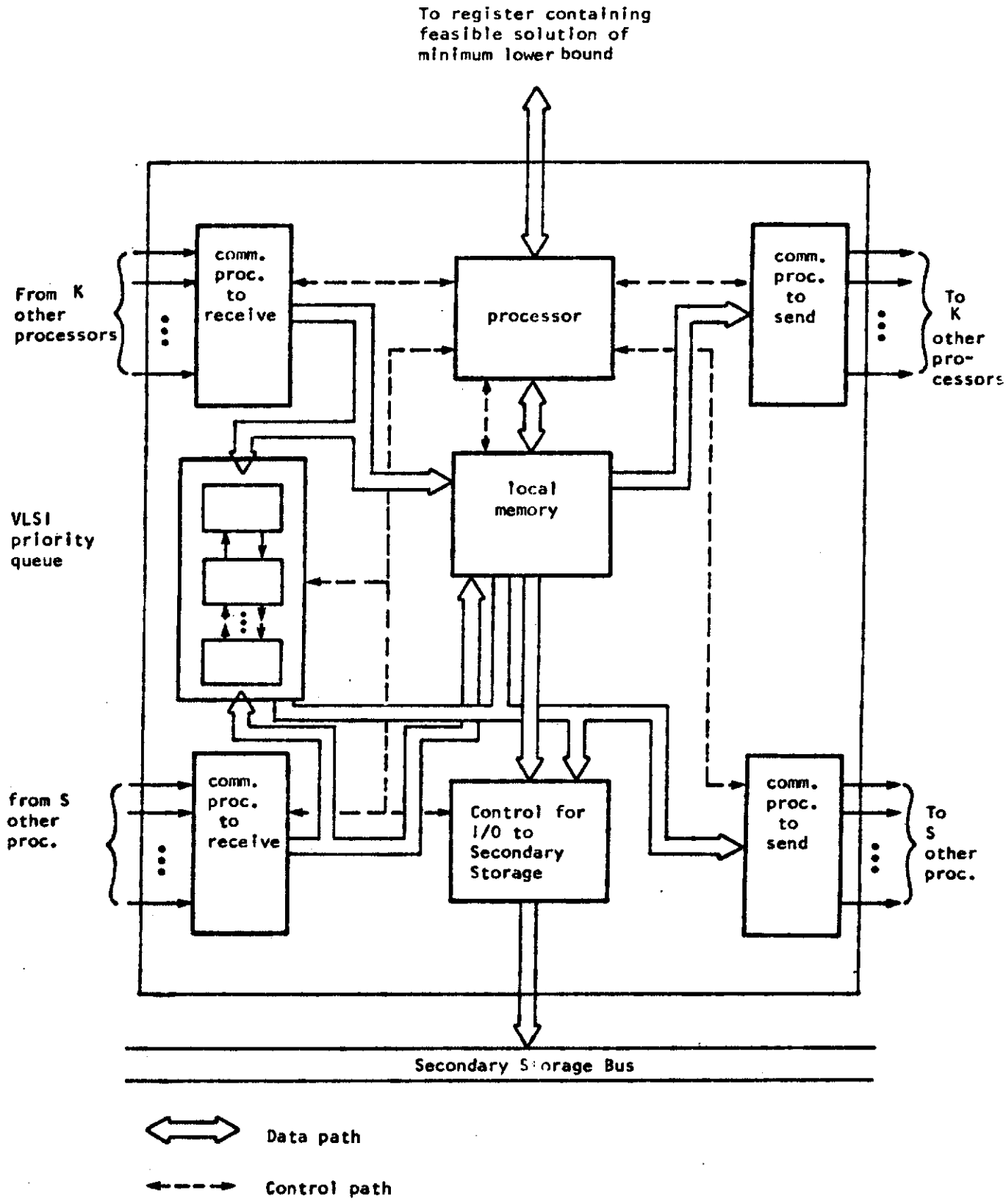


Figure 12. Processor Architecture

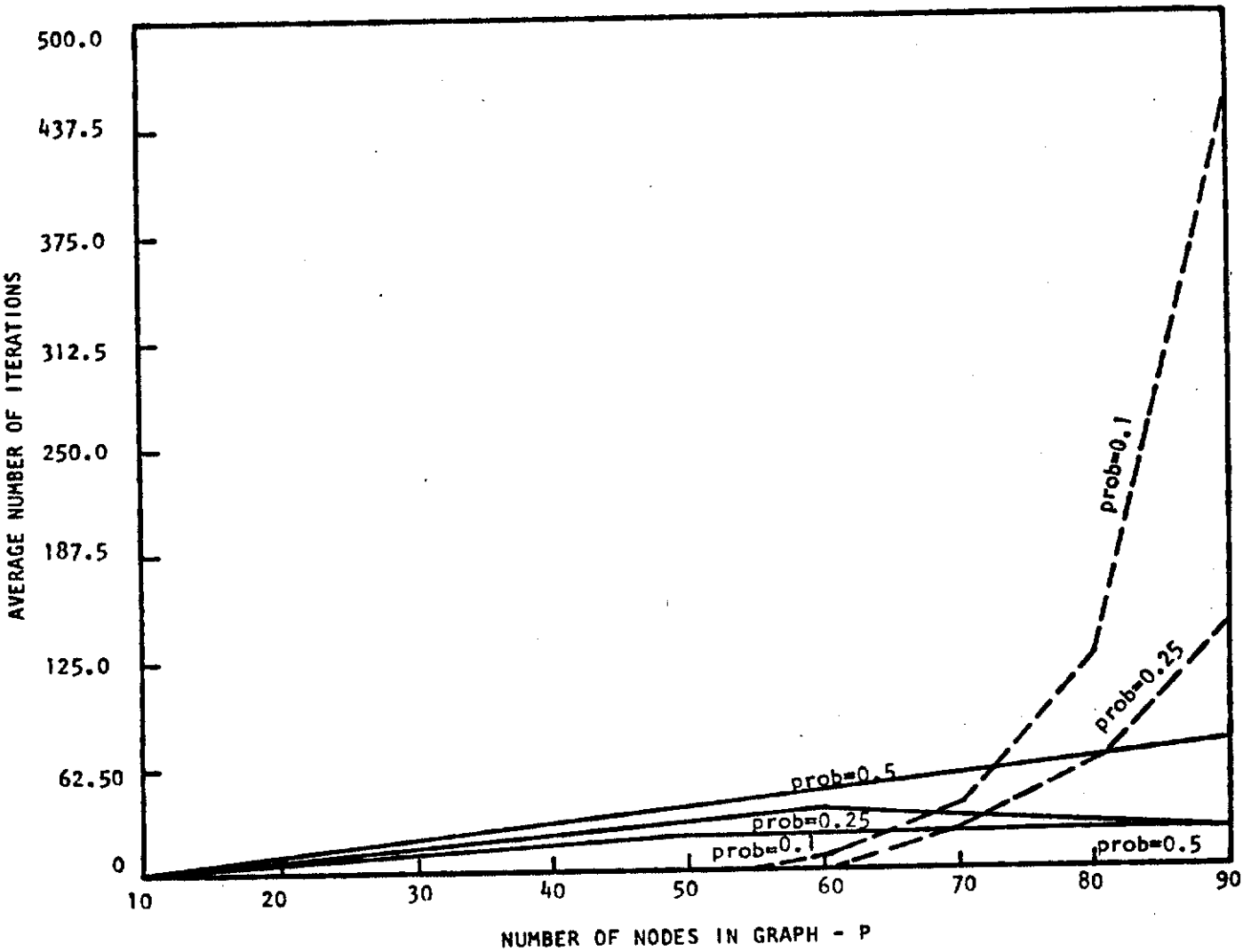


Figure 13. Average Iterations Required by 64-Processor System to Solve Vertex-Covering Problem of Different Sizes

- **Best-First Search = Solid Line**
- **Depth-First Search = Broken Line**

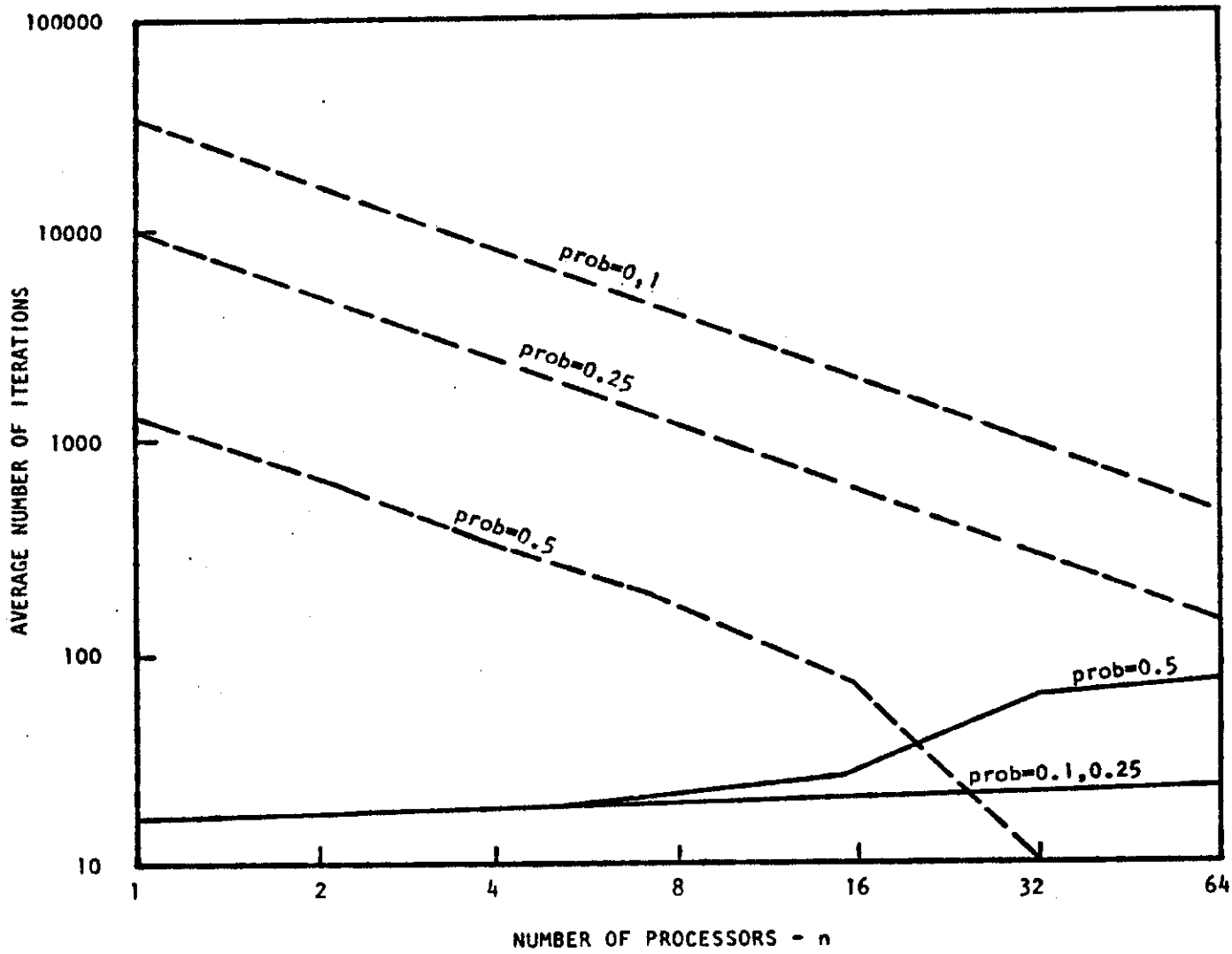


Figure 14. Average Iterations Needed to Solve Vertex-Covering Problem with 90 Nodes Using Different Numbers of Processors

- **Best-First Search = Solid Line**
- **Depth-First Search = Broken Line**

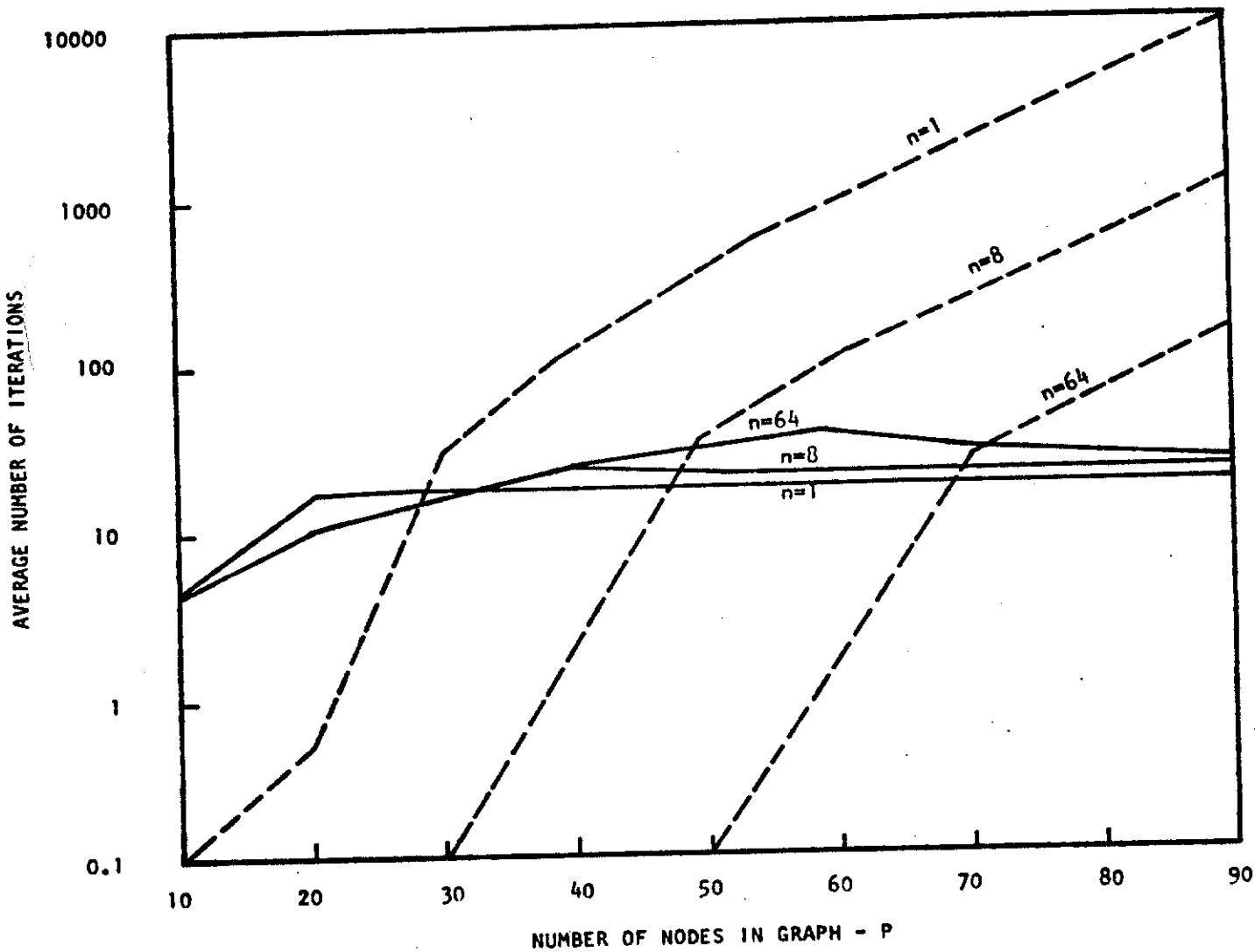


Figure 15. Improvement in Average Iterations when Larger Numbers of Processors Are Used (n)
- **Probability of a Graph Edge = 0.25**
- **Best-First Search = Solid Line**
- **Depth-First Search = Broken Line**