Tree-Searching Algorithms on Parallel Architectures

Mala Mebrotra
College of William & Mary - Arts & Sciences

TREE-SEARCHING ALGORITHMS ON PARALLEL ARCHITECTURES

A Thesis

Presented To

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Master of Science

by

Mala Mehrotra

1985

APPROVAL SHEET

This thesis is submitted in partial fulfillment of
the requirements for the degree of

Master of Science

Mala Mehrotra
Author

Approved, March 1985

John R. Van Rosendale
J. R. Van Rosendale

W. Robert Collins
W. R. Collins

Robert E. Noonan
R. E. Noonan

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I am deeply indebted to my advisor, Dr. John Van Rosendale, for his invaluable guidance during the course of this research. From him I have learned not only many of the basic tenets of Computer Science, but also the art of conducting research.

I am also grateful to ICASE, for providing me with excellent facilities and a very inspirational atmosphere to conduct this research. In addition, I have benifited immensely from my association with the professors of College of William and Mary, in particularly Dr. Collins, Dr. Noonan and Dr. Feyock. I would also like to express my gratitude to Dr. Dennis Gannon, for his keen interest and enthusiasm in my work.

It would not be out of place to appreciate all the encouragement I got from my husband, Piyush, at all stages of this thesis. Finally, I would like to acknowledge the moral support that I have received from my parents, parents-in-law and friends during this period.

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

LIST OF GRAPHS

LIST OF GRAPHS contd.

ABSTRACT

Tree searching is a fundamental technique in computer science, having applications in combinatorial optimization, artificial intelligence, robotics, database searching, operations research, numerical analysis, and computer game-playing. The research here focuses on the problem of mapping tree searching algorithms to non-shared memory multiprocessor architectures. Though there is substantial parallelism in most tree searching problems, this parallelism can be difficult to exploit on highly parallel machines interconnected via a packet-switched network. Our work is directed at discovering heuristically based adaptive load distribution algorithms which can distribute the load nearly uniformly over the processors, without incurring excessive communications costs. Factors affecting the performance of these strategies, such as interconnection network topology, network size, communication speed, and the particular search tree problem, are studied via simulation. Results obtained show that relatively simple load balancing strategies can map tree searching algorithms to parallel architectures quite well.

TREE-SEARCHING ALGORITHMS ON PARALLEL ARCHITECTURES

CHAPTER 1

INTRODUCTION

1.1. Importance of parallel tree searching

Tree searching is a fundamental technique in computer science, with applications in many diverse areas. Tree searching algorithms are used primarily in problems where a sequence of choices must be made in finding the solution. For example, in the field of artificial intelligence, tree searching algorithms are used in game playing and in natural language understanding. In robotics, tree searching is used in path planning algorithms. It is also used in combinatorial optimization problems, such as the traveling salesman problem and bin packing problems. Such combinatorial optimization problems occur in a variety of contexts, such as maximizing the utilization of the available resources in operating systems and factory production lines. Tree searching also occurs in adaptive numerical algorithms, computer graphics, and robot collision detection.

Tree searching algorithms have been seriously studied for many years because of their wide range of applications. An important characteristic of many tree search problems is their high computational complexity[2]. Combinatorial search problems, in particular, often have complexity exponential in the problem size. Various pruning techniques, such as alpha-beta pruning and branch and bound strategies, are often employed to reduce this complexity[19]. These techniques greatly reduce the size of the tree to be searched. However, they are problem specific and applicable only on a subset of tree search problems. Many problems remain in which the construction of a large search tree is the only reasonable algorithm known.

Recently, with the advent of parallel computers, the possibility of exploiting multiprocessor architectures in tree searching has begun to receive serious attention[6,10,30]. There is a natural match between tree search algorithms and parallel architectures because of the inherent parallelism of these algorithms. This is especially true since it appears possible to split tree search problems into relatively independent subproblems, which can be executed in parallel.

In this thesis, we study the issues involved in mapping tree searching algorithms onto non-shared memory architectures. The proper distribution of the subproblems on the processors becomes critical as one tries to maximize the utilization of the processors. Our research is directed at discovering heuristically based adaptive load distribution algorithms which can distribute the work nearly uniformly over the processors, without incurring excessive communication costs.

1.2. Multiprocessor Architectures

A variety of multiprocessor architectures have been designed and built. These multiprocessor architectures can be characterized on the basis of several factors, such as the type of memory access and interprocessor communication hardware. Architectures can be classified as shared memory or non-shared memory depending on whether all the memory is directly addressable by any processor. In shared memory systems, such as the Carnegie Mellon CM*[9] and the CRAY XMP, processors communicate by accessing shared data structures in the common memory. Non-shared memory architectures, such as the Caltech Cosmic Cube[24] the Purdue Pringle[13,14] and the NASA Finite Element Machine[12] are characterized by the fact that each processor has only local memory. In such systems, processors communicate via message passing, rather than by reading and writing in the shared memory. The time required to communicate in non-shared memory systems depends strongly on the interconnection structure.

Non-shared memory systems can be further classified according to whether their interconnection networks are multi-stage or single-stage[29]. In a multi-stage network, interprocessor communication is established via several layers of switches interconnecting the processors. In single-staged networks, the processors are directly interconnected, with no intervening switches. Multi-stage networks can be circuit-switched or packet-switched. In circuit-switched networks a direct electrical path is established between source and destination, while in packet-switched networks store and forward communication is employed. Single-stage networks are universally packet-switched. Similar classifications apply to processor-memory interconnection networks in shared memory systems.

In this research, we concentrate on non-shared memory architectures consisting of a number of interconnected processors, each having its own local memory. These processors are interconnected via a single-stage packet-switched communications network. Such architectures are attractive, since they are highly scalable, and computers with large numbers of processors can be built in this fashion. However, such architectures are also more difficult to program than the more expensive shared memory architectures.

1.3. Parallelism in Tree Search Algorithms

Tree search algorithms contain a great deal of parallelism. This is true since subtrees rooted at sibling nodes of a tree can be searched independently. Consequently, once a node has been assigned to a processor, this processor can search the whole subtree rooted there, without requiring any communication with other processors. The need for communication arises only when work needs to be redistributed between processors, and also when the answers produced by searching subtrees are combined to give the final answer. Thus, unless dynamic load balancing is used, the need for communication is quite minimal. Note that this is true only for an exhaustive tree search, where pruning is not employed. Pruning techniques generally require knowledge of all parts of the tree currently being expanded, which in turn requires additional communication. Our goal is to study parallelism in tree searching problems under the most favorable assumptions, and thus we did not look at pruning issues.

The performance of a tree search algorithm on non-shared memory architectures seems to be dependent on many factors, including:

(1) the interconnection network topology

(2) the load distribution strategy

(3) the ratio of the speed of communication to computation

The network topology in non-shared memory systems dictates the penalties incurred due to the communication demands of a parallel algorithm. A particular interconnection pattern may be very well suited to a particular parallel algorithm, while being totally unsuited for some other algorithm. The networks studied here are: a bus, a complete connection, a hypercube, a ring, and a tree network. These five networks were selected as being representative of packet-switched communication networks used in multiple microprocessor architectures.

A second important issue is load distribution. The purpose of a load distribution strategy is to optimize the utilization of a parallel system by balancing the work load on the processors; and the performance of a parallel architecture can vary dramatically with the load distribution strategies used. Several simple load distribution strategies requiring few assumptions about the underlying interconnection topology and about the characteristics of the tree being searched are examined in this thesis.

The relative speed of communication network in a non-shared memory architecture is also clearly important. The faster the network, the more potential there is to spread work evenly throughout the network. However, it maybe possible to compensate for slow communication by using a more sophisticated load distribution strategy. This is one of the types of questions we wish to answer through our study. To this end, load distribution strategies of varying sophistication, and possibly of varying message intensities were chosen for study.

For this research, we picked four problems for our study: the eight queens problem, the knight's tour problem, the traveling salesman problem and an adaptive quadrature problem. Each problem is solved through the generation of a large search tree. We simulate the execution of the tree search algorithms on non-shared memory architectures, in order to assess the effectiveness of different load balancing strategies and the effect of different networks and different communication speeds.

1.4. Background

Two recent papers, have also addressed some of the issues we are studying. Gray, McCormack and Haralick[10] have experimented with the mapping of the consistent labeling problem on two interconnection topologies, a ring and a hypercube. Their approach is to parameterize the important factors in their load distribution strategy, in order to study the effects of varying these parameters in simulation experiments. Their load distribution strategy is based upon a forward checking pruning algorithm which provides knowledge about the size and complexity of the subproblem that is yet to be handled.

Several different parameters were varied in their study. Two different search strategies breadth-first and depth-first were utilized. Secondly, the size and number of subproblems passed to a neighbor was varied. The cut-off point, at which a processor completes its subproblem rather than subdivide and pass it, was also varied. Simulation runs were made for two different networks: the ring and a 64 processor hypercube.

Their load distribution strategy is based on polling. Each processor polls its neighbors, after processing each node, and passes work to idle neighbors. This is in contrast to other approaches, where processors interrupt neighbors and request work. The minimum execution time was found for the depth-first strategy, when the subproblem size passed was large and the number of subproblems passed was approximately half the total work on the processor.

Burton and Huntbach[6] in their research, have viewed the tree being searched as a tree of processes. They define their load distribution algorithm for these processes, together with the interconnection network chosen, as a virtual tree machine. They have chosen their interconnection network from a family of networks called binary n-cubes. Each processor is capable of running more than one process. Their load distribution strategy is priority based; children nodes having a priority one greater than their parent. Thus, a priority queue of processes resides on each processor during run time. A processor, in the absence of any communication with its neighboring processors, carries out a simple depth first search for the node with the maximum depth in the tree. When a processor has to decide which process to transfer to another processor, it chooses one with the least priority, thus transferring as much work as possible in one transmission. A process can migrate only once. Hence, only local communication is required.

In order to balance the workloads of all processors, a processor sends work to a neighboring processor only if the destination processor has less work than itself. Thus, every processor communicates its current workload to each of its neighbors at every communication cycle. Computation and communication proceed in parallel. One difficulty they encountered was that the information a processor received from its neighbors was often out of date, thus reducing the effectiveness of their load distribution algorithm.

Burton and Huntbach have chosen to measure the performance of their virtual tree machine by processor utilization. Their simulation results, based on a "well behaved" tree algorithm show that the execution time of a program can be divided into three parts: start-up period, when work diffuses through the network; main period, when processor utilization is almost a 100%; and wind-down period, when little or no work is left for distribution purposes. Their wind down period is longer than the start up period because processes cannot migrate more than once. It was found that the main period of execution was longer on larger problems. But as the number of processors increases, larger problems are needed to saturate the architecture and achieve high utilization.

1.5. Overview of Thesis

Our research, though similar in spirit to the work described in the last section, focuses on issues that have not been investigated before. Our approach is to investigate several algorithms and several load distribution strategies, comparing the difference in their performances. Chapter 2 discusses the different network topologies considered. Chapter 3 describes the tree search problems studied. The search trees generated by these problems are classified according to several parameters. Chapter 4 describes our load distribution strategies and the reasons for choosing these strategies. Chapter 5 presents the results of our simulation experiments. It examines the performance of these load distribution strategies from a variety of points of view. Finally, the thesis concludes with a short summary in Chapter 6.

CHAPTER 2

ANALYSIS OF NETWORKS STUDIED

The internode communication network in a multi-microprocessor system is perhaps the most critical part of the system, having a strong impact on both system performance and algorithm choice. In this chapter we describe the five interconnection network topologies studied in this thesis. Next we examine their theoretical performance and cost effectiveness. This theoretical analysis is intended as a background to prepare the way for understanding the simulation results given in Chapter 5. Before presenting the five networks, some of the factors distinguishing various networks are described.

2.1. INTERCONNECTION NETWORKS

A wide variety of networks have been proposed and many have been implemented recently, largely because of the availability of low cost microprocessors and the advances in VLSI technology[18]. However, there does not seem to be any "best" interconnection network because the cost-effectiveness of a particular network depends on many factors. These factors include the complexity of the computational tasks to be performed on it, the desired data transfer rate, and the practical realizability of the system.

Our attention is focussed on single-stage packet-switched networks. The reason for choosing this class of networks is two-fold. First, the load distribution strategies being explored here generate communication patterns which are not known a priori. Packet-switched networks give us the flexibility to develop load distribution strategies which are independent of network topology. Circuit-switched networks[3,15,23] are difficult to use unless all data movements are preplanned. Second, the higher bandwidth possible with multi-stage networks seems unnecessary for tree searching problems. We would expect to be able to extract the parallelism of these problems without massive communication, and thus the less expensive single-stage networks should suffice for this problem class.

In this chapter we examine the theoretical performance of the networks studied based on several criteria, the total cost of the network, the mean internode distance, and the network throughput. This analysis follows closely that given in Reed[21]. It is also possible to study interconnection networks by looking at the set of permutations they pass[26] or by using queueing theory techniques[21]. Looking at the set of permutations passed is a valuable approach for circuit-switched networks but is not applicable here. Queueing theory, on the other hand, is a dual approach to the analytical approach followed here. Queueing theory has advantages, such as the ability to treat the case of bounded buffers, but also gives only approximate answers. For the situations here, either approach would be equally appropriate.

2.2. NETWORKS STUDIED

In the following sections we describe and analyze the five interconnection networks that were simulated. In this section we describe the five interconnection networks. Section 2.3 gives the routing algorithms needed in each of the networks to implement long distance packet switched communication. Next, section 2.4 defines the criteria by which we look at the performance of these networks. In section 2.5 the five networks considered are analyzed according to these criteria. The chapter ends with some remarks in section 2.6 on the relative utility of these various networks. In the following discussion, the total number of nodes in the system is represented by P.

The five interconnection networks are a bus, complete connection, hypercube, ring, and a tree. These networks seem to be fairly representative of single-stage packet-switched networks. All have been used in multi-microprocessor architectures.

Bus

In a bus network all P nodes are connected directly to a single shared bus, as shown in Figure 2.1. Bus arbiters, also known as bus controllers, resolve contentions. The Intel 432[1] is an example of a bus network. CMU's Cm*[9] is a loosely coupled bus network, where there is a hierarchy of bus levels. Buses are easy to implement and are efficient for small numbers of nodes. With a large number of nodes, bus contention limits the utility of this system.

Complete Connection

The complete connection network consists of interconnections between each pair of nodes (Figure 2.2). Thus each node requires P-1 ports to connect to all other nodes. Its main disadvantage is the large number of channels and ports involved. But it is quite practical for a moderate number of nodes and is studied here partly because it provides an upper bound on network performance.

Hypercube

A hypercube network joins nodes such that a node can communicate with any of the nodes whose address, when written in binary, differs from it in any one bit. The hypercube network is based on the concept of arranging nodes on the vertices of a

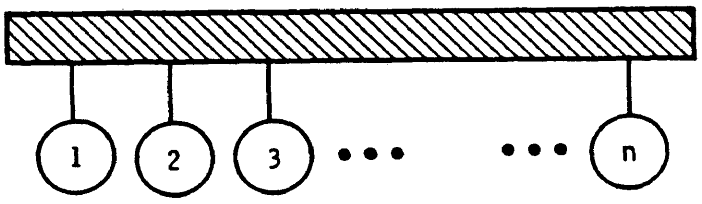


Figure 2.1 Bus Interconnection Network

Figure 2.2 Complete Interconnection Network

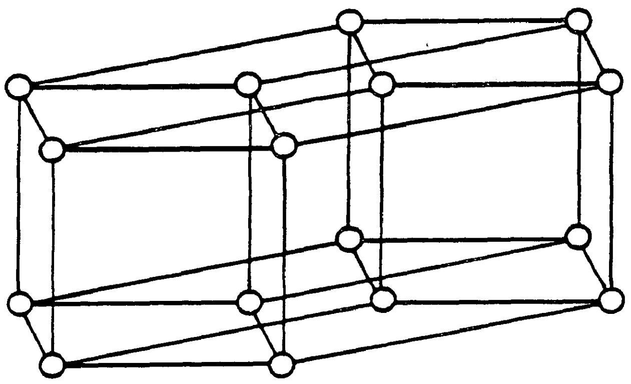


Figure 2.3 Hypercube Interconnection Network

hypercube of dimension D where $D = \log P$. Thus each node is connected to $\log P$ nodes as shown in Figure 2.3. The Caltech Cosmic Cube[24] is based on a hypercube network.

Ring

A ring network consists of P nodes, connected together in a circle (Figure 2.4). Node i is connected to nodes $i+1 \bmod P$ and $i-1 \bmod P$. Messages are passed around the ring in one or both directions. The ring network studied here is bidirectional. University of Maryland's ZMOB[22] consists 256 nodes connected by a unidirectional ring.

Tree

The tree network consists of a binary tree with processors residing at every node as shown in Figure 2.5. Each node is connected to its parent and its right and left children. The root of the tree has only left-child and right-child connections, while the leaf nodes have only a parent connection. We consider here only complete binary trees with $P = 2^k - 1$ nodes, for some k. The tree network is the only asymmetric network we have considered in our study, which means that the network topology is different as seen from different nodes. The tree machine being built at University of North Carolina[17] is an example of a tree network. A tree machine has also been built at Caltech[4,5].

2.3. Routing Algorithms

In packet-switched networks, each node in the network receives messages and forwards them towards their destination through a sequence of intermediate nodes. In

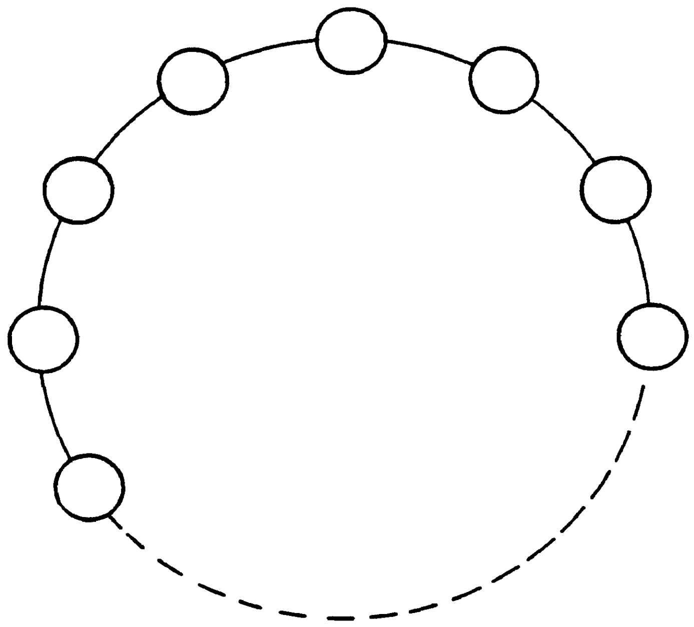


Figure 2.4 Ring Interconnection Network

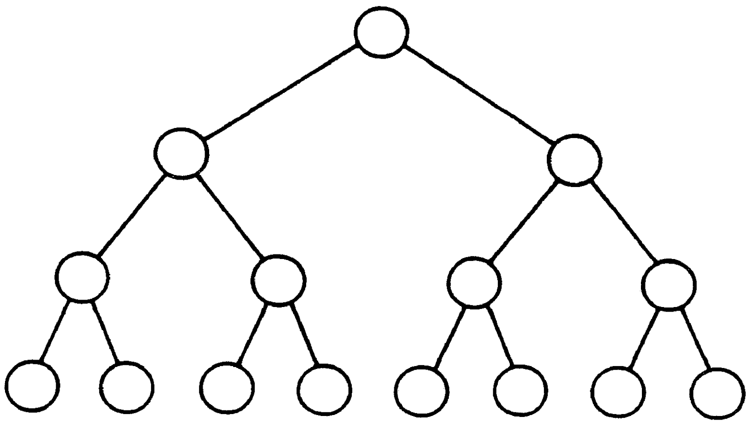


Figure 2.5 Tree Interconnection Network

the event that the communication channel required for forwarding a message is busy, the message is stored in a buffer and is forwarded on once the channel becomes clear. In our analysis, we presume that infinite buffers are available at each node for storing such messages.

The algorithm required to route messages at intermediate points is dependent on the network under consideration. For several of the networks studied here, the routing algorithm is completely trivial and little needs to be said. For others, there are subtleties involved, and the routing algorithm selected can affect the performance of the network.

Consider first the bus network. Hardware bus arbiters are used in a bus to resolve bus contention. A node attempting to send a message contends for the bus, and when it gets control of the bus, the message is transmitted immediately to the destination node. No intermediate nodes are involved.

In the complete connection network, each node is connected to all other nodes, so messages are again directly transmitted from source to destination. With this network, in each communication cycle, P messages can be transmitted.

In a hypercube network, a node is directly connected to $\log P$ neighbors. As the message is routed through the network the following algorithm can be used to determine the next node:

```
xor_vector := present_address XOR destination_address
i := position of a random non-zero bit in the xor_vector
send message to the i-th neighbor
```

Note for P nodes, the address of a node requires $\log P$ bits, so that the *xor_vector* has $\log P$ bits. It can be shown that using this routing algorithm, algorithm, a message requires at most $\log P$ steps to reach its destination. The random choice introduces non-determinism in the sense that two messages sent from the same source to the same destination may take different routes and hence arrive out of order. There exist alternate routing algorithms which are determinate and preserve the order of message arrival[16].

The routing algorithm for bidirectional ring networks is simple. Every message is forwarded around the ring in the clockwise or counter clockwise direction, depending on which direction leads to the shorter path. Note that the average distance traveled by a message is $\frac{P}{4}$.

The routing algorithm for a tree is more complex. If a message is not at its destination, then the following algorithm can be used to forward the message one step closer to its destination:

```
if (destination node in subtree rooted at the present node)
    then
        if (destination node in subtree rooted at the right child of present node)
            then
                send to right child
            else
                send to left child
    else
        send to parent
```

This routing strategy requires a simple kernel routine, dependent on the node numbering scheme, to determine whether a node is in a given subtree.

2.4. Network Performance Criteria

The performance of the networks considered can be studied from many viewpoints. In this chapter we consider several standard performance measures. Some of these measures depend on the assumption one makes on the locality of message traffic. The worst case assumption, taken here, is to assume a uniform message distribution. This means that all nodes are equally likely to originate messages, and the destinations of these messages are uniformly distributed.

Cost

The total cost, C, of a network is proportional to the total number of nodes P, plus the number of communication channels L. Note that in counting the number of communication channels, in most cases, it is easiest to determine the number of channels attached to each node, multiply by the number of nodes, and then divide by two.

Mean Internode Distance

The mean internode distance gives an estimate of the average communication time for messages, assuming there is no contention between the messages. The mean internode distance is the average number of channels traversed by a message as it filters through the network, assuming a uniform message distribution.

For a symmetric network, we can calculate the mean internode distance in the following manner. The probability that a message requires m channel traversals to reach its destination is:

$$\phi(m) = \frac{k(m)}{(P-1)},$$

where $k(m)$ is the number of nodes at distance m from any given node. Then the average number of channels traversed by a message, i.e., the mean internode distance μ, is:

$$\begin{aligned} \mu &= \sum_{m=1}^{m_{\max}} m\phi(m) \\ &= \sum_{m=1}^{m_{\max}} m\frac{k(m)}{(P-1)} \end{aligned}$$

where $m_{\max}$ is the maximum number of channels a message can traverse.

Throughput

The throughput of a network is the average number of messages reaching their destination per unit time, under the assumption of a uniform message distribution. This definition assumes that messages are never lost and eventually reach their destinations. Assuming infinite buffers, throughput is quite easy to calculate.

The idea is as follows. One assumes that each node sends a message to every other node. Some fraction ν of these $P(P-1)$ messages will pass through any given channel. One then computes the total time taken by this channel to forward these $\nu P\ (P-1)$ messages. Dividing this total time by $P(P-1)$ gives the average amount of time required by this channel to service any of the $P(P-1)$ messages. Note that this time may be much less than the time taken by this channel to forward a message, since many messages may avoid this channel, thus requiring zero service time. This analysis can be applied to all channels in the network.

The next step is to select one of the bottleneck channels. For most symmetric networks, all channels are topologically equivalent, and it does not matter which one is selected. For asymmetric networks, like the tree, the bottleneck channel must be more carefully selected. Now, assuming that uniformly distributed messages are being continually sent as fast as the network can handle them, and assuming infinite buffers, one can conclude that the bottleneck channel will have 100% utilization. Given 100% utilization, and given that this channel is a bottleneck, it follows that the number of messages that the network can deliver per second is the inverse of the amount of time it takes this channel to service the average message. Thus the throughput of a network can always be calculated by looking at the bottleneck channel.

For the special case of symmetric networks, the throughput can be calculated easily from μ, the mean internode distance. This analysis requires the additional assumption that all channels are topologically equivalent. The mean internode distance is the average number of channels traversed by a message. If a channel can forward messages simultaneously in both directions in time τ, the time to service the average message in the system is

$$service\_time = \frac{\mu\tau}{2L}$$

where L is the total number of channels in the system. Thus the throughput will be:

$$\theta = \frac{2L}{\mu\tau}.$$

2.5. Analysis of Networks

The five networks being considered can be studied in terms of the performance measures described above. These measures are perhaps not sufficient to indicate exactly how a network will perform in any given application, but they give a strong indication of their relative suitability.

Bus

In a bus network, each node is connected to the bus through a communication channel. Hence, there are as many channels as nodes. Using the cost measure defined earlier, the total cost C of a bus network is equal to

$$C_{BUS} = P + L = 2P.$$

This analysis does not include the cost of the bus itself. The mean internode distance

$$\mu_{BUS} = 1$$

because each node has direct access to all other nodes.

In a bus based system, the bus itself is the bottleneck, because only one node at a time can get access to the bus. The throughput of a bus can be calculated by viewing the bus as a critical resource. Since only one message can be transmitted on the bus at any time, the throughput must be:

$$\theta_{BUS} = \frac{1}{\tau}.$$

Note that the throughput of the bus is independent of the number of nodes on it.

Complete Connection

Since each node in a complete connection network is attached to $P-1$ channels, the total number of channels in the network is:

$$L_{COMPLETE} = \frac{P(P-1)}{2}.$$

Hence the total cost of a complete connection is:

$$C_{COMPLETE} = P + \frac{P(P-1)}{2} = \frac{P^2+P}{2}.$$

This quadratic cost in the number of nodes is the principal reason that a complete connection network is considered impractical. The mean internode distance on a complete connection network is one, since each node is connected to all other nodes. The throughput on a complete connection network is easily calculated from the formula for symmetric networks given earlier:

$$\theta_{COMPLETE} = \frac{P(P-1)}{\tau}.$$

Hypercube

In a hypercube network with P nodes, each node is connected to $\log P$ channels. Therefore,

$$L_{HYPERCUBE} = \frac{P \log P}{2}.$$

Thus the total cost, in this case, is:

$$C_{HYPERCUBE} = P + \frac{P \log P}{2} \quad = P\left(\frac{\log P}{2} + 1\right).$$

The mean internode distance is calculated as follows. The number of links a message travels in a hypercube is equal to the number of bit positions that are different in the source and destination address pair. Messages are all uniformly distributed, thus the probability that a particular bit in the source and destination address is different, is half. Since the number of bits is $\log P$, the average number of bits different in the source and destination address pair is $\frac{\log P}{2}$. This analysis includes messages sent to itself by a node. Since our model excludes those messages, a correction factor of $\frac{P}{(P-1)}$ has to be included. Thus the mean internode distance is

$$\mu_{HYPERCUBE} = \frac{\log P}{2}\left(\frac{P}{P-1}\right) \quad = \frac{P \log P}{2(P-1)}$$

which asymptotically simplifies to:

$$\mu_{HYPERCUBE} \approx \frac{\log P}{2}.$$

The throughput of a hypercube is

$$\theta_{HYPERCUBE} = \frac{2\left(\frac{P\log P}{2}\right)}{\tau\left(\frac{P\log P}{2(P-1)}\right)}.$$

Simplifying,

$$\theta_{HYPERCUBE} = \frac{2(P-1)}{\tau}.$$

Ring

The total cost of a ring is

$$C_{RING} = 2P,$$

since there are P nodes and P channels. The mean internode distance is

$$\mu_{RING} = \frac{P^2}{4(P-1)}$$

for a bidirectional ring with an even number of nodes. This is approximately $\frac{P}{4}$ for large P, indicative of the fact that the average message needs to travel only one-fourth of the way around the ring.

Using the formula for symmetric networks, we can calculate the throughput of a ring as follows:

$$\theta_{RING} = \frac{2P}{\left(\frac{P^2}{4(P-1)}\right)\tau} = \frac{8(P-1)}{P\tau},$$

which is asymptotically

$$\theta_{RING} \approx \frac{8}{\tau}.$$

Tree

In a tree with P nodes there are $P-1$ channels. One calculates the number of channels by thinking of each node, except the root node, as "owning" the channel between this node to its parent. Since there are P-1 channels, the total cost of a tree is

$$C_{TREE} = 2P - 1.$$

Unlike the other networks considered here, calculating the mean internode distance of a tree network is quite complex. A careful analysis is given in the appendix of Reed's thesis[21]. The main result is that the mean internode distance is given by:

$$\frac{2\,k\;2^{k-1}(2^k+1)}{(2^k-1)(2^{k-1}-1)} - \frac{6 2^{k-1}}{(2^{k-1}-1}$$

where $P = 2^k-1$. This expression is complex, but is clearly asymptotic to:

$$\approx 2\,k - 6 = 2\,\log(P+1) - 6\,.$$

On a tree network, the channels attached to the root node are the bottleneck channels[7,25]. Suppose each node sends one message to every other node. To compute the throughput one calculates the percentage of these $P(P-1)$ messages which will pass through either of the channels attached to the root. Let $q = \frac{P-1}{2}$ be the number of nodes in each of the subtrees of the root. Then there are $2q^2$ messages passing through the root from between the nodes of these subtrees. There are also $2q$ messages passing between either subtree and the root. Thus there are

$$2q^2 + 2q$$

of the $P(P-1)$ messages passing through the bottleneck channel, which becomes:

$$\frac{(P-1)\,(P+1)}{2}.$$

Thus the percent of messages passing through the bottleneck is:

$$v = \frac{P+1}{2P}.$$

It follows that the throughput of this network is:

$$\theta_{TREE} = \frac{1}{v\tau}$$

$$= \frac{2P}{\tau(P+1)}.$$

As one can easily see, the throughput of a tree is approximately constant, independent of the number of nodes:

$$\theta_{TREE} \approx \frac{2}{\tau}.$$

2.6. COMPARISON OF NETWORKS

The performance measures computed for the five networks are summarized in Table 2.1. In several cases lower order terms have been neglected and we give only the asymptotic results.

Viewing throughput as the most critical performance measure, the networks here fall into two classes. The complete connection and hypercube have throughput growing at least linearly in P. On the other hand, the bus, ring and tree have throughput which is asymptotically constant. Thus, for large number of processors, the latter three networks are probably inappropriate for message intensive problems.

The higher performance of the complete and hypercube networks is reflected in their higher cost. The quadratic cost of the complete connection may make it impractical in many cases. However, high performance and $PlogP$ cost of the hypercube shows the great attractiveness of this network, and related networks, such as the shuffle[28] and cube-connected cycles[20] not considered here.

Networks	*Number of Channels* L	*Cost* C	*Mean Internode Distance* μ	*Throughput* θ
Bus	P	$2P$	1	$\frac{1}{\tau}$
Complete	$\frac{P(P-1)}{2}$	$\frac{P^2+P}{2}$	1	$\frac{P(P-1)}{\tau}$
Hypercube	$\frac{P\log P}{2}$	$P\left(\frac{\log P}{2}+1\right)$	$\approx\frac{\log P}{2}$	$\frac{2(P-1)}{\tau}$
Ring	P	$2P$	$\approx\frac{P}{4}$	$\approx\frac{8}{\tau}$
Tree	$P-1$	$2P-1$	$\approx 2\log(P+1)-6$	$\approx\frac{2}{\tau}$

Table 2.1 Network Parameters

There may be problems where the throughput is not the critical parameter and instead the message delays are important. If the network is lightly loaded and thus almost contention free, message delays will be proportional to mean internode distance. Viewed this way, the ring is by far the worst and the bus and the complete connection are somewhat better than the hypercube or tree. It remains to be seen which of these factors plays an important role in our simulation experiments.

CHAPTER 3

TREE-SEARCHING ALGORITHMS

Tree searching problems occur in a variety of application areas. In this chapter, we describe the problems chosen for study here. We also characterize the search trees generated by these problems on the basis of several parameters such as tree breadth and depth.

3.1. ALGORITHMS STUDIED

For the purpose of this study, we have chosen four tree-searching problems. These problems are: eight queens, knight's tour, traveling salesman and adaptive quadrature. The eight queens and knight's tour algorithms are examples of combinatorial problems where one searches for feasible solutions. The traveling salesman problem is a combinatorial optimization problem, where feasible solutions are apparent and one searches for an optimal solution[31]. Finally, the adaptive quadrature algorithm here, is a simple example of numerical and graphics algorithms based on spatial divide and conquer techniques[11]. The next four sections provide a high level view of these algorithms.

Eight Queens

The eight queens problem is the well known problem of putting eight queens on an eight by eight chess board in such a way that no queen checks another. Using the fact that two queens cannot be placed in the same column, we can represent the positions of the queens on the board as an array, *queens*, where *queens*[*i*] gives the

```
program eight_queens;
begin
   for column = 1 to 8 do
         queens[column] := 0;
   board_gen (queens, 1);
end;

procedure board_gen (queens: array[1..8] of integer; column: integer);
begin
   for queens[column] = 1 to 8 do
   begin
      if test(queens) then
         if column < 8 then
               board_gen (queens, column+1)
           else
               printboard (queens);
   end;
end;
```

Listing 3.1 Eight Queens

position of the queen in the i-th column. Given this representation, Listing 3.1 presents a recursive algorithm which computes all feasible solutions to this problem. Here *test* is a procedure which tests whether any of the queens on the board checks another. If none do, the algorithm proceeds recursively to place queens on the next column. When queens have been successfully placed on all eight columns, one of the solutions to this problem has been found, and this answer is printed. Eventually all solutions will be found.

Knight's Tour

The knight's tour problem is another chess problem, but one that is considerably harder than the eight queens problem. Given a chess board, of size m by n, and a fixed

```
program knights_tour;

var path  = array[1.m*n] of integer;
    board = array[1..m,1..n] of (visited, not_visited);

begin
     initialize(board, path);       /* initialize array board to not_visited
                                                  and path to 0 */

     row := 1;    col := 1;
     board[row, col] := visited;  /* put knight on square 1,1 */

     move_num := 1;
     tree_search(move_num, row, col, path, board);
end.

procedure tree_search (move_num, old_row, old_col: integer;
                       path: array[1..m*n] of integer;
                       board: array[1..m,1..n] of (visited, not_visited));

begin
     for direction := 1..8 do
     begin
         col := old_col + col_change(direction);
         row := old_row + row_change(direction);

         if valid_move(board, row, col) then
              begin
                  board[row, col] := visited;
                  path[move_num] := direction;
                  if (move_num = m*n) then
                         print(path);
                  else
                         tree_search (move_num+1, row, col, path, board);
              end;

         board(row, col) := not_visited;
     end;
end;
```

Listing 3.2 Knight's Tour

starting point for the knight, the problem is to find all paths on the board in which the knight visits each square exactly once. A depth first search algorithm for this problem is given in Listing 3.2. The array *board*, in the algorithm, indicates the positions on the chess board already *visited*. Array *path* gives the sequence of directions the knight moves in performing the tour. Kernel routines *col_change* and *row_change*, give the position change for each of the knight's eight possible moves. Routine *valid_move* checks whether the new position will be on the chess board, and if so whether the position has already been visited.

Traveling Salesman

The traveling salesman problem belongs to a very important class of problems called NP-complete. Various other problems like warehouse location, job-shop scheduling, graph partitioning, dynamic storage allocation and register allocation in program optimization belong to the same complexity class. In the traveling-salesman problem, a salesman desires a minimum mileage trip which visits each of n cities exactly once, and returns to the starting city[19]. The array *adjacency_matrix*, in the algorithm presented in Listing 3.3, gives the mileage between the cities. If two cities are not connected, the mileage between them is set to -1. The array *cities* indicates the cities already *visited*, while the array *path* gives the sequence in which the cities are visited. Procedure *find_minimum* determines whether the current tour is shorter than the shortest previous tour, and copies it into array *minimum_path* if so.

```
program travel_salesman;

var adjacency_matrix    : array[1..n, 1..n] of integer;
    path, minimum_path : array[0..n] of integer;
    cities              : array[1..n] of (visited, not_visited);

begin
    initialize (adjacency_matrix, path, cities);

    path[0] := 1;
    cities[1] := visited;
      move_num := 1;

    path_find (move_num, path, cities);
      print(minimum_path);

end;

procedure path_find (move_num : integer;
                          path      : array[0..n] of integer;
                            cities   : array[1..n] of (visited, not_visited));
begin
      last_city := path[mov_num - 1]

      for next_city = 1 to n do
      begin
          if ( adjacency_matrix[last_city, next_city] <> -1)
               and ( cities[next_city] = not_visited) ) then
          begin
               path[mov_num]      := next_city;
               cities[next_city] := visited;
               if (move_num = n) then
                   find_minimum(path);
               else
                   path_find (move_num + 1, path, cities);
          end;

          path[move_num]     := 0;
          cities[next_city] := not_visited;
      end;
end;
```

Listing 3.3 Traveling Salesman

Adaptive Quadrature

There are many important mathematical functions for which no analytical formula for the integral is available. The standard numerical technique for computing the integral of such functions is adaptive quadrature. We consider here the problem of numerically integrating a real valued function of two variables over a rectangular region. This can be done by dividing the region into cells and then estimating the value of the integral over each of the cells using one of the standard numerical quadrature formulas[11]. The total value of the integral in the region is then the sum of the values of the integrals over the constituent cells.

While the integral value over each cell is being calculated the error in the approximation can also be estimated. If the error in a cell is greater than a specified tolerance level, the cell can be further subdivided and the integral and error over the new cells evaluated. This subdivision of cells can be viewed as the construction of a tree whose nodes represent the quadrature cells. Refinement of these cells is continued until the error in each of the subcells meets the specified tolerance level. Thus smaller cells will be created for areas where the function is changing rapidly and the quadrature formulas are less effective. A recursive adaptive quadrature algorithm is given in Listing 3.4.

In this algorithm, function *quad_formula* computes the integral on a cell. Function *error_estimate* returns an estimate of the error in the approximation used in *quad_formula*. If this estimated error is excessive, the cell is subdivided into four subcells, and the function *adapt* is called recursively for each of these subcells.

```
program quadrature;

begin
   x0 := 0.0;  x1 := 1.0;
   y0 := 0.0;  y1 := 1.0;      /* region for integration */

   tolerance := 1.0e-2;
   value := adapt(f, x0,x1,y0,y1);
end;

function adapt(function funct: real; x0,x1,y0,y1: real) : real;
begin
   error := error_estimate (funct, x0,x1,y0,y1);

   if (error < (x1-x0) * (y1-y0) * tolerance) then
        adapt := quad_formula(funct, x0,x1,y0,y1)
   else
     begin
        xmid := 0.5*(x0 + x1);
        ymid := 0.5*(y0 + y1);

        sum1 := adapt(funct, x0,  xmid,y0,  ymid);
        sum2 := adapt(funct, x0,  xmid,ymid,y1);
        sum3 := adapt(funct, xmid,x1,  y0,  ymid);
        sum4 := adapt(funct, xmid,x1,  ymid,y1);

        adapt := sum1 + sum2 + sum3 + sum4;
     end;
end;
```

Listing 3.4 Adaptive Quadrature

3.2. CLASSIFICATION OF TREES

For our simulation experiments, we generated five different search trees. In this section we will make an attempt to classify these trees based on several parameters. Each of these parameters is a computable measure of one property of the tree. The parameters are: depth, maximum breadth, average breadth, average branching factor, balance factor and size.

Depth

The depth of a tree is the maximum number of levels in the tree. Given infinite parallelism, the execution time of an algorithm may be expected to be proportional to the the depth of the tree. In an ideal situation, if there is a one to one mapping of the tree on the architecture, and all the parallelism is being extracted, one can expect the limiting factor to be wholly dependent on the depth of the tree.

Maximum Breadth

The maximum breadth of a tree is the maximum number of nodes at any level of the tree. Maximum breadth of a tree gives the maximum parallelism inherent in the problem. Hence, the maximum breadth provides an upper bound on the number of processors which can be used for a problem.

Average Breadth

The average breadth of a tree can be defined as the average number of nodes in a level. It is therefore,

$$\begin{aligned} average\_breadth &= \frac{\sum_{levels} number\ of\ nodes\ in\ level}{number\ of\ levels} \\ &= \frac{total\ number\ of\ nodes}{depth\ of\ tree} \end{aligned}$$

Average breadth of the tree gives a measure of the average parallelism in the problem. This factor largely determines the optimum number of processors needed for a problem.

Average Branching Factor

The average branching factor of a tree is the average number of children of all non-leaf nodes. The average branching factor turns out to be:

$$\frac{\textit{total number of nodes} - 1}{\textit{number of non-leaf nodes}}$$

There is no obvious relation between the branching factor and the ability to exploit parallelism in tree searching.

Balance Factor

Balance of a tree is a more difficult concept to define. Our measure of the balance of a tree is determined by computing a balance parameter at each node of the tree, and then combining these to generate a single parameter. The balance of a node is defined as the difference between the number of nodes in its largest subtree and the number of nodes in its smallest subtree. The balance factor is then defined as:

$$\textit{balance\_factor} = \frac{\sum_{\textit{nodes}} \textit{balance\_of\_nodes}}{\textit{total number of nodes}}$$

The balance of a tree is important, since the more unbalanced the tree, the more difficult it will to be to distribute the load effectively across a parallel architecture. The balance factor parameter defined here will be large for unbalanced trees, and zero for a perfectly balanced tree.

The above six parameters, can be used to characterize the search trees generated by the algorithms under study. We looked at five different search trees generated by four algorithms. We solve the knight's tour problem for a 4 by 4 chess board and also the eight queens problem on an 8 by 8 chess board. A traveling salesman tour for a dense graph of 7 cities was computed. Finally, two different functions were integrated by the adaptive quadrature algorithm, since the type of tree produced by this algorithm was strongly dependent upon the function integrated. The two functions used are:

$$f_1 = (x^2 + y^2)^{\frac{1}{8}}$$

$$f_2 = abs\sqrt{(x^2 + y^2)} - 0.6$$

The parameters characterizing these five search trees are given in Table 3.1. In each of these problems, there are free parameters, such as the size of the chess board used for the knight's tour or the tolerance in adaptive quadrature. These free parameters were selected so that all trees here would be approximately the same size. Thus differences in multi-processor performance on these different tree search problems, must be due to other factors, such as breadth or balance, rather than the size of the tree.

Algorithms	*Depth*	*Maximum Breadth*	*Average Breadth*	*Average Branching*	*Balance Factor*	*Size*
Queens	*8*	*568*	*228.56*	*1.56*	*0.52*	*2057*
Knights	*14*	*448*	*148.2*	*1.52*	*0.49*	*2223*
TSP	*6*	*924*	*272.71*	*1.95*	*0.35*	*1944*
Quad_1	*57*	*256*	*35.95*	*4.0*	*69.17*	*2058*
Quad_2	*11*	*512*	*176.08*	*4.0*	*17.84*	*2113*

Table 3.1 Characteristics of Trees

As seen here, most of the trees have depth about 10. The exception is one of the quadrature problems, Quad_1, which has a depth of 57, since it is performing deep refinement at a point singularity. For the same reason, the average breadth of this problem is almost an order of magnitude lower than the rest of the problems and the maximum breadth is also smaller.

It is interesting to note that all the combinatorial problems give rise to branching factors below two. This is due to the fact that most of the nodes in the tree are near the leaves, where few combinatorial choices remain.

The most striking difference between the quadrature problems and the combinatorial problems is in the balance factor. The quadrature trees are highly unbalanced because cells in regions where the functions are singular, will become highly refined while the rest will not be. The combinatorial problems, where we have to make a sequence of choices, seem to have a completely different character. It appears that each of the early choices are almost equally likely to lead to solutions. For example, in the traveling salesman problem (TSP), no matter which cities are visited first, we will probably be able to complete a tour. Thus most of the nodes higher up in the tree, tend to be well balanced.

In this chapter we have presented four different problems and characterized their search trees. The search trees produced will be used in the simulation studies described in Chapter 5 and the parameters characterizing these trees will be related to the performance results obtained.

CHAPTER 4

LOAD DISTRIBUTION STRATEGIES

In the last chapter we discussed a number of tree-search algorithms and the types of trees they generate. The goal in this thesis is to effectively map these algorithms onto non-shared memory architectures based on the networks discussed in Chapter 2. The non-shared memory architectures being considered here do not have a central controller, and also there is no a priori knowledge of the structure of the search trees being generated. Thus, the principal issue in mapping these problems to such architectures is the dynamic balancing of the workload.

This chapter discusses a family of load distribution strategies. We begin by presenting the architectural assumptions made for our simulation experiments. After that, four load distribution strategies which were developed during the course of this study, are described. These range from relatively simple strategies, used to give a baseline for comparisons, to complex strategies, which seem to achieve high utilization even on problems where load balancing is quite difficult. Finally, we discuss the way in which the strategies here relate to the general problem of load distribution on non-shared memory architectures.

4.1. ARCHITECTURE MODEL

As noted before, we are assuming non-shared memory architectures, wherein the processors operate in an asynchronous MIMD mode. The run-time environment used is based on typical low-level message passing primitives. We look first at the processor model employed, and then discuss the run time environment.

Processor Model

The parallel systems that are to be studied in our research consist of a set of processor nodes, interconnected by a packet switched network. Each of these processor nodes or processing elements (PE) is a simple microprocessor along with its local memory.

In order to allow computation and communication to proceed in parallel, each PE has a co-processor dedicated to input-output, as shown in Figure 4.1. In the absence of this co-processor, the processor would have to be interrupted every time a message had to be processed. This issue is particularly important with packet-switched networks,

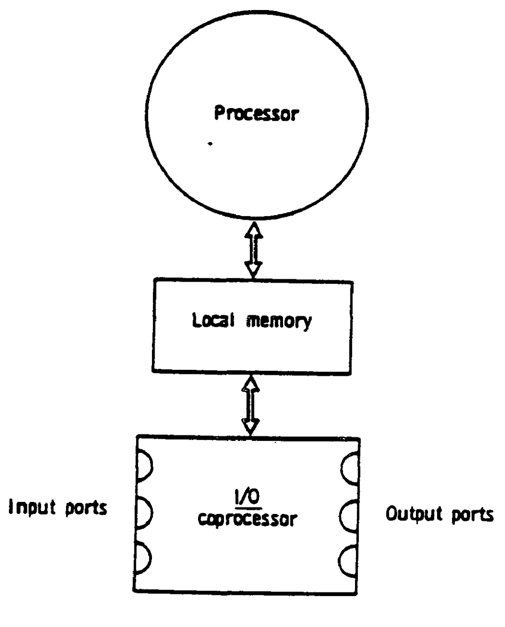


Figure 4.1 Model of a Processing Element

because of the large number of intermediate stores and forwards. The co-processor is a micro-processor of simpler design than the processor, since its function is more trivial. The local memory is dual-ported so that the co-processor can directly access it.

When a processor needs to send a message to another processor, it places the message in a buffer region of memory, which the co-processor services when it has time. Similarly, when a message is received from the network, the co-processor places the message in the appropriate buffer region of memory and the processor fetches it when required. Co-processor also maintains buffers in memory where they keep messages being stored and forwarded in the packet-switched network. Since all buffers here are regions in the local memory, whose sizes vary dynamically, they can be potentially quite large. This is in contrast to other non-shared memory architectures where small hardware buffers are employed. With the hardware assumption here, the buffers are for practical purposes infinite, just as we assumed in Chapter 2.

We assume a simple run-time environment in which the processors are not multiprogrammed. That is, a single process runs on each processor. One process is started up on each processor at the beginning of execution, and execution continues until all processors become idle. In situations such as ours, the children of any node can be processed by invoking new processes for them, or by using recursive subroutine calls. The context-switching overhead associated with implementing processes on a single processor is generally higher than the overhead associated with a subroutine call. Thus, if the granularity of parallelism is low, for example, if the amount of processing needed at each node is very small, the process overhead would become significant. Hence we have chosen not to allow multiprogramming on the processors.

Communication Primitives

The processors communicate with each other via message passing. Each message begins with a header containing, an origin processor number, a destination processor number, and time stamp. The body of the message is a fixed length data field.

There are three simple message passing primitives, a "send" construct, and two kinds of "receive" constructs. The *send* primitive can be utilized to send a message to any other processor. This *send* is non-blocking, that is, after executing a send, a processor does not have to wait for the message to be received, but can continue executing immediately.

The run-time environment provides two means of receiving a message, a non-blocking receive and a blocking receive. When a process executes a non-blocking receive, it receives a message if it is present in the buffer of input messages. If the input buffer is empty, the processor is not blocked but continues execution. On the other hand, if a processor executes a blocking receive, an empty buffer forces the processor to wait for the next message to arrive. Execution of a receive returns the earliest message that arrived. No receive primitive that selectively receives messages from a specific source processor has been provided here.

4.2. STRATEGIES

In this section we describe the load distribution strategies developed for mapping tree searching algorithms onto non-shared memory architectures. The first two strategies here are relatively obvious and are studied to provide a baseline. We also explored two other strategies which are more sophisticated. The latter two strategies were tuned to give high utilization over a range of problems.

In presenting these strategies the issue of program termination has been ignored. The process executing on each processor is described as an infinite loop. Execution actually terminates at the point where all processors and input-output co-processors become idle.

A related issue is the reporting of results at the end of the computation. The computation is begun by processing the root node of the search tree on processor 0, and one would ordinarily like the results reported back to this processor. This can be done as a separate step at the end of the computation, but requires only $logP$ steps, and has been ignored here.

Tree Strategy

The simplest approach to tree-search problems is to map the search trees generated directly onto a tree architecture. One way of doing this is to let the root processor in a binary tree architecture process the root node of the search tree. Half of the children of the root node are sent to the left child of the root processor and the other half is sent to the right. Each of the processors below the root similarly send half the children spawned to their left child and the other half to their right child. This process continues until the leaf processors are reached. At that point, the leaf nodes carry out a simple depth first search of the nodes that they have.

There are two main difficulties with this approach. For large search trees, the leaves of the tree architecture end up doing all the work. Since in a binary tree architecture, half of the total number of processors are leaf processors, the utilization with this strategy, can be expected to be only close to 50%. The other drawback of this strategy is that there is no load balancing. Thus, on unbalanced trees, the utilization can be expected to drop even further, as there will be some overworked leaf processors, expanding large subtrees, while other leaf nodes remain idle. It remains to be seen, however, how effective other sophisticated strategies are, in dealing with the case of different trees.

Round Robin

The round robin strategy is a simple strategy, using a large amount of communication. In this strategy, each processor, when processing a node, spawns all its children, and sends them to other processors in a round robin fashion. A high level view of the algorithm is presented in Listing 4.1.

As is evident from the algorithm this is a very message intensive strategy. Each node in the tree leads to a message, and hence the total number of messages in the network, under this strategy, is only one less than the total number of nodes in the tree. No heuristics are used for distributing the workload among all the processors and the idea of exploiting neighborhood connections is not used either. Due to the long distance packet-switching communication used, this strategy may be able to distribute workloads very uniformly among the processors. However, the important research issue is whether adequate load balancing can be achieved with less message intensive strategies.

Askwork

The next strategy considered, *Askwork*, is one of the strategies we have developed, which makes use of heuristics to obtain better load balancing. In *Askwork*, work is distributed on the basis of work request messages sent to neighboring processors when a

```
begin   -- robin strategy
   prime := 87;
                  -- a prime number is chosen so as to avoid clustering
               -- of messages near the low numbered processors.

   if ( processor number = 0 ) then
        next_processor := prime mod number_of_processors
        foreach child of node 0 do
            send ( next_processor, child)
            next_processor := ( next_processor + 1 ) mod number_of_processors
        end_foreach
   end_if

   loop  -- all processors execute this code.

        work_message := block_receive ()
        next_processor := (processor_number * prime) mod number_of_processors

        foreach child of work_message.node do
            send ( next_processor, child)
            next_processor := ( next_processor + 1 ) mod number_of_processors
        end_foreach

   end_loop

end -- robin
```

Listing 4.1 Robin Strategy

processor becomes idle. Thus *Askwork* does not employ long distance communication, but uses communication between processors directly connected in the network.

Initially, processor 0 distributes all the children of the root node of the search tree in a round robin fashion. Each processor maintains a list of unexpanded nodes, *work_list*. It successively performs a depth first search on the subtrees rooted at each node in this list. During this process, if a work request message arrives from a neighboring processor, an unexpanded node from the end of the *work_list* is sent. The

```
begin  -- askwork strategy

   if ( processor number = 0 ) then -- do a round robin distribution of children of root node
        next_processor := 1
        foreach child of node 0 do
            send ( next_processor, child)
            next_processor := ( next_processor + 1 ) mod number_of_processors
        end_foreach
   end_if

   loop
        while  not_empty (work_list)  do
            while ( work_request_message ) and ( length( work_list ) > 2 ) then
                          -- If there is a pending work_request message and
                          -- number of nodes in the work list is greater than 2,
                          -- honor the work request message by sharing work.
                 node := extract_from_end ( work_list)
                 send( requesting_processor, node )
            end_while
                          -- Process a node
            node := extract_from_top ( work_list )
            foreach child of node do
                 push_on_top ( work_list, child )
            end_foreach
        end_while
                          -- Work list is empty at this point, process work messages if any,
                          -- otherwise request work from neighbors
        work_message := non_block_receive ();
        if ( work_message <> null) then
            push_on_top ( work_list, work_message.node )
        else
            foreach neighbor heard from since last request message sent
                 send ( neighbor, work_request_message)
            end_foreach
            work_message := block_receive ()
            push_on_top ( work_list, work_message.node )
        end_if

   end_loop

end   -- askwork
```

Listing 4.2 Askwork Strategy

processor honors work request messages only if its *work_list* has more work than a predefined cut-off value. When a processor becomes idle, it sends out work request messages only to those neighbors who have sent work since the last round of work request messages were sent. A high level view of this algorithm is presented in Listing 4.2.

There are three ways in which the *Askwork* strategy tries to minimize the message traffic. First, work is always sent from the bottom of the *work_list*, so that as much work as possible is transferred in one send. Secondly, an experimentally chosen constant serves as a cut-off point, for honoring work request messages. If the workload falls below this value, the processor simply expands its own nodes, ignoring work request messages sent by its neighbors. Finally, work request messages are sent out sparingly. If a processor has not heard from one of its neighbors since the last time request messages were sent out then there is no need to send it another work request message.

Knowledge

The last strategy developed here, *Knowledge*, uses heuristics as well as "knowledge" of the workload of neighboring processors, to achieve load balancing. Like *Askwork*, the *Knowledge* strategy exploits neighborhood connections during communication, but it does not use the concept of work request messages. Instead, if a processor "believes" that it has a lot more work than one of its neighbors, it sends work to that neighbor. For this purpose each processor maintains a table, *neighbor_work_estimates*, that reflects the state of the workload on all of its neighbors. The current workload of the processor is transmitted along with each work message sent to a neighbor. The receiving processor

```
begin -- knowledge strategy
    initialize ( neighbor_work_estimates )
    if ( processor number = 0 ) then      -- do a round robin initial distribution
        next_processor := 1
        foreach child of node 0 do
            send ( next_processor, child)
            next_processor := ( next_processor + 1 ) mod number_of_processors
        end_foreach
    end_if

    loop
        while  not_empty (work_list)  do
            while ( length(work_list) > 1.5 * minimum(neighbor_work_estimates) )
                            and ( length(work_list) > 2 ) do
                node := extract_from_end ( work_list)
                send( processor_with_minimum_workload, node )
                update (neighbor_work_estimates)
            end_while
                -- Process a node
            node := extract_from_top ( work_list )
            foreach child of node do
                push_on_top ( work_list, child )
            end_foreach
        end_while         -- Work list is empty at this point

        work_message := non_block_receive ()
                -- if there is a work message, receive it, else wait for one to arrive

        while  ( work_message <> null) do
            push_on_top ( work_list, work_message.node )
            update( neighbor_work_estimates )
            work_message := non_block_receive ()
        end_while

        if ( empty (work_list) then
            work_message := block_receive ()              -- wait for a work message
            push_on_top ( work_list, work_message.node )
            update( neighbor_work_estimates )
        end_if
    end_loop
end -- knowledge
```

Listing 4.3 Knowledge Strategy

updates its *neighbor_work_estimates* table based on this information.

As in the strategy *Askwork*, after an initial round robin distribution, each processor maintains a list, *work_list* of unexpanded nodes. During the process of performing a depth-first search each processor tries to keep track of the neighbor with the minimum workload. If the minimum is less than some predefined factor of the processor's current workload, the processor sends out a node from the end of the *work_list* to this neighbor. It then updates its own *neighbor_work_estimates* table by incrementing the workload of its neighbor.

When a processor becomes idle, it services any pending work messages. If there are none, it is blocked until work arrives. A high level view of the strategy is presented in Listing 4.3.

In this strategy, knowledge about the workload of the neighbors is maintained, in an attempt to keep the load distribution balanced. Since communication about workloads is piggy-backed on the work messages being sent, this strategy may lead to balanced workloads without the communication intensity of *Askwork*.

4.3. COMPARISON OF STRATEGIES

In this chapter we have presented four different strategies for mapping tree-searching algorithms onto non-shared memory architectures. The first of these, the *Tree* strategy, is an obvious strategy for tree connected architectures. The other three strategies have been designed without any assumptions about either the underlying network or the topology of the search tree being mapped.

Of the other three strategies, *Robin*, was chosen as a baseline even though it may be quite message intensive. One of the issues here is how well the simple round robin distribution strategy performs relative to more subtle strategies.

The last two strategies being proposed here, are attempts to exploit our knowledge of multiprocessor environments to obtain even load distributions with lower message traffic. In the first of these, *Askwork*, processors distribute work only to idle neighbors requesting work. The final strategy, *Knowledge*, is similar in that both strategies use only local communication. This strategy differs from *Askwork* in that processors maintain knowledge tables on which decisions on whom to send work to are based. Because this approach uses workload information piggy backed on work messages, instead of separate work request messages, it may turn out to have lower message traffic than *Askwork*. The issue is whether it will be as effective at balancing workload.

Strategies similar to *Knowledge* have been considered by Gray et al[10] and by Burton and Huntbach[6]. Dissimilarities between our approach and theirs exist both because their architectural assumptions are significantly different, and also because they use different mechanisms for obtaining estimates of the neighbor's workload. Gray et al utilize a polling mechanism wherein each processor polls its neighbors to check their status. If any of the neighbors is idle, and the processor has enough work it transmits a part of its workload to the neighbor.

Burton and Huntbach[6] maintain workload estimates of their neighbors as in our system. The processors send work to their neighbor if the destination processor has less work. The workload information, as in our strategy, is piggy-backed onto the work message. This information is sent to the neighboring processors, in each communication cycle. Thus if in a communication cycle, if a processor does not need to send work to its neighbor, it sends a dummy message containing just its workload. This is in contrast to our strategy, wherein messages are sent only if work has to be transmitted.

CHAPTER 5

SIMULATION EXPERIMENTS

5.1. Introduction

The objective of the research here is to study the performance of tree searching on non-shared memory architectures. There are several ways of approaching this objective. The principal alternatives are the use of theoretical tools such as queueing theory, the use of existing parallel architectures, and the use of simulation techniques.

The approach followed here is simulation, since simulation offers strong advantages over the other two approaches. Queueing theory is an elegant approach, but cannot deal with algorithms of the complexity of the load distribution strategies proposed here. Also, one of the issues of interest here is the impact of search tree topology on execution speed. It would have been impossible to accurately assess this impact via queueing theory methods.

A second approach is the use of working parallel architectures. We had access to the eight processor NASA Finite Element Machine[12], and could have obtained remote access to a Denelcor HEP[27]. There is a clear advantage to this approach; by using working parallel architectures one necessarily takes into account all relevant details. On the other hand, this approach is highly inflexible and makes it difficult to vary architectural parameters, such as the number of processors, interconnection topology, and the relative speed of communication.

The great advantage of simulation is flexibility, since it allows one to vary architectural parameters simply by changing a parameter in the simulator. Thus it is possible to rapidly explore a wide range of architectural and algorithmic options. With simulation, there is always the issue of the level of detail required, and whether the simulation results accurately reflect performance realities. However, with sufficient care, simulation is one of the most useful approaches to parallel computing research.

5.2. Simulator Design

During our research two different simulators were employed. The first of these was an instruction level multiprocessor simulator written by D. Gannon and students at Purdue University. With this simulator, the user writes programs in a parallel dialect of C, which are then compiled into assembly code to execute on a hypothetical architecture consisting of networked microprocessors. The simulator then interprets the code executing on each microprocessor.

With a simulator of this level of detail, the simulated execution time of a program will be virtually identical to its execution on corresponding hardware. On the other hand, with this level of simulation the computation time required can be quite large. For our problems, simulating the execution of tree search problems having search trees with two thousand nodes required twelve CPU hours on a VAX 11/750. Such large execution times would have precluded the exploration of more than a small number of architectural and algorithmic options.

In order to accomplish our goal of studying the interaction between tree search algorithms, architectures, and load balancing strategies, we switched to a higher level and less detailed simulator. There are several high level architecture simulators available. We used one written by J. Van Rosendale and P. Mehrotra in connection with their research on parallel languages and parallel run-time environments.

As with the simulator written by D. Gannon, this simulator executes a dialect of C, but is not an instruction level simulator, and is in fact much closer in concept to the Simon simulator, Fujimoto[8]. This simulator was specifically designed to execute very rapidly, thus permitting the study of MIMD architectures having thousands of processors. This is in sharp contrast to the situation with instruction level simulators, where simulation of large programs and complex architectures is prohibitively expensive. Since our goal was to study a large number of choices of algorithms and architectures, this high level simulator was ideal for our purposes.

This simulator executes a CSP-like dialect of C, which is compiled by the standard Unix C compiler, after a special extra pass through the C preprocessor. Our basic scheme for using this simulator is shown in Figure 5.1. The process simulator and network simulator are separate modules, which execute concurrently as coroutines. Thus, one can change networks and processor modules independently. The processors are simulated using standard event queue simulation techniques, while the packet-switched communication networks, which operate synchronously, are simulated without the use of event queues.

Unlike the related Simon simulator, which times VAX instructions, the various system clocks here are incremented by special instructions the user inserts in his code. Thus one can arbitrarily set the time of each operation of interest, thus controlling the architectural assumptions. In our work, we allocate time to process each node in the search tree, to spawn each child of a node, and to send, receive, and forward messages.

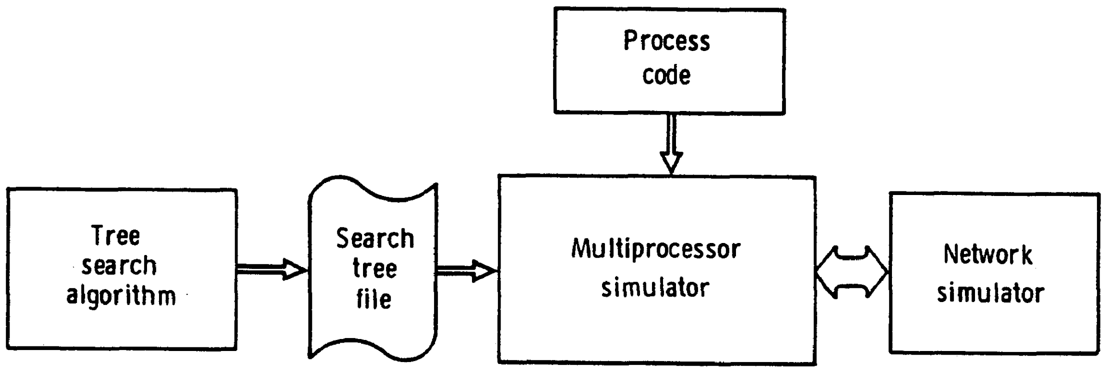


Figure 5.1 Organization of the Simulator

The processors in this simulator execute asynchronously, and each has its own associated clock. All operations we perform, except message routing and forwarding, consume time on these processor clocks. Routing and forwarding messages is the responsibility of the co-processors, which operate synchronously under the global network clock. Time on these co-processors is consumed by sending and receiving messages, and also by forwarding messages.

Following the assumptions in Chapter 2, we make the assumption here that the co-processor can transmit a message on each communication channel in one clock cycle, requiring time τ. Here τ is a relative time, meaningful only in relation to the time the processors require to process or spawn a node, assumed here to take unit time.

Under these assumptions, a message requires between four and six τ units to go from one processor to a neighboring processor, assuming no message contention. The lower bound of 4 arises as follows. It takes one clock cycle for a processor to place the message in the buffer region of its co-processor. The co-processor requires one clock cycle to take the message from the buffer and route it to the input port of the destination PE. The co-processor of the destination processor requires another clock cycle to transfer the message from the input port to the buffer region of its processor. The destination processor finally requires another cycle to "receive" the message. Hence a total time of at least four τ is required. This is the minimum possible. Messages may take up to 6 cycles, since they may have to wait part of a network clock cycle as they pass between the processors, which run under asynchronous clocks, to the co-processors, which run under the global network clock.

To utilize the simulator, the user writes a process in a C dialect, which consists of the C language with extensions for communication. The compiled version of this process executes on each processor in the simulated system. For each of the search problems to be simulated, a separate sequential program was run to generate the underlying search tree, as shown in Figure 5.1. This approach allowed the load distribution strategy to be coded independently of the search problem.

Besides the search problem and the load distribution strategy, in each run of the simulator the following factors were set: the interconnection network, the number of processors P, and the relative network speed, ρ. Here ρ is the inverse of τ, the time per network clock cycle. The simulator generates several performance statistics, the most important of which is the average utilization or the speed-up achieved. These quantities are related as follows:

$$\textit{speed-up} = \textit{utilization} * P.$$

5.3. Analysis of Simulation Experiments

To study the behavior of the load distribution strategies, a number of simulation experiments were carried out. Five factors were varied during these experiments. These factors and the values over which they were varied are listed in Table 5.1. Note that ρ is varied from 0.1 to 10.0. Here $\rho = 1.0$ means that τ, the network clock cycle, is equal to the time required to process or spawn a node. Smaller values of ρ imply a slower network and conversely. This combination of factors gave rise to more than 2000 simulation runs, generating a large amount of data which can be studied from many different viewpoints.

Factor	*Values*
P	$2^2, 2^3, 2^4, 2^5, 2^6$
ρ	0.1, 0.32, 1.0, 3.16, 10.0
Problems	*Knights*, *Queens*, *TSP*, *Quadrature_1*, *Quadrature_2*
Networks	*Bus*, *Complete*, *Hyper*, *Ring*, *Tree*
Strategies	*Tree*, *Robin*, *Askwork*, *Knowledge*

Table 5.1 Factors varied during simulation experiments

To facilitate this study, we divide the issues into two broad categories which we analyze separately; the effect of network topology and the effect of the load balancing strategies. We study the effect of network topology, by looking at the performance of the different networks under varying circumstances. The results achieved are discussed in relation to the theoretical analysis of the networks done in Chapter 2.

After that we examine load distribution strategies, looking first at strategy *Tree*, which maps search trees onto a tree architecture. The performance of this strategy is compared to that of the other load distribution strategies when they are used on the tree network.

Finally we study the the other three load distribution strategies. We look at the way the performance of these strategies is affected by the speed and size of the networks and by other factors. In particular we look at the underlying search tree generated by the different problems and relate the parameters of the search trees discussed in Chapter 3 to the performance of the load distribution strategies.

5.3.1. Network Topology

The five networks under consideration were theoretically analyzed in Chapter 2. In this section we study the relative merits of these different networks when they are used with the load distribution strategy *Robin*. This strategy is more communication intensive than the other strategies and uses long distance packet-switched communication rather than neighborhood connection.

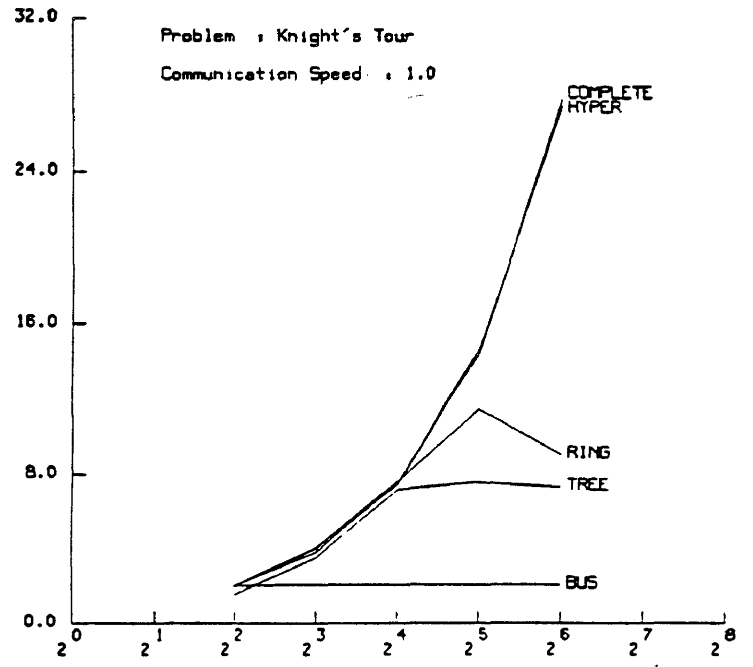
We present two sets of graphs, Graphs 5.1 and 5.2, in this section. In the first set, Graphs 5.1, the first graph plots speed-up versus the number of processors, while the second graph plots speed-up versus the communication to computation speed ratio, ρ.

In both graphs the knight's tour search tree is used.

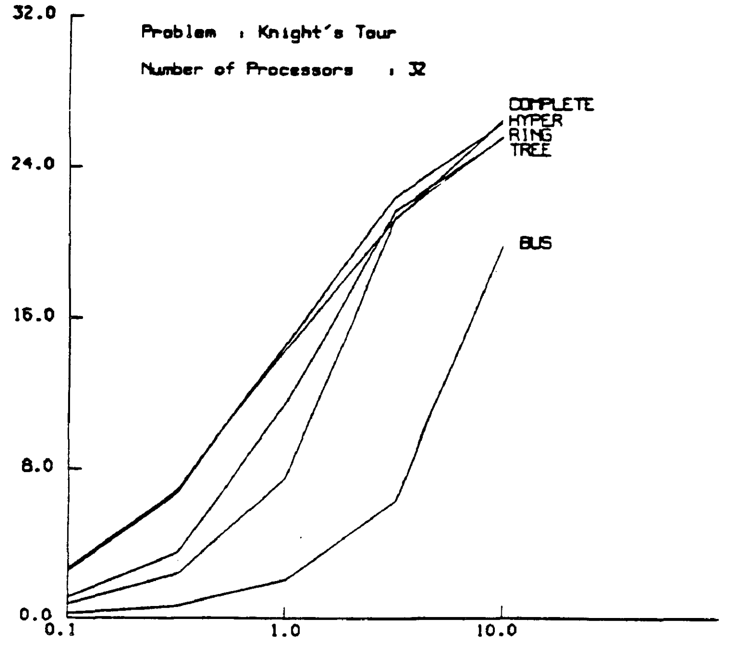
Graph 5.1a shows the speed-up that can be obtained on different networks as P is increased from 4 to 64[1]. The value of ρ is 1.0, that is, the time needed to spawn a child of a node in the search tree is equal to the time needed for a message to be sent across a channel. As the number of processors is increased, the performance of different networks seems to mirror the throughput column of Table 2.1. The throughput of the bus, ring and tree is essentially constant, independent of the number of processors, in the network and this is reflected in the flattening out of the speed-up curves. In fact for a bus there is essentially no change with respect to P while for ring and tree the flattening occurs at slightly later points reflecting the higher constants in the throughput.

In contrast to the bus, the hypercube network shows an almost linear increase in its speed-up as the number of processors is increased. The throughput of the hypercube was seen to be proportional to P, the number of processors, reflecting this linear speed-up. The complete network also achieves linear speed-up, which does not seem to reflect the $O(P^2)$ theoretical throughput. This is a consequence of the fact that the architectural assumptions of the simulator are slightly different from the ones assumed in our theoretical analysis. Here each processor can send only one message per communication cycle, since each message has to be transmitted to the co-processor, requiring time τ. Consequently, each processor can send only one message per network clock cycle. Hence the throughput for a complete network is $O(P)$ rather than $O(P^2)$, as theoretically calculated in Chapter 2. Note that this difference in architectural assumptions does not affect the other networks, since their throughput is at most $O(P)$

[1] For the tree network the number of processors is always 2^k-1 rather than the listed 2^k.



Graph 5.1a: Speed-up vs. number of processors for ROBIN strategy



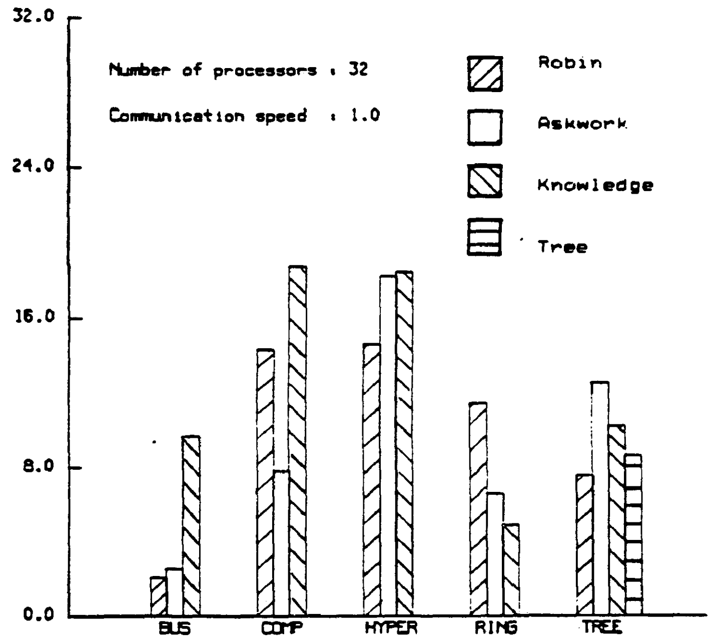
Graph 5.1b: Speed-up vs. communication speed for ROBIN strategy

anyway.

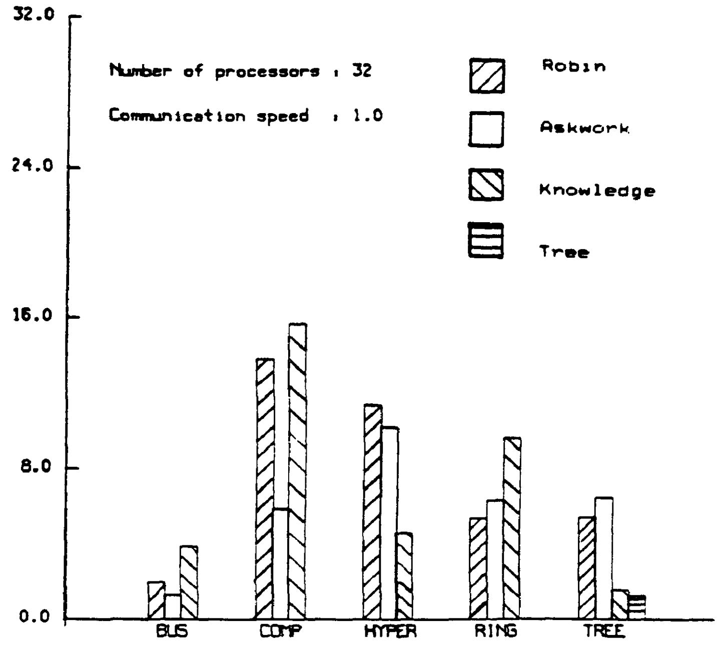
The next Graph 5.1b shows the speed-ups achieved as a function of relative communication speed, ρ. In this graph, the number of processors, P, is 32, and the underlying search tree problem is the knight's tour. With faster communication (higher values of ρ) performance differences between the networks become less evident, and all but the bus do about equally well. When communication is slower, a correspondence between the throughput column of Table 2.1 and the performance of networks can be seen again. That is, message contention and throughput determines the speed-up.

The next set of graphs, Graphs 5.2, show the speed-up that can be achieved by each of the load distribution strategies on the different networks. For the tree network, we present four strategies, including the *Tree* strategy. Here the number of processors on each of the networks is 32 and the relative communication speed ρ is 1.0. The first graph, Graph 5.2a presents the data for the knight's tour problem, which has a balanced search tree, while the second graph, Graph 5.2b looks at the problem quadrature 1, which has a highly unbalanced search tree.

For the knight's tour problem, most of the networks seem to perform well. The higher bandwidth complete and hypercube networks do better than the rest, while the bus does worse. By contrast, on an unbalanced search tree, such as that generated by the first quadrature problem, the performance of all networks has dropped. The balanced search tree leads to a better distribution of the work and to higher speed-ups, while for an unbalanced search tree, some processors finish their work sooner than others, leading to poor utilization. This observation holds for all networks except the bus network, which has uniformly poor performance.



Graph 5.2a: Speed-up vs. networks for KNIGHT'S TOUR problem



Graph 5.2b: Speed-up vs. networks for QUADRATURE 1 problem

In this section, we have presented several graphs in order to characterize the performance of the various networks under consideration. All the graphs presented lead to the same conclusion, that the complete and the hypercube networks perform much better than the other networks. The ring and the tree networks have intermediate performance while the bus has the poorest performance. This closely reflects the theoretical analysis given in Chapter 2.

Of course, the high performance of the complete and hypercube networks is achieved at a cost. The cost of the hypercube network, as shown in Table 2.1, is a factor of $\log P$ more than that of the bus, ring or tree. But with the hypercube interconnection, speed-ups a factor of three higher than that of a ring or tree are achieved with 64 processors. Thus if the cost of the network is not the dominant cost in the computer, the hypercube interconnection seems strongly preferable to a tree or ring interconnection.

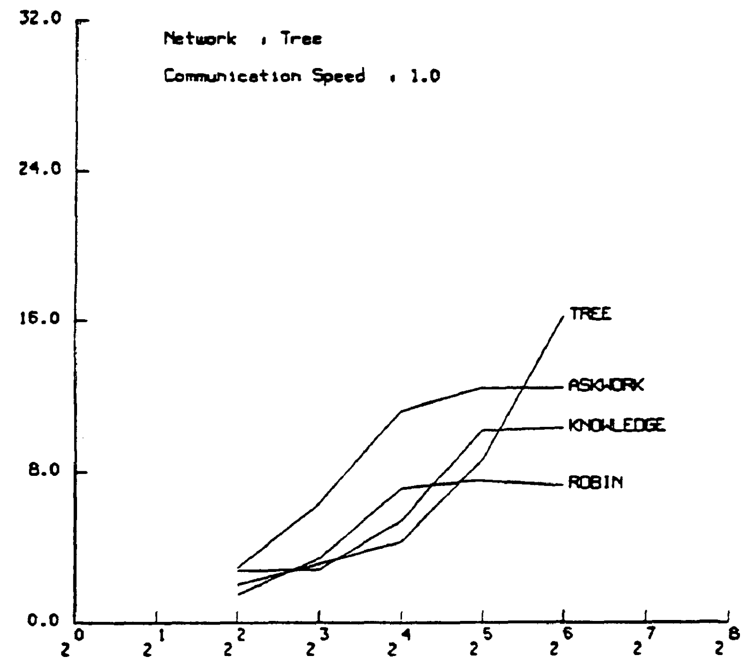
With the assumptions made in the simulator, the complete connection network does not show up well. Our simulator assumptions imply that it has only $O(P)$ throughput, while its cost is $O(P^2)$. This network would look better under different assumptions, but it is generally acknowledged that the complete connection network is too expensive to construct for large number of processors. The hypercube network, and related networks like the shuffle and cube-connected cycle network, having mean internode distance $O(logP)$, appear to be generally the most cost effective networks.

5.3.2. Performance of the Tree Strategy

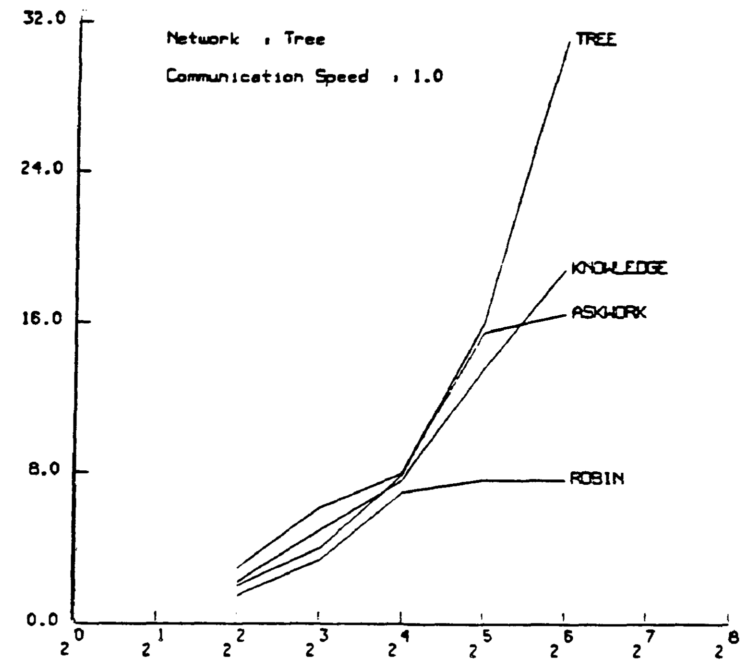
It is natural to study the possibility of performing tree searching on a tree architecture. In the last chapter, a strategy *Tree*, was developed, that was tailored to this architecture. In this section, we compare the performance of this strategy with that of the other strategies on the tree network.

The first set of graphs, Graphs 5.3, represents the performance of the different strategies as the number of processors on the tree network is varied. The *Tree* strategy seems to perform well for problems with a balanced search tree. This can be seen from the Graphs 5.3a-c for the knight's tour, the eight queens, and the traveling salesman problem respectively. For unbalanced search trees, the *Tree* strategy does not perform as well as the other strategies, as seen in Graphs 5.3d-e. This is as expected, given the manner in which the *Tree* strategy distributes load. Work is divided between the two children of the current processor node in the network. This division of work is continued until the leaf processors are reached, which process the entire subtree rooted at each of the nodes that they are given. Thus, for a balanced search tree, all leaf processors end up with approximately equal work. This leads to high utilization of the leaf processors, with little communication costs.

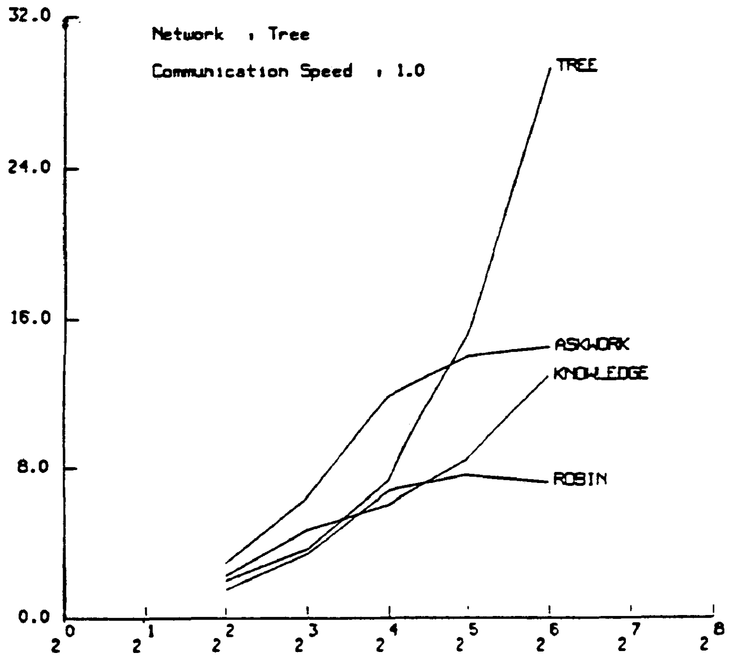
On the other hand, when the search tree is unbalanced, the leaf processors end up with an uneven distribution of work, leading to poor utilization. In such a situation, strategies such as *Askwork* and *Knowledge* perform better, since work can migrate between nodes to achieve better balance. The *Tree* strategy is much worse for unbalanced trees, as can also be noted from Graph 5.4, where the speed-up achieved by the various strategies are presented as a bar graph.



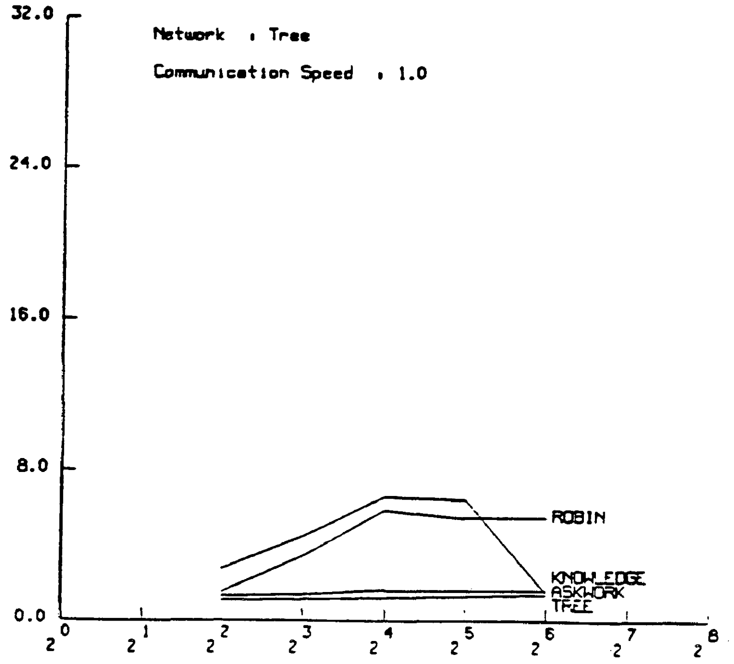
Graph 5.3a: Speed-up vs. no. of processors for KNIGHT'S TOUR



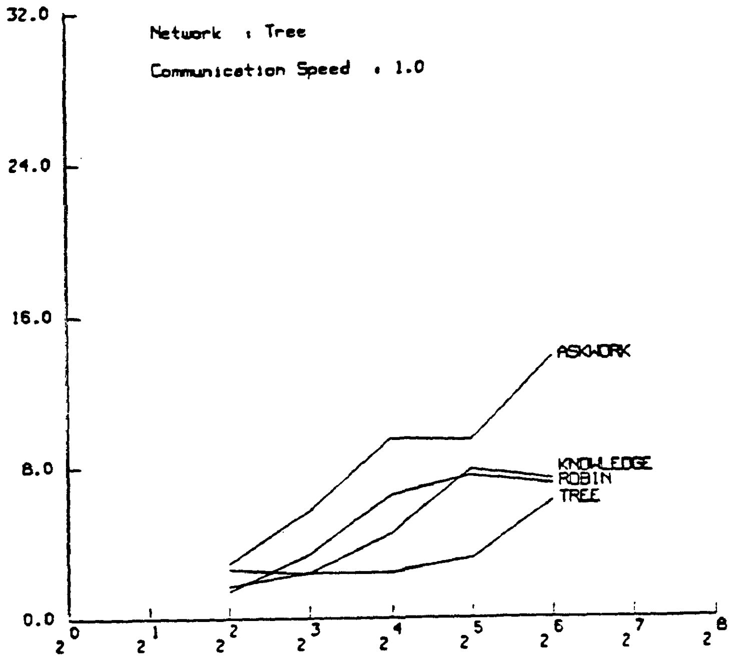
Graph 5.3b: Speed-up vs. no. of processors for EIGHT QUEENS



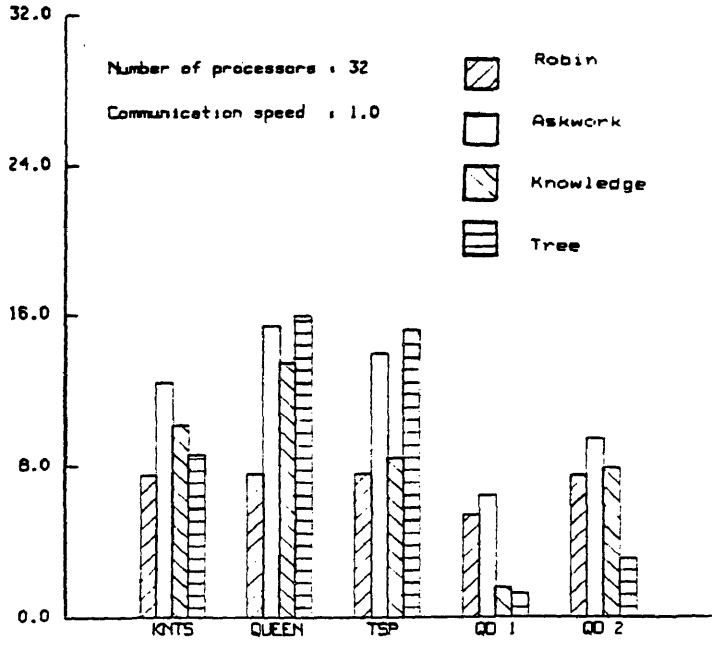
Graph 5.3c: Speed-up vs. no. of processors for TSP



Graph 5.3d: Speed-up vs. no. of processors for QUADRATURE 1



Graph 5.3a: Speed-up vs. no. of processors for QUADRATURE 2



Graph 5.4: Speed-up vs problems for the Tree network

For balanced search trees, the performance of the *Tree* strategy seems to be dependent on the structure of the search tree. The strategy does better for queens and the traveling salesman problem than for the knight's tour. The search tree generated by the knight's tour has a smaller average breadth than the queens and the TSP search trees. Also the knight's tour search tree has a smaller number of nodes at levels nearer the root node. For example, since the knight's initial position is in the corner of the board, there are only two possible initial moves, and thus there are only two children for the root node. Thus for the knight's tour, the processors in the tree architecture nearer the root finish their part of their work quickly, since they do not have many nodes to process. This leads to poorer utilization.

On the other hand, for the eight queens problem or the traveling salesman problem, depending upon the number of processors in the network, work may reach the leaf nodes before processors higher up in the tree have finished their processing. For the eight queens problem, the root node has 8 children, and for the traveling salesman problem, the root node has, in our case, 6 children. This seems to lead to a pipelining effect, in which all processors in the tree architecture are simultaneously active. The result is a surprisingly high utilization of the processors for these problems.

The set of graphs presented in this section show that the *Tree* strategy does as well or better than the other strategies for balanced search trees and especially for balanced and bushy search trees. This is particularly true with large number of processors (i.e. $P = 32 \textit{ or } 64$) and must be because the *Tree* strategy is tuned to the tree network and can distribute the load to the leaf processors with little communication overhead. With smaller number of processors the distinction between strategies is not as evident.

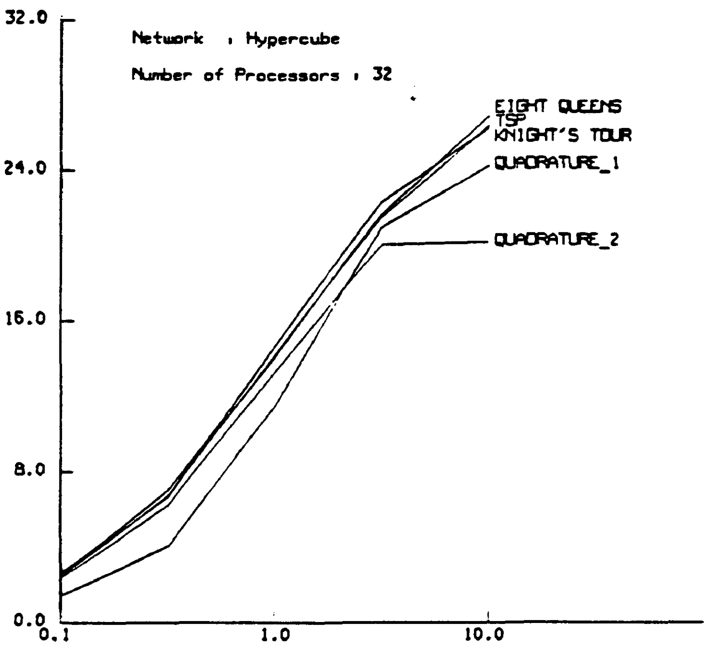
On the other hand, with unbalanced trees the *Tree* strategy performs very poorly. As was seen in the last section, the hypercube and complete networks performed better than the tree network for both balanced and unbalanced search trees. Thus, if a priori knowledge of the tree is not available, one should not choose the *Tree* strategy.

5.3.3. More General Strategies

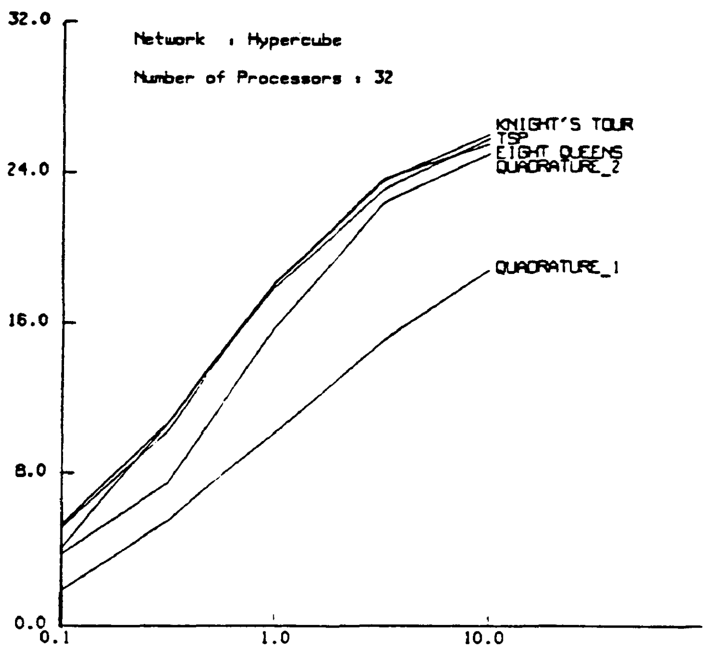
In the last section we concentrated on the *Tree* strategy. In this and the next subsection we study the other three strategies. Here we look at basic issues such as how the communication speed and the network size affect the performance of the strategies. In the next section we look in more detail at each of the three strategies studying both their speed-ups and utilization.

Performance as a Function of Communication Speed

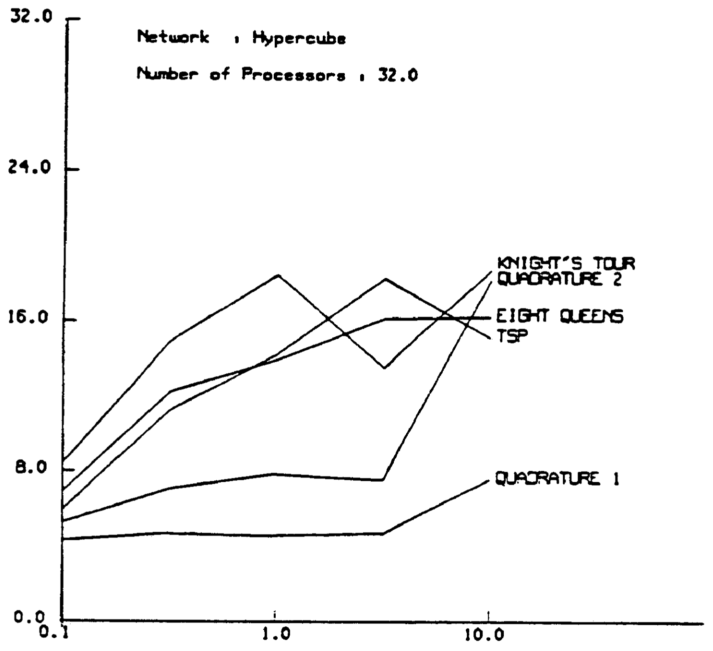
Graphs 5.5 present the performance of the three strategies for the different search trees, as the communication speed ρ is varied. For these graphs the number of processors, P, is fixed at 32, while the network is the hypercube. As expected, all three strategies do a better job of distributing work if the search trees are balanced and if communication is fast. The strategies *Robin* and *Askwork* achieve almost linear increases in speed-up as the communication becomes faster. The effect of increasing communication speed is unpredictable for the *Knowledge* strategy. This strategy attempts to send very few messages and thus factors other than communication speed seem to be important in this case. However, at low communication speed, the strategy *Knowledge* achieves higher speed-up than the other two strategies. This is true for all the problems.



Graph 5.5a: Speed-up vs. communication speed for ROBIN strategy



Graph 5.5b: Speed-up vs. communication speed for ASKWORK strategy

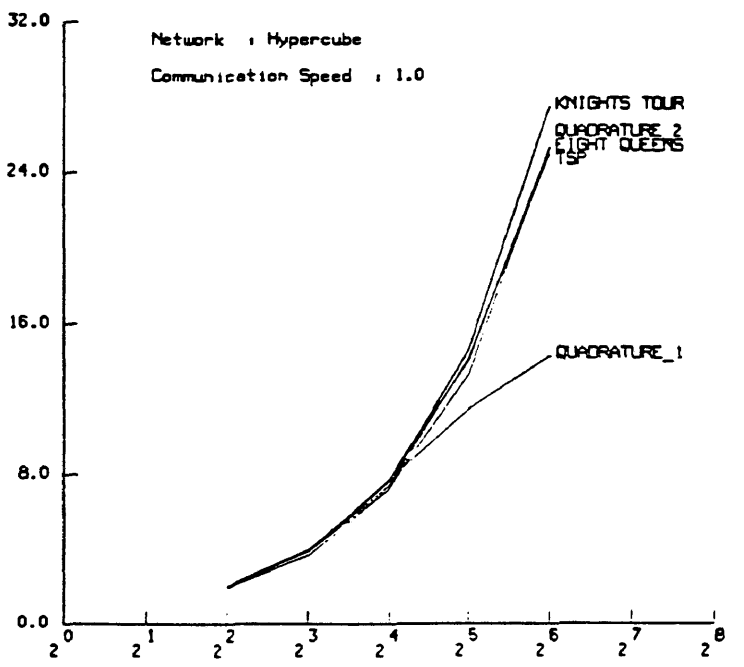


Graph 5.5c: Speed-up vs. communication speed for KNOWLEDGE strategy

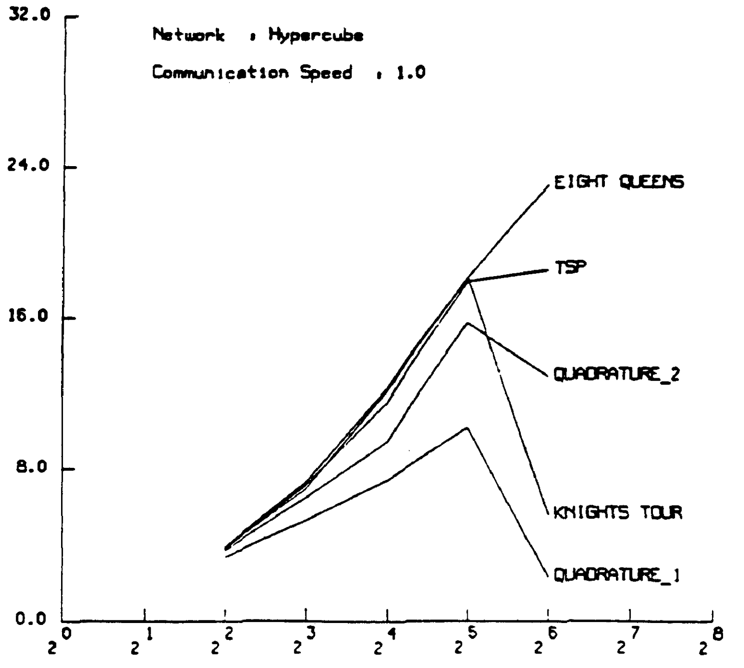
Performance as a Function of Network Size

The next set of graphs, Graphs 5.6, show the performance of the three strategies as a function of the number of processors P. The network is again set to be the hypercube, while the communication speed is set to 1.0. Graph 5.6a shows that the speed-up achieved by the *Robin* strategy is approximately linear in the number of processors. As will be shown later, the utilization is almost independent of the number of processors for this strategy on networks like the hypercube having $O(P)$ throughput. This is a consequence of the fact that this strategy generates the same number of messages regardless of the network or number of processors.

The second strategy, *Askwork*, does well for moderate numbers of processors, but does poorly for 64 processors. In this strategy a processor sends out work-request



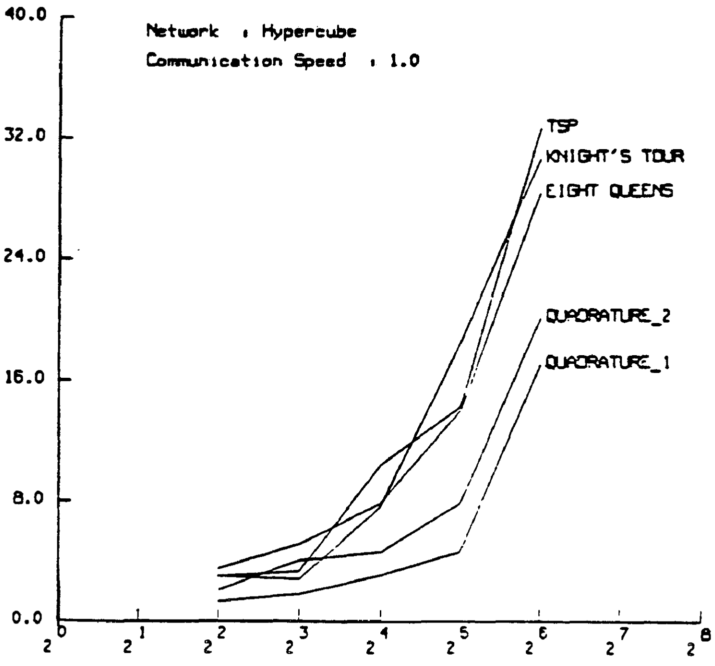
Graph 5.6a: Speed-up vs. number of processors for ROBIN strategy



Graph 5.6b: Speed-up vs. number of processors for ASKWORK strategy

messages to its neighbors when it becomes idle. These messages are overhead messages which do not perform any "useful" work. It may be that even for the hypercube network, where each processor has only log*P* neighbors, the total number of work-request messages sent by this strategy is too large, and saturates the network if there are a large number of processors. This issue would be less critical for larger problems having more inherent parallelism.

The last strategy, *Knowledge*, seems to perform well on all problems except quadrature_1, which generates a narrow and deep search tree. This strategy has a behavior similar to that of *Robin* though the "envelope" of performance is more spread out. As already stated, the behavior of this strategy is subtle and difficult to interpret.



Graph 5.6c: Speed-up vs. number of processors for KNOWLEDGE strategy

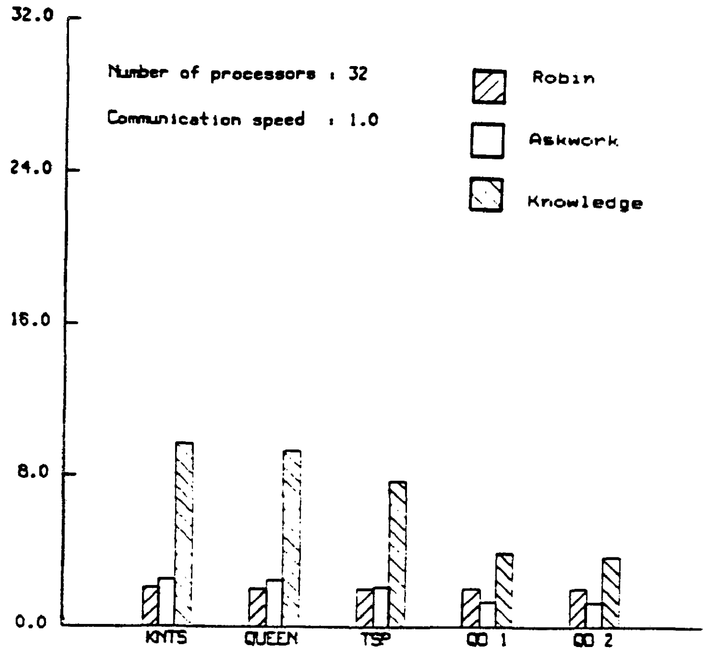
Interaction of Networks with Strategies

With the next set of graphs, Graphs 5.7, we compare the behavior of the strategies on each of the networks, attempting to understand the way the strategies and network topologies interact. Graph 5.4 in the last section gave similar data but only for the tree network.

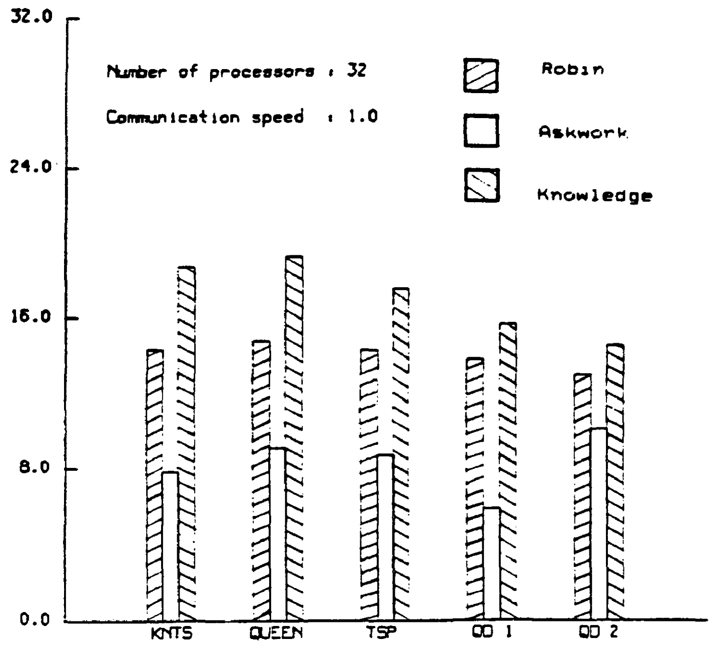
On the complete connection, Graph 5.7b, the strategy *Knowledge* performs very well, because each processor can keep track of the workload of all other processors. This system wide information allows the strategy to distribute the workload uniformly. The rich interconnection pattern helps the strategy *Robin* also in the distribution of the workload, though it does not perform as well as *Knowledge*.

On the other hand the rich interconnection pattern reduces the efficiency of the *Askwork* strategy. This can be attributed to the fact that in *Askwork* a processor sends work-request messages to all neighbors when it is idle. The rich interconnection pattern thus implies that the network is flooded with work-request messages, causing contention which delays actual-work messages. Thus the *Askwork* strategy performs poorly on the complete network. In principle, this problem could be fixed by considering a modified *Askwork* strategy, in which each processor sends work-request messages to only a limited subset of its neighbors.

A similar behavior is seen for the bus network in Graph 5.7a. Here again each processor is considered a neighbor of every other processor, but in this case the bus itself acts like a bottleneck reducing the overall performance. The *Knowledge* strategy, which minimizes communication, performs much better than the other strategies on the bus network. The high message traffic, generated by both *Robin* and *Askwork* reduce



Graph 5.7a: Speed-up vs. problem for BUS network



Graph 5.7b: Speed-up vs. problem for COMPLETE network

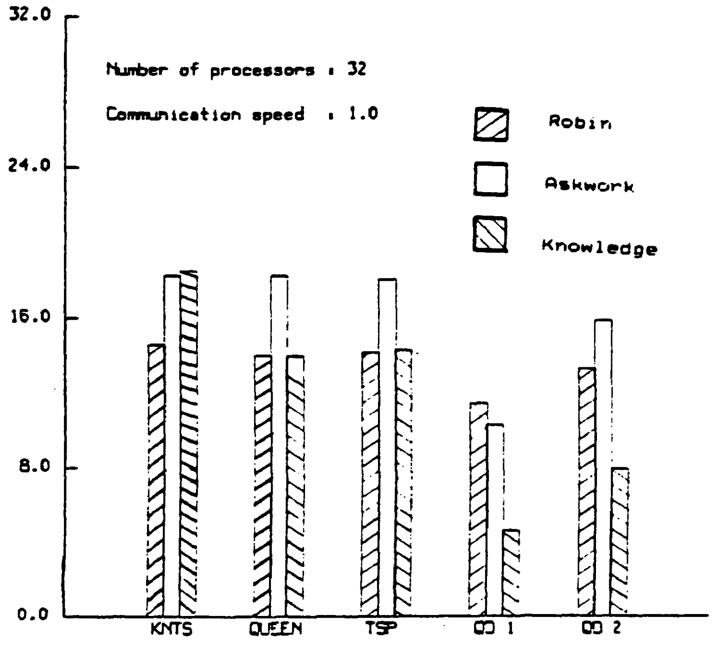
their performance considerably.

The hypercube network, Graph 5.7c, provides an interesting contrast to the above two networks. This network has low mean internode distance and high throughput, but the number of neighbors of each processor is only $\log P$, for P processors. Here the *Askwork* strategy comes into its own, performing better than the other two strategies, in most cases.

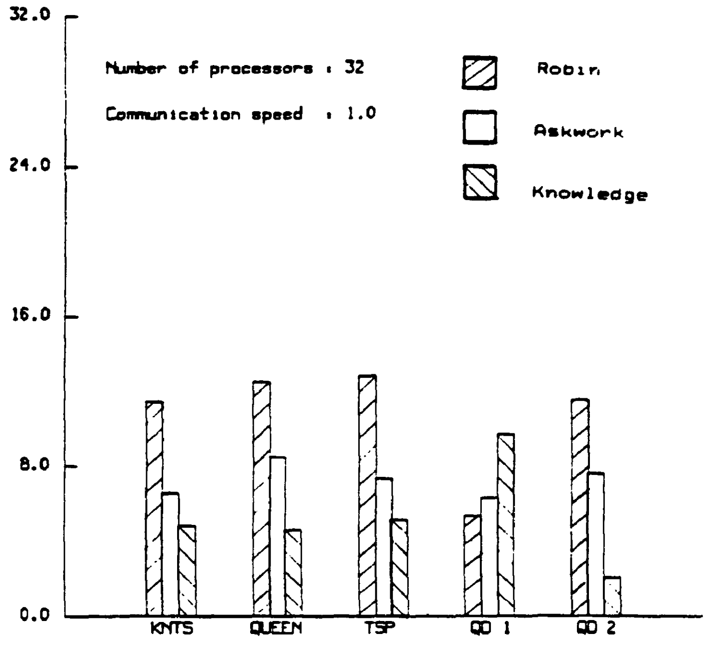
On the ring network, in Graph 5.7d, we find *Robin* doing better than the other two strategies. The reason for this is that in *Askwork* and *Knowledge*, work is passed only to neighbors. Thus after the initial round robin phase, work moves in wavefronts around the ring. With the *Robin* strategy, on the other hand, the long distance packet-switched communication yields better load balance and higher speed-ups.

The data for the *Tree* network was presented in Graph 5.4 in the last section, where we saw that *Askwork* performed better than the other two strategies. Here the situation is similar to the hypercube. The tree interconnection is less richly interconnected than the hypercube network, but has smaller mean internode distance than the ring, allowing *Askwork* to distribute work better. In a tree network each node has at most three neighbors, thus limiting *Knowledge*'s ability to efficiently distribute the workload.

The set of Graphs 5.7, indicates that the strategy *Knowledge* performs better on networks which have a rich interconnection pattern, since then each processor has more knowledge about the system. On the other hand, *Askwork* does poorly if the number of neighbors of each processor is large, since in this case too many work-request messages are sent out. It is interesting to note that on the highly unbalanced search tree,



Graph 5.7c: Speed-up vs. problem for HYPERCUBE network



Graph 5.7d: Speed-up vs. problem for RING network

quadrature_1, the brute-force strategy *Robin* out performs the other strategies on most networks.

5.3.4. Detailed Analysis of Strategies

In this subsection we look at each of the three strategies, studying their performance as functions of the number of processors and relative communication speed. Here we freeze the knight's tour as the underlying search problem. Though in principle one gets the same information by plotting speed-up or utilization, it is often difficult to visually infer utilization from speed-up and conversely. Because of this, we plot both speed-up and utilization here as functions of the number of processors. We also plot speed-up as a function of the communication speed ρ, for the specific case of 32 processors.

Robin

The first set of graphs here show the *Robin* strategy. Note that Graph 5.8a and 5.8c duplicate Graph 5.1a and 5.1b. From Graph 5.8b one can see that the utilization for hypercube and complete networks is nearly independent of P, the number of processors, while it degrades for the bus, ring and tree. This is due to the fact that the throughput for the hypercube and complete connection networks is proportional to P, and hence the network is able to handle the increased traffic generated by *Robin*. For the other three networks the throughput remains constant and thus as the network size (and message traffic) increases the utilization drops rapidly.

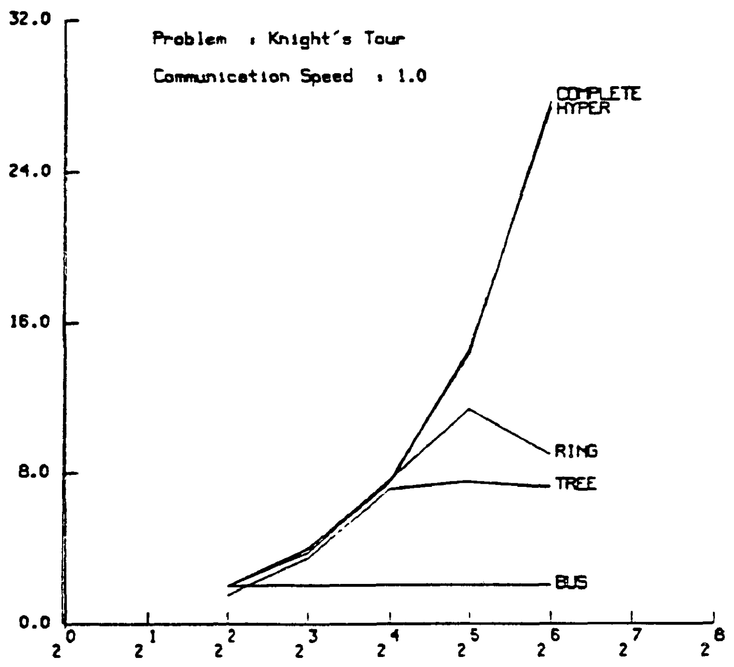
Graph 5.8c presents the speed-ups achieved by *Robin*, as a function of the network speed ρ. As noted before, for lower communication speeds (smaller values of ρ) the performance reflects the throughput of the network. With increasing communication speed, ρ, as the penalty for communication decreases, the *Robin* strategy performs well on all networks.

Askwork

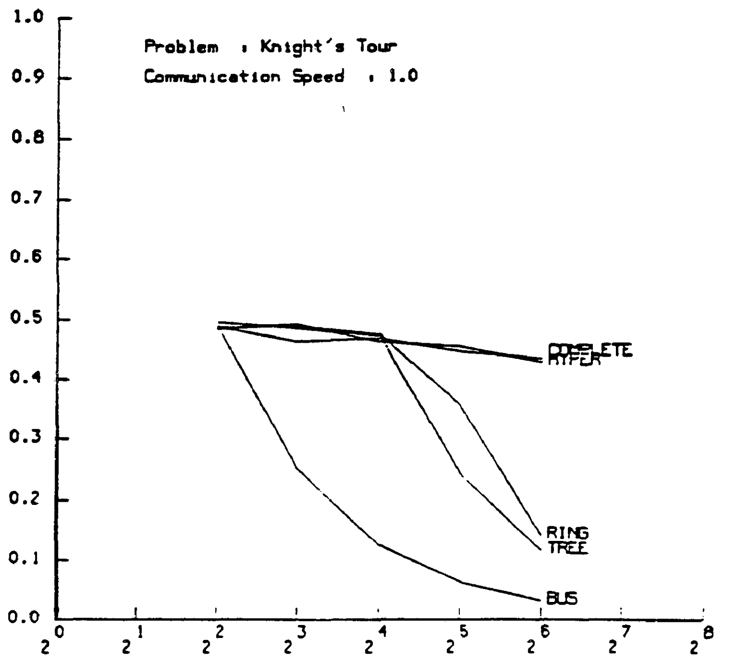
The second set of graphs, Graphs 5.9, show the performance of the *Askwork* strategy. When the number of processors is small, *Askwork* achieves very high utilization on all networks. As the number of processor increases, the topology of the networks starts influencing the performance. The performance of *Askwork* is worst for the bus and complete connection network, as reflected by both Graph 5.9a and 5.9b. For networks with smaller number of neighbors, *Askwork* does a good job of distributing the workload.

Knowledge

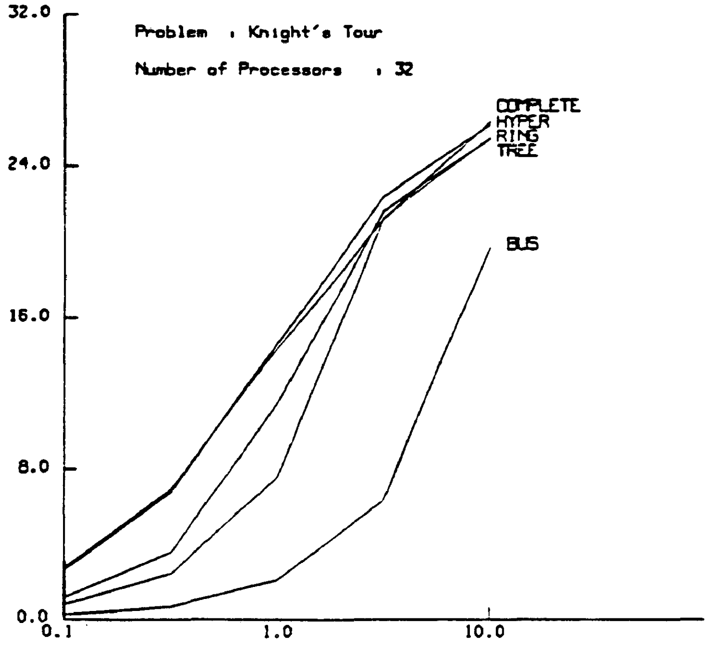
The last set of graphs, Graphs 5.10, depict the performance of the strategy *Knowledge* as a function of the number of processors, and the communication speed. The *Knowledge* strategy generates the least number of messages of all the strategies studied here. However, subtle effects due to the order of execution of the nodes of the search tree can arise with this strategy. Such effects were not seen for the other strategies, since the number of messages they generated was large and communication dominated all other factors. The performance of *Knowledge* appears to be far less predictable than the other strategies, and is difficult to analyze.



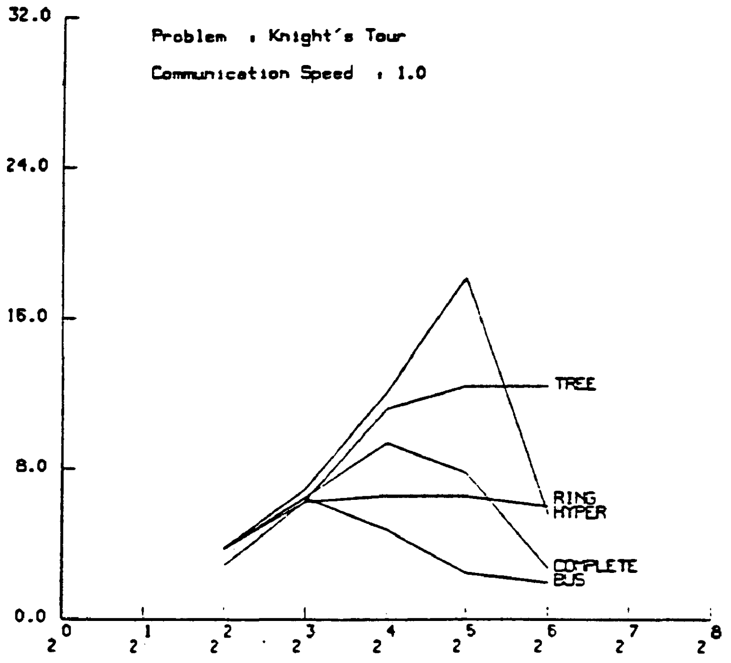
Graph 5.8a: Speed-up vs. no. of processors for ROBIN strategy



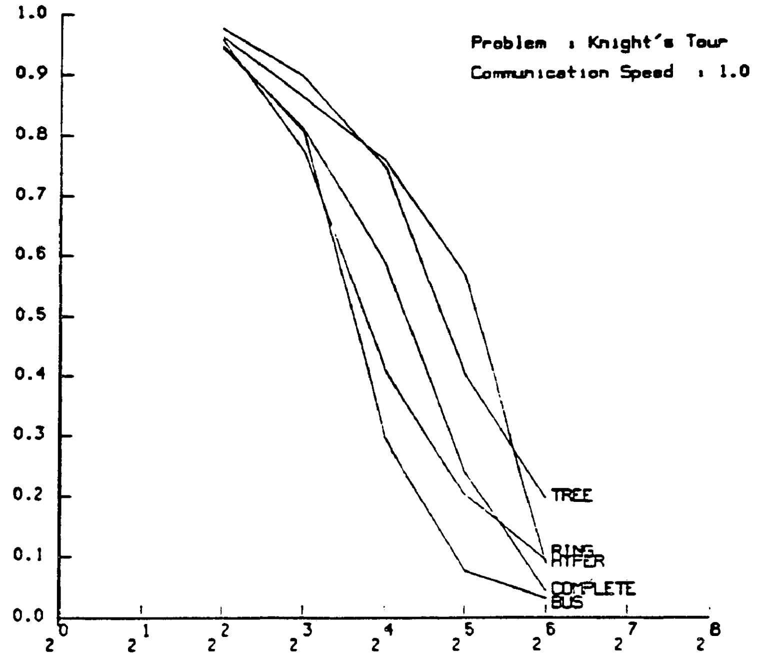
Graph 5.8b: Utilization vs. no. of processors for ROBIN strategy



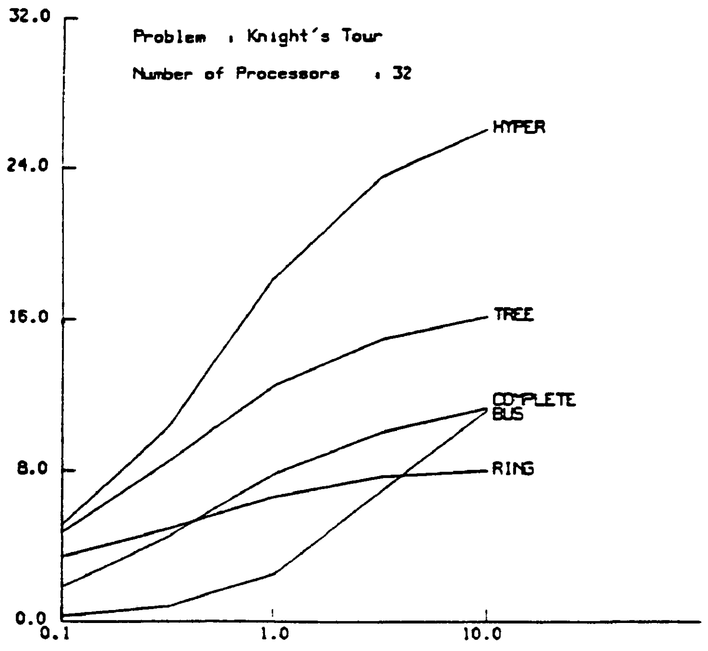
Graph 5.8c: Speed-up vs. communication speed for ROBIN strategy



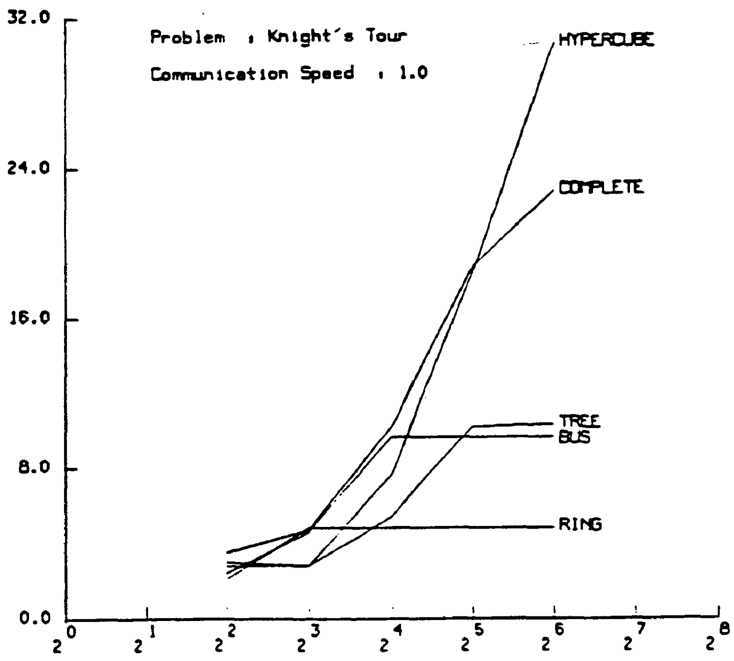
Graph 5.9a: Speed-up vs. no. of processors for ASKWORK strategy



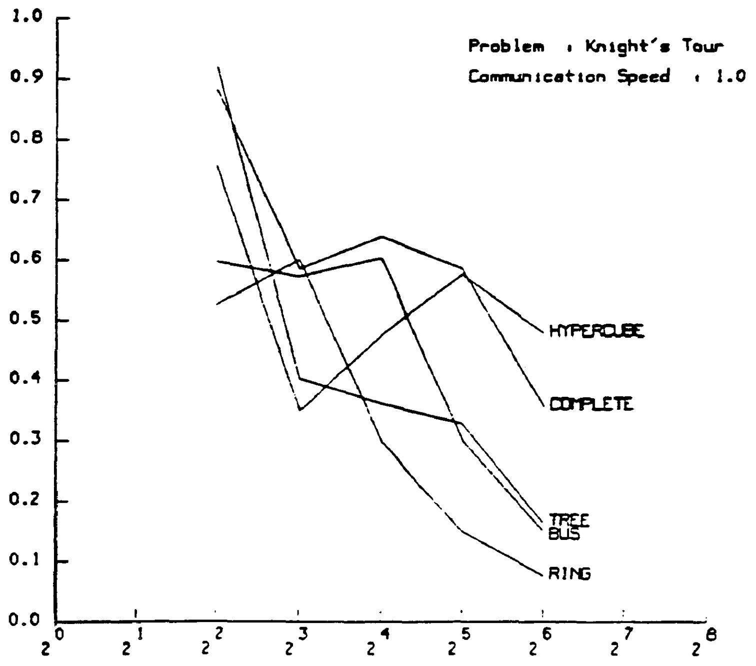
Graph 5.9b: Utilization vs. no. of processors for ASKWORK strategy



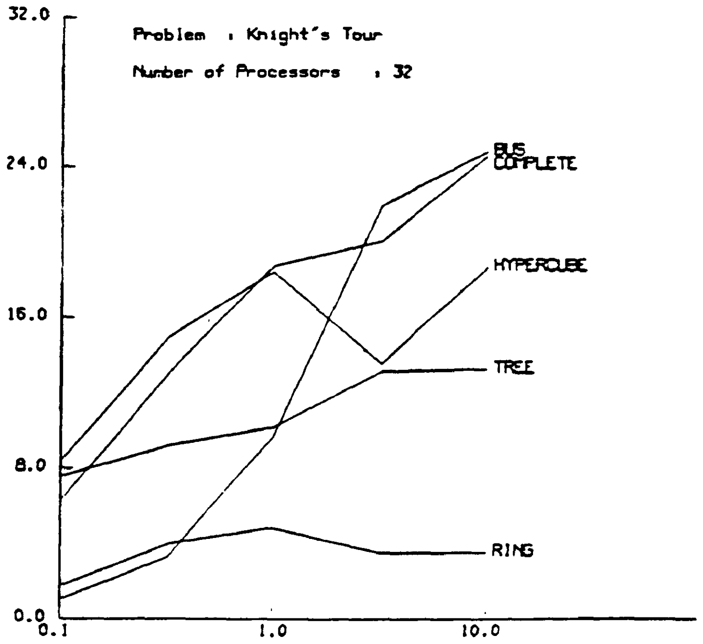
Graph 5.9c: Speed-up vs. communication speed for ASKWORK strategy



Graph 5.10a: Speed-up vs. no. of processors for KNOWLEDGE strategy



Graph 5.10b: Utilization vs. no. of processors for KNOWLEDGE strategy



Graph 5.10c: Speed-up vs. communication speed for KNOWLEDGE strategy

CHAPTER 6

CONCLUSIONS

In this thesis we have studied some of the issues involved in mapping tree searching algorithms onto non-shared memory multiprocessor architectures. Tree searching is employed in a wide variety of application areas. We chose four different algorithms for the purpose of this study. These were the eight queens problem, the knight's tour on a 4 by 4 board, the traveling salesman problem on a network of seven cities, and two problems in numerical quadrature. As seen in Chapter 3 the first three problems generated well-balanced search trees while the quadrature problems gave rise to narrow and unbalanced search trees.

These tree searching problems were mapped onto five topologically different architectures. Besides the interconnection pattern, the size and speed of the network were also varied. We devised four different load distribution strategies. The *Tree* strategy was tuned to the tree network, while *Robin* was a simple minded strategy which distributed work in a round robin fashion. The *Askwork* strategy used the concept of work-request messages. Work was sent out by a processor on receiving such a message from its neighbor. The *Knowledge* strategy attempted to maintain the workload information about the neighbors on each processor. Such information was then utilized to parcel out work to neighbors. All of these strategies were independent of the network topology (except for the *Tree* strategy) and used no a priori knowledge of the search tree. This approach was taken in order to find strategies which are both effective and robust.

6.1. General Conclusions

Several broad conclusions can be drawn from our study. We did not find any one strategy which performs uniformly well on all networks and on all problems. The variety of factors characterizing the networks and the problems have subtle effects on the performance of the strategies, leading to complex behavior.

For example, consider the "richness" of the interconnection pattern. On a richly interconnected network such as the complete connection network, the potential for distributing work is high and we see that *Knowledge* performs very well. However, the larger number of neighbors works against the *Askwork* strategy, which generates work-request messages for all neighbors. Similar behavior patterns are seen with the bus, and of course, the bus is itself a bottleneck.

The *Askwork* strategy performs better on networks such as the hypercube and the tree, because of the small number of neighbors there. In such a situation, *Knowledge* does not have enough system wide information to perform well. The *Robin* strategy does not do as well as either *Askwork* or *Knowledge* on any of those networks. Surprisingly, *Robin* works better on the ring network than the other strategies apparently because it does long distance communication rather than communication with neighbors, as in the other strategies. This results in better load distribution on this network. All of these effects were observed on both balanced and unbalanced search trees, the difference being that on unbalanced trees all strategies did worse than on balanced trees.

The behavior of the strategies with increasing number of processors is also complex. For networks with small number of processors, all strategies performed very well, including *Robin*, achieving utilization of 90% or higher. For larger networks the amount of inherent parallelism in the problem started to play a more important role. In general, for large number of processors we found that the strategy *Knowledge*, which utilized the least amount of communication in distributing the workload, did better than the other two strategies, *Askwork* and *Robin*.

Similar behavior was seen while varying the communication speed for the various networks. For faster networks all strategies performed well since communication was not penalized. With slow communication *Knowledge* did well because of its lower message traffic.

The fourth strategy, *Tree*, was devised specifically for the tree network. It was found that this strategy performed well only when the underlying search tree was extremely balanced and bushy near the root node. Under such circumstances, a pipelining effect aided in achieving higher speed ups.

In comparing networks we found that the hypercube network proved to be the most cost-effective of the five networks studied. This is not surprising in view of its high theoretical throughput and moderate cost. The complete connection network did equally well under most circumstances but is considerably more costly than the hypercube. The tree network performed surprisingly well considering its constant throughput, perhaps because mean internode distance also plays a significant role. As one could have anticipated, the ring and the bus performed poorly, the bus being the worst of the two.

6.2. Limitations of Study

Several simplifying assumptions were made during our study, and given more time one could have made a more detailed study. The principal assumption made here was that separate subtrees of a search tree can be searched in parallel. This is a severe restriction as it rules out several important algorithms such as branch and bound, and alpha-beta pruning[19].

There were also several relatively minor simplifying assumptions made here. One of these was that the time to execute or spawn a node in the search tree was taken to be a fixed constant, independent of the problem or the particular node in the search tree. This is a reasonable simplifying assumption, but, for example, the time taken to test a board position in the eight queens problem may depend on the number of queens already present. Better treatment of this issue might alter our results slightly, but would not have a major effect.

We also assumed that the messages sent between processors were of constant size, independent of the problem. This is also a natural simplifying assumption, but one which could be altered in a more detailed study.

In simulating the distribution strategies, we have also ignored the overhead due to the strategy itself. This penalizes simple strategies such as ***Robin*** which do not have much overhead over more sophisticated ones such as, ***Knowledge***, in which knowledge tables have to be maintained. A more detailed simulation would have included this overhead.

Though the simplifying assumptions just described are particular to the simulation study performed, similar assumptions are inevitable with computer simulation studies.

Given more time we would have liked to study tree searching and load balancing issues on working parallel architectures such as Pringle and NASA Finite Element Machine. Such experiments would be useful not only in validating our simulation results, but also in demonstrating the practical value of this approach to parallel computation.

REFERENCES

1. *Introduction to the iAPX432 Architecture,* Intel Corporation (1981).
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley (1974).
3. G. Broomell and J. R. Heath, "Classification Categories and Historical Development of Circuit Switching Topologies," *Computing Surveys* **15** (2) p. 95 (June 1983).
4. S. A. Browning, "Computations on a Tree of Processors," *Proceedings of the Caltech Conference on VLSI*, (January 1979).
5. S. A. Browning and C. L. Seitz, "Communication in a Tree Machine," *Proceedings of the Second Caltech Conference on VlSI*, (January 1981).
6. F. W. Burton and M. M. Huntbach, "Virtual Tree Machines," *IEEE Transactions on Computers* **C-33** (3) p. 278 (March 1984).
7. A. Despain and D. Patterson, "X-TREE: A Structured Multiprocessor Computer Architecture," *Proceedings 5-th Annual Symposium on Computer Architecture IEEE*, p. 144 (April 1978).
8. R. M. Fujimoto, "SIMON: A Simulator of Multicomputer Networks," UCB/CSD 83/140, Computer Science Division (EECS), University of California, Berkely, California (September 1983).
9. S. H. Fuller, A. K. Jones, and L. Durhamm, "Cm* Review, June '77," Technical Report, Dept. of Computer Scienc, Carnegie-Mellon University, Pittsburgh, Pa. (June 1977).
10. F. G. Gray, W.M. McCormack, and R. M. Haralick, "Significance of Problem Solving Parameters on the Performance of Combinatorial Algorithms on Multi-Computer Parallel Architectures," *International Conference on Parallel Processing*, p. 185 (1982).
11. R. W. Hamming, "Integration," in *Introduction to Applied Numerical Analysis*",, McGraw-Hill, Inc (1971).
12. H. Jordan, "A Special Purpose Architecture for Finite Element Analysis," *Proceedings of the 1978 International Conference on Parallel Processing IEEE*, p. 263 (1978).
13. A. Kapauan, K. Wang, D. Gannon, J. Cuny, and L. Snyder, "The Pringle: An Experimental System for Parallel Algorithm Design and Testing," *Proceedings of the International Conference on Parallel Processing*, (August 1984).
14. A. Kapauan, D. Gannon, L. Snyder, and T. Field, "The Pringle Parallel Computer," *Proceedings of the International Conference on Computer Architecture IEEE-Sigarch*, (1984).

15. K. Kummerle and H. Rudin, "Packet and Circuit Switching: Cost/Performance Boundaries," *Computer Networks* **2** p. 3 (1978).

16. C. R. Lang, Jr, "The Extension of Object-Oriented Languages to a Homogeneous Concurrent Architecture," Report No. 5014, California Institute of Technology, Pasadena, California (1982).

17. G. Mago, "A Cellular Language Directed Architecture," *Proceedings of the Caltech Conference on Very Large Scale Integration*, p. 435 (January 1979).

18. C. Mead and L.Conway, *Introduction to VLSI Systems*, Addison-Wesley (1980).

19. N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*, McGraw Hill (1971).

20. F. Preparata, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Communications of the ACM* **24** p. 300 (1981).

21. D. A. Reed, "Performance Based Design and Analysis of Microcomputer Networks," *Ph.d Thesis*, Purdue University, (May 1983).

22. C. Rieger, "ZMOB: A Mob of 256 Cooperative Z80-based Microprocessors," *Proceedings Image Understanding Conference, Los Angeles, CA*, p. 25 (1979).

23. R. D. Rosner, "Packet Switching and Circuit Switching: A Comparison," *Proceedings of the National Telecommunications Conference, NTC*, p. 421 (December 1975).

24. C. L. Seitz, "The Cosmic Cube," *Communications of ACM* **28** (1) p. 22 (January 1985).

25. C. H. Sequin, A. M. Despain, and D. A. Patterson, "Communication in X-TREE, A Modular Multiprocessor System," *Proceedings of the ACM National Conference*, (December 1978).

26. H. J. Siegel, "Interconnection Networks for SIMD Machines," *Computer* **12** (6) p. 57 (June 1979).

27. B. Smith, "A Pipelined, Shared Resource MIMD Computer," *Proceedings 1978 International Conference on Parallel Processing*, p. 6 (1978).

28. H. S. Stone, "Parallel Processing with Perfect Shuffle," *IEEE Transactions on Computers* **20** p. 153 (Feb 1971).

29. H. S. Stone, "Parallel Computers," pp. 327 in *Introduction to Computer Architecture*, ed. H. S. Stone,Science Research Associates, Chicago (1980).

30. B. W. Wah, Y.W.Eva, and Ma, "MANIP - A Multicomputer Architecture for Solving Combinatorial Extremum- Search Problems," *IEEE Transactions on Computers* **C-33** (5) p. 377 (May 1984).

31. Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall Inc. (1976).

VITA

Mala Mehrotra was born in Patna, Bihar, India on May 12, 1954. She attended St. Joseph's Convent, Patna and graduated from there in 1970. From 1971 to 1974, she was in Lady Brabourne College, Calcutta University, Calcutta, India, studying for a B.Sc degree in Physics(Hons). In 1977 she obtained an M.Sc degree in Physics from the University of Delhi, Delhi, India.

She came to the United States in 1978 and went back to school in 1982 for further studies. From September 1982 to May 1985 she has worked towards a Master's degree in Computer Science from the College of William and Mary. While working on her Master's she was a research assistant for a year at the College of William and Mary and later worked for ICASE at NASA Langley Research Center, Hampton, Va. as a student assistant.