SYNCHRONIZATION ISSUES OF THE PARALLEL A STAR SEARCH

by

Daniel Cameron Daly

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements

for the degrees of

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September, 1993

[February 1994]

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science, Sept 13, 1993

Certified by [illegible]
Thesis Supervisor (Academic)

Certified by [illegible]
Company Supervisor (Cooperating company)

Accepted by [illegible]
Campbell L Searle, Chair, Department Committee on Graduate Students

SYNCHRONIZATION ISSUES OF THE PARALLEL A STAR SEARCH

by

DANIEL CAMERON DALY

Submitted to the Department of Electrical Engineering and Computer Science on September 13, 1993 in partial fulfillment of the requirements for the degrees of Bachelor of Science and Master of Science in Electrical Engineering

Abstract

An experimental study was carried out on the performance of synchronous and asynchronous implementations of the A* Search on a multiprocessor network. Master-Slave parallelism was used to distribute the Search among the 8 processor nodes of a transputer network. The test programs were run on 4 different types of maps. Measurements were taken in the form of percentages of time spent in computation and communication in each cycle of the search as an artificial delay in the computation phase was increased.

The results from the map tests showed that the asynchronous implementation spent a larger percentage of each cycle performing calculations rather than communicating or waiting for communications as the artificial delay was increased. This means that the efficiency of the asynchronous approach increases more rapidly than the efficiency of the synchronous approach as the computational complexity of a parallel program is increased. This was found to be true for all artificial delays on all test maps for the Master-Slave A* Search. The results might vary with different implementations and search methodologies.

Thesis Supervisor:	Dimitri P Bertsekas, Ph.D.
	Professor of Electrical Engineering
Company Supervisor:	Ed Leung

Chapter 1 - Introduction

Technological advancements over the past 40 years have allowed computers to be improved in effective performance by an order of magnitude every decade or less. Despite this tremendous rate of advancement, there are still problems which take huge amounts of time to solve by computer due to the large amount of information processed or extremely complex calculations required. Intelligent design of a computer system can lead to significant performance improvements without a need for better technology. By incorporating multiple processors into a system, performance can be enhanced greatly if the correct preparations are made and the work is split up among subsystems. There are different ways to go about building multiprocessor systems, and severe performance penalties can result from poor design. In this project a graph searching algorithm known as the A Star (A*) Search will be used to illustrate the issues that arise in the parallelization of a program.

The key aspect of the implementation of a particular task or problem is the parallelization of the problem. A normal computer would perform the task sequentially, by analyzing the current data for each step in order until all the required operations had been performed. A program running correctly implemented for a parallel system could perform different parts of the required calculations or operations concurrently, reducing the time needed to perform the processing. Ideally, a doubling in the number of processors would halve the amount of time needed to reach a solution. There are penalties involved in using a parallel approach , however, which cannot be ignored. The main difficulty occurs as a result of the process of communicating between the processors.

Since each processor is operating independently, data must be transferred from one processor to another by some sort of link. The choice of what data needs to be transferred will have a large impact on the performance of the system, since it will take a finite amount of time to transfer data from one processor to the next. If the processors spend a lot of time communicating instead of processing the data needed to solve the problem, it will take longer to solve. The parallelization approach used to implement a program has a significant impact on the effectiveness of the multiprocessor system.

There are basically three different types of parallelism that can be used. The first is algorithmic parallelism, in which the program being implemented is divided into modules, each of which can execute concurrently on different processors. The data is sent from one module to the next as each step is completed, allowing a different set of data for each processor used, in a fashion comparable to an assembly line. Each processor acts upon a section of data in a certain way, and when the data has passed through all the modules it is 'done'. This type of parallelism is fairly easy to implement, and requires a moderate amount of communication.

The second approach is geometric in nature. If the problem space is known and fairly regular, it can be divided among the processors. A good use of geometric parallelism would be for image processing. An image being operated upon could be divided into a number of identical sections equal to the number of processors, and each processor could operate without needing to communicate with the others on its own sub-image. When all the processors were done, the data would be gathered and the final image recreated. This approach is more complex than algorithmic parallelization, because the data space must be subdivided and reassembled, ideally without altering it unnecessarily.

The third and final method is called farming. It can be used when the input data is not regular enough for geometric parallelism, but can still be subdivided. Each processor operates independently of the others on its own section of the problem space. The important difference between this and geometric parallelism is that the topology of the network is independent of the application geometry. Farming is best suited for applications where the unit of work varies in complexity.

In this project, a combination of algorithmic parallelism and farming will be used to implement the A* Search on a multiprocessor system. The details will be discussed later, but the basic idea is to divide the program into two sections. The first section will handle the higher level aspects of the search algorithm and prepare data for the other processors. This section will be called the Master, and it will be run on one processor. The rest of the processors will be called Slaves, and they will receive data from the Master, perform some operation on it and return the new data to the Master. It will then operate on the collected data and prepare the next phase of the search. This implementation may not be the best for this algorithm, but it is used here as a vehicle for analysis of the communication between processors.

There are two different approaches to controlling the interaction between the master processor, which will direct the search, and the Slave processors which will examine nodes for the search. These two possibilities are defined by the manner in which they control the transfer of data, or synchronization. The synchronous approach will operate in a controlled fashion, allowing the root processor to decide when each Slave sends and receives data. This could result in performance penalties if the Slaves are left idling while the Master interacts with a particular Slave that requires more attention than the others. The alternative is the asynchronous approach which allows the Slaves to specify to the Master when they are ready to send or receive data. This will allow the Master to only interact with the Slaves that are ready for transmission of data, which should benefit performance. One side effect of the asynchronous approach, however, is that the Master will have to be informed by the Slave it is communicating with what the Slave was working on. The tradeoffs involved in the selection of the synchronization will have an effect upon the performance of the A* Search.

This thesis is intended to examine various aspects of a multiprocessor system through the parallelization of a normal sequential algorithm. The chosen algorithm is the A Star search, which is used to determine the optimal path among the arcs between two nodes, given a directed graph containing those nodes. In this particular application, the graph represents an area to be traversed by the Draper Lab simulated Unmanned Undersea Vehicle (UUV). The problem is to find the optimal path through the area given that there are obstacles in the way. The A Star search is a simple algorithm which has been around for decades. Researchers at Draper have been working on normal sequential versions for quite some time, but this parallel implementation will be significantly different.

The fabrication of simple yet advanced processors has been taken on by several different companies. The processors being used for this project are a product of the INMOS corporation, which calls each processor a Transputer Module, or TRAM. A group of these processors connected together to perform a task is called a transputer. Each TRAM typically has a main processor, connected to memory and a system of links to connect the processors together. The transputer used for the project will be described in greater detail later.

The ultimate purpose of this thesis is to examine the effects of synchronous and asynchronous algorithms. The fact that high performance transputers have only recently come into existence means that only a fairly small amount of research has been performed on actual systems. Work has been done on different possibilities of synchronous and asynchronous algorithms [3], and this project examines synchronization issues on a real multiprocessor system. The A Star Search is being implemented on a transputer system using the Master-Slave approach of parallelization. This approach may not be ideal for this search, but it serves as a vehicle for experimentation on the synchronization issues that arise in the porting of a sequential algorithm to a parallel multiprocessor system. The effects of the synchronization approach upon the performance of the search will be examined. This is done by comparing the granularity of the parallel processes with the communications overhead of each synchronization approach and the delays associated with them. The next chapter will go into greater detail on the Master-Slave approach and its effect upon the relationship between communication and computation.

Chapter 2 - Master-Slave Parallelization

The three basic types of parallelism are algorithmic, geometric, and farming. Each of these approaches can be used to transform a sequential program into one that can run on a system of multiple processors. The key difference between the algorithmic and geometric approaches is that in one case the program is divided and in the other case the data is divided. The implementation of geometric parallelism is more complex than algorithmic because the data space must be subdivided evenly so that the processors have approximately equal amounts of work to do. A less organized alternative is called farming, in which the processors each get an independent set of data, but the processor topology is not related to the problem topology. There would be a number of processors each performing the same operation on the data. Different amounts of data are sent to each processor and the complexity of the operation on the data will be variable. This approach reduces the problem of mapping data onto the processors for problem spaces that are not regular, but there is more overhead in coordinating the system to handle varied loads.

One obvious consideration of geometric parallelism and farming is the division of the data space, which will require some amount of preprocessing and postprocessing before the multiple processors can get the appropriate subsection of the data and begin work. The algorithmic parallelism in this project will be used to divide the program into two major sections. The first section will contain the code that initiates and controls the entire program, which will run on one processor, and the second section will be farmed out to the multiple processors. This combination will be known as a Master-Slave relationship. The basic idea behind this Master-Slave relationship is the master's motivation to get as many slaves as possible doing the master's work.

The master decides what work needs to be done and prepares a list of things to do. The slaves are then given the tasks by the Master in the order of their priority. When each slave is done, it reports back to the master and is given more work.

An important aspect in the transformation of a sequential program to a parallel program is the communication protocol between the processors. With an ideal data transfer system, no time would be spent establishing communications between processors and transferring data. In the real world, however, communication takes a real amount of time to initiate and perform. In the Master-Slave configuration, there is no reason for the Slaves to exchange information with each other directly, so communication is all between the Master and Slaves. This places a large burden on the part of the Master to communicate with all its Slaves. Since information exchanges must be received and processed as rapidly as possible, if there is any performance difference among the processors, the best one should be used as the Master. If there is no difference among the processors, the division of labor between the Master and Slaves is crucial. For each phase of the program, the Master will consult its list of things to do, and prepare data for a Slave. A Slave will then take the data and operate on it according to its programming, and return its results to the Master. If the time it takes a Slave to process the data is the same as the amount of time it takes for the Master to prepare the list in preparation for the Slave's next request for data, then one Slave will be enough for the Master. Since they operate at the same time, the Master will be updating the list while the Slave works. When that Slave comes back to the Master for its next assignment, the Master will be ready to give out the next task. Even if there are a hundred Slave processors available, they will never improve the performance of the program unless the Master is made faster. A different

Slave processor might be called upon to do the work each time, but still only one of the hundred would be working, and modern microprocessors don't really need time to rest between jobs.

The significance of all this is that the division of processing labor will become more effective as the Slave takes more time to complete its data processing. If the average Slave processing time is 20 times longer than the amount of time the Master needs for each communication cycle with a Slave, then 20 Slave processors could be effectively used, and the task would be accomplished approximately 20 times faster than if only one processor is used. In this example, each Slave would be processing data for 20 times as long as it spent on communication on each task, or spending over 95% of the time computing rather than communicating. As the percentage of time spent computing by the Slaves is increased, the communication percentage would decrease, and the efficiency of the Master-Slave implementation is increased. This measure of the relative granularity of the Slave work is the best way to analyze the performance of the communication implementation. The more the Slaves are working, the better the implementation. When the implementation is inefficient, the Slaves will be computing less, and communicating and being idle more.

A similar measure could be made for the Master, but the values yielded would not be a clear indicator of the performance of the Master. There are two possibilities for the results for the Master: it is either computing more or communicating more. If it is the case that the Master is computing a larger percentage of the time on each cycle, that could be read as either a positive or negative indicator. Either the processor is capable of communicating quickly, which is advantageous, or the computation for each cycle is complex, which is detrimental. The results are ambiguous. If the Master is communicating more than it is computing, it could just be that the Slaves are all being used regularly and efficiently, and the Master is in communication mode waiting for a Slave to be ready, which is the ideal situation. An alternative is that the Master and Slaves have a poor communication protocol or noisy lines and less data should be transmitted. The possible causes for variations in the computation percentage for the Master are sufficiently varied that it would be difficult to evaluate the performance of the system from the analysis of the Master's computation and communication times. Limiting the waste of time in communication while the Master is waiting for some Slave to be ready for a new task is an issue of synchronization.

There are two different approaches to the synchronization issue, synchronous and asynchronous control of information transmission. In the synchronous case, the Master processor knows which Slave should be ready to send and receive data at all times. The process begins when the Master sends data to the first Slave and ends when the last Slave is given its data. The Master processor then stops processing until the first Slave has completed its operations and is ready to return data. When that point has been reached, they exchange data and the first Slave goes to work on its next set of data. The Master then waits for the second Slave, which should be done by this point if the processors are taking approximately equal amounts of time for each set of data. This process continues until the program terminates. With this approach, the Master can easily keep information on what it told each Slave to do, and no extra communication is required when the Slave is ready to respond. The major disadvantage to this system is that variations in the Slave processing time will cause the entire system to perform more slowly. If a particular Slave takes longer than the others to complete the needed analysis, the Master will be stuck waiting for that processor to finish, adding to the communication delay time. The alternative to this is called asynchronous operation.

In an asynchronous system, no assumptions are made for the Master program about the state of the Slaves. Whenever it is ready to receive and transmit data, it polls the Slaves to determine which of them is ready as well, and initiates communication with that Slave. In this fashion, the Slaves will have greater control over the operation of the system. If a particular Slave had an extremely large amount of data to process for whatever reason, it might take several times longer to process than the other Slaves, but the Master would have no real knowledge of that. Other Slaves could continue to perform smaller tasks until the heavily loaded Slave was done, without blocking the Master. The major side effect of the asynchronous approach is that the Master and Slaves need to transmit more information than in the synchronous case, because the Master doesn't know what set of data the Slave was working on and needs to be 'reminded' before it can accept the data from the Slave. The asynchronous approach would probably be more useful for systems with more variation in the amount of time each Slave takes in its computation, while the synchronous approach would be better suited for applications where the data could be split up into fairly even groups, to keep all the Slaves working and the Master constantly switching from one to the next.

The purpose of this project is to examine the effects of the different synchronization approaches. A combination of algorithmic parallelism and farming has been combined in a Master-Slave configuration to create a multiprocessor system to test these different approaches. A search algorithm called the A* Search will be the particular program implemented as a vehicle for experimentation, and it will be explained and the implementation described in the next chapter.

Chapter 3 - The A Star Search

The A Star is a basic search algorithm which can be found in computer science texts as an introduction to searches. It was first detailed in a 1968 paper by Hart, Nilsson and Raphael [1], although the exact description used in this project is from E. Rich's Artificial Intelligence [2]. In order to understand the parallelization of the A* Search, it is first necessary to understand the search itself.

The object of the A* Search is to find the optimal path defined as a sequence of arcs from a starting node to a goal node of a directed graph. The arcs and nodes are assigned a cost or weight which describes the difficulty in traversing the arc or entering the node. The optimal path is defined as being the path from start node to goal node which has the lowest possible total cost. It is the job of the search algorithm to minimize the total cost. This is done by a carefully ordered process which attempts to check every possibility.

The search begins when the start node is examined by the program and its neighbors, or successors, recorded. After the first node has been examined, an appropriate heuristic function is applied to the successor nodes to determine which represents the most promising direction towards the optimal path. This process is repeated on the most promising successor node. A more precise definition of the A* Search follows.

Each node of the directed graph being searched has several characteristics that are either fixed or changed as the search algorithm is applied. The fixed attributes are the coordinates of the node, a value which represents the cost of entering that node, and costs for each of the eight arcs departing from the node in the vertical, horizontal and 4 diagonal directions. The modifiable characteristics include a pointer back along the best path that traverses that node, a value describing the cost to reach it (G), a heuristic estimate of the distance to the goal (H), and a total cost estimate for that node and the course containing it(F = G + H). Obviously, the last four elements of the node are only relevant when the search function has analyzed the node as part of a search for the optimal path. There will also be two lists associated with the search, the Open and Closed lists. The Open list contains all the nodes which have been generated and have had the heuristic function applied to them, but which have not had their successors generated. This list will act as the raw material for the next step of each iteration of the search, as the most promising element of the Open list will be examined and its successors generated. The Closed list is a collection of all the nodes that have had their successors generated.

The description of the potential of a particular path is represented by the value of F for a particular node. F is the sum of G, the exact cost of reaching a particular node, and H, an estimate of the cost to get from the current node to the goal node, and is therefore an estimate of the total cost of the best path containing the node. The estimation of H is where knowledge of the problem domain can be exploited, by modifying the estimate for known conditions. It is important that H always be an underestimate of the actual cost to reach the goal. If it were an overestimate, then the search might decide on a path which is not the optimal solution, since another path which is actually less costly will seem more expensive and not be explored. The more accurate the estimation of H is, the more rapidly will the Search converge towards the optimal goal. If H is a perfect estimate of the actual cost, the algorithm should converge quickly to the goal, since any node along the optimal path will have the lowest value of F and should be immediately obvious, provided no back tracking occurs. So it can be seen that a better estimate can yield a quicker search. The values of G and H must not be negative, otherwise loops of negative cost could exist which the algorithm would get trapped in, since each pass through the loop would make the current path more and more ideal and yet no closer to the goal.

The roles of G and H can also be modified to change the nature of the Search easily. By incorporating G into the value of F, which is the estimate of the cost of the path which includes the current node, we can affect the choice of which node to expand next by considering not only how good the node itself looks, but also on the basis of good the path to the node was. If the optimal path is desired, this is very important. If all that is needed is a path from start to goal node, the value of G can be set to 0, thereby forcing the Search to always chose to examine the node that seems closest to the goal. To find the path that takes the fewest number of steps and ignores the cost for each arc or node, the value of G can be set to 1. If the arc costs and weights are used then the path determined will be the optimal one. The value of H can be modified as well, depending on the reliability of the estimate, as mentioned previously. If it is set to 0, then the search will be controlled purely by the cost of reaching a node.

Here is a more mathematical description of the A* Search algorithm [4]: Assume a directed graph A has a cost of a_{ij} for each arc (i, j) and w_j for each node. The cost of a path $(i, i_1, i_2, \ldots, i_k, 1)$ from node i to node 1 is defined to be the sum of the arc and node costs $(a_{ii_1} + w_1 + a_{i_1i_2} + w_2 + \ldots + a_{i_k1})$. Furthermore assume each $a_{ij} \geq 0$ for all (i, j) and $w_j \geq 0$ for all (i) in A. The algorithm makes use of a set of nodes L, a start node s and a scalar g_i for each node i.

Initially $L = \{s\}$, g_i = infinity for all $i \neq s$ and $g_s = 0$. Let $h_i \geq 0$ be a known underestimate of the shortest distance from node $i <> 1$ to the destination node 1 and let $h_1 = 0$. Perform the following steps:

Step 1: Remove a node i from L. For each j neighbor of A(i), if $g_i + a_{ij} + w_j < \min\{ g_j, g_1 - h_j\}$ then set $g_j = g_i + a_{ij} + w_j$, and if $j \neq 1$ and j is not in L, place j in L. if j = 1, then stop.

Step 2: If L is empty stop; otherwise go to Step 1.

The implementation of the A* Search on a multiprocessor system is done with the previously explained Master-Slave configuration. The Open and Closed lists are placed on the Master, along with the parts of the program that deal with the moving of nodes to and from those lists and the procedures that decide which node has the best F value. On each cycle after initialization has been completed, the Master will accept from a Slave the data for all the neighbors of the node the Master last gave to the Slave, and give to that Slave the node with the current best F value, not including the new nodes just processed by that Slave. By giving the Slave a preprocessed node, the Master avoids the need to delay the Slave while it calculates the new best F valued node. To keep the communication as minimal as possible, both the Master and Slave programs are written to perform all needed calculations before initiating communication, then transmitting all the needed data in as quick a burst as is possible. Obviously, communication with the Master will become a bottleneck once enough Slaves are added, so it will become necessary to add subprocessors to the Master which can assist with the routing of communications. That addition further complicates the issue and will not be dealt with explicitly here. The addition of subprocessors should be transparent to the synchronization and parallelism issues in any case if implemented correctly.

Once a Slave receives a node from the Master, with the appropriate value for G, the cost to the node so far, the Slave will determine the nearest neighbors and the values of G, H and F associated with them. It will then signal to the Master that it has completed computation and will wait for the communication to begin. Of course, the Slaves each need a copy of the full map of nodes and weights, which they will receive from the Master as part of the program initialization. In an ideal program, the A* Search would not be improved by this parallelization, because each cycle of the program is really just a single addition and comparison for each neighbor. Once real world delays are taken into account, however, the process of determining the neighbors and calculating their G, H and F values becomes a more time consuming process, and the Search will benefit from the division of labor represented by the Master-Slave approach. Although there might exist better ways to incorporate parallelism into the A* Search algorithm, this allows an easy comparison between the two different types of synchronization.

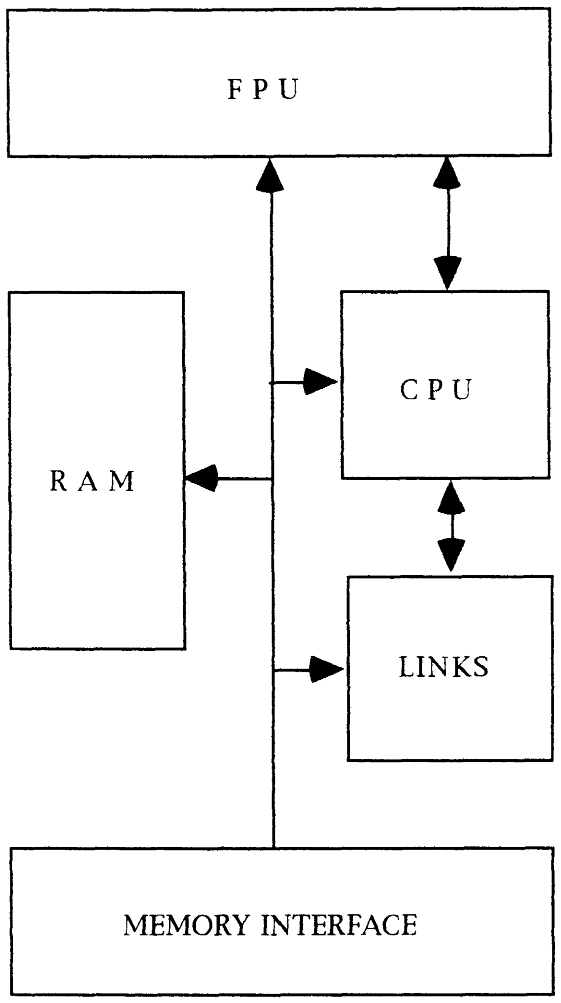
The synchronous approach is the most easily implemented. The Master processors simply runs in large loop, with a sequential ordering of Slave communication. Each Slave is accessed in the same order, meaning that the Master can store data for each Slave and always know when the appropriate Slave is being communicated with. The precise communication control is handled by a system of Channel communications between processors which can send data in or out. In any communication instance, one processor will activate a Channel out with a piece of data waiting to travel over it, and wait until the target processor signals ready by sending a Channel in notification. At that time, the data will be transmitted and communication terminated. If no Channel in response is received, the sending processor will wait forever, or until a certain amount of time has passed if that option is used. These timed

Channel communications could be very useful to add fault tolerant capabilities to the system for future development.

In the asynchronous case, the Master uses a special procedure that selects from a provided list whatever Channel is ready to communicate, and exchanges the required data. The Slaves are the exact same as in the synchronous case, except now more information needs to be transferred since the Master doesn't already know what node the Slave was given in the last cycle to expand. With this system, the Slaves will only request communication with the Master when they have completed their work and are ready for a new node to expand. The Master is unaware of what the Slaves are doing, it just takes data and hands out a new node every time it is aware that one of the Slaves is ready for it. Of course, in both cases the Master is always ready for a Slave to notify it that the solution has been found, in which case it ceases normal function, assembles the shortest path solution and sends it out to whatever process activated the entire program. The next chapter details some of the real world aspects of the project, such as the nature of the hardware the search will be implemented on and the possible applications.

Chapter 4 - Transputer System

The hardware and most of the software to be used are produced by the INMOS corporation, which has been at the forefront of parallel systems research since they introduced the IMS T414 transputer in September 1985. The INMOS transputer family is a range of system components, each of which combines 16 or 32 bit processing, fast memory and interconnection links in a single VLSI chip. The latest generally available version is called the T805, and it differs from the T414 primarily in that a microcoded floating-point unit (FPU) which operates concurrently with and under the control of the CPU has been added and the processor can run at a higher speed. Both the T414 and the T805 are 32 bit processors. The transputers created by INMOS take better advantage of some aspects of VLSI technology than some more conventional processors do. One important consideration in VLSI design is that communication between devices on separate chips is much slower than communication within a device. By including the central processor, the FPU, 4 kilobytes of fast RAM, 4 serial links for connection with other processors and a memory interface for use of larger external memory, the transputer is able to process information more quickly than if the devices were mounted on separate chips. A block diagram of the T805 is shown below.



When the T805 is mounted onto a small circuit board and 4 MB of RAM are connected to it via the memory interface, it is called a transputer module, or TRAM. For this project, the system consists of a group of eight TRAMs located on a board which can change the serial links between them on command. In this way, the processors can be connected together in a pipeline, ring or whatever configuration is needed for the parallel operation being performed. The TRAMs run at 25 and 30 MHz (4 of each) and the links can move data at up to 20 megabits per second. The TRAM in slot 0 on the board acts as a link for connecting the board to outside systems, which are connected via a special device, the B300 Ethernet Link Box, on the Draper Lab ethernet. With the B300, any computer on the lab network can access the transputer and run programs on it, if it has the correct software installed. The machine used for this project is a Sun SPARCStation which is accessed through a Macintosh SE acting as a terminal. All of this equipment has been provided for the project by Draper.

Chapter 5 - Experimentation

5.1 - Experiments

As has been previously stated, the main purpose of this thesis is to explore different types of synchronization using the A* search. The two possibilities examined here are synchronous and asynchronous implementations of the Master-Slave relationship. The basis of comparison of the synchronization approaches is the percentage of time spent in computation and communication by the Slaves in either case. As the efficiency of the implementation increases, so will the percentage of time spent computing by the Slaves. Since the Slave is either computing or communicating at any time and no other states exist, the percentage of computation time was used for analysis purposes and the percentage of communication time can be deduced by subtracting the computation percentage from 100%.

In order to examine the percentage of time spent in computation, it was necessary to vary the amount of time actually spent computing by the A* search. Varying the communication time could produce the same results, since the target information is simply a ratio, but in this implementation the computation time was increased by the addition of a delay loop in the Slave program. The delay induced by this loop could be set to either a single value or programmed to be a randomly selected value over an evenly distributed range. All of the Slaves were modified to keep track of the percentage of time spent computing, to insure that the observation of the data does not change the results by handicapping one Slave, but only one was used to actually print the final result at the end of the search. Both the synchronous and asynchronous versions of the program were run in debugging mode before the actual tests to insure that the programs were behaving in the proper fashion and that Slaves were being selected with the correct synchronization method.

There were two main series of tests, the mine series and the random node weight series. In the mine series, each node had a 25% chance of having the node cost set to 100, to represent a mine, and a 75% chance of having the cost set to 0, to represent free space. The horizontal and vertical arcs connecting adjacent nodes had a cost of 1, while the diagonal arcs had a cost of 1.41. These costs could be used to simulate an environment which is uniform except for a number of potentially dangerous spots which should be avoided. The second series was the random node weight series. It was characterized by a random integer cost for each node from the uniformly distributed range 0 to 9. The arc costs for the horizontal and vertical arcs were set to 1, while the diagonal arcs were set to a number large enough to make them unused. This is done because the diagonal arcs represent a shortcut of sorts, in that only one node entry cost is added when a diagonal arc is traversed instead of the two which would be added if the equivalent vertical and horizontal arcs were covered. This effectively makes the diagonals half as expensive as the horizontal and vertical equivalent, so the diagonal course will almost always be taken and the algorithm will terminate quickly. In this series, the desired effect was to run a complex test that will not find a solution quickly, so the diagonal arcs were removed. The random node weights caused the search to examine a large number of nodes while finding the optimal path, which should have provided more data for the Slave computation percentage. It is important to remember that this is not a performance test of the A* algorithm, so the series were chosen to obtain data on the performance of the synchronous and asynchronous implementation of the Master-Slave parallelism.

Each series was divided into two sections, a set delay section and a random delay section. In the set delay section, the delay was preset to a particular value for all Slaves for that run. In the random delay section, a maximum delay was specified and the actual delay for each node expansion for each Slave was randomly selected from a uniform range from 0 to the specified maximum for each step of the search. The asynchronous implementation would probably perform better with the random delay, since it was designed to take advantage of variable computation time for each Slave.

Each of the two sections of each of the series was performed in the same fashion. The delay set or maximum delay possible was varied in a logarithmic fashion to cover a range from around 200 all the way up to 100,000. For each delay setting, the software was run on the 8 processor net with 5 different map sizes for each of the 2 synchronization possibilities. The percentage of time computing for each cycle of the search was averaged into a mean percentage for the entire run of the search at that map size, then the mean percentage for each of the 5 map sizes was averaged into a final computation time percentage for each delay. These results were then plotted and their significance analyzed.

The graphs on the following pages show the results of the experiments. Figures 1 and 2 are for the mine series, figure 3 and 4 display the results of the random node weight series. The odd numbered figures present information for the set delay sections for each series, while the even numbered figures are for the random delay series. An explanation of the results follows the graphs.

figure 1 - Mine Series Set Delay Section

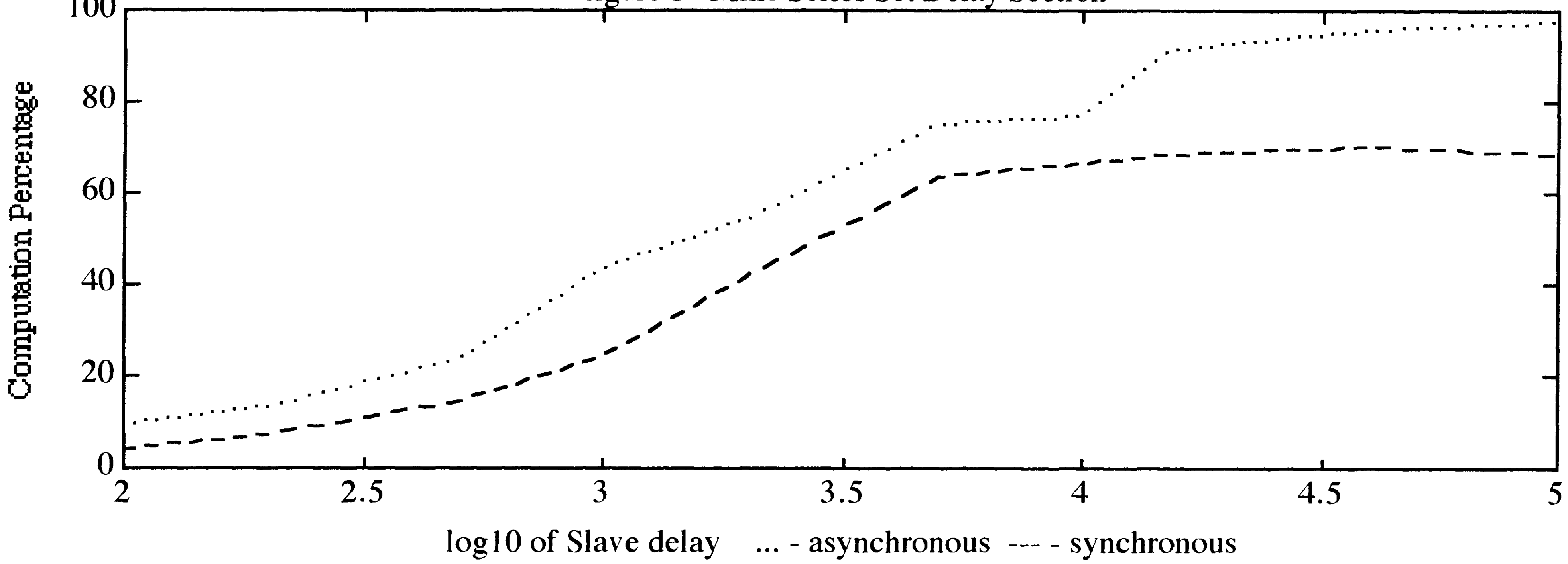


figure 2 - Mine Series Random Delay Section

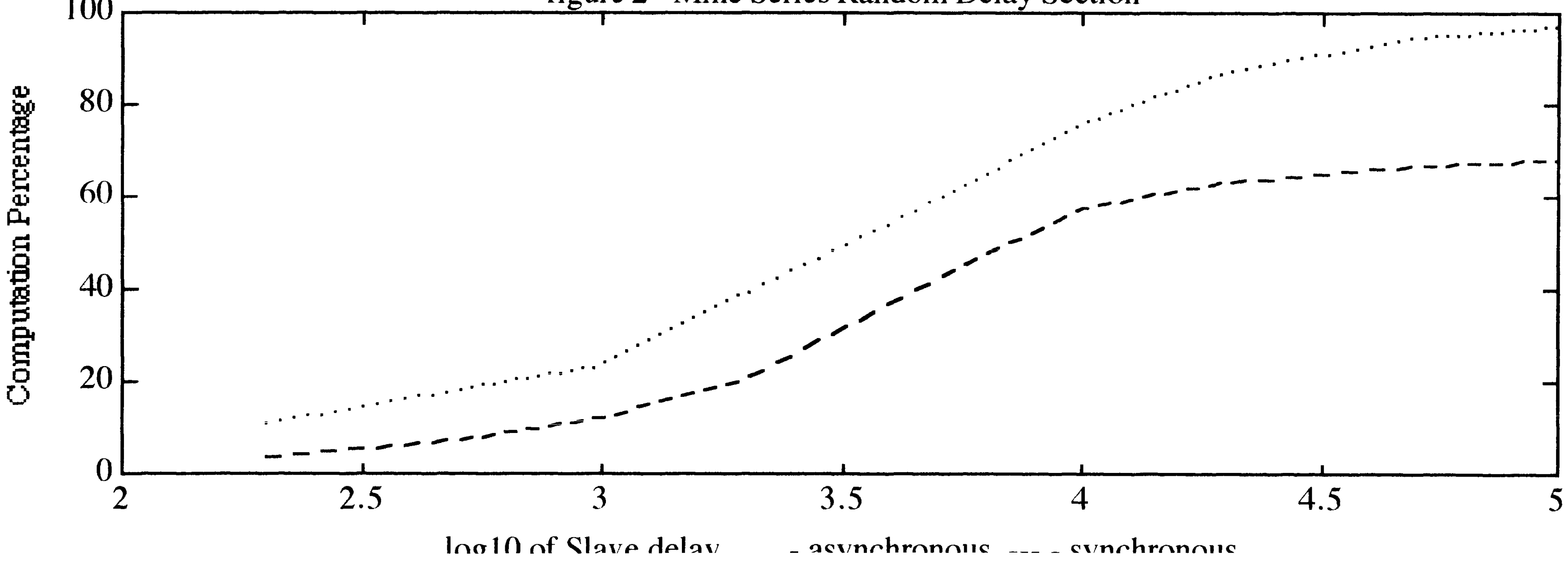
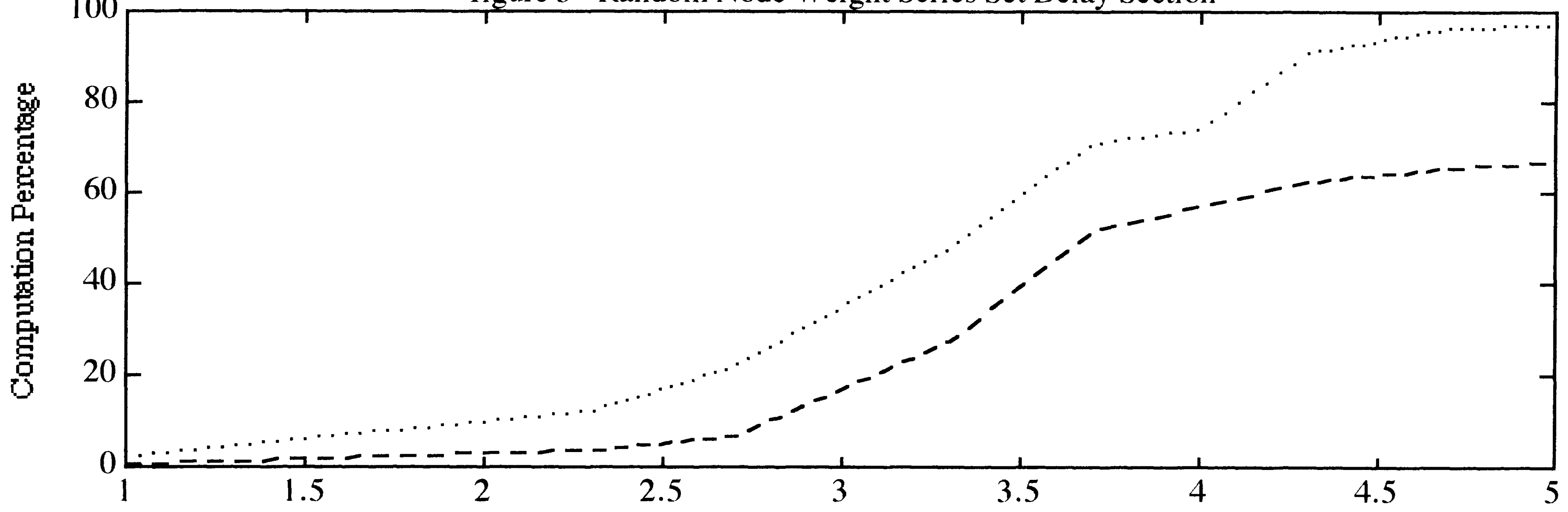
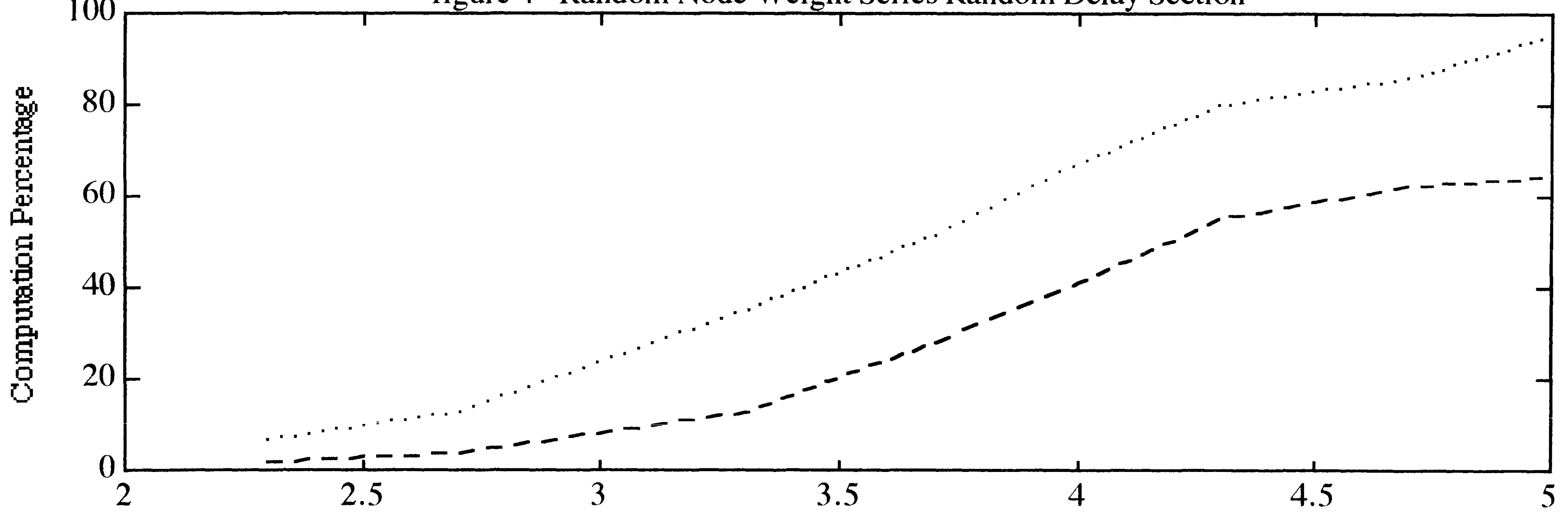


figure 3 - Random Node Weight Series Set Delay Section



log10 of Slave delay ... - asynchronous --- - synchronous

figure 4 - Random Node Weight Series Random Delay Section



log10 of Slave delay ... - asynchronous --- - synchronous

Section 5.2 - Results

In both series, the results were very similar. In all cases, the asynchronous case turned out to be the most efficient implementation of the Master-Slave relationship. The computation percentages were fairly close for the lower delays in some of the sections, but as the Slave delay was increased the asynchronous approach became more effective. Since each Slave is taking more time in computation with the increase in delay, it was less likely that more than one Slave was ready to receive a new task from the Master at any one time. This means that as soon as a Slave finished the expansion of one node, a new one was available from the Master. In most cases with a large delay, the only action of the communication phase was the actual communication. No processing time for the Slave was lost waiting for the Master to respond. In the synchronous case, the Master rotates through the Slaves, waiting for each to complete its work before communicating and proceeding to wait for the next Slave. There will always be delays as the Master waits for each Slave to finish. In the random delay section of the random node weight series, the synchronous approach can be seen as being even less efficient than in the set delay section due to the variable delay introduced.

It is not obvious from the graphs, but in both series the general effect of the random delay was to cut the computation percentage to approximately one half of the set delay equivalent. This makes sense when it is considered that the average of a uniformly distributed group of numbers from 0 to some maximum will be half of that maximum. For example, a set delay of 10,000 results in the same computation percentage as a maximum random delay of 20,000. Given that there are several thousand computation percentages averaged for each delay, this seems statistically reasonable.

Chapter 6 - Conclusions

The experimental results for the four different types of test maps show that are very similar in terms of computation and communication percentages. In all cases, the asynchronous approach is more efficient than the synchronous. This is due to the fact that the knowledge of which Slave was being accessed only yielded mild communication savings in this implementation of the A* search. A different implementation or a different program might have produced different results, an area which could be a source of future study. With the maximum delay of 100,000, Slaves in the asynchronous case were communicating or idle and waiting for communication only 3% of the time. This means that up to approximately 30 Slaves could be used efficiently by this implementation, if the communication percentage doesn't increase. In the synchronous case with a delay of 100,000, the 6 Slaves that are connected aren't being used efficiently, as shown by the fact that each of them is spending over 30% of the time communicating and waiting for the Master. In both cases, however, extremely low delays reveal that the Master-Slave implementation of the A* search is inefficient, since the Slaves spend almost all their time in communicating with the Master or waiting idle for it to get to them. This is due to the fact that the transputer modules are capable of processing the relatively simple A* search node expansions much more quickly than they can communicate. A much more computation intensive program would be a better choice for future examination of the synchronization issues of parallel programming.

References

[1] Hart, P.E., N.J. Nilsson, & B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on SSC*, Vol. 4, 1968.

[2] Rich, E., Artificial Intelligence, University of Texas at Austin, McGraw-Hill Inc., 1983.

[3] Z Cvetanovic, and C. Nofsinger, "Parallel Astar Search on Message Passing Architectures", *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences*, 1990.

[4] Bertsekas, Dimitri P., and Tsitsiklis, John N., Parallel and Distributed Computation, Englewood Cliffs, New Jersey, Prentice Hall, 1989.

Hamilton, Andy, "Some issues in the scientific-language application porting and farming using transputers", *INMOS Technical Notes*, Vol 53, INMOS Corporation, 1989.