IMPLEMENTATION AND UTILIZATION OF A HETEROGENEOUS MULTICOMPUTER CLUSTER FOR THE STUDY OF LOAD BALANCING STRATEGIES

by

PER ANDERSEN B.ENG

A THESIS

IN

COMPUTER SCIENCE

Submitted to the Graduate faculty

of Texas Tech in

Partial Fulfillment of

the Requirements for

the Degree of

MASTER OF SCIENCE

Committee Members:

Dr John Antonio (Chairperson)

Dr. Noe Lopez-Benitez

Dr. Donald Gustafson

ACKNOWLEDGMENTS

I could not have carried out this thesis without the support and assistance of many people. The support and assistance of my thesis advisor, Dr. John Antonio, has been invaluable. His patience and continued good advice were greatly appreciated. I would also like to recognize and thank my committee members Dr. Donald Gustafson and Dr. Noe Lopez-Benitez, for their support and encouragement.

I would also like to thank the faculty and staff of Computer Science; their help has been invaluable. In particular, I would like to thank Fred Dautermann for his cooperation and support in providing a large part of the hardware that went into the cluster.

Finally I need to thank my wife Sue and my two sons, Matthew and Timothy, for putting up with me during this period. This master's thesis has kept me away from them for many long hours and during that period I received their full support and much needed encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Multicomputer clusters of workstations can be characterized based on many different parameters and features; examples include workstation types, network types, and distributed computing model. The workstation types and whether or not they are binary compatible can be used to define, in a sense, whether the cluster is homogeneous or heterogeneous. This characteristic has a major impact on the ability to dynamically share processes (e.g., task migration) and efficiently share data. However, even binary compatible machines can be considered heterogeneous if they have different processors, different memory configurations (e.g., with or without cache), different amounts of memory, different internal bus implementations, and/or different clock rates. The network type influences the communication strategies and load sharing schemes that should be utilized. For example, strategies to make effective use of an FDDI ring network could be quite different from those that make effective use of a bus-based Ethernet configuration. Finally, the distributed computing model impacts the design of the application program. For instance, a message passing model gives the programmer more explicit control of data distribution and communications between machines than does a shared memory model. A shared memory model creates a virtual shared memory space in an attempt to hide some of the primitives typically associated with a distributed memory model.

The focus of this thesis is the implementation and utilization of a heterogeneous multicomputer cluster (HMC) based on a message passing model. The classic single visit Traveling Salesman Problem (TSP) was selected to exercise the HMC using various search techniques and load balancing strategies. Empirical studies are conducted that compare performance for various load balancing techniques and HMC configurations.

A vast amount of performance data is collected and analyzed. The impact of varying parameters of the load balancing schemes, TSP, and HMC is studied. This research provides a characterization of the HMC constructed. This characterization will be important for future users of the HMC as different applications are implemented.

1.1 Document Overview

Chapter II is a summary of the literature review for this thesis. Besides discussing some general motivations behind multicomputer clusters, overviews of existing work in parallel search algorithms and load balancing schemes are covered. Overviews of the Linux operating system (a freeware UNIX operating system), Local Area Multicomputer/Message Passing Interface (LAM/MPI, the parallel development software utilized here), and the system hardware are provided in Chapter III. Chapter IV describes the particular load balancing strategies implemented for the empirical studies and provides insight into the general advantages and disadvantages of these strategies. The methodology used to test and collect data on the HMC and the rationale for these tests are presented in Chapter V. An analysis of the data collected and

conclusions are provided in Chapter VI. Representative source code (for one of the four programs developed for this thesis) and details on related MPI software packages are found in the appendices.

CHAPTER II

LITERATURE REVIEW

2.1 Multicomputer Clusters

There are at least two advantages to constructing a cluster of workstations instead of, for example, purchasing a parallel supercomputer. First, a cluster allows users to take advantage of idle workstations. Gropp *et al*. [12] use the phrase "Supercomputers At Night" or SCANs when describing workstation clusters. Cost is another reason: purchasing tens or hundreds of off-the-shelf workstations or personal computers can be orders of magnitudes cheaper than purchasing a single supercomputer having comparable peak performance. Pfister [23] points out that the price/performance ratio of using small off-the-shelf computers in a cluster is very attractive. Gropp *et al*. states that "It's easier to add more oxen to pull a wagon than to grow one giant ox" [12].

There are many different network types available to support a cluster of workstations. Dietz [6] gives a summary of various network configurations for possible multicomputer installations, including estimates of cost and performance.

Two popular distributed computing models were reviewed for this thesis: Parallel Virtual Machine (PVM) [10] and Message Passing Interface (MPI) [7]. Geist *et al*. [9] discuss the advantages and disadvantages of PVM versus MPI, and suggest that the principle advantage of MPI over PVM is speed. However, Lawton *et al*. [17] have generated results that show PVM out-performing MPI. These apparently

conflicting findings are probably a good indicator that there are plenty of opportunities to further investigate the performance of clusters or workstations under these two (and perhaps other) models. Although these opportunities for investigation exist, the focus of this thesis involved the implementation and utilization of a multicomputer cluster on the UNIX operating system Linux [35] and the Ohio Supercomputer Center's implementation of MPI, which is called LAM (Local Area Multicomputer) [19].

2.2 Parallel Search Techniques

There are three processing phases associated with implementing a parallel search algorithm on a multicomputer cluster that are considered in this thesis: the startup phase; the working phase; and the shut-down phase [13]. For the discussion in this chapter, no particular search application is assumed (More details on the particular search implemented for this thesis, for solving the Traveling Salesman Problem [3], are given in Chapter IV). The only assumption here is that the search is conceptualized as a search tree in which the root of the tree represents the starting point of the search, the internal nodes of the tree represent partial solutions to the search, and the leaves of the tree represent candidate solutions to the search.

2.2.1 The Startup Phase

Within the startup phase, four possible initialization methods are described by Henrich [13]: root initialization, enumerative initialization, selective initialization, and direct initialization. Root initialization is the simplest and most common method of

starting a parallel search. Root initialization starts at a single "root" workstation and expansion of the nodes (of the search tree) continues on the root workstation until the load balancing scheme is activated (load balancing is described in Section 2.3).

Enumerative initialization is similar to root initialization except that the root node of the search tree is broadcast to all workstations (not just one "root" workstation). Workstation *p* then expands the *pth* node of the search tree. Strategies are required for handling situations where more nodes exist than workstations.

Selective initialization starts the same as enumerative initialization except each workstation is restricted to generating only one path to a certain depth within the search tree. The advantages of this method is less communication overhead; the disadvantage is poor node selection.

In the fourth method, direct initialization, nodes are generated to a depth that allows each workstation a unique root node of a sub-tree. If there are more nodes than workstations, then a workstation will take on multiple nodes.

The general advantages and disadvantages (in qualitative terms) of each of the four startup phase types are compared in Table 2.1 [13]. Communication overhead indicates the amount of necessary communication between workstations during the startup phase (too much communication is considered poor). Duplication of effort measures the number of multiple processed identical nodes, i.e., to what degree the workstations are working on distinct nodes. Idle workstation level is an indicator of how idle the workstations are (too many idle workstations is considered poor).

Table 2.1: Comparison of Four Types of Startup Phases [13].

Initialization	Root	Enumerative	Select	Direct
Communication overhead	Poor	Good	Good	Good
Duplication of effort	Good	Poor	Fair	Good
Idle workstation level	Poor	Good	Fair	Good
General purpose	Good	Good	Good	Poor
Branch pruning	Good	Good	Good	Poor
Best-first node selection	Good	Good	Poor	Poor

General purpose is an indication of how well the initialization method works with all search problems. Branch pruning indicates how well the method handles search trees with non-constant branching factors or whether or not some branches can be pruned due to poor cost estimation. The last criterion indicates how well the nodes have been selected in a best-first fashion at the end of the startup phase.

2.2.2 The Working Phase

During the working phase (which follows the startup phase), as a workstation completes it's work, it has to either request more work from busy workstations (idle initiated) or respond to requests to take work from busy workstations (busy initiated) [27]. Two possible methods of dividing work between workstations for search applications are: a queue-sharing method [34] and a fixed work packet method [25]. Queue-sharing is the method investigated in this thesis.

Queue-sharing is divided into two possible sub-methods. In the first, a busy workstation transfers it's entire queue when it receives a request for work. A rationale for this is relevant for heterogeneous clusters, where it is (possibly) more likely that powerful machines finish work sooner.

In the second, a busy workstation splits the queue and transfers only part of the queue. How to split a queue is an interesting problem. When there is a wide range in performance with respect to the workstations, like in the HMC implemented in this thesis, it may become beneficial to split the queues based on workstation capability. The results of empirical studies involving different queue-splitting strategies are covered in Chapter V.

2.2.3 The Shutdown Phase

The shutdown technique used depends on the type of search algorithm that is implemented. If an exhaustive search is implemented, shutdown occurs when a selected machine detects that all other machines have finished processing. For other types of search algorithms, such as an non-exhaustive search, one workstation may broadcast a termination message to all the other workstations when it determines a solution has been found.

2.3 Load Balancing Schemes

Load balancing, which is activated during the working phase of a search, can be divided into two general methods: static and dynamic. Briefly, static load balancing

involves only one initial assignment of tasks (i.e. partitions of the search space) to workstations before program execution begins. In dynamic load balancing, tasks can be redistributed to workstations during program execution.

The advantage of static load balancing is that the overhead cost is incurred only once, at compile time [26], resulting in a more efficient execution environment during the working phase. One disadvantage of static load balancing is that it generally involves solving a NP-complete scheduling problem to arrive at an optimal distribution of the load. Therefore it is generally not practical to pre-determine an optimal task distribution before execution. A second disadvantage is the complexity associated with determining the actual amount of work associated with each task (e.g., consider how to partition an unbalanced search tree).

Dynamic load balancing has the advantage over static in that it is adaptive and the system does not require a complete characterization of the run-time behavior of the application before execution. This makes dynamic load balancing a good choice for systems consisting of a heterogeneous network of workstations. Some possible disadvantages of dynamic load balancing include:

1. synchronization requirements;
2. overhead associated with setup for task migration; and
3. overhead associated with the actual transfer of the task between processors.

Three different methods of dynamic load balancing are investigated in this thesis [15]:

1. asynchronous round robin (ARR);

2. random polling (RP); and

3. global round robin (GRR).

In ARR, each workstation maintains its own local variable that identifies a target for the next work request. In RP a workstation randomly selects a processor when it needs to make a work request. GRR maintains a single variable (stored on a selected workstation), which is requested by a workstation when it wants to make a work request.

The total load on a workstation, which includes the load created by the parallel application and "background" load, plays a major role in the workstation's effectiveness in participating in a multicomputer cluster. Cermele *et al.* [2] make the point that for a heterogenous cluster of workstations, the cluster has a "relative available capacity" for processing. This relative available capacity is based on the cumulative available capacity of the workstations for parallel applications. To determine the available capacity on each workstation Cermele *et al.* investigated two methods, a passive (i.e., static) method and an active (i.e., dynamic) method. The passive method is attractive because it avoids the overhead costs of the active method; however, the active method does a better job potentially of estimating the actual load at any given time.

For this thesis a passive approach to estimating the capacity of each workstation is employed because the HMC was tested as a dedicated system. Passive

load estimates are derived from executing a significant portion of the application code on a single workstation, excluding code related to communications and synchronization between processes.

CHAPTER III

HMC SYSTEM OVERVIEW

In this thesis, two different types of networks are investigated: a bus-based Ethernet (10base2) and a switched Ethernet (10baseT). Part of the research will focus on the advantages and disadvantages (in terms of measured performance) of using these two types of networks. The workstations in the HMC (heterogeneous multicomputer cluster) are Intel-based platforms including, 386, 486 and Pentium based systems. Linux was chosen as the operating system and LAM/MPI as the parallel application development package. These two software distributions are over viewed in Sections 3.1 and 3.2, followed by an overview of the HMC hardware configuration.

3.1 Overview of the Linux Operating System

The Linux operating system grew out of a desire to provide users with a low cost UNIX operating system that ran on off-the-shelf personal computers. In 1991, Linus Torvalds [35] started developing Linux; after an early beta release, other developers began to assist in the development of Linux. In a very short period, thousands of programmers all over the world were developing and troubleshooting the Linux operating system. Many ports of well-known UNIX shells and tools were added to Linux. Eventually, after several years of development as a beta release, Version 1.0 of Linux was released in 1994.

A brief description of the current version of Linux, published by the *Linux*

Journal [14], is quoted below.

> Linux is a multi-user, multi-tasking operating system that runs on many platforms including Intel processors, 386 and higher. It implements a superset of the POSIX standard...
>
> This complete operating environment includes:
>
> - hundreds of programs including compilers, interpreters, editors and utilities;
> - tools that support connectivity, including Ethernet, SLIP and PPP, and interoperability;
> - reliable, production releases of software, as well as cutting-edge development versions;
> - a development team located around the world working to make Linux portable to new platforms, as well as supporting a user community as diverse in needs and location as the development team itself. (p. 5)

In addition to being a very complete and mature operating system, Linux is free and can be copied and redistributed without paying any royalty fees. Linux is licenced under the Free Software Foundation's General Public License.

The latest Linux kernel is Version 2.0.30, which is the kernel that is used on all of the systems in the HMC. Many of the daemons and other programs that normally run on a Linux installation were disabled or were not installed at all. For example, sendmail and the line printer daemon were both prevented from loading. By keeping all non-essential applications from loading, workstation resources available for HMC operation and parallel application execution were maximized.

3.2 Message Passing Interface

3.2.1 Background on MPI

The development of the Message Passing Interface (MPI) standard grew out of a desire by researchers in both academia and industry to provide a standard that would motivate and promote the development of parallel applications. In the 1980s and early 1990s, it was apparent that there existed the potential for many vendors to develop and release message passing interfaces that would be proprietary and restrictive. In an attempt to discourage this trend, researchers meet in April 1992 at the Workshop on Standards for Message Passing in a Distributed Memory Environment in Williamsburg Virginia. Over 80 people from 40 organizations participated in the development and approval of a standard for message passing. In November 1992 the first preliminary draft for the standard was released; this was followed by the release of the first official MPI Standard, Version 1.0, in June 1994 [28].

Since the release of the MPI standard, a number of commercial and publically available (or freeware) versions of MPI have been released. Many of these packages both meet and exceed the functionality defined in the standard. The latest version on the MPI Standard is Version 1.1.

Some of the key features in the MPI 1.1 Standard are as follows [18]:

1. point-to-point communications;
2. collective operations;
3. process groups;
4. communication domains;
5. process topologies;

6. environmental management and inquiry;

7. profiling interface; and

8. bindings for both Fortran 77 and C.

Two of the freely available implementations of MPI that were reviewed for this thesis are LAM/MPI and MPICH. MPICH has the advantage of a Win32 version. This version, because it installs under Microsoft Windows, can run on a large percentage of the computers in the world. LAM/MPI runs on UNIX systems only. For this thesis, LAM/MPI was chosen because it readily installs on Intel platforms running the Linux operating system. Appendix A lists many of the other available implementations of MPI and the supported platforms.

3.2.2 LAM/MPI

LAM/MPI was developed and is distributed freely by the Ohio Supercomputer Center (OSC) at Ohio State University. It follows the MPI 1.1 Standard as proposed by the MPI Forum and includes the following enhanced features [19]:

1. extensive monitoring and debugging tools;

2. heterogeneous network support;

3. dynamic processing element (PE) addition and deletion;

4. PE fault detection and recovery;

5. MPI extensions and LAM programming supplements;

6. fast client-to-client communications;

7. dynamic MPI process spawning; and

8. is freely available under a GNU license.

MPI has been described as complex by some and large but not complex by others [12]. The MPI 1.1 Standard has 125 functions and LAM/MPI adds an additional 40 functions to this list. Although 165 functions is a lot, many message passing applications can actually be developed using only six primary functions. These functions are:

MPI_INIT	Initialize MPI;
MPI_COMM_SIZE	Find out how many MPI identifiers are defined;
MPI_COMM_RANK	Find out calling process MPI identifier;
MPI_SEND	Send a message;
MPI_RECV	Receive a message; and
MPI_FINALIZE	Terminate MPI.

An example of an application that makes use of a small number of MPI functions is Lawrence Livermore's Accelerated Strategic Computing Initiatives simplified version of the Piecewise Parabolic Method benchmark program [16]. This implementation uses only the above six functions plus MPI_WAIT and MPI_ALLREDUCE.

MPI also has a number of additional features that were useful for the empirical studies conducted in this thesis. Besides the above listed six functions, the following additional LAM/MPI functions were used in developing the software application for this thesis:

MPI_IRECV	Non-blocking receive;

MPI_ISEND	Non-blocking send;
MPI_WAIT	Wait for completion of non-blocking send or receive;
MPI_TEST	Test for non-blocking send or receive completion;
MPI_IPROBE	Non-blocking receive or send buffer probe; and
MPI_WTIME()	Precision timing function.

The non-blocking functions provide the opportunity to, for example, send a message to one system and then to proceed with computations related to the problem domain. The MPI_IPROBE non-blocking function returns an indication that a message can be received. In combination with a blocking receive it emulates a non-blocking receive and proved to be more stable than the non-blocking receive, particularly in cases where a non-blocking receive is setup to receive a message from any process in the HMC. All of the code developed in the thesis utilized an asynchronous communication style. Every attempt was made to avoid performance degragation related to communication synchronization through the use of non-blocking (i.e. asynchronous) communication functions.

Fairness is a potential problem with MPI-based asynchronous communications. For example, it was discovered that it is possible for a process to post a send to another process that never gets received. This send-receive failure occurs when the receiving process posts a receive using the receive from any source as its source argument.

Under these conditions other sending processes can repeatedly send messages to the

same receiving workstation, and eventually overtake the original send. For LAM/MPI, it is the programs responsibility to manage these types of situations.

LAM/MPI is described by OSC as a four layer product. The first layer consists of a set of command procedures, LAM applications, and Graphic User Interfaces. The second layer consists of client-to-server support; the third layer consists of network messaging architecture; and the fourth (and bottom) layer defines local message handling and client management.

A LAM/MPI based cluster has a LAM daemon running on every computer in the cluster. The daemon itself is designed as a nano-kernel and provides a simple message-passing, rendez-vous service to the local processes. The daemon architecture supports developers in testing and debugging a parallel message passing application. Through the LAM/MPI daemon, which buffers all messages in the cluster, it is possible to run LAM/MPI command procedures, like `mpitask`, and examine a cluster's message passing state. Once debugging is complete, the developer can quickly begin testing parallel applications using direct client-to-client (C2C) message passing. The switch from daemon-based message passing to C2C message passing is initiated, not in code, but by the LAM/MPI startup command procedure `mpirun`.

One important issue that has to be addressed when switching from daemon mode to C2C mode is related to the buffering of messages.

The following quote is taken directly from the LAM/MPI online man (help) pages,

> LAM has a property called "Guaranteed Envelope Resources" (GER) which serves two purposes. It is a promise from the implementation to the application that a minimum amount of envelope buffering will be available to each process pair. Secondly, it ensures that the producer of messages that overflows this resource will be throttled or cited with an error as necessary.
>
> A minimum GER is configured when LAM is built. The MPI library uses a protocol to ensure GER when running in daemon mode. The default C2C mode (TCP/IP) does not use a protocol, because process-pair protection is provided by TCP/IP itself. Errors are only reported to the receiving process in C2C mode. An option to mpirun(1) disables GER.

What this means is that it is up to the developer to ensure that there's enough buffer space allocated for messages when the switch is made from daemon mode to C2C mode. Buffering became an issue during GRR testing after GER overflow errors were reported during C2C testing. To increase the buffer for messages, the LAM/MPI libraries and applications had to be rebuilt. The GER buffering resources were increased from the default value of 8 to a new value of 16. This increase in buffer resources corrected nearly all occurrences of GER overflow errors.

3.2.3 LAM/MPI Tools

In addition to LAM/MPI itself, OSC distributes a number of related MPI tools including XMPI [20]. XMPI is a diagnostic tool and debugger developed for SunOS, HP-UX, DEC/OSF, SGI/IRIX and IBM/AIX UNIX systems. For this thesis, XMPI was initially ported to Linux. (An official Linux version of XMPI was released by OSC part way through this thesis.) The key features of XMPI are:

a. runtime snapshot of MPI process synchronization;

b. runtime snapshot of unreceived message synchronization;

c. single process focus detailing communicator, tag, message length and datatype;

d. runtime and post-mortem execution tracing with timeline and cumulative visualization;

e. highly integrated snapshot from communication trace timeline;

f. process group and datatype type map displays;

g. manages MPI applications from local or remote programs; and

h. easy startup and takedown of applications.

XMPI was very useful in the early stages of software development, particularly with its ability to illustrate situations where processes sent messages to other processes that had already terminated or had not setup a receive to accept a message. XMPI's other strong feature is it is easy for a developer to quickly start and stop processes running on the HMC; for heavy debugging and testing periods, this feature was invaluable (see Figure 3.1 for an example screen view of XMPI).

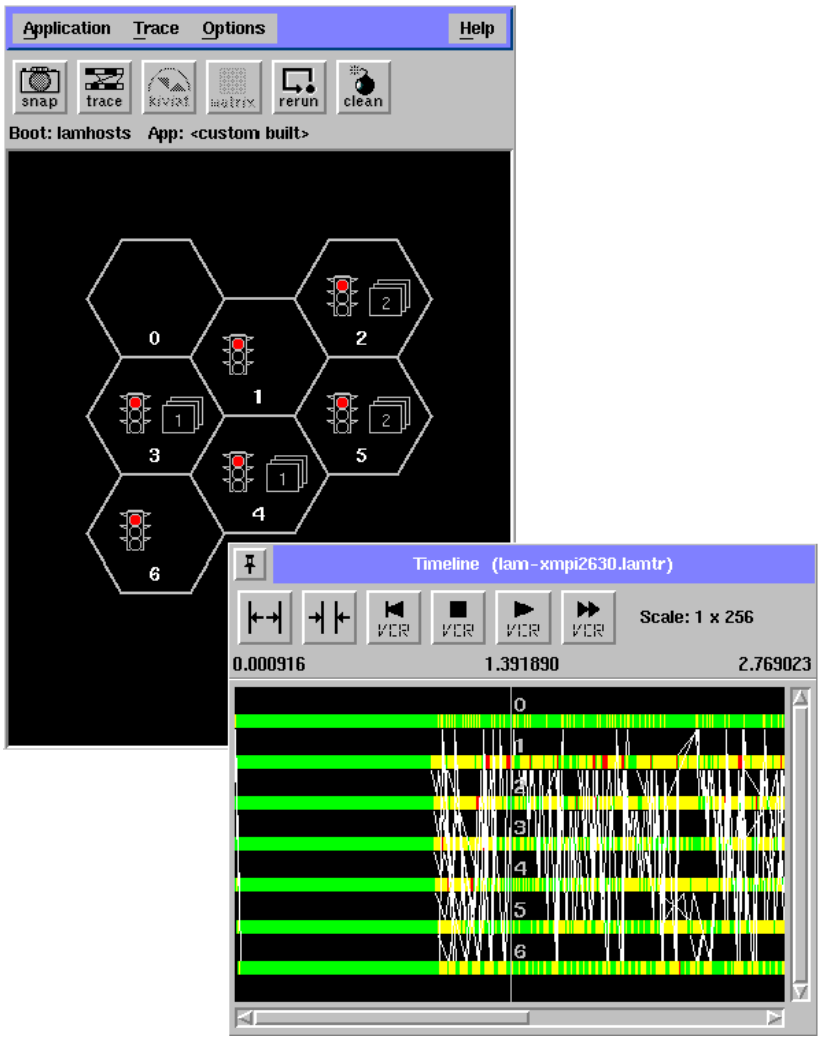


Figure 3.1. Screen View of LAM/MPI XMPI Tool.

3.3 HMC Hardware Configuration

There are (currently) seven Intel-based workstations in the HMC: two Pentium 100s each with 32Mbytes, a 486/DX100 with 20Mbytes, a 486/DX66 with 8 Mbytes, a 486/DX40 with 8 Mbytes, and two 386/DX40s each with 8 Mbytes. Each system has a 3Com 509 network interface card (NIC). One system, the 486/DX66, has a SCSI hard drive and CDROM drive, all the other workstations had IDE hard drives and no CDROMS.

The Ethernet switch used in the HMC is a 24-port 3Com Superstack II 1000 [31]. The switch reduces collisions and provide a higher aggregate bandwidth compared to bus-based Ethernet networks, because concurrent pair-wise communication is possible [29]. The HMC is connected to the campus-wide network, but can be isolated from it by simply disconnecting the campus-wide network connection from the switch. The 3Com 509 card is a combo card which means the card can be networked with twisted pair cables, thin wire coax and thick wire coax (not concurrently). For this thesis the HMC will be configured with twisted pair cable when the Ethernet switch is in use and thin wire coax cables as an alternative network configuration. See Figure 3.2 for the HMC hardware schematic.

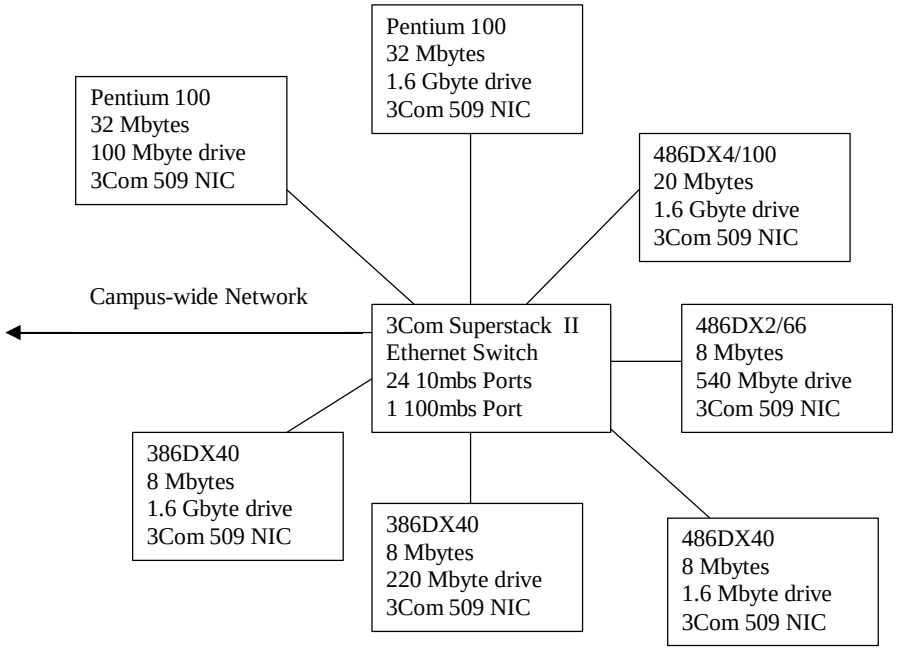


Figure 3.2. HMC Hardware Configuration.

CHAPTER IV

LOAD BALANCING STRATEGIES AND THE PROBLEM DOMAIN

4.1 Load Balancing for a Class of Single Program Multiple Data Computations

Two broad programming models for parallel and distributed computing, which are applicable to cluster computing, are MIMD (Multiple Instruction, Multiple Data) [8] and SPMD (Single Program, Multiple Data) [5]. In the MIMD model, multiple independent instruction streams (e.g., processes or programs) are executed across multiple processors. The SPMD model is a restricted case of the MIMD model in which every processor is executing the same program on different data sets. The focus of this thesis is to study load balancing schemes for SPMD computations.

Discrete optimization algorithms represent an example class of search algorithms that can be implemented in SPMD mode. A sub-class of discrete optimization algorithms is the depth first search (DFS). For this thesis, two types of DFS algorithms are implemented: exhaustive depth first search and best distance branch and bound search. These two algorithms are described in Section 4.2.

An example of a generic search tree associated with a simple DFS is illustrated in Figure 4.1. Each box represents a node in the search space: the number on the left indicates the order in which the node was generated and the number on the right indicates the order in which the node was expanded (i.e., the order in which the node's

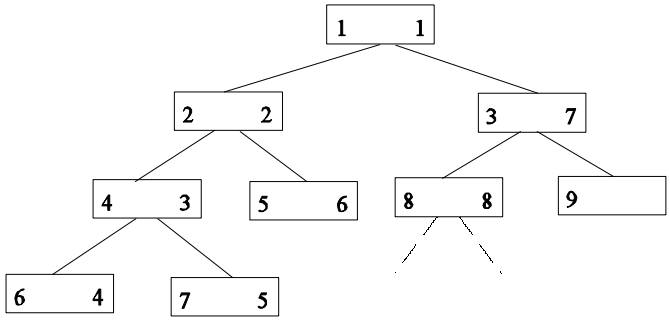


Figure 4.1. Search Tree Expansion.

children nodes were generated). In a load balancing scheme involving such a search space, the tree structure will be stored in a queue as the search proceeds.

The only difference in the execution of the SPMD parallel program on each workstation is associated with the startup and shutdown phases (refer to Sections 2.2.1 and 2.2.3 for general descriptions of these phases). One workstation acts as a root server, which initializes and terminates searches. In this thesis, initialization of a search is done using root initialization and a version of direct initialization. In root initialization a root server is selected to start the search for a solution. The root server will start building up its queue with search paths as it attempts to find a solution. At the same time, the other workstations will either start making requests for work (idle initiated) or will start checking for the arrival of a request to accept work (busy initiated). Eventually, the root server's queue size will exceed the busy threshold, and at that point the root server will either start accepting requests to share its queue (idle initiated) or it will begin to look for an idle workstation to accept part of its queue

(busy initiated). Flow diagrams of the idle initiated and busy initiated algorithms can be viewed in Figures 4.2 and 4.3.

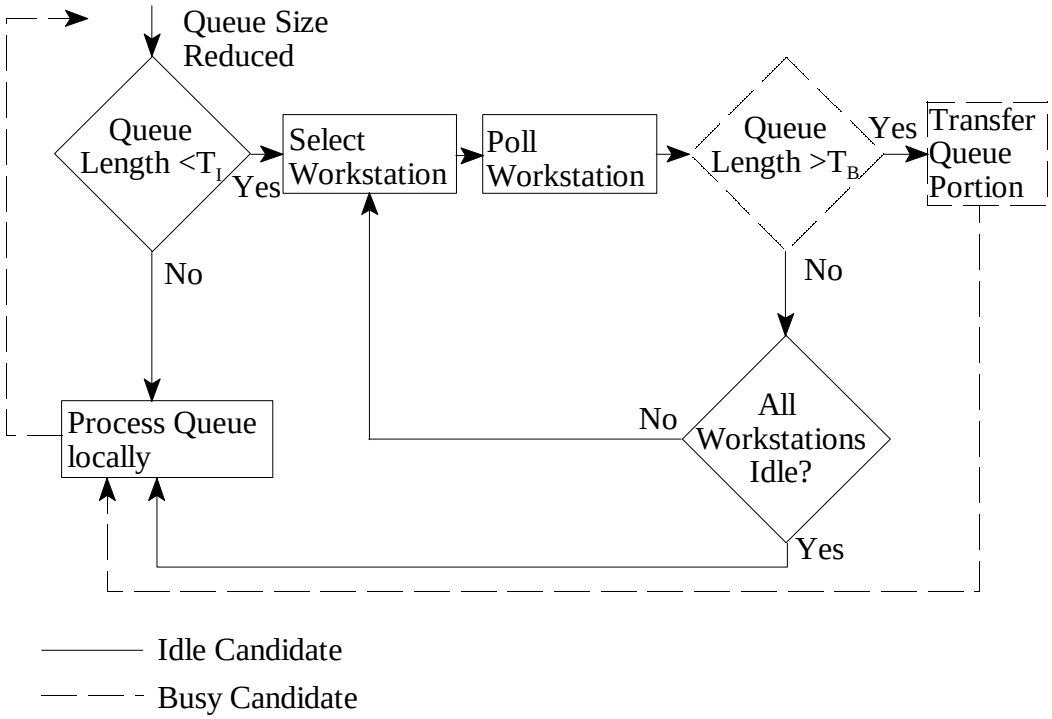


Figure 4.2. Idle Initiated Algorithm.

For direct initialization, the root server expands the root node and shares the resulting paths to the other workstations before the search proceeds. Because of the small size of the HMC (7 workstations) there is always enough paths to share. In this thesis, the code was designed so that the developer could control the parameters of the direct initialization scheme using command line arguments during HMC startup.

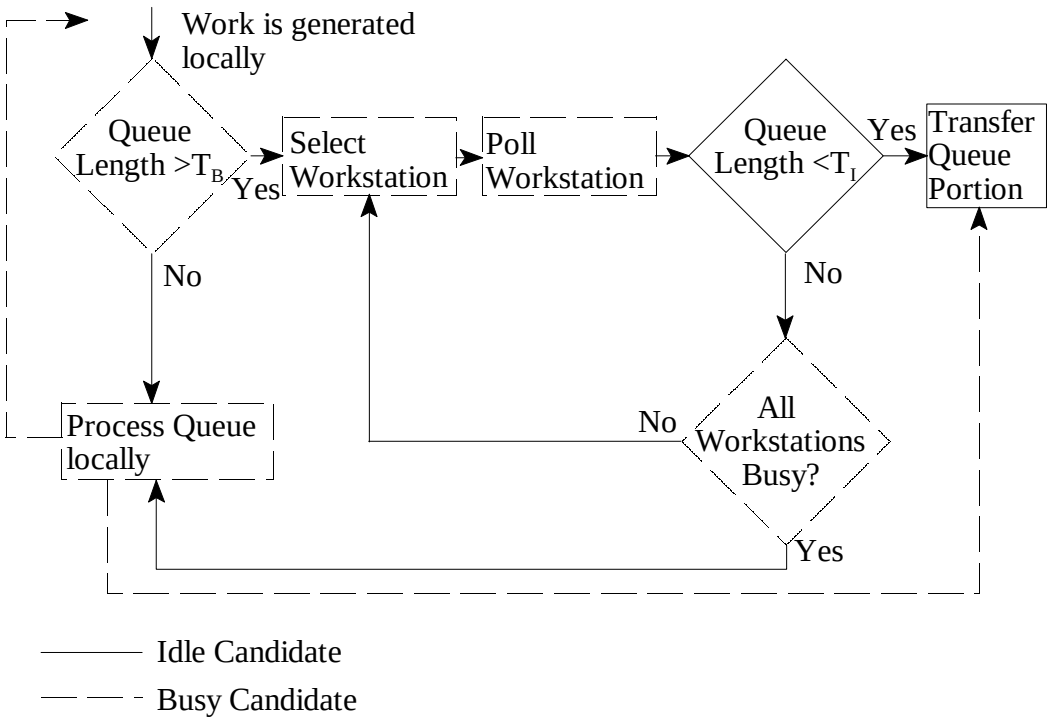


Figure 4.3. Busy Initiated Algorithm.

The dynamic load balancing schemes implemented are based on queue splitting or data sharing (refer to Section 2.2.2 for details). An illustration of queue splitting is given in Figure 4.4.

Each vertical column of the array is divided in to two parts, the upper two positions of a column contain information on the number of nodes in a path and the associated (intermediate) path distance. The rest of the column is dedicated to storing the actual nodes in a search path. Shown at the top of the figure is the contents of Workstation A's queue before splitting. The bottom two arrays illustrate the content of Workstation A's queue after splitting its queue and giving a portion to Workstation B (which was assumed to be out of work before receiving a portion of Workstation A's queue). An important point related to queue splitting is that the paths left behind on

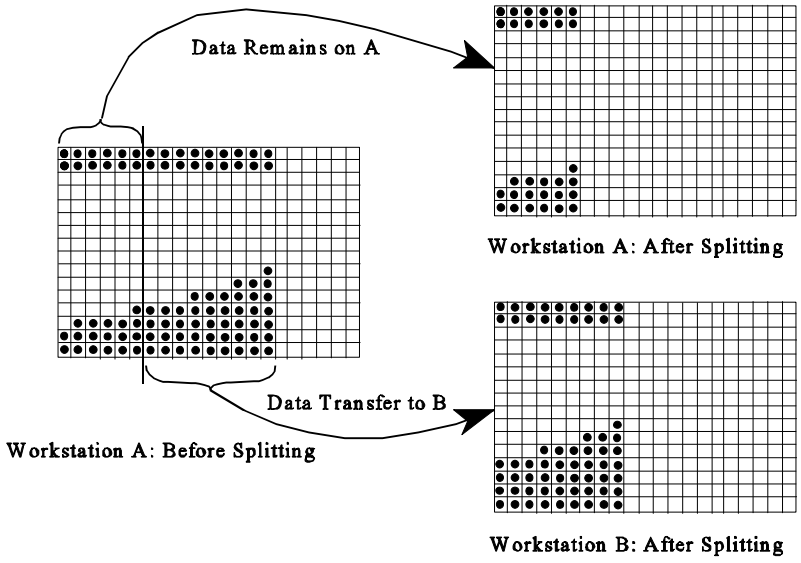


Figure 4.1 Example of Queue Splitting

Workstation A after queue splitting has more future expansion capability than the paths transferred to Workstation B, because they are closer to the root of the search tree. As a result, Workstation B has less work than A and can potentially empty its queue before Workstation A. Ideally, it would be best if both workstations shared equal amounts of work after a single queue splitting operation. This fact is important when evaluating the performance of queue splitting.

Queue splitting is initiated based on the value of a busy threshold parameter. When a workstation's queue size reaches this busy threshold value, queue splitting will occur. For this thesis, two queue splitting methods were implemented. In the first method, the queue is split in half, and in the second the queue is split into portions whose sizes are proportional to each workstation's performance index. For the remainder of this document the first queue splitting method will be referred to as fixed queue splitting and the second method as variable queue splitting.

The queue is populated with paths generated by either an exhaustive depth first search or a branch and bound search. The exhaustive depth first search algorithm for a sequential (i.e., single workstation configuration) is as follows. The search is initiated by pushing the root node of the search tree onto the queue. The process then enters a loop which creates and deletes search paths. The path last pushed on to the queue is popped off and expanded, creating new paths for each node not already appearing in the parent path. The resulting paths are pushed back onto the queue. The process then loops back and pops another path off of the queue and repeats the expansion process. Each time the bottom of the search tree is reached, the current path distance is

calculated. If the search path is the best so far it is saved otherwise the path is discarded. Eventually all paths that were pushed onto the queue are popped off. When the queue is empty the search is finished, and the best distance and path is returned.

In a single workstation configuration, the branch and bound search algorithm tracks the current value of the best distance found so far. Therefore, when a poor distance is encountered in a partial path, the partial path can be discarded before it is expanded any further. The branch and bound search algorithm creates an interesting problem for a parallel system. For a parallel system the best distance found on a given workstation is not known to the other workstations until that information is communicated. The result of this is that some workstations may search down paths that would not normally be searched on a single workstation implementation. Additionally, it is also possible that in the parallel implementation a workstation may communicate a better distance to another workstation before a path, normally expanded on a single workstation, is expanded. This may result in less total work being necessary for the parallel implementation.

Queue splitting tests were carried with both fixed queue splitting and variable queue splitting. For fixed queue splitting the current queue was divided in half. For variable queue splitting each workstation is assigned an index based on the workstation's performance capability (this index was determined offline based on benchmarking results). The workstation's index and the index of the workstation sharing the queue are used to determine what portion of the queue each will receive.

Although Bytes UNIX [1] benchmark program was originally used as a method for indexing workstation performance, a decision was later made to use the results of the single system exhaustive depth first search to get relative performance indexes of each system [2]. These performance indexes were derived form the exhaustive depth first search code, excluding any code related to communications and process synchronization.

This technique is passive in nature and follows Cermele *et al.* [2] approach on how to implement passive load estimating for a heterogeneous multicomputer cluster. Although Cermele *et al*. discarded this method for a more active method of estimating load capacity, unlike Cermele's cluster no external loads unrelated to the parallel search were simulated on the HMC. Therefore, passive load estimates proved to be acceptable for the purposes of this thesis. This HMC was dedicated to parallel processing during testing and every attempt was made to reduce workstation load from any unnecessary system programs.

All three dynamic load balancing techniques, ARR, RP and GRR (overviewed in Section 2.3), were implemented and their performance measured and compared using an exhaustive root initialized, idle initiated depth first search. For ARR busy initiated and idle initiated test results were compared.

A simple static load balancing scheme was investigated and compared to the dynamic schemes. The static testing was done without any communications between systems. Each system reports its results when it completes the search. Both exhaustive and branch and bound searches are compared to a dynamic load balancing scheme.

4.2 Problem Domain

The minimum tour traveling salesman problem (TSP) is a classic NP-complete problem [3]. There are many different techniques for trying to find the shortest route among all cities and there are many variations on the problem formulation itself. Possible types of TSP problems include:

many-visit TSP;

time dependent TSP;

bottleneck TSP;

stochastic TSP;

multi-salesman TSP;

clustered TSP; and

multi-criterion TSP.

The TSP problem that was the focus of this thesis is the classic one-visit per city shortest distance traveled. To solve this problem, two different search techniques were used:

Exhaustive Search (ES) [21]; and

Best Distance Branch and Bound Algorithm (B&B) [4,11].

Although a third more involved heuristic was considered during the early phases of this research, it became apparent during testing that the above two choices would provide a sufficient problem domain to exercise the HMC. It was also felt that another search technique would be a distraction from the main purpose of the thesis, which is to focus on comparing different load balancing techniques.

The TSP search space increases quickly as the number of cities increase. An exhaustive search of a 50 city graph has about 10^{62} Hamiltonian paths (or (n-1)! in general). Because of the large search space associated with the TSP, techniques have been developed to reduce the search space. The best distance branch and bound search greatly reduces the search space, unfortunately the gain is usually lost with the addition of 2 or 3 more cities. Table 4.1 contains example values for the size of the search space for exhaustive depth first search for various problem sizes.

Table 4.1. Exhaustive Depth First Search, Problem Size.

Cities	8	9	10	11	12
Paths	13699	109600	986409	9864100	108505111

For Branch and Bound Depth First Search, the size of the search space varies depending on the city associated with the root node, whether the search starts at the right or left side of the search tree, and the distances between the cities. On a parallel system, best distance communication time overhead can also effect the size of the space that is searched.

The cities and distance measurements used in this thesis are based on fourteen selected Texas cities and the direct distances between them.

CHAPTER V

HMC TESTING AND DATA COLLECTION

LAM/MPI is released with C, C++, and Fortran libraries, any one of which could have been used as the programming language for this thesis. The decision to use C was one of convenience related to programming experience. (In addition, the Fortran compiler distributed with Linux is actually a Fortran to C translator rather than a pure Fortran compiler. This fact could have lead to inefficiencies if translation overhead was substantial.) A representative source code example is listed in Appendix B.

5.1 Sequential Testing

Before any HMC testing was done, it was important to first test and gather data on the performance of a single workstation's solution time to the TSP. Two different data structure schemes for the queue were investigated and compared. The first scheme was based on a data driven approach. It involved the use of a linked list of linked lists: the queue was represented with one linked list and each search path was represented by a linked list. With this scheme search conditions were tracked using standard techniques that checked for end of list. By using the structure of a list to drive the search, the search procedures were implemented with a minimum amount of code. The linked list approach also had the advantage of freeing the programmer from having to predetermine the size of the search space in order to allocate an appropriate amount of space on heap at compile time.

After testing both the exhaustive search and branch and bound code on a single workstation, development then began on the parallel code using the linked list approach. It soon became clear that packing and unpacking the linked lists for transfer was inefficient. Because of this, code was then developed for the HMC that implemented the queue using a two dimensional array for the queue instead of a linked list. This code explicitly tracks the queue and the last position in the queue through a series of control variables.

The implementation of the queue using the array structure was illustrated in Figure 4.1. Although memory space is wasted with this type of structure, it is not a substantial penalty. The justification for the development of this array based code was to improve performance related to message passing. Before continuing with parallel code development, the array based code was tested on a single workstation. The results of the testing are listed in Table 5.1.

Table 5.1. Exhaustive Search Execution Times on a Pentium 100.

Cities	Link Listed Search (s)	Array Based Search (s)	Max Queue Size
8	.51	.23	22
9	4.7	2.05	29
10	49.8	20.7	37
11	574.9	230.0	46
12	NA	2825.53	56

The dramatic difference in performance between the link list based code and the array based code resulted in the array based code being used for all single workstation and HMC testing.

The sequential program was executed (individually) on all the workstations in order to establish performance differences between the systems. Although HMC execution times were always compared against the fastest workstation available, it was important to establish peak aggregate HMC performance capability based on execution times for each individual workstation. Table 5.2 lists the performance of each workstation in the HMC individually executing the sequential exhaustive depth first search. The workstation's execution times are normalized with respect to the Pentium 100 in Table 5.3. These normalized values are the performance indexes used for variable queue splitting. By normalizing the performance with respect to the fastest system, it is also possible to establish a peak potential speedup capability for the HMC as each workstation is added. For example, the potential peak speedup of a cluster consisting of the two Pentium 100s is 2.0; with all the seven workstations participating, the potential speedup is 3.38. These speedup estimates ignore the possible effects of superlinear speedup. Superlinear speedup occurs when the parallel algorithm performs less work than the serial and speedup exceeds the speedup directly related to increased CPU capability [15].

Table 5.2. Execution Times for Exhaustive Search (s).

Cities	P100	P100	486/100	486/66	486/40	386/40	386/40
8	.24	.24	.49	.53	.89	2.40	2.41
9	2.15	2.16	4.38	4.84	8.04	21.86	21.88
10	21.72	21.70	44.42	49.12	81.84	222.84	223.53
11	242.16	241.73	498.10	563.62	ND	ND	ND

ND - No Data

Table 5.3. Normalized System Performance.

Workstation	P100	P100	486/100	486/66	486/40	386/40	386/40
Normalized Performance	1	1	0.49	0.44	0.27	0.09	0.09
Potential HMC Speedup	1	2	2.49	2.93	3.23	3.29	3.38

5.2 Parallel Testing

5.2.1 HMC Latency Testing

Before testing the HMC with the parallel TSP code, ping-pong tests were carried out. In order to determine the latency in sending and receiving a message, not including the actual time to transfer data, a zero length message was repeatedly sent back and forth between a workstation pair. The average send-receive-send-receive time was divided by two to give the average zero length message latency.

The latency testing took place on both the coax and Ethernet switch network configurations. The Ethernet switch is capable of operating in four different modes:

cut-through, store and forward, intelligent flow control, and fragment free mode. Although latency testing was done for all four modes, a decision was made to operate the switch in cut-through mode only for testing the various load balancing schemes. Table 5.4 contains the results of ping-pong tests between the two Pentium 100s using the coax network and the four modes associated with the switch.

Table 5.4. Coax and Switch Latency (μs).

Network Configuration	Coax	Intelligent Flow Control	Cut-Through	Fragment Free	Store and Forward
P100 to P100	320	359	367	389	412

The ping-pong test was carried out twice for each workstation pair on the coax network. For the second test the role of message initiator was reversed. This was done in order to determine if the initiator of the message sequence affected the results. Latency testing is summarized in Tables 5.5 and 5.6.

Initially (before testing) it was believed that there were two pairs of identical systems in the HMC, two Pentium 100s and two 386/DX40s. The difference in message passing latency between the two supposed identical 386/DX40 workstations is because they were not identical. One 386/DX40 had 64k bytes of cache memory while the other has 128k bytes of cache memory. Besides the already recognized differences in CPUs, this is an example of how cache memory configurations contribute to the heterogeneous nature of the multicomputer cluster.

Table 5.5. HMC Latencies for a Coax Network (µs).

CPU	386/40	386/40	486/40	486/66	486/100	P100	P100
386/40	ND	1394	1089	928	901	715	739
386/40	1327	ND	1353	1088	1215	1209	967
486/40	1092	1227	ND	845	855	706	653
486/66	1012	1211	904	ND	745	550	543
486/100	877	1044	907	716	ND	524	523
P100	650	810	675	499	531	ND	332
P100	687	849	649	549	524	320	ND

ND - No Data

Table 5.6. HMC Latencies for the Ethernet Switch in Cut-Through Mode (µs).

CPU	386/40	386/40	486/40	486/66	486/100	P100	P100
386/40	ND	1377	1121	933	991	834	ND
386/40	1433	ND	1353	1089	1295	988	ND
486/40	ND	ND	ND	1027	ND	680	ND
486/66	ND	ND	ND	ND	ND	622	ND
486/100	ND	ND	903	734	ND	569	ND
P100	ND	ND	ND	ND	ND	ND	367

ND- No Data

Each test between the workstation pair was repeated 100 times; a sample of two of these tests is illustrated in Figures 5.1 and 5.2. It is evident from Figure 5.2 that there are random events that can effect the HMC testing.

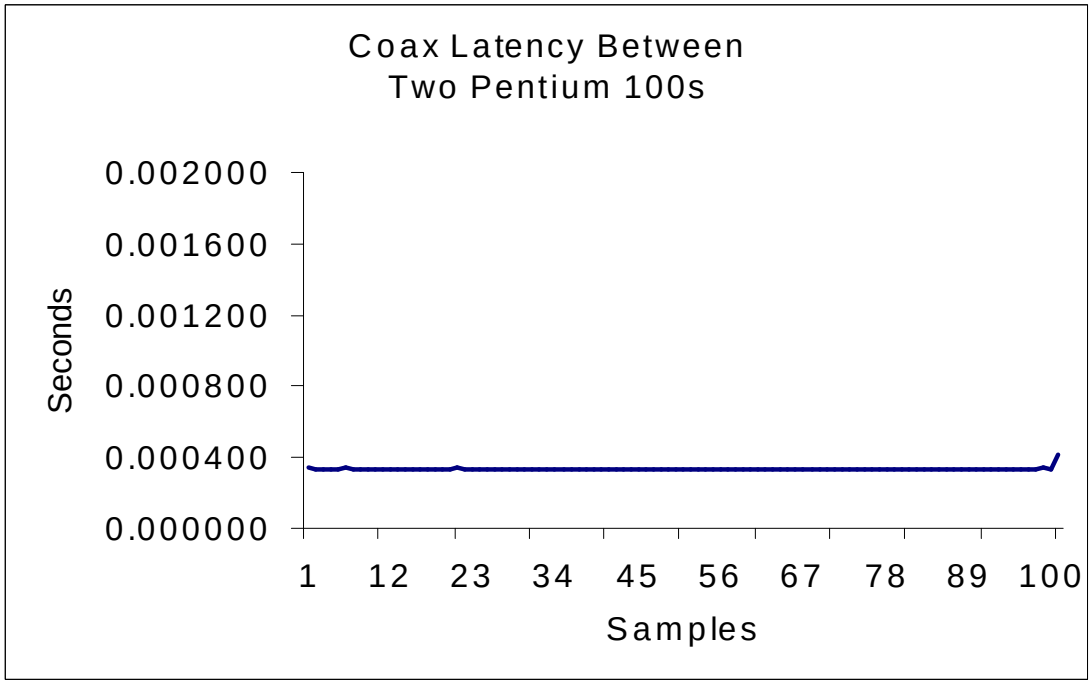


Figure 5.1 Coax HMC Latency.

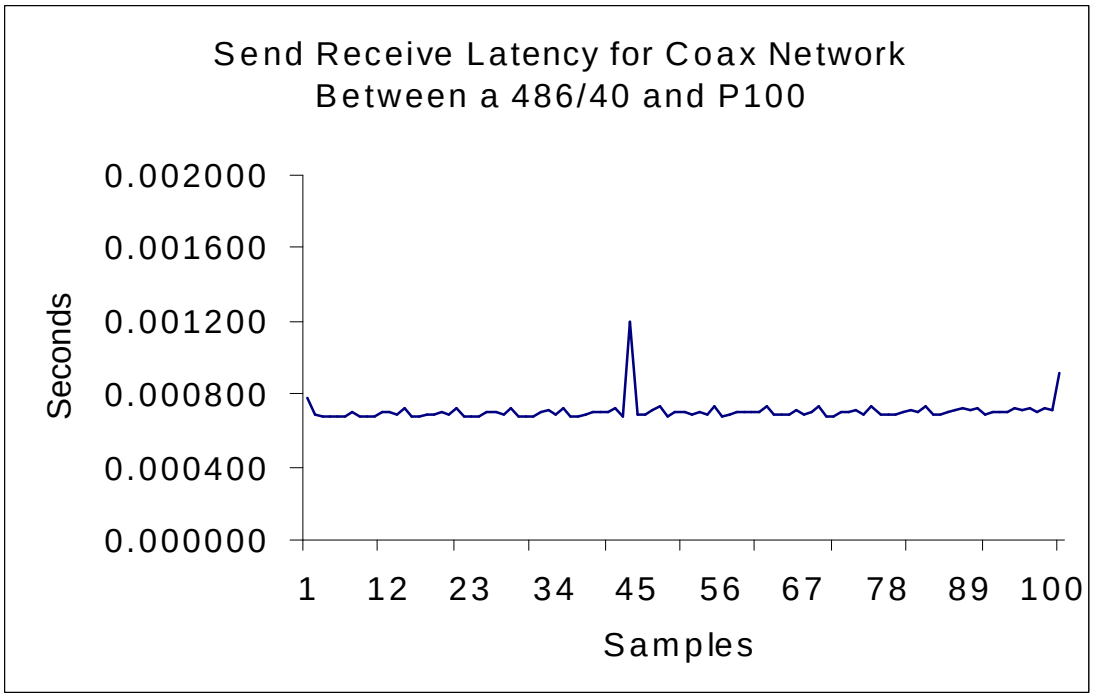


Figure 5.2. HMC Latency Illustrating a Random Event.

In addition to the zero message length tests, contention tests were also conducted. A 2 Mbyte message was sent between a workstation pair consisting of a Pentium 100 and a 386/DX40 and a second similar workstation pair. The concurrent communication transfer time for one workstation pair was 6.29s and for the other pair 5.35s; the sequential communication time on the coax was 6.11s and 4.60s respectively. With the switch the concurrent test results were 5.93s and 4.58s, the sequential communication transfer times for the switch were 6.02s and 4.63s Although there is a margin of error, it is still clear that the switch provides contention free transfers of large data sets.

5.2.2 Parallel Search Testing and Debugging

For parallel execution of the TSP, message sizes varied from 12 bytes in length to approximately 1000 bytes in length for a problem size of 10 cities. Table 5.7 summarizes the message passing requirements of a 10 city, root initialized, idle initiated, exhaustive depth first search, fixed queue splitting, for a cluster of 2 workstations.

Table 5.7. Message Passing Data for Two Pentium 100s .

	P100	P100
# Request for Work Msgs	1	302
# Work Msgs Received	0	300
Amount of Work Received (bytes)	0	139888
# Work Response Msgs	302	1
# Work Msgs Sent	300	0
Amount of Work Sent (bytes)	139888	0

By calculating the average message size passed between two systems, which was 466.29 bytes per work message and 12 bytes for all other messages, and then rerunning ping-pong tests for these average message sizes, it was possible to estimate the overhead associated with message passing. The ping-pong test determined that, on average, it takes 1318 µs to transfer one average sized work message. The actual data transfer time, work message time minus the latency, is 986 µs. Thus, the total data transfer time is 986 * 300 = 0.2958s. Adding in the latency overhead the total time spent sending work messages is 0.3936s.

Ping-pong testing revealed that the time to transfer the response and request messages or handshaking messages is 530 μs, the total time for this example is 0.1606 s. This particular search took 11.58s to complete, the total communication overhead time is 0.5542s, which is 4.8 % of the time spent on the 10 node search.

For most of the debugging phase, overhead was calculated directly in the code and relative times for communication overhead vs. work was obtained. The debugging

phase is entered by recompiling the parallel code with all MPI timers enabled. These timers monitor time spent in each part of the code. For this debugging mode, the time to complete the search is 21.0s. The time spent doing work on one Pentium is 13.2s and on the other 12.62s. The rest of the time, besides a small amount of time for communication overhead, is overhead associated with calls to the MPI timers. From this search, it can be assumed that each workstation does approximately half the work. The exhaustive test takes, on average, 21.6s to complete on a single Pentium. To complete half the work on each system it would take approximately 10.8s. Adding the previously calculated messaging time and the total is 11.35s, this compares well within 11.58s for the actual search. Although a little crude, the numbers compare reasonably well.

5.2.3 Empirical Data Collection for Parallel TSP

Initial HMC test results showed poor performance for a number of reasons. First it was not clear what the busy threshold value should be for either idle initiated or busy initiated searches. Another problem was that very high communication overhead was occurring. It also became apparent latter on that for search loops of 100 million or more paths, the LAM/MPI timers, which were used for debugging, created a significant overhead.

It was determined, in debugging mode, that most of the communication overhead was wasted time spent checking to see if a request for work had come in from another workstation. In an attempt to control communication overhead, a series

of MPI timers where introduced as time delay controllers. These controllers increased the time interval between requests for work as well as the time between checks for incoming work requests. Using MPI timers as time delay controllers proved to be a poor choice, when it became apparent that the MPI timer calls in debugging mode created significant overhead. A better choice for time delay controllers was countdown counters. Initially, countdown values were estimated by making use of the MPI timers. Even though the timers do affect overall performance, it was possible to locate the timers in the code so that each timer affected its part of the code by about the same relative degree. The result was a set of relative numbers associated with the total time spent on any given part of the cluster program. From this set of timing values, reasonable countdown values could be estimated.

After the first set of queue splitting tests were complete, a new series of tests were run on the countdown counters for both idle initiated searches and busy initiated searches, the results are shown in Figures 5.3 and 5.4. Having established reasonable values for the communication delay controllers, the queue splitting tests were repeated.

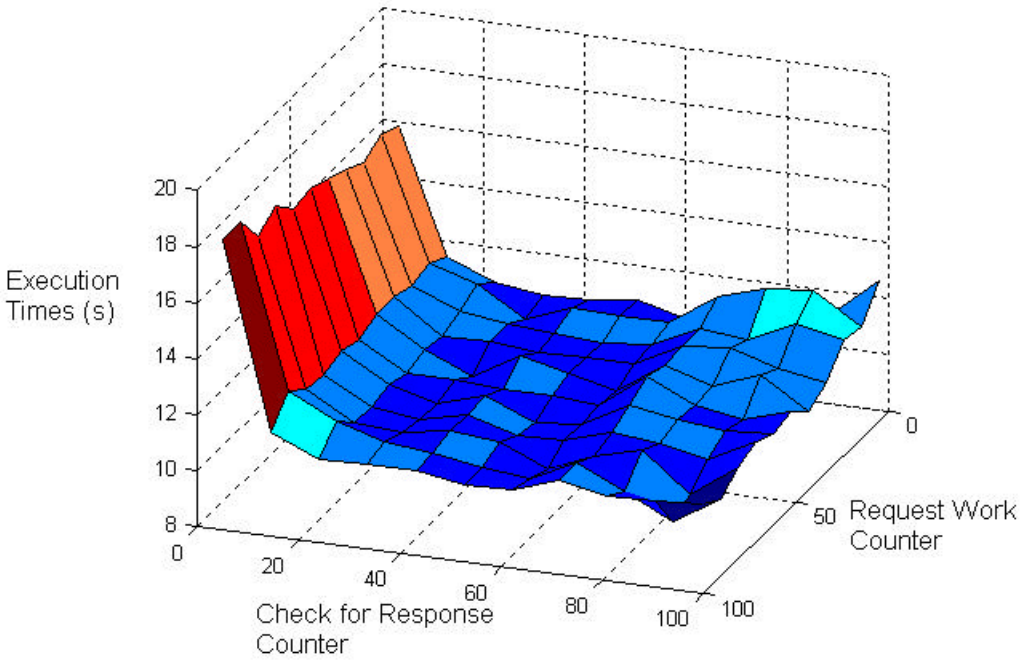


Figure 5.3. Idle Initiated, Exhaustive Search, Delay Counter Test.

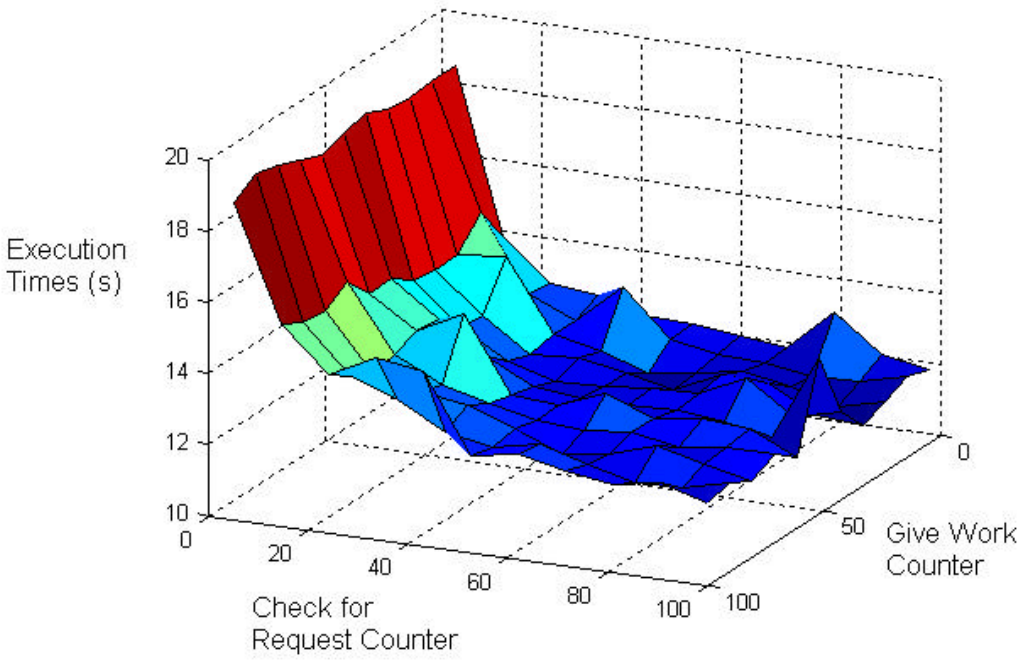


Figure 5.4. Busy Initiated, Exhaustive Search, Delay Counter Test.

The tests on queue splitting and the determination of the optimal idle-threshold and busy-threshold values proceeded as follows. For idle initiated searches, idle threshold values were varied from 0 and 6 while busy threshold values were varied between 2 and 30 for each value of idle threshold. For each test, execution times were recorded. The tests were repeated for busy-initiated searches. For busy initiated searches, a busy threshold of 2 was determined to be unreasonable. With this low a value, solution times were greater than the single system solution times. For busy-initiated searches, the busy threshold was started at 15 and increased to 40. The idle-threshold was again varied from 0 to 6. The results of these two tests is a graph of busy and idle thresholds versus execution time for exhaustive and branch and bound searches (see Figures 5.5, 5.6, 5.7, and 5.8).

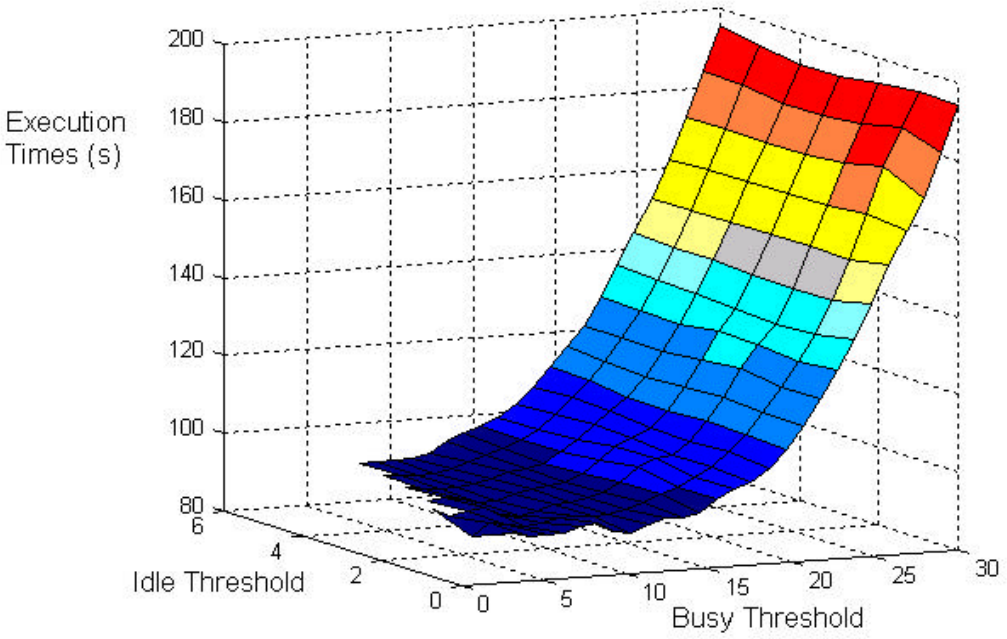


Figure 5.5. Idle Initiated, Exhaustive Search, Queue Threshold Test.

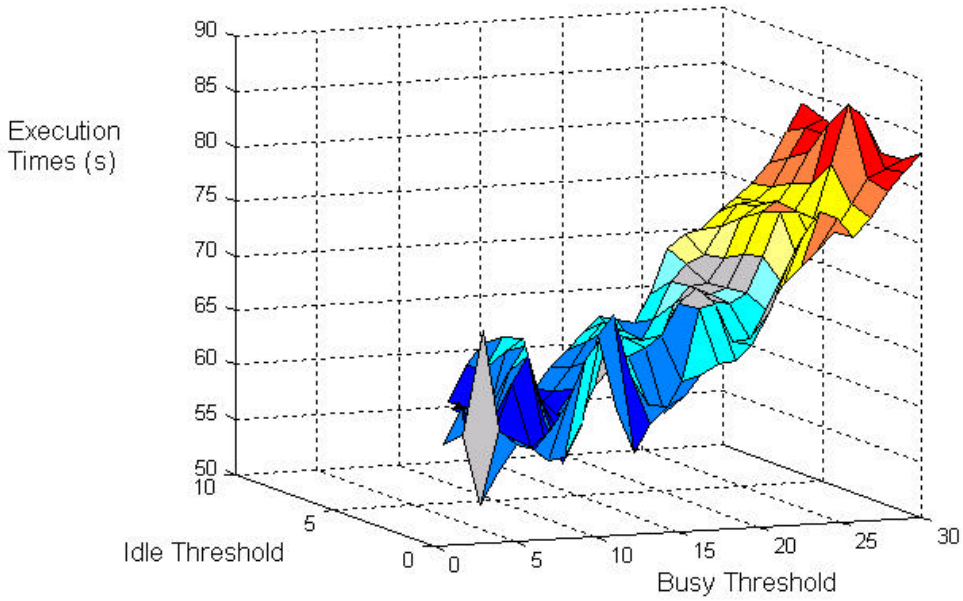


Figure 5.6. Idle Initiated, Branch and Bound Search, Queue Threshold Test.

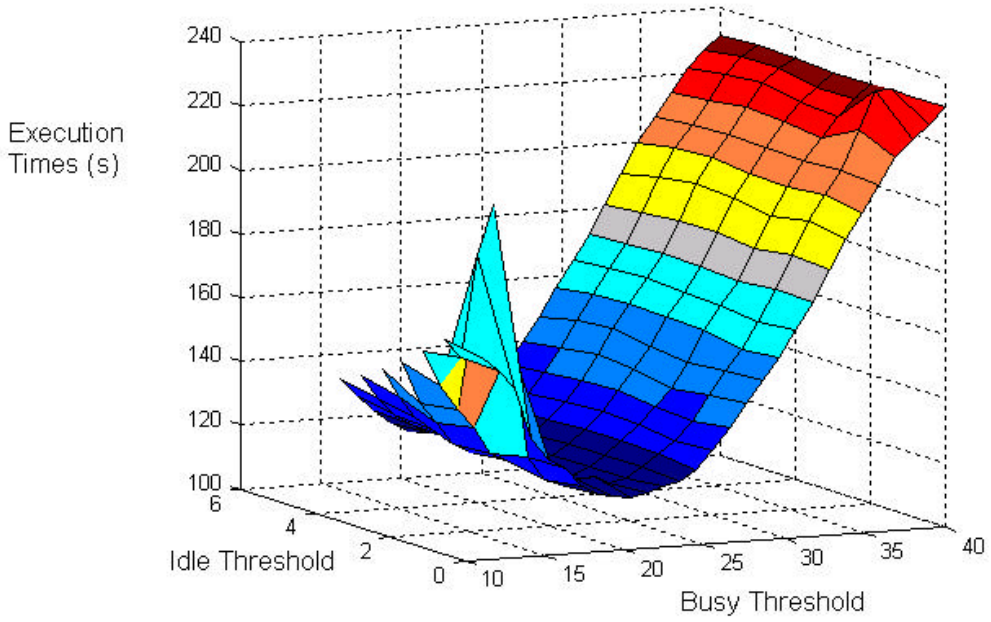


Figure 5.7. Busy Initiated, Exhaustive Search, Queue Threshold Test.

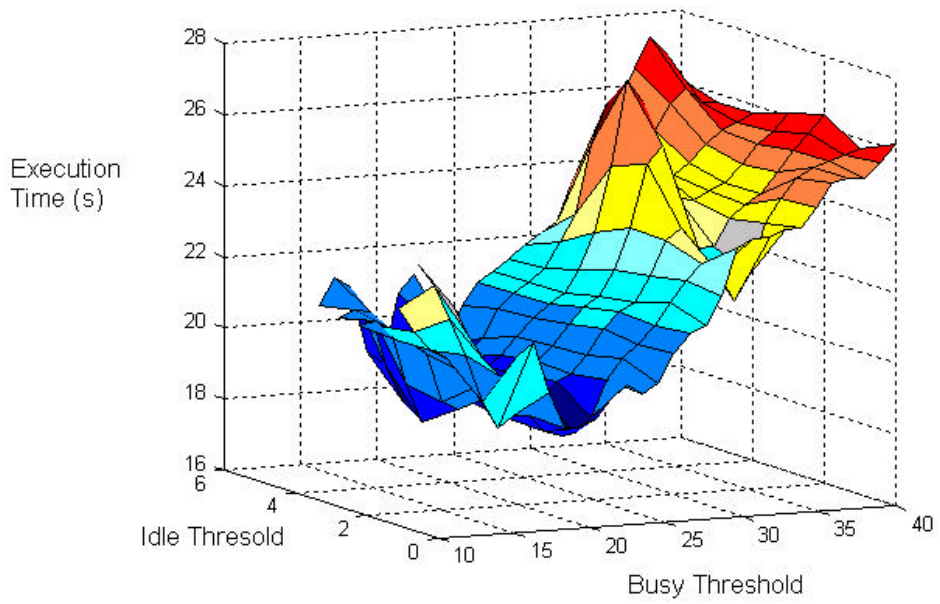


Figure 5.8. Busy Initiated, Baranch and Bound Search Queue Threshold Test.

A difficulty encountered during busy initiated testing was how to terminate the search. During busy initiated searches, only busy workstations can initiate communications. Once a workstation becomes idle, it waits for a request to accept work. When the search is complete, all workstations should be idle, resulting in all of the workstations waiting for a request to accept work. To solve this problem, a root server was chosen, and when this root server becomes idle it starts sending requests to terminate to the other workstations. If a negative response is received (i.e., at least one other workstation is still busy) the root server backs off and tries again latter.

In the code developed for the HMC, two different algorithms for pushing paths onto the queue were implemented and tested. The first algorithm follows the same execution path as the single workstation algorithm. In the second algorithm, only one descendant node (child node) is expanded (visited) before the process loops back to carry out HMC communications. On the next call to the search code, the next descendant node is expanded; this continues until all descendant nodes have been expanded and their paths pushed onto the queue before the next path is popped off the queue. Most of the testing on the HMC used the same search algorithm as the sequential algorithm. The testing that was done using the one node at a time expansion algorithm revealed that this method was about 8% slower for an 11 city exhaustive depth first search. The slower execution times held true as queue threshold were varied. Therefore the rest of the testing on the HMC used the first algorithm.

Most of the HMC testing during the debugging phase involved all 7 systems. It was believed that having all 7 systems in a test would push communications and queue

splitting to the limit, revealing any possible weaknesses in the design of the parallel code. To push the communications to the maximum, tests were run with busy-thresholds set to 2 and idle-thresholds set to 1. Under these circumstances, network traffic is heightened. In the case of busy initiated searches, thrashing was observed at low busy threshold values. Once the debugging phase was completed, all testing was done in C2C mode.

5.2.4 Initialization Testing

Direct initialization testing took place and was compared with root initialization testing. Figure 5.9 is a graph that compares the performance of idle initiated, root initialization versus direct initialization. Figure 5.10 is a graph of busy initiated, root initialization versus direct initialization.

The actual definition of direct initialization testing is as follows: nodes of the search tree are generated to a depth that allows each workstation a unique root node, which is actually a root of a sub-tree of the entire search tree. If there are more nodes than workstations, then a workstation will take on multiple nodes. For this thesis, a slight variation to this approach was used. Instead of having each system expand the

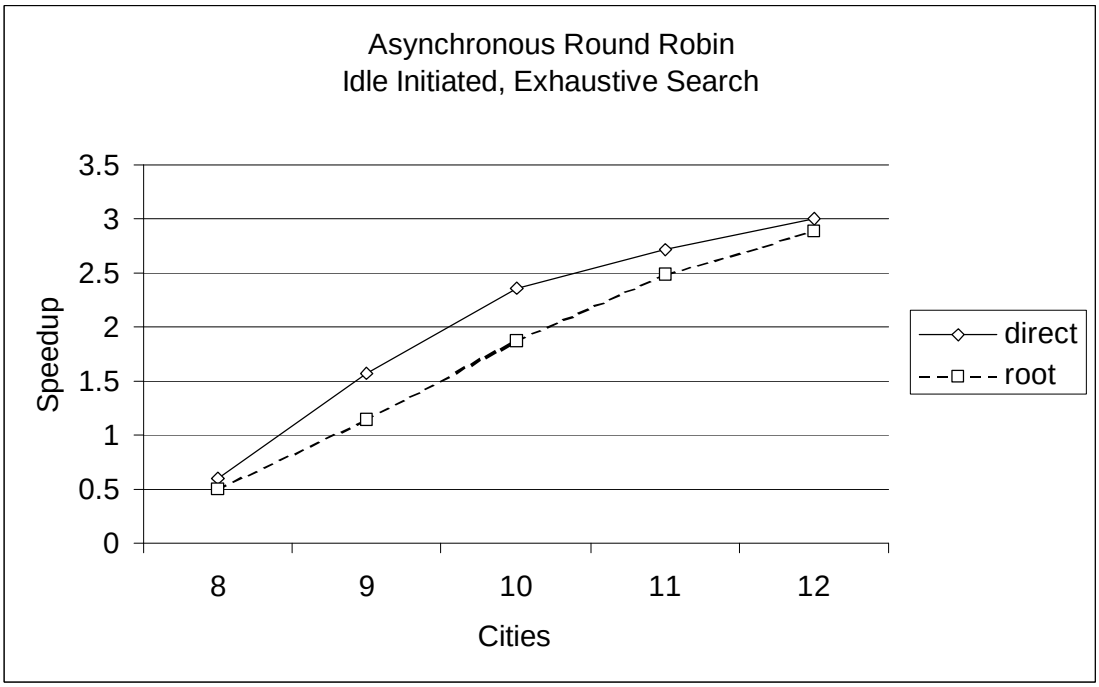


Figure 5.9. Idle Initiated, Root Initialization versus Direct Initialization.

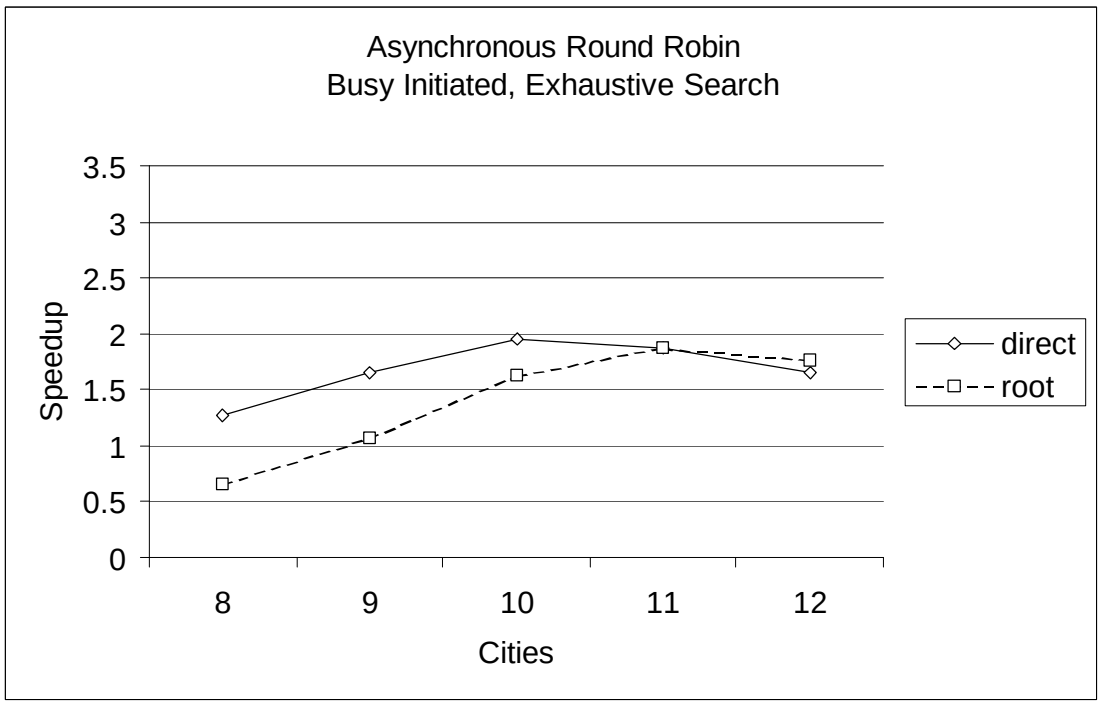


Figure 5.10. Busy Initiated, Root Initialization versus Direct Initialization

root and choose a unique node, it was decided that the root server would seed each workstation with a unique node. This was done because of the wide range in performance among the HMC workstations and the desire to have more control over initialization. With this method it is possible to seed only selected workstations leaving some workstations without any initial work. A command line variable controls which servers get seeded and how many nodes they get seeded with.

It could be argued that this is not actually direct initialization because the root server is instructed to make sure each system starts with its own unique root node rather than each workstation doing that for itself. Whether it is direct (based on the definition given in [13]) or not is debatable, what is of real interest is how well the HMC performed when all or some of the workstations have work from the start. The results of direct initialization testing are in Figures 5.11 and 5.12. The horizontal axis represents the number of nodes initially seeded to each of the seven workstations. Each digit in the string are hexadecimal. So, the string “C,0,0,0,0,0,0” indicates that twelve nodes are initially seeded on the first workstation and zero nodes are seeded on the remaining six workstations.

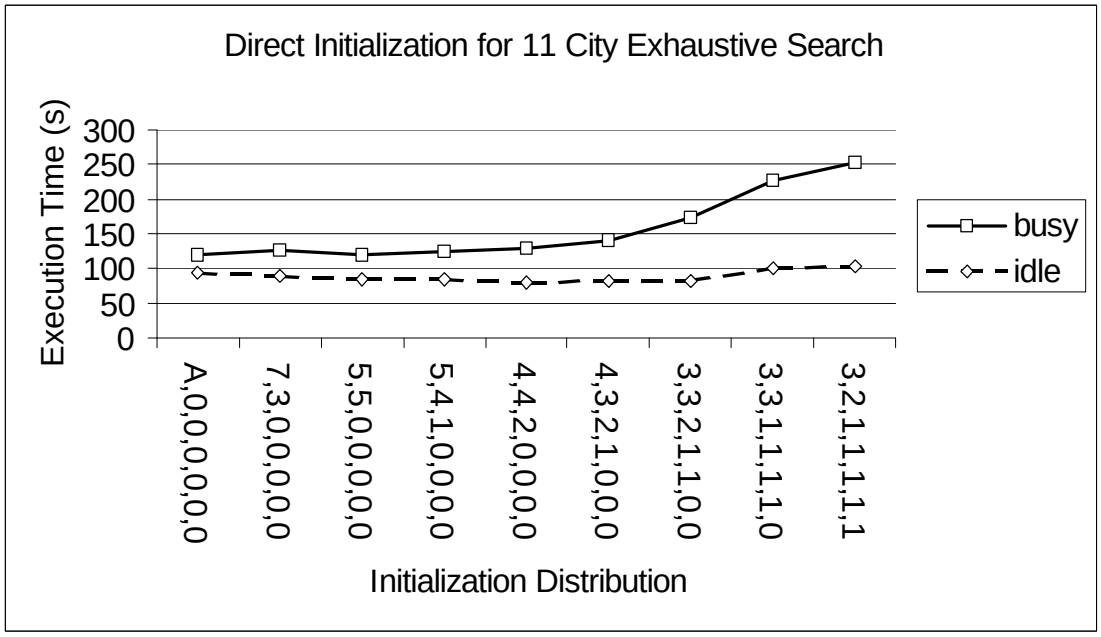


Figure 5.11. Initialization Comparison for an Exhaustive Search.

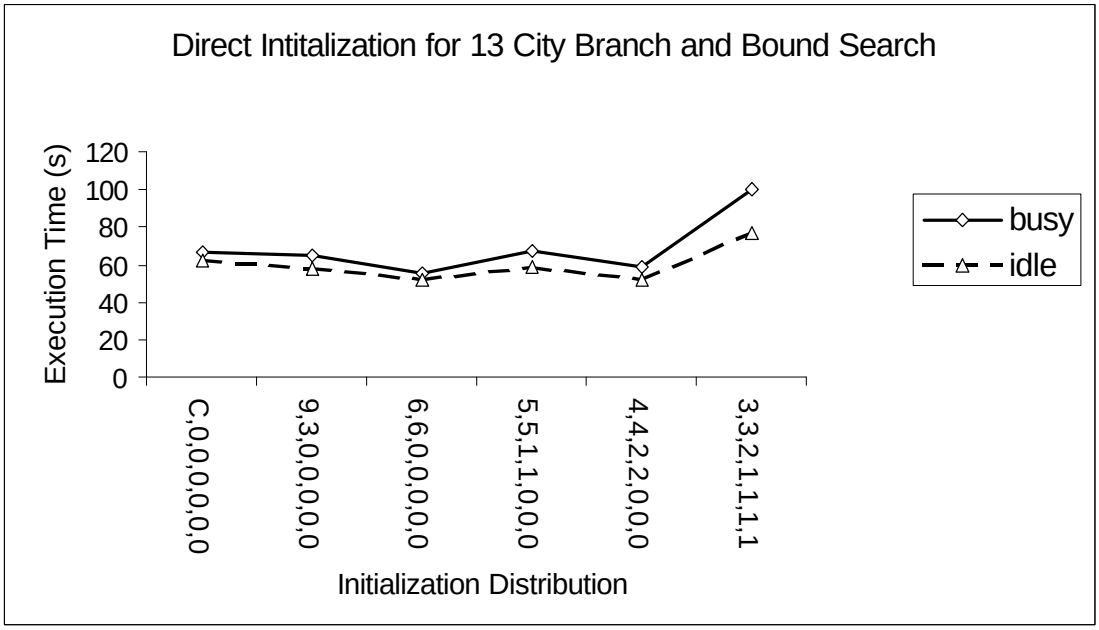


Figure 5.12. Initialization Comparison for a Branch and Bound Search.

5.2.5 Branch and Bound Testing

As previously stated, two different type of searches were implemented: the exhaustive depth first search and best distance branch and bound depth first search. So far the discussion on HMC testing has been relevant to both search procedures. The branch and bound search procedure has an additional communication requirement. When a best distance branch and bound search is executing on a single workstation, the process always knows the best distance found so far. As the search proceeds down the search tree, a path distance calculation is made after each node expansion. This distance is checked against the best distance found so far; if the current search path distance is greater than the best distance so far, the path is discarded and backtracking occurs.

For HMC-based branch and bound searches the best distance found so far may not be known to all systems at the same point in time. It is necessary during branch and bound testing to communicate the best distance found so far to each workstation whenever a workstation finds a better distance. In order to determine the effects of best distance communications, three tests were run. The first test is a "passive" approach where best distances are contained in the same messages that request work and the same messages that respond to work requests. The next approach is more active, where each time a better distance is found the workstation explicitly communicates this information to all HMC workstations. The additional communications overhead must be minimized. The final approach is a test to determine the effects of not passing the

best distance around the HMC (i.e., only local best distance is used by each

workstation). Tables 5.8, 5.9 and 5.10 contain the results of the three tests.

Table 5.8. Passive Best Distance Updating for Branch and Bound Searches.

# Systems in HMC	Average Execution Time (s)	Minimum Execution Time (s)	Maximum Execution Time (s)	Speedup vs Pentium 100	Total # Paths Processed
2	378.8	362	399	1.82	17502200
3	318.7	302	341	2.16	17485600
4	283.9	270	297	2.43	17554100
5	273.6	254	292	2.52	17573900
6	275.0	265	287	2.50	17368800
7	291.1	271	317	2.37	17677300

Table 5.9. Active Best Distance Updating for Branch and Bound Searches.

# Systems in HMC	Average Execution Times (s)	Minimum Execution Time (s)	Maximum Execution Time (s)	Speedup vs Pentium 100	Total # Paths Processed
2	381	362	404	1.81	17628900
3	315	303	340	2.19	17287800
4	287	277	314	2.40	17810400
5	270	255	285	2.55	17409100
6	278	261	304	2.46	17527100
7	286	263	322	2.40	17593000

Table 5.10. No Best Distance Updating for Branch and Bound Searches.

# Systems in HMC	Average Execution Time (s)	Minimum Execution Time (s)	Maximum Execution Time (s)	Speedup vs Pentium 100	Total # of Paths Processed
2	405	394	414	1.70	18849700
3	367	341	385	1.88	20361200
4	338	329	348	2.04	21377400
5	330	311	311	2.09	22157100
6	342	330	330	2.02	22574000
7	357	342	342	1.93	23445100

Branch and bound testing produced more variation on execution times than exhaustive depth first search testing (see Figure 5.13). The standard deviation for the branch and bound test was 4.45 and the average execution time was 60.59s. For the exhaustive test, the standard deviation was 1.79 with an average execution time of 92.74s. Even with the longer execution times, the exhaustive depth first search had more predictable execution times. As a result, exhaustive testing was done whenever comparisons were made between different load balancing schemes. When branch and bound tests were run under the different load balancing schemes comparisons were made, but the true purpose of the test was to determine if the character of the branch and bound search affected each load balancing scheme differently. During branch and bound testing, the queue pop count, or number of paths processed, was monitored.

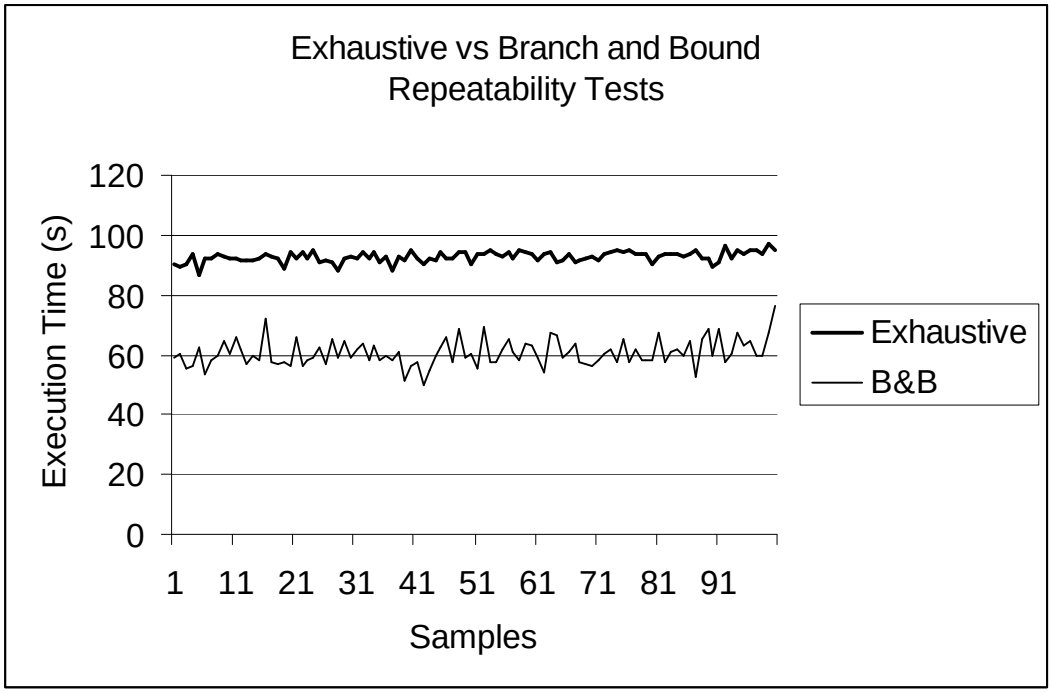


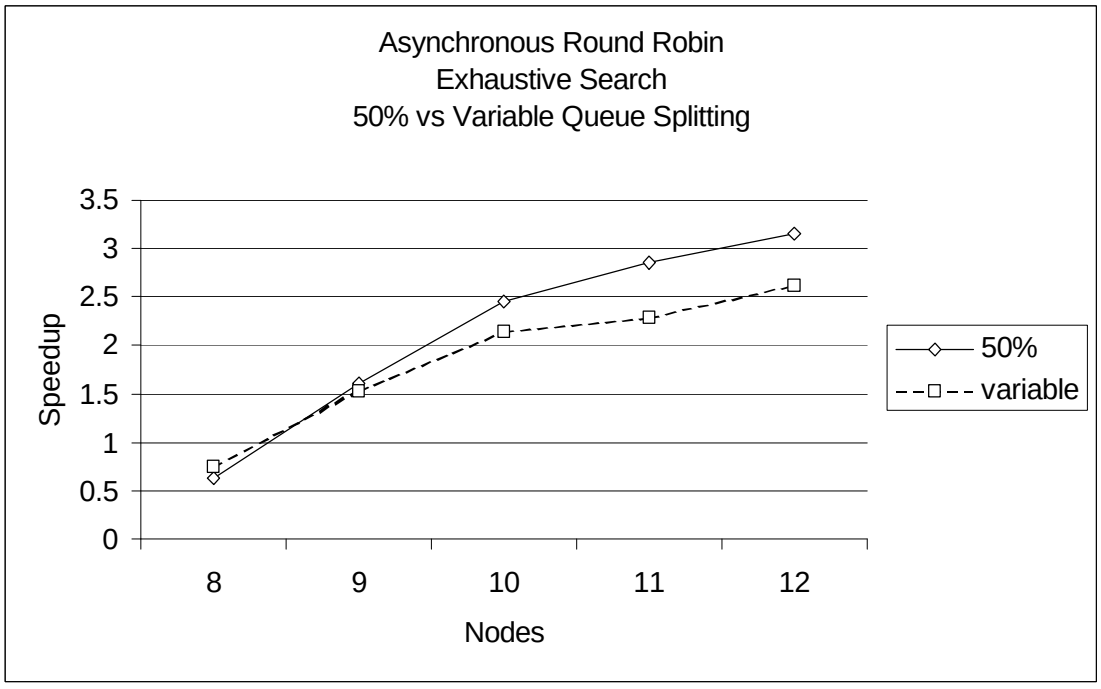
Figure 5.13. Comparison of Randomness of Execution Times

This was done in order to determine if any superlinear speedup effects could be observed. For the passive test on a 5 workstation HMC an execution time was 254s and the total number of paths processed was 16067377 paths. For the corresponding single workstation case the total was 16719163 paths. This is evidence of superlinear speedup, where the parallel algorithm is able to reduce the size of the search space with respect to the sequential algorithm. This reduction in search space can increase the speedup over and above the speedup increase from the additional workstations.

5.2.6 Queue Splitting

The next series of tests involves varying the queue splitting index based on each

workstation's performance index. All of the tests before this point have been all carried out with a fixed queue splitting value of 50%, disregarding the performance indexes. For idle initiated queue splitting, queue sizes are usually around 6 when splitting occur. With a 10 to 1 difference in performance between a Pentium and a 386 the resulting split would give 6 paths to the Pentium and 0 paths to the 386. Therefore unless a workstation has enough work to share, queue splitting may not occur. For busy initiated searches the queue splitting occurs at larger queue sizes, 20 or more. Therefore, a 0 in the queue splitting calculation should be rare. The testing was done for exhaustive depth first searches on an HMC configuration that included all the systems. The results for idle initiated queue splitting are in Figures 5.14 and 5.15. For



busy initiated Figure 5.14. Graph of Queue Splitting Performance.

queue splitting the same trends were observed.

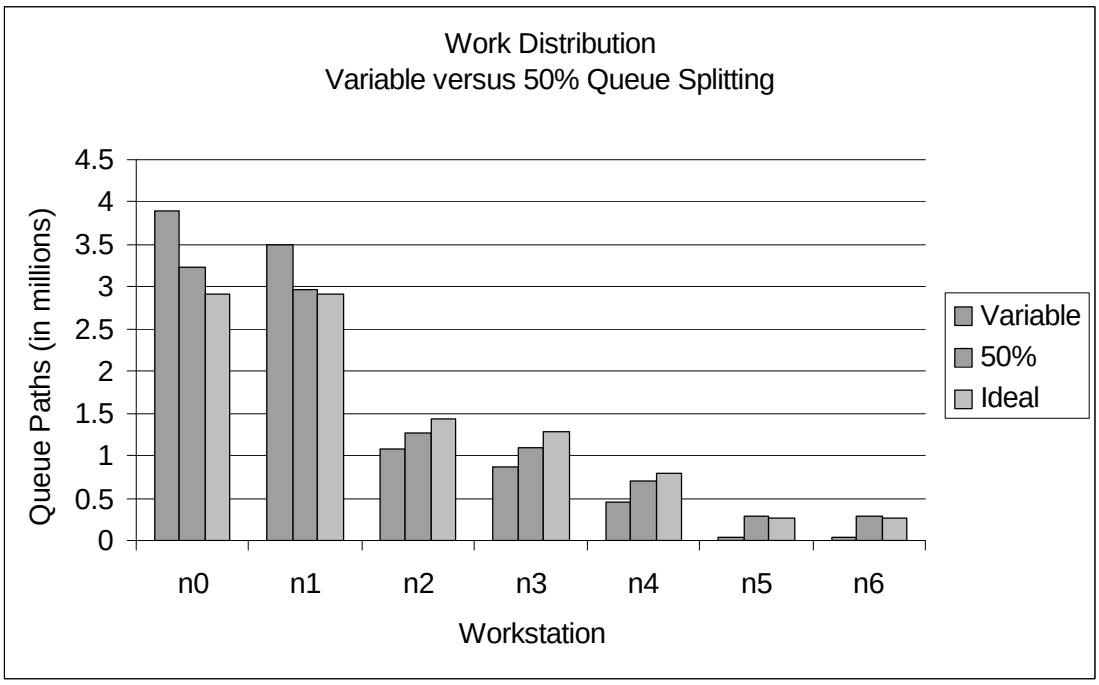


Figure 5.15. Path Count for each Workstation.

5.2.7 ARR, RP and GRR Polling

A series of tests on 7 workstations using asynchronous round robin (ARR), random polling (RP), and global round robin (GRR) was conducted. The tests were done with all parameters fixed and only the method of selecting a system for a work request was variable. Although the tests were carried out using both exhaustive search and branch and bound search algorithms, the exhaustive search fixes the search tree size, which eliminates another variable when comparing the various polling techniques.

One of the problems with ARR is the placement of the workstations, since work sharing requests are generated starting with the nearest neighbor. Because of the large number of workstation orderings it was decided that the workstations would be

arranged in only one configuration, from fastest to slowest. The result is the nearest neighbor to the root server, one of the Pentium 100s, is a 386/DX40. The RP and GRR polling methods should reveal whether or not workstation placement in the HMC is important, at least for this problem domain.

GRR polling requires additional communications to take place between each workstation and the root server, which manages the global polling index. It was at this point, during GRR testing, that the message buffers started overflowing on the root server and LAM/MPI had to be rebuilt. Tests already completed had to be rerun in order to ensure that the recompile did not affect the results.

Only during GRR testing was the workstation order varied, this was done in order to evaluate the effects on the HMC of workstations of different capability functioning as the root server. The direct initialization method was used to shift the search space processing away from the root server to the second and third workstations (see Table 5.11 for the results). The results of the polling tests are graphed in Figure.5.16.

Table 5.11. GRR Polling with Different Root Workstations.

Workstation	P100	486/100	486/66	486/40	386/40
Execution Time (s)	12.93	9.75	10.30	10.78	11.45

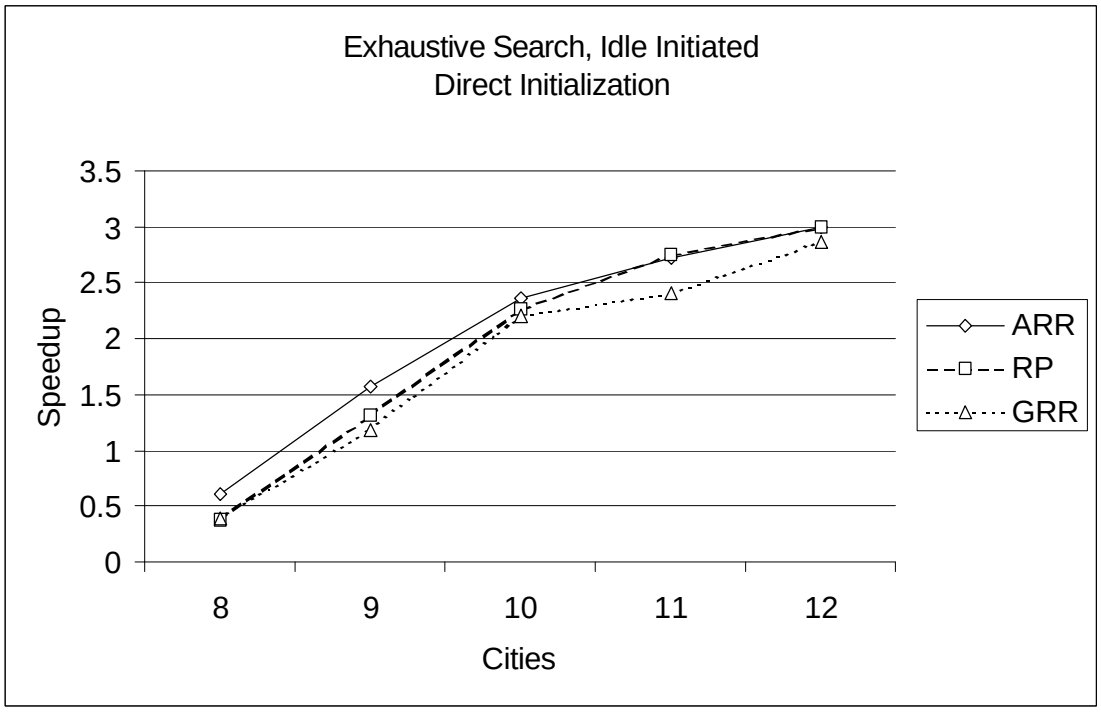


Figure 5.16. Comparison of Polling Techniques.

5.2.8 Switch versus Coax Testing

This set of tests involve a comparison of HMC performance between the closed and open HMC configuration using the Ethernet switch, and a closed and open coax based network configuration. An open HMC receives broadcast traffic from the campus network as opposed to the closed HMC which is physically disconnected from the campus network during testing. The results of this test are in Table 5.12.

Table 5.12. Coax Network versus Ethernet Switch Network .

Cities	8	9	10	11	12
Open Coax	.44	1.78	11.06	92.12	982.58
Closed Coax	.40	1.69	11.15	91.59	974.75
Open Switch	.41	1.74	11.27	92.53	979.41
Closed Switch	.42	1.71	11.05	92.25	972.17

5.2.9 Problem Size Testing

For most of the testing period, the problem size was kept constant. An execution time of 10s to 30s seemed a reasonable length for debugging purposes. During these short test periods it was obvious that the two slower workstations were of no real value in the HMC. It was only as the problem size was increased that it appeared that the slower workstations were a benefit to the HMC. A series of problem size tests were carried out in which the number of workstations in the HMC were varied with the problem size (see Figures 5.17 and 5.18).

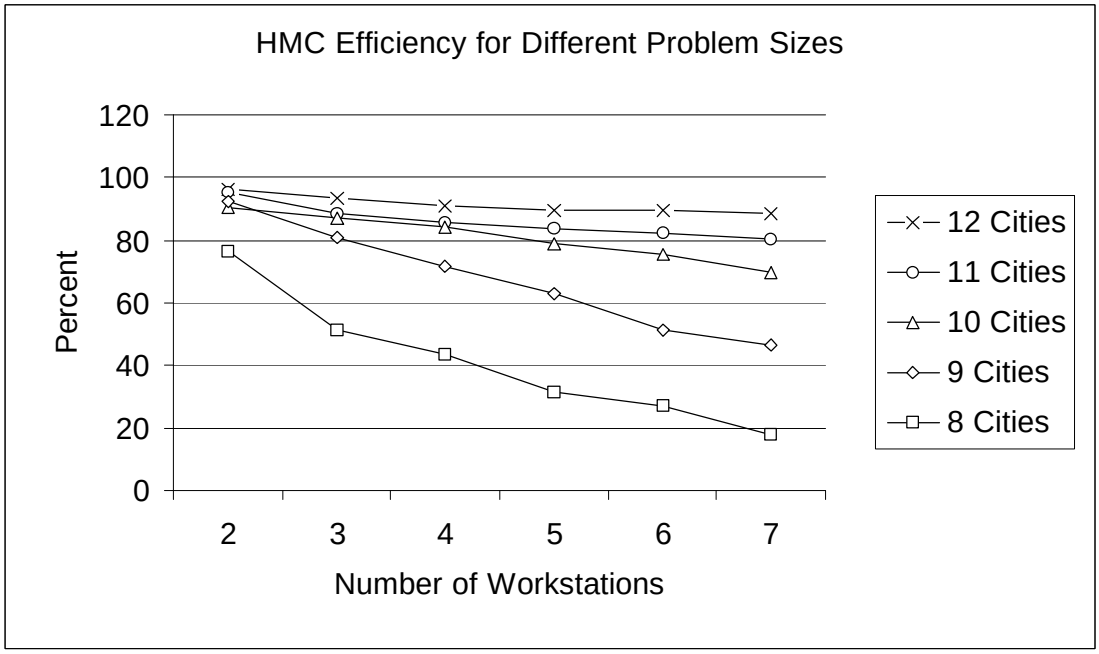


Figure 5.17. Effect of Problem Size on HMC Configuration.

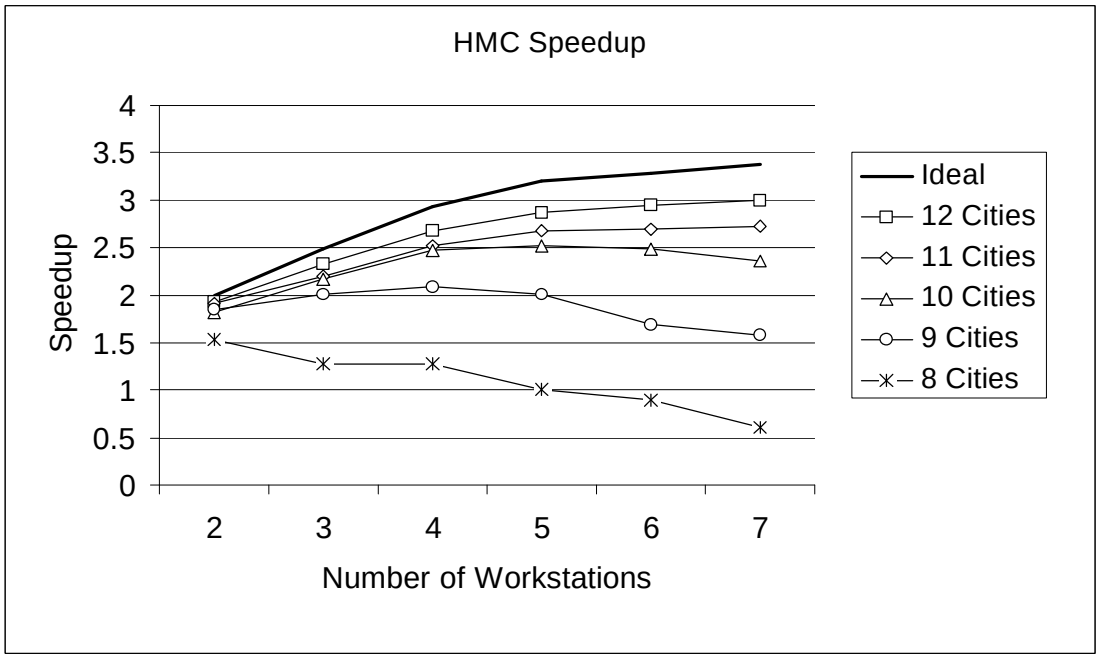


Figure 5.18. Speedup Curves for Different Problem Sizes on HMC.

5.2.10 Static Testing

The static testing algorithm was partially a breath first search (BFS) and partially a DFS. In order to improve the static load balancing the search tree was expanded using BFS to the second level, the root node being at level zero. The result was between 56 and 110 paths for distribution among the workstations. Each workstation executes the same BFS and then selects its own portion of the expanded paths for DFS. The accuracy in dividing up the second level fluctuated due to rounding quantization. For example, for a 12 city search, the slowest workstation should have received 4.4 paths but got only 4 since fractions of paths are not possible. By going deeper in the search tree it is possible to improve performance but the end result is more likely a series of compile, tune and execute sessions. The performance of static load balancing versus the best dynamic load balancing session is compiled in Table 5.13 and 5.14.

Table 5.13. Exhaustive Search for Static and Dynamic Load Balancing.

Nodes	8	9	10	11	12
Static	.09	.90	9.46	84.04	1047.37
Dynamic	.38	1.30	8.78	84.55	941.57

Table 5.14. Branch and Bound Search for Static and Dynamic Load Balancing.

Nodes	8	9	10	11	12	13	14
Static (s)	.04	.24	1.02	4.84	28.26	108.46	692.46
Dynamic (s)	.22	.40	.86	3.45	14.81	54.19	253.65

CHAPTER VI

ANALYSIS OF THE RESULTS

One of the goals of this thesis was to develop a parallel application in which the load on the HMC is automatically and dynamically controlled. Experiments were conducted using various dynamic load balancing schemes. Within each of these schemes, “delay counters” were introduced as a means of tuning the frequencies at which workstations attempt to shed or accept work. The testing described in the previous chapter revealed that once these delay counters are set above a minimum value, they have very little effect on execution times, until the counters start to become very large (1000+). For example, for one set of tests a countdown value of 100 yielded an execution time of 93.4 seconds and a countdown value of 1000 yielded an execution time of 98.5 seconds. The final choice was to set the counters at 100 for both the get work request and check for work calls.

The absence of using delay counters corresponds to the case where both counters of Figures 5.3 and 5.4 are set to zero. Thus, had the HMC been allowed to operate without the time delay counters, execution times would have been greater. Based on Figures 5.3 and 5.4, there is a substantial penalty that occurs when a workstation checks too often for incoming requests. It is the asynchronous communication model that creates the need to tune this action.

The effects of queue splitting strategies was also determined by extensive testing, it was a surprise that the best value ended up so low for idle initiated searches.

For example, for a 14 city TSP, the maximum possible queue size is 78. The best busy threshold value for idle initiated searches (determined by empirical studies) was between 2 and 6 (see Figures 5.5 and 5.6). Once the busy threshold gets above 6, good search performance can be maintained only by raising the idle-threshold so that the difference of 6 remains constant. Busy initiated searches behaved quite differently. The best performance occurs around a busy threshold of 15 (see Figure 5.7 and 5.8). This is because of the increased communication overhead that occurs with low busy threshold values (systems that should be working are instead frequently trying to share work). The busy initiated search also has the disadvantage that it is difficult to terminate. In contrast, it is fairly easy to have a selected root workstation terminate a search during idle initiated searches, because it only has to wait for a work request from all the other workstations in the HMC. However, for busy initiated searches, a series of requests for termination has to be initiated at some point by the root workstation. The result is an additional communication overhead at the end of a search.

The root initialization and direct initialization behaved differently for the idle initiated searches as compared to the busy initiated searches. Idle initiated completed faster when direct initialization was used. For busy initiated, direct initialization performed poorly for a 10 city TSP problems and larger (see Figure 5.10). For a 9 cities TSP or smaller the busy initiated searches out performed idle initiated queue sharing using direct initialization. Busy initiated benefitted from the fact that the busy threshold was high enough that the busy initiated requests were not satisfied before the search completed. The result is that the search completes on the initialized workstations

before queue sharing occurs. For idle initiated, requests for work are made immediately by idle workstations. This interferes with the initialized workstation's ability to complete the search before queue sharing occur. This indicates that for small search spaces, queue sharing is not productive.

Asynchronous round robin (ARR) polling and random polling (RP) had comparable performance results, and global round robin (GRR) performed slightly poorer (see Figure 5.16). The reason for the poorer GRR performance is communication overhead. Using a selected workstation to distribute the polling index to all other workstations (as done in GRR) increased the communication overhead by adding a time delay associated with each poll. The performance of GRR could be improved by setting up one of the slower workstations as the root server and then start a search by directly initializing several of the faster workstations (see Table 5.11).

Variable queue splitting did not perform nearly as well as fixed queue splitting. The chart on work distribution (see Figure 5.15) reveals the reason for this. Ideally each workstation should process an amount of work proportional to its capability. Under variable queue splitting, this did not occur. The reason for this is because it is more effective to distribute the work based on a relationship between the amount of polling a workstation does and the workstation's capability to complete some amount of work than on the percentage of work shared each time the queue is split. Even though 108,505,111 paths are processed during a 12 city exhaustive search, the maximum possible queue size is 56. Thus, it is often not possible to get a decent amount of work during variable queue splitting because there is such a large ratio

between the total search space size and the queue size.

There was little difference in performance between the passive best distance message passing and active best distance message passing for the branch and bound search. The active best distance communication search was about 3% faster than the passive based search. It is important that the best distance gets communicated between workstations, this is evident from the test where the best distance is not communicated at all (see Table 5.10).

The problem size and HMC efficiency are closely linked. Figure 5.17 reveals that as the problem size is increased, HMC efficiency improves. For small problem sizes, the percentage of time spent performing communication is high. For larger problem sizes, although the amount of time spent communicating increases, the percentage of time spent communicating decreases. From Tables 6.1 and 6.2 it is evident that two things are happening. First, for smaller problem sizes more messages are sent per second, this is due to the short execution time.

Table 6.1 Message per second, 7 workstation HMC.

Cities	8	9	10	11	12
Messages/second	211	224	109	44	9.3

Table 6.2 Rate of Success Requesting Work, 7 workstation HMC

Cities	8	9	10	11	12
Success Rate	.117	.288	.348	.406	.447

Secondly, small problem sizes get more negative responses to work requests. Figure 5.17 also reveals that as more workstations are added, the efficiency always goes down. This does not mean that execution time remains constant or goes up. Figure 5.18 shows that speedup increases as the number of HMC workstations increase. What the efficiency curves indicate is that as workstations are added to the HMC, the total communication overhead of the HMC is increasing.

The static loading tests revealed that it is possible to make good guesses and end up with good or better execution times than dynamic load balancing, but as the problem size gets larger or changes dynamically (i.e., branch and bound), static load balancing soon loses its effectiveness (see Tables 5.13 and 5.14).

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

It is evident from the data collected and presented in this thesis that for depth first exhaustive searches on this HMC, idle initiated queue sharing is preferred over busy initiated queue sharing for significant problem sizes. Also evident through out testing is the impact of the two 386s is varied at times: they have a negative impact on HMC performance under some conditions and a positive impact under other conditions.

The size of the packets, which where monitored at less than the size of one Ethernet packet (1500 bytes), made the use of the Ethernet switch no better than standard coax bus-based connections. Even with an open HMC, performance of the HMC was not substantially affected by the campus network broadcast traffic. Part of the reason for this may be a change in the campus network configuration that occurred shortly before HMC testing started. At the start of this thesis, the campus network was not sub-netted, therefore broadcast traffic from any system on the network went to every other system on the network. Shortly before the start of testing the engineering network segment was sub-netted with a router. This router blocked broadcast traffic from systems outside the engineering sub-net, the result was a dramatic drop in network broadcast traffic.

The final conclusion is that low-cost off-the-shelf workstations and network hardware was successfully implemented and utilized in an HMC.

7.2 Future Work

Future work includes a second phase of testing of applications that require large message packets. It is believed that an algorithm involving matrix multiplication with striped partitioning could be implemented on the HMC. Large message packets should provide a better understanding of the potential benefits of a switched network in an HMC configuration (instead of a bus-based network).

Additional future work should investigate the installation of LAM/MPI on other UNIX platforms available in the Department of Computer Science. This type of HMC configuration should work well with the depth first search implemented in this thesis. The network packets created by queue splitting are small and network traffic should not be a major factor.

Another area of investigation is PVM (Parallel Virtual Machine). Of particular interest is the distributed shared memory capability of PVM. It would be interesting to try algorithms on the HMC that can take advantage of a distributed shared memory HMC.

REFERENCES

[1] Byte UNIX Benchmark Program, http://www.silkroad.com/linux-bm.html, 1992.

[2] M. Cermele, M. Colajanni, G. Necci, "Dynamic Load Balancing of Distributed SPMD Computations with Explicit Message-Passing," *Sixth Heterogeneous Computing Workshop*, IEEE Society Press, CA. Apr. 1997

[3] J. Choi, *"NP-complete Problems"*, http://csc.hanyang.ac.kr/~jmchoi/class/a...e/node7.html, 1996.

[4] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[5] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister, "A Single-Program-Multiple-Data Computational Model for EXPEX/FORTRAN," *Parallel Computing*, Vol. 7, pp. 11-24, Apr. 1988.

[6] H. Dietz, *"Linux Parallel Processing Using Clusters,"* Purdue University School of Electrical and Computing Engineering, http://yara.ecn.purdue.edu/~pplinux/ppcluster.html, 1996.

[7] N. Doss, " *Message Passing Interface (MPI) FAQ,*" ftp://rtfm.mit.edu/pub/usenet/news.answers/mpi-faq, May 1996.

[8] M.J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, Vol. 54, No. 12, pp. 1901-1909, Dec. 1996.

[9] G.A. Geist, J.A. Kohl, and P.M. Papadopoulos "PVM and MPI: a Comparison of Features," *Calculateurs Paralleles*, Vol. 8, No. 2, pp. 137-150, June 1996.

[10] G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.Sunderam, *PVM Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*, The MIT Press, Cambridge, MA, 1994.

[11] M. Ginsberg, *Essentials of Artificial Intelligence*, Morgan Kaufmann Publishers, San Francisco, CA, 1993.

[12] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI. Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, MA, 1994.

[13] D. Henrich, "Initialization of Parallel Branch-and-Bound Algorithms," *Second International Workshop on Parallel Processing for Artificial Intelligence*, Chambery, France, Aug. 29 1993.

[14] P. Hughes, "What is Linux?," *Linux Journal Special Issue, 1997 Buyer's Guide*, Specialized Systems Consultants Inc., Seattle WA., pp 5, June 1997.

[15] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, The Benjamin/Cummings Publishing Co., Redwood City, CA, 1994.

[16] Lawrence Livermore, "*The ASCI sPPM benchmark code readme file,*" http://www.llnl.goc/asci_benchmarks/asci/limited/ppm/sppm_readme.html, Feb. 1996.

[17] J. Lawton, J. Brosnan, M. Doyle, S. Riordain, and T. Reddin, "Building a High-performance Message-Passing System for MEMORY CHANNEL Clusters," *Digital Technical Journal*, Vol. 8, No. 2, pp. 96-115, 1996.

[18] MPI Forum, "MPI: A Message-Passing Interface Standard," *International Journal of Supercomputing Applications*, Vol. 8 No. 3/4, Fall/Winter 1994 updated; University of Tennessee, Knoxville Tennessee 1995.

[19] *"MPI Primer/ Developing With Lam,"* Ohio LAM 6.1 Ohio Supercomputer Center, Ohio State University, Nov. 11 1996.

[20] *"XMPI - A Run/Debug GUI for MPI,"* Ohio Supercomputer Center, Ohio State University, 1996.

[21] J. Nievergelt, R. Gasser, F. Maser, and C. Wirth, *"All the Needles in a Haystack: Can Exhaustive Search Overcome Combinatorial Chaos?,"* http://nobi.ethz.ch/febi/ex_search_paper/paper.html, 1995.

[22] M. Palis, J. Liou, and D. Wei. "Task Clustering and Scheduling for Distributed Memory Parallel Architectures," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No.1, pp. 46-55, Jan. 1996

[23] G. Pfister, *In Search of Clusters. The Coming Battle in Lowly Parallel Computing*, Prentice-Hall, Upper Saddle River, NJ, 1995.

[24] S.P. Rajpal and S. Kumar, "Parallel Heuristic Search Algorithm for Message Passing Multiprocessor Systems," *Computer Science and Informatics*, Vol. 23 No. 4, pp. 7-18, 1993.

[25] A. Reinefeld and V. Schnecke, "*Work-Load Balancing in Highly Parallel Depth-First Search*", Proc. Scalable High Performance Computing Conf. SHPCC'94, IEEE Comp. Sc. Press, pp. 1-8, 1994.

[26] B. Shirazi, A. Hurson, and K. Kavi, *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press. pp. 1-5, Los Alamitos, CA, 1995.

[27] N.G. Shivaratri, P. Krueger, and M. Singhal, *"Load Distributing for Locally Distributed Systems,"* IEEE Computer, pp. 33-44, Dec. 1992.

[28] M. Snir, *MPI: The Complete Reference*, MIT Press, Cambridge, Mass. 1996.

[29] P. Steenkiste, "Network-Based Multicomputers: A Practical Supercomputer Architecture," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 8, pp. 861-875, Aug. 1996.

[30] T. Sterling *et al.*, "Beowulf: A Parallel Workstation for Scientific Computation," *Proceedings of the 1995 International Conference of Parallel Processing*, Vol. I, Aug. 1995, pp. I-11 - I-14 (also see http://cesdis.gsfc.nasa.gov/linux/beowulf/icpp95.html,beowulf.html,beowulf1.html,white2.html).

[31] "SuperStack II Desktop Switch High Performance, Dedicated Switching for End Stations," *3Com Buyers Guide*, Nov. 1996.

[32] A. Tucker, *Applied Combinatorics*, (3rd ed), John Wiley & Sons, Toronto, Canada, 1995.

[33] X. Wang and E.K. Blum, "Parallel Execution of Iterative Computations on Workstation Clusters," *Journal of Parallel and Distributed Computing*, Vol. 34, No. 2, pp. 218-226, 1996.

[34] F. Westerberg, "*Load Balancing and PRAMs,*" http://www-dsc.doc.ic.ac.uk/~nd/surprise_95/journal/vol2/fcw/article2.html

[35] M. Welsh and L. Kaufman, *Running Linux*, O'Reilly & Associates, Inc., Sebastopol, CA, 1995.

[36] Z. Xu and K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on IBM SP2," *IEEE Parallel & Distributed Technology*, pp. 9-23, Spring 1996.

APPENDIX A

LIST OF MPI PACKAGES

MPI Implementations compiled by Ohio Super Computer Center

CHIMP/MPI

Supplier: Edinburgh Parallel Computing Centre (EPCC)
Current Version: v2.1.1c
MPI 1.1 Functions Implemented: all except MPI_Intercomm_merge
Source Code Available?: yes
Help Contact: chimp@epcc.ed.ac.uk
Supported Systems:

Sun SPARC (SunOS 4, Solaris 5), SGI (IRIX 4, IRIX 5), DEC Alpha (Digital UNIX),HP PA-RISC (HP-UX), IBM RS/6000 (AIX), Sequent Symmetry (DYNIX), Meiko T800, i860 and SPARC Computing Surfaces, Meiko CS-2

CRI/EPCC MPI for Cray T3D

Supplier: Cray Research Incorporated via Edinburgh Parallel Computing Centre
Current Version: 1.5a
MPI 1.1 Functions Implemented: all
Source Code Available?: libraries and headers only
Help Contact: t3dmpi@epcc.ed.ac.uk
Supported Systems: Cray T3D

LAM

Supplier: Ohio Supercomputer Center
Current Version: 6.1
MPI 1.1 Functions Implemented: all
Source Code Available?: yes
Help Contact: lam@tbag.osc.edu
Supported Systems:

Sun (Solaris 5.4-5), SGI (IRIX 6.2), IBM RS/6000 (AIX V3R2), DEC Alpha(OSF/1 V4.0), HP PA-RISC (HP-UX 10.01), Intel X86 (LINUX v2.0.24)

MPIAP

Supplier: Australian National University - CAP Research Program
Current Version: 1.6
MPI 1.1 Functions Implemented: all
Source Code Available?: yes
Help Contact: sits@cafe.anu.edu.au
Supported Systems: Fujitsu AP1000 (CellOS), AP1000+ (CellOS,

APLINUX)

MPICH

Supplier: Argonne National Laboratory & Mississippi State University
Current Version: 1.0.13
MPI 1.1 Functions Implemented: all
Source Code Available?: yes
Help Contact: mpi-bugs@mcs.anl.gov
Supported Systems:

MPP's, IBM SP, Intel Paragon, SGI Onyx, Challenge and Power Challenge, Convex (HP) Exemplar, NCUBE, Meiko CS-2, TMC CM-5, Cray T3D, TCP-connected networks of: SUN (SunOS and Solaris), SGI, HP, RS/6000, DEC Alpha, Cray C-90, most above machines

MPICH/NT

Supplier: Mississippi State University
Current Version: 0.8b
MPI 1.1 Functions Implemented: MPICH
Source Code Available?: yes
Help Contact: shane@erc.msstate.edu, boris@cs.msstate.edu
Supported Systems: Intel X86 Windows NT 3.51, 4.0b1

MPI-FM

Supplier: University of Illinois, Concurrent Systems Architecture Group
Current Version: 1.0
MPI 1.1 Functions Implemented: all
Source Code Available?: (contact for more information)
Help Contact: fast-messages@red-herring.cs.uiuc.edu
Supported Systems: Suns w/ Myrinet (both SunOS and Solaris)

RACE-MPI

Supplier: Hughes Aircraft Co.
Current Version: 3.0
MPI 1.1 Functions Implemented: all from MPICH July 22 1994, exceptMPI_Bsend
Source Code Available?: yes
Help Contact: llewins@msmail4.hac.com
Supported Systems: Mercury Race i860

W32MPI

Supplier: Universidade de Coimbra - Portugal

Current Version: v0.8b
MPI 1.1 Functions Implemented: same as MPICH 1.0.12 (ch_p4)
Source Code Available?: no
Help Contact: fafe@eden.dei.uc.pt
Supported Systems: Windows 95 and NT

Vendor MPI Implementations

Entries in this section are either provided by computer vendors for their own products or are provided by independent software vendors. They may or may not be bundled, or have additional cost.

Alpha Data MPI
Supplier: Alpha Data parallel systems ltd.
Current Version: 1v0
MPI 1.1 Functions Implemented: all that MPICH 1.0.12 offers
Help Contact: sales@alphadata.co.uk
Supported Systems: Alpha Data Systems based on AD164, AD64 and AD66

HP MPI
Supplier: Hewlett-Packard, Co.; Convex Division
Current Version: 1.1
MPI 1.1 Functions Implemented: all
Help Contact: romero@convex.hp.com
Supported Systems: SPP1200, SPP1600, D-, K-, S-, and X-Class Exemplar

IBM Parallel Environment for AIX - MPI Library
Supplier: IBM Corporation
Current Version: 2.1
Ordering Information: product number 5765-543
MPI 1.1 Functions Implemented: all
Help Contact: your IBM rep or Sharan Scanlon (sp_info at kgn.ibm.com)
Supported Systems: Risc System/6000, RS/6000 SP

MPI
Supplier: Hitachi, Ltd.
Current Version: 1
MPI 1.1 Functions Implemented: all

Help Contact: haradake@soft.hitachi.co.jp
Supported Systems: Hitachi SR2201

MPI

Supplier: Silicon Graphics Inc.
Current Version: 2.0
Ordering Information: Array 2.0 CD, order number SC4-PCAS-2.0
MPI 1.1 Functions Implemented: all except MPI_Cancel, MPI_Test_cancelled
Help Contact: salo@sgi.com
Supported Systems: all 64-bit SGI machines running IRIX 6.2

MPI/DE

Supplier: NEC Corporation
Current Version: 0.96
MPI 1.1 Functions Implemented: all
Help Contact: denen@csl.cl.nec.co.jp
Supported Systems: NEC Cenju-3 (Cenju-3/DE)

Paragon OS R1.4

Supplier: Intel Corp.
Current Version: R1.4
MPI 1.1 Functions Implemented: all that MPICH 1.0.12 offers
Help Contact: support@co.intel.com
Supported Systems: Paragon

T.MPI

Supplier: Telmat Multinode
Current Version: 1
MPI 1.1 Functions Implemented: all that MPICH 1.0.11 offers
Help Contact: flaagel@telmat.fr
Supported Systems: Telmat TN110 and TN300 series (T9000 transputer)

TransMPI

Supplier: PERIASTRON
Current Version: 1.0-C
MPI 1.1 functions implemented: all of MPI-1.0, C bindings only
Help Contact: thomasd@netcom.com
Supported Systems: all transputers 32 bits or above

APPENDIX B

SOURCE CODE

```
/* dfs_ex_idle.h */
/* include file for idle initiated, exhaustive search program */
/* Author: Per Andersen     */
/* June 1997 */

#define STARTTAG            1
#define GETWORKTAG          2
#define WORKTAG             3
#define NOWORKTAG           4
#define ENDTAG              5
#define SENDWORKTAG         6
#define WORKREQTAG          7
#define QUITTAG             8

/* for a depth of 20 maximum queue length is 171 */
#define DEPTH               14
#define WORKBUFFER          1000
#define QUEUE_SPLITTER          2
#define TERMINATE_COUNT         100
#define GET_WORK_COUNT          100
#define WORK_REQUEST_COUNT      100
#define UPDATE_COUNT        100
/* because 20 nodes creates such a huge search space fix node matrix

   at 20x20, which has (20-1)! paths or about 6.22 billion paths */

/* work receive buffer DEPTH+2*180 a little extra in case I'm wrong */
int work[WORKBUFFER];
int work1[WORKBUFFER];
int buf1, buf2[3], buf3, buf4;

/* last to ints in depth is position of next entry and distance */
int queue[250][DEPTH + 2];
int best_trip[DEPTH + 2];
int nodes;              /* total number of vertices */
int root;               /* root vertice */
int queue_size;             /* size of queue */
int init_queue_size;
int max_queue_size;         /* max size of queue */
int queue_threshold;        /* queue splitting threshold */
int best_distance;
int best_distance1;
int idle_threshold;
```

```
int busy_threshold;
int ntasks;
int myrank;
double start;
double end;
int keep_asking;

int pending_recv[1];
int check_buffer[3];
int response_from_get_work;
/* can send up to ~10 queue pathes */
int enum_data[220];
int dist_factor[7];
int all_nodes[7];
int queue_init;
int get_work_counter[7];
int check_get_work_recv1[7], check_get_work_recv2[7],
check_get_work_amount[7];
int send_some_work_send1[7], send_some_work_send2[7],
send_some_work_amount[7];
int pop_counter;
int get_work_count, work_request_count;
int tour[14][14] =
{
   0, 108, 222, 211, 339, 257, 214, 316, 506, 395, 423, 548, 684, 525,
   108, 0, 117, 105, 282, 160, 138, 281, 443, 318, 331, 456, 559, 421,
   222, 117, 0, 19, 230, 111, 152, 314, 420, 281, 266, 390, 484, 330,
   211, 105, 19, 0, 247, 98, 134, 296, 407, 269, 265, 385, 482, 331,
   339, 282, 230, 247, 0, 384, 424, 541, 643, 502, 478, 581, 650, 489,
   257, 160, 111, 98, 384, 0, 77, 220, 309, 168, 175, 299, 407, 267,
   214, 138, 152, 134, 424, 77, 0, 163, 304, 183, 213, 335, 451, 326,
   316, 281, 314, 296, 541, 220, 163, 0, 216, 173, 237, 331, 502, 381,
   506, 443, 420, 407, 643, 309, 304, 216, 0, 141, 183, 178, 178, 285,
   395, 318, 281, 269, 502, 168, 183, 173, 141, 0, 72, 166, 290, 211,
   423, 331, 266, 265, 478, 175, 213, 237, 183, 72, 0, 125, 237, 139,
   548, 456, 390, 385, 581, 299, 335, 331, 178, 166, 125, 0, 126, 125,
   684, 559, 484, 482, 650, 407, 451, 502, 286, 290, 237, 126, 0, 161,
   525, 421, 330, 331, 489, 267, 326, 381, 285, 211, 139, 125, 161, 0};

/* functions supporting search algorithm */

int in_path(int *, int);
int push_path(int *, int, int);
```

```
int get_distance(int, int);
void push_queue(int *);
void init_queue();
void pop_queue(int *);
void delete_path(int *);
int last_node(int *);
int get_best_path(int *);
void print_queue();
void copy_path(int *, int *);
void print_path(int *apath);
void best_dfs_1();

void best_dfs_2();

/* functions supporting parallel processing */

void seed_servers();
void wait_for_start();
int test_for_termination();
void quit_program();
int check_for_work_request();
void send_some_work();
void get_work();
void get_all_results();
void split_queue();
int setup_queue();
void initialize_search();
void search_result();
void all_quit();
void print_results();
void send_termination();
void process_work_request();
void clear_recv();
int check_get_work();
void direct();
void expand_queue();
int reduce_queue_by(int);
void reduce_queue(int);
int check_node_work_status(int);
void update_distance();
void queue_reduction_parse();
```

```
/* dfs_ex_idle.c */
/* idle initiated, exhaustive search program */
/* Author: Per Andersen     */
/* June 1997 */

#include <mpi.h>
#include "dfs_ex_idle.h"
#define DEBUGGER         0

main(argc, argv)
int argc;
char *argv[];

{
   int some_server, i, total_pathes, try_all_nodes;
   int get_work_cntdwn, work_request_cntdwn, terminate_cntdwn;
   int get_work_counter_tl;
   int check_get_work_recv1_tl, check_get_work_recv2_tl,
check_get_work_amount_tl;
   int send_some_work_send1_tl, send_some_work_send2_tl,
send_some_work_amount_tl;

#if DEBUGGER
   double timers[12];
#endif

   MPI_Init(&argc, &argv);  /* initialize MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);          /* process rank,
0 thru N-1 */
   MPI_Comm_size(MPI_COMM_WORLD, &ntasks);  /* work space size */

/* pass the number of vertices from the command line */
   if (argc < 5) {
        printf("invalid syntax usage:  #nodes root_node idle_threshold
busy_threshold\n");
        MPI_Finalize();
        exit(0);
   }
   nodes = atoi(argv[1]);
   if ((nodes < 0) || (nodes > 14)) {
        printf("out of range node, %d\n", nodes);
        MPI_Finalize();
        exit(0);
```

```
    }
    root = atoi(argv[2]);
    if ((root < 0) || (root > 13)) {
        printf("out of range starting point, %d\n", root);
        MPI_Finalize();
        exit(0);
    }
    busy_threshold = atoi(argv[3]);
    idle_threshold = atoi(argv[4]);
    queue_init = atoi(argv[5]);
    get_work_count = atoi(argv[6]);
    work_request_count = atoi(argv[7]);
    total_pathes = 0;
    if (queue_init) {
        queue_reduction_parse();
        for (i = 0; i < ntasks; i++)
            total_pathes = total_pathes + dist_factor[i];
        if (total_pathes > (nodes - 1)) {
            printf("invalid direct distribution, max pathes %d\n", (nodes - 1));
            MPI_Finalize();
            exit(0);
        }
    }
    if (idle_threshold < 0) {
        printf("out of range idle threshold %d\n", idle_threshold);
        MPI_Finalize();
        exit(0);
    }
/* initalize some parameters */
#if DEBUGGER
    for (i = 0; i < 12; i++)
        timers[i] = 0;
#endif

    for (i = 0; i < ntasks; i++) {
        all_nodes[i] = 0;
        get_work_counter[i];
        check_get_work_recv1[i] = 0;
        check_get_work_recv2[i] = 0;
        check_get_work_amount[i] = 0;
        send_some_work_send1[i] = 0;
        send_some_work_send2[i] = 0;
        send_some_work_amount[i] = 0;
```

```
    }
    get_work_counter_tl = 0;
    check_get_work_recv1_tl = 0;
    check_get_work_recv2_tl = 0;
    check_get_work_amount_tl = 0;
    send_some_work_send1_tl = 0;
    send_some_work_send2_tl = 0;
    send_some_work_amount_tl = 0;
    pop_counter = 0;
    best_distance = 999999;
    best_distance1 = 999999;
    max_queue_size = 0;
    queue_size = 0;
    keep_asking = 1;
    bzero(best_trip, DEPTH + 2);

    if (myrank == 0) {
        init_queue();
        if (queue_init)
            direct();
        init_queue_size = queue_size;
        seed_servers();
    } else {
        wait_for_start();
    }

    terminate_cntdwn = TERMINATE_COUNT;
    work_request_cntdwn = work_request_count;
    get_work_cntdwn = get_work_count;
    response_from_get_work = 1;
    try_all_nodes = 0;

    start = MPI_Wtime();
    while (1) {
        if (keep_asking) {
            if (queue_size <= idle_threshold) {
                if (!get_work_cntdwn--) {
                    get_work_cntdwn = 0;
                    if (try_all_nodes++ < (ntasks - 1)) {
                        get_work_cntdwn = get_work_count;
                        try_all_nodes = 0;
                    }
#if DEBUGGER
```

```
                    timers[10] = MPI_Wtime();
#endif

                    get_work();

#if DEBUGGER
                    timers[1] = timers[1] + (MPI_Wtime() - timers[10]);
#endif

                }
                if (!response_from_get_work) {
                    keep_asking = check_node_work_status(check_get_work());
                }
            }
        }
#if DEBUGGER
        timers[8] = MPI_Wtime();
#endif
        if (myrank && (!terminate_cntdwn--)) {    /* only slave servers test for
termination */
            terminate_cntdwn = TERMINATE_COUNT;
            if (test_for_termination()) {
                quit_program();
                break;
            }
        }
#if DEBUGGER
        timers[9] = timers[9] + (MPI_Wtime() - timers[8]);
#endif

        if (!work_request_cntdwn--) {
#if DEBUGGER
            timers[4] = MPI_Wtime();
#endif
            work_request_cntdwn = work_request_count;
            process_work_request();
#if DEBUGGER
            timers[5] = timers[5] + (MPI_Wtime() - timers[4]);
#endif
        }
        if ((myrank == 0) && (!keep_asking)) {
            process_work_request();
            send_termination();
```

```
            get_all_results();
            break;
        }
/* do work inline rather than system call */
#if DEBUGGER
        timers[2] = MPI_Wtime();
#endif

/* best_dfs_1: one descendant at a time expansion */
/*      best_dfs_1(); */

/* best_dfs_2: full descendant expansion before continuing */
        best_dfs_2();

#if DEBUGGER
        timers[3] = timers[3] + (MPI_Wtime() - timers[2]);
#endif

    }
    end = MPI_Wtime();
    /* master has different point for capturing end time */
    if (!max_queue_size)
        printf("n%d queue size always 0 therefore did no work\n", myrank);
    if (pending_recv[0])
        clear_recv();
    printf("n%d # of requests for work: ", myrank);
    for (i = 0; i < ntasks; i++) {
        printf("%d ", get_work_counter[i]);
        get_work_counter_tl = get_work_counter_tl + get_work_counter[i];
    }
    printf("\n");
    printf("n%d # of responses to work requests received: ", myrank);
    for (i = 0; i < ntasks; i++) {
        printf("%d ", check_get_work_recv1[i]);
        check_get_work_recv1_tl = check_get_work_recv1_tl +
check_get_work_recv1[i];
    }
    printf("\n");
    printf("n%d # of work msgs received: ", myrank);
    for (i = 0; i < ntasks; i++) {
        printf("%d ", check_get_work_recv2[i]);
        check_get_work_recv2_tl = check_get_work_recv2_tl +
check_get_work_recv2[i];
```

```
    }
    printf("\n");
    printf("n%d # of integers received in work msgs: ", myrank);
    for (i = 0; i < ntasks; i++) {
        printf("%d ", check_get_work_amount[i]);
        check_get_work_amount_tl = check_get_work_amount_tl +
check_get_work_amount[i];
    }
    printf("\n");
    printf("n%d # of work request responses sent: ", myrank);
    for (i = 0; i < ntasks; i++) {
        printf("%d ", send_some_work_send1[i]);
        send_some_work_send1_tl = send_some_work_send1_tl +
send_some_work_send1[i];
    }
    printf("\n");
    printf("n%d # of work msgs sent: ", myrank);
    for (i = 0; i < ntasks; i++) {
        printf("%d ", send_some_work_send2[i]);
        send_some_work_send1_tl = send_some_work_send1_tl +
send_some_work_send1[i];
    }
    printf("\n");
    printf("n%d # integers sent in work msgs: ", myrank);
    for (i = 0; i < ntasks; i++) {
        printf("%d ", send_some_work_amount[i]);
        send_some_work_amount_tl = send_some_work_amount_tl +
send_some_work_amount[i];
    }
    printf("\n");
    printf("n%d pop count %d\n", myrank, pop_counter);
#if DEBUGGER
    printf("n%d time spent asking for work: %f\n", myrank, timers[1]);
    printf("n%d B&B time: %f\n", myrank, timers[3]);
    printf("n%d time spent checking for work requests: %f\n", myrank,
timers[5]);
    printf("n%d time spent sending best distance: %f\n", myrank, timers[7]);
    printf("n%d time spent testing for termination: %f\n", myrank, timers[9]);
#endif
    if (!myrank)
        print_results();
    MPI_Finalize();
    exit(0);
```

```
}

void seed_servers()
{
   int i, num_pathes;

   for (i = 1; i < ntasks; i++) {
        enum_data[0] = -1;
        if (queue_init)
           if (queue_size)
                reduce_queue(reduce_queue_by(i));

        MPI_Send(&enum_data[0],         /* message buffer */
                 220,            /* zero data item */
                 MPI_INT,     /* data item is an integer */
                 i,                /* destination process rank */
                 STARTTAG,/* user chosen message tag */
                 MPI_COMM_WORLD);    /* always use this */
   }
}

/* if one of these cases is reduced either more pathes will stay on n0
   or someone needs an increase see the include file for the pattern */
int reduce_queue_by(int id)
{
   int total_factor = 0, i;

   for (i = 0; i < ntasks; i++)
        total_factor = total_factor + dist_factor[i];
   return (dist_factor[id]);
}

void queue_reduction_parse()
{
   dist_factor[0] = queue_init / 1000000;
   dist_factor[1] = (queue_init - (queue_init / 1000000 * 1000000)) / 100000;
   dist_factor[2] = (queue_init - (queue_init / 100000 * 100000)) / 10000;
   dist_factor[3] = (queue_init - (queue_init / 10000 * 10000)) / 1000;
   dist_factor[4] = (queue_init - (queue_init / 1000 * 1000)) / 100;
   dist_factor[5] = (queue_init - (queue_init / 100 * 100)) / 10;
   dist_factor[6] = (queue_init - (queue_init / 10 * 10)) / 1;
}
```

```
void reduce_queue(int reduce)
{
   int temp_size;
   int i, j = 1, k;

   if (reduce == 0) {
        enum_data[0] = -1;
        return;
   }
   temp_size = queue_size - reduce;
   bzero(enum_data, 220);
   for (i = temp_size; i < queue_size; i++) {
        for (k = 0; k < DEPTH + 2; k++) {
            enum_data[j] = queue[i][k];
            j++;
        }
   }
   if (j > 220)
        printf("n%d enum_data buffer overflow\n", myrank);
   enum_data[0] = j;
   queue_size = temp_size;
}

void wait_for_start()
{
   int i;
   MPI_Status status;

   MPI_Recv(&enum_data[0],      /* message buffer */
            220,                /* zero data item */
            MPI_INT,            /* data item is a int */
            0,                  /* receive from 0 */
            STARTTAG,                /* receive STARTTAG messages only */
            MPI_COMM_WORLD,  /* always use this */
            &status);           /* info about received message */
   if (!(enum_data[0] == -1))
        expand_queue();
}

void expand_queue()
{
   int i, j = 1;
```

```
    if (enum_data[0] == 0)
        return;
    while (j < enum_data[0]) {
        for (i = 0; i < DEPTH + 2; i++) {
            queue[queue_size][i] = enum_data[j];
            j++;
        }
        queue_size++;
    }
}

int check_node_work_status(int actual_work)
{
    int i, work_request = 0;

    work_request = 0;
    if (queue_size == 0)
        all_nodes[myrank] = 1;
    if (check_buffer[0] == -1) {
        all_nodes[check_buffer[2]] = 1;
        for (i = 0; i < ntasks; i++) {
            if (all_nodes[i])
                work_request++;
        }
        if (work_request == ntasks) {
            return 0;
        }
    } else
        all_nodes[check_buffer[2]] = 0;
    return 1;
}

void clear_recv()
{
    int workreq[3];

    MPI_Send(&workreq,          /* message buffer */
             3,                 /* one data item */
             MPI_INT,           /* data item is an integer */
             myrank,            /* source is any server */
             WORKREQTAG,        /* get work tag */
             MPI_COMM_WORLD); /* always use this */
}
```

```
void process_work_request()
{
   int some_server;

   some_server = check_for_work_request();
   if (!(some_server == -1)) {
        send_some_work(some_server);
   }
}


int test_for_termination()
{
   int endflag, end;
   int prbflag;
   MPI_Request endreq;
   MPI_Status status, prbstatus;

/* if a Irecv is pending don't set it up again */
   MPI_Iprobe(0,
              ENDTAG,
              MPI_COMM_WORLD,
              &prbflag,
              &prbstatus);

   if (prbflag == 1) {
        MPI_Recv(&end,              /* message buffer */
                 1,             /* zero mesage item */
                 MPI_INT,       /* data item is a int */
                 0,             /* receive message from 0 */
                 ENDTAG,    /* receive ENDTAG messages only */
                 MPI_COMM_WORLD,      /* always use this */
                 &status);      /* MPI_Request structure for testing */
/* test for termination message */
        return 1;
   }
   return 0;
}

void get_work()
{

   static MPI_Request sendreq;
```

```
    MPI_Status sendstatus;
    int sendflag;
    static int reqrank, first_pass = 1;

/* only request work if previous request has received a response */

    if (first_pass) {
        reqrank = myrank + 1;
        first_pass = 0;
    }
    buf2[0] = 0;
    if (!queue_size)
        buf2[0] = -1;
    buf2[1] = best_distance;
    buf2[2] = myrank;
    if (response_from_get_work) {
        if (reqrank == ntasks)
            reqrank = 0;
        if (reqrank == myrank)
            reqrank = myrank + 1;
        if (reqrank == ntasks)
            reqrank = 0;
        get_work_counter[reqrank]++;
        MPI_Isend(&buf2[0],         /* message buffer */
                  3,                /* one data item */
                  MPI_INT,    /* data item is an integer */
                  reqrank,      /* send msg to a rank */
                  WORKREQTAG,   /* work request tag */
                  MPI_COMM_WORLD,     /* always use this */
                  &sendreq);
        response_from_get_work = 0;
        reqrank++;
    }
    MPI_Test(&sendreq, &sendflag, &sendstatus);
}


void update_distance()
{

    MPI_Request sendreq;
    MPI_Status status;
    int sendflag, i;
```

```
/* only request work if previous request has received a response */

    buf2[0] = -2;
    buf2[1] = best_distance;
    buf2[2] = myrank;
    for (i = 0; i < ntasks; i++) {
        if (!(i == myrank)) {
            MPI_Isend(&buf2[0],      /* message buffer */
                    3,  /* one data item */
                    MPI_INT,/* data item is an integer */
                    i,   /* send msg to a rank */
                    WORKREQTAG,         /* work request tag */
                    MPI_COMM_WORLD, /* always use this */
                    &sendreq);
        }
        MPI_Test(&sendreq, &sendflag, &status);
    }
}

int check_get_work()
{
    static MPI_Request recvreq;
    MPI_Status status, prbstatus;
    static int recvflag = 0, prbflag;

    MPI_Iprobe(MPI_ANY_SOURCE,
                SENDWORKTAG,
                MPI_COMM_WORLD,
                &prbflag,
                &prbstatus);

    if (prbflag == 1) {
        MPI_Recv(&check_buffer[0],       /* message buffer */
                    3,              /* one data item */
                    MPI_INT,     /* data item is an integer */
                    MPI_ANY_SOURCE,       /* any server */
                    SENDWORKTAG,  /* get work tag */
                    MPI_COMM_WORLD,      /* always use this */
                    &status);      /* MPI_Request structure for testing */
        recvflag = 1;
        response_from_get_work = 1;
        check_get_work_recv1[status.MPI_SOURCE]++;
    } else
```

```
        recvflag = 0;

/*   process_work_request(); */

   if (recvflag == 1) {
        if (check_buffer[1] < best_distance) {
           best_distance = check_buffer[1];
        }
        if (check_buffer[0] != -1) {
           if (check_buffer[0] > 0) {
                MPI_Recv(&work[0],          /* message buffer */
                        check_buffer[0],     /* data items */
                        MPI_INT,     /* data item is an integer */
                        check_buffer[2],     /* any server */
                        WORKTAG, /* get work tag */
                        MPI_COMM_WORLD,      /* always use this */
                        &status);      /* MPI_Request structure for testing */
                check_get_work_recv2[status.MPI_SOURCE]++;
                check_get_work_amount[status.MPI_SOURCE] =
check_get_work_amount[status.MPI_SOURCE] + work[0];
                setup_queue();
                return work[0];
           }
        }
        return -1;
   }
   return 0;
}

int check_for_work_request()
{

   static MPI_Request recvreq;
   MPI_Status status, prbstatus;
   int temp;
   static int workreq, control1 = 1;
   int recvflag, prbflag;

   MPI_Iprobe(MPI_ANY_SOURCE,
              WORKREQTAG,
              MPI_COMM_WORLD,
              &prbflag,
              &prbstatus);
```

```
    if (prbflag == 1) {
        control1 = 1;
        pending_recv[0] = 0;
        MPI_Recv(&buf2[0],/* message buffer */
                3,              /* one data item */
                MPI_INT,    /* data item is an integer */
                MPI_ANY_SOURCE,       /* source is any server */
                WORKREQTAG,   /* get work tag */
                MPI_COMM_WORLD,     /* always use this */
                &status);

/* test for receive complete which resets flag to a one otherwise it's a zero */
        if (buf2[0] == -2) {
           if (buf2[1] < best_distance) {
#if DEBUGGER
                printf("n%d received a better distance %d from n%d\n", myrank,
buf2[1], status.MPI_SOURCE);
#endif
                best_distance = buf2[1];
                return -1;
           }
        }
/*      if((buf2[0] == -1) && (queue_size <
busy_threshold))all_nodes[status.MPI_SOURCE] = 1; */
        pending_recv[0] = 0;
        return status.MPI_SOURCE;
    }
    return -1;
}

void send_some_work(int requestor)
{
    MPI_Request sendreq;
    MPI_Status status;
    int msg1[3];

    bzero(work1, WORKBUFFER);
    if (queue_size > busy_threshold) {
        split_queue();
        msg1[0] = work1[0];
    } else {
        if (queue_size > 0)
           msg1[0] = 0;
```

```
        else
            msg1[0] = -1;
    }
    msg1[1] = best_distance;
    msg1[2] = myrank;
    MPI_Send(&msg1[0],              /* message buffer */
             3,                     /* size of data buffer */
             MPI_INT,               /* data item is an integer */
             requestor,             /* destination process rank */
             SENDWORKTAG,           /* user chosen message tag */
             MPI_COMM_WORLD); /* always use this */
    send_some_work_send1[requestor]++;

    if (msg1[0] > 0) {
        MPI_Send(&work1[0],         /* message buffer */
                 msg1[0],       /* size of data buffer */
                 MPI_INT,       /* data item is an integer */
                 requestor,     /* destination process rank */
                 WORKTAG, /* user chosen message tag */
                 MPI_COMM_WORLD);     /* always use this */
        send_some_work_send2[requestor]++;
        send_some_work_amount[requestor] =
send_some_work_amount[requestor] + work1[0];
    }
}

void send_termination()
{
    int i;

    for (i = 1; i < ntasks; i++) {
        MPI_Send(&buf3,             /* message buffer */
                 1,                 /* size of data buffer */
                 MPI_INT,       /* data item is an integer */
                 i,                 /* destination process rank */
                 ENDTAG,    /* user chosen message tag */
                 MPI_COMM_WORLD);     /* always use this */
    }
}

void quit_program()
{
```

```
    int best[DEPTH + 2], i;

    for (i = 0; i < DEPTH + 1; i++)
        best[i] = best_trip[i];

    best[DEPTH + 1] = best_distance;

    MPI_Send(&best[0],          /* message buffer */
             DEPTH + 2,         /* zero data item */
             MPI_INT,           /* data item is an integer */
             0,                 /* destination process rank */
             QUITTAG,           /* user chosen message tag */
             MPI_COMM_WORLD); /* always use this */
}

void get_all_results()
{
    MPI_Status status;
    MPI_Request recreq;
    int recvflag;
    int i, j;
    int best[DEPTH + 2];

    for (i = 1; i < ntasks; i++) {
        MPI_Irecv(&best[0], /* message buffer */
                  DEPTH + 2,/* zero data item */
                  MPI_INT,    /* data item is a int */
                  i,          /* receive from anyone */
                  QUITTAG,  /* receive STARTTAG messages only */
                  MPI_COMM_WORLD,     /* always use this */
                  &recreq);    /* info about received message */
        while (1) {
            MPI_Test(&recreq, &recvflag, &status);
            if (recvflag == 1)
                 break;
            process_work_request();
        }
        if (best[DEPTH + 1] < best_distance) {
            best_distance = best[DEPTH + 1];
            for (j = 0; j < DEPTH; j++);
            best_trip[j] = best[j];
        }
    }
```

```
}

void print_results()
{
    int i;

    printf("best distance %d for trip: ", best_distance);
    for (i = 0; i < nodes; i++)
        printf("%d ", best_trip[i]);
    printf("\n");
    printf("lapse time = %f %d %d %d\n", end - start, get_work_count,
work_request_count, queue_init);
}

void split_queue()
{
    int new_size, temp_size;
    int i, j = 1, k;
    new_size = queue_size / QUEUE_SPLITTER;

    temp_size = queue_size - new_size;

    bzero(work1, WORKBUFFER);
    for (i = temp_size; i < queue_size; i++) {
        for (k = 0; k < DEPTH + 2; k++) {
            work1[j] = queue[i][k];
            j++;
        }
    }
    work1[0] = j;
    if (work1[0] >= WORKBUFFER)
        printf("%d had overflow on work1 buffer\n", myrank);
    queue_size = temp_size;
}

int setup_queue()
{
    int i, j = 1;

    if (work[0] == 0)
        return 0;
    while (j < work[0]) {
        for (i = 0; i < DEPTH + 2; i++) {
```

```
            queue[queue_size][i] = work[j];
            j++;
        }
        queue_size++;
    }
    if (queue_size) {
    }
    return work[0];
}

void initialize_search()
{
/* initialize the queue with the root node */
    init_queue();
}

void search_result()
{
    int i;
    printf("best distance %d for trip ", best_distance);
    for (i = 0; i < nodes; i++)
        printf(" %d ", best_trip[i]);
    printf("\n");
    printf("max queue size %d\n", max_queue_size);
}

/* check if a node is already in the path */
int in_path(int *apath, int node)
{
    int i;

    for (i = 0; i < apath[DEPTH]; i++) {

        if (apath[i] == node)
            return 1;
    }
    return 0;                    /* return true if not already in path */
}

/* push a node on to a path */

int push_path(int *apath, int src_node, int dst_node)
{
```

```
    if (src_node == dst_node)
        return 0;

    apath[apath[DEPTH]] = dst_node;
    apath[DEPTH] = apath[DEPTH] + 1;
    apath[DEPTH + 1] = apath[DEPTH + 1] + get_distance(src_node,
dst_node);

    return 1;

}

/* return distance between two verticies */

int get_distance(int node1, int node2)
{
    return tour[node1][node2];
}

/* push a path onto the queue */

void push_queue(int *apath)
{
    int i;

    for (i = 0; i < DEPTH + 2; i++) {
        queue[queue_size][i] = apath[i];
    }
    queue_size = queue_size++;
    if (max_queue_size < queue_size)
        max_queue_size = queue_size;
}



/* initialize the queue */

void init_queue()
{
    queue_size = 1;
    queue[0][0] = root;
    queue[0][DEPTH] = 1;
    queue[0][DEPTH + 1] = 0;
```

```
}


/* pop a path off the queue */

void pop_queue(int *apath)
{
    int i;

    for (i = 0; i < queue[queue_size - 1][DEPTH]; i++)
        apath[i] = queue[queue_size - 1][i];
    apath[DEPTH] = queue[queue_size - 1][DEPTH];
    apath[DEPTH + 1] = queue[queue_size - 1][DEPTH + 1];
    queue_size--;
}

int last_node(int *apath)
{
    return apath[apath[DEPTH] - 1];
}

int get_best_path(int *apath)
{

    int i, j, temp;

    temp = best_trip[2];
    if ((apath[DEPTH + 1] + tour[last_node(apath)][apath[0]]) > best_distance)
        return 0;
    if (apath[DEPTH] == nodes) {
        j = apath[DEPTH + 1];
        j = j + get_distance(last_node(apath), apath[0]);
        if (j < best_distance) {
            best_distance = j;
            for (i = 0; i < nodes; i++) {
                best_trip[i] = apath[i];
            }
        }
    }
    return 1;
}

void print_queue()
```

```
{
    int i, j;

    printf("queue size %d\n", queue_size);
    for (i = 0; i < queue_size; i++) {
        for (j = 0; j < queue[i][DEPTH]; j++)
            printf("%d ", queue[i][j]);
        printf("No.= %d dist= %d\n", queue[i][DEPTH], queue[i][DEPTH + 1]);
    }
}

void copy_path(int *new_path, int *old_path)
{
    int i;

    for (i = 0; i < old_path[DEPTH]; i++) {
        new_path[i] = old_path[i];
    }
    new_path[DEPTH] = old_path[DEPTH];
    new_path[DEPTH + 1] = old_path[DEPTH + 1];
}

void print_path(int *apath)
{
    int i;

    for (i = 0; i < apath[DEPTH]; i++) {
        printf("%d ", apath[i]);
    }
    printf("No. %d dist %d\n", apath[DEPTH], apath[DEPTH + 1]);
}

/* dfs exhaustive search one node expanded per call */
void best_dfs_1()
{
    int *new_path, new_path1[DEPTH + 2];
    static int dst_node = 0, src_node, old_path[DEPTH + 2];

/* setup something for temporary storage */
    new_path = &new_path1[0];

/* run through queue expanding nodes and push all the descendants */
    if (queue_size) {
```

```
/* check if expansion is complete */
        if (dst_node == nodes)
            dst_node = 0;
/* if this is first pass through expansion pop a path and expand the path */
        if (dst_node == 0) {
            pop_queue(old_path);
            pop_counter++;
            src_node = last_node(old_path);
        }
        while (in_path(old_path, dst_node))
            dst_node++;
        if (dst_node < nodes) {
            copy_path(new_path, old_path);
            push_path(new_path, src_node, dst_node);
            get_best_path(new_path);
            push_queue(new_path);
            dst_node++;
        } else
            dst_node = 0;
    }
}

/* dfs exhaustive search, complete descendant expansion each call */
void best_dfs_2()
{
    int *new_path, new_path1[DEPTH + 2], i;
    static int dst_node = 0, src_node, old_path[DEPTH + 2];

/* setup something for temporary storage */
    new_path = &new_path1[0];
    if (queue_size) {
        dst_node = 0;
        pop_queue(old_path);
        pop_counter++;
        src_node = last_node(old_path);
        for (i = 0; i < nodes; i++) {
            if (!in_path(old_path, i)) {
                copy_path(new_path, old_path);
                push_path(new_path, src_node, i);
                get_best_path(new_path);
                push_queue(new_path);
            }
        }
```

```
    }
}

void direct()
{
    int new_path[DEPTH + 2], i;
    int src_node, old_path[DEPTH + 2];


    if (queue_size) {
        pop_queue(old_path);
        src_node = last_node(old_path);
        for (i = 0; i < nodes; i++) {
            if (!in_path(old_path, i)) {
                copy_path(new_path, old_path);
                push_path(new_path, src_node, i);
                push_queue(new_path);
            }
        }
    }
}
```