THE DESIGN, IMPLEMENTATION AND PERFORMANCE OF A QUEUE MANAGER FOR PVM

Douglas J. Sept

Computer Science Department

CS-93-196 August 1993

The Design, Implementation and Performance of a Queue Manager for PVM

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Douglas J. Sept

August 1993

Acknowledgements

I thank my thesis advisor, Dr. Michael Berry, for his support and guidance. I appreciate very much his insightful recomendations and endless patience. The amount of time he spent helping me work through the details and results of the QM was above and beyond the call of duty. I thank Karen Minser, author of the map analysis application, Victor Eijkhout who wrote the Conjugate Gradient application and Bob Manchek who wrote the Mandelbrot application and answered my never-ending PVM questions. I thank Dr. Jack Dongarra who served on my committee and helped guide the application, and Al Geist who served as my advisor at ORNL in the summer of 1992. I also thank Dr. David Straight who agreed to serve on my committee on very short notice.

This research was supported in part by the U.S. Department of Energy under contract number DE-AC-05-84OR21400.

Abstract

The PVM Queue Manager (QM) application addresses some of the load balancing problems associated with the heterogeneous, multi-user, computing environments for which PVM was designed. In such environments, PVM is not only confronted with the difficulties of distributing tasks among machines of variable loads, it must also contend with machines of varying performance levels in the same virtual machine. The QM addresses both of these problems using two different load balancing techniques, one static, the other dynamic. In its simplest (static) mode, the QM will initiate PVM processes for the user on demand, taking into account information such as the peak megaflops/sec and actual load of each machine. In addition to the initiation of processes, the QM will also accept tasks to be completed by a specified PVM process type. These tasks are shipped to the QM where they are kept in a FIFO queue. Worker processes in the virtual machine send idle messages to the QM when they are ready for a task, and the QM ships a task to the process if there is one (of a type matching the process) in the queue. The QM also maintains a list of idle processes and chooses the *best* one for the task, should one arrive when several processes are idle. Since faster machines typically send more idle messages (and receive more tasks) than slower ones, this provides a level of dynamic load balancing for the system. Three applications have already been implemented using the QM within PVM: a Mandelbrot image generator, a conjugate-gradient algorithm, and a map analysis program used in landscape ecology applications. Benchmarks of elapsed wall-clock time comparing standard PVM versions with the QM-based versions demonstrate substantial performance gains for both methods of load balancing. When processing a 1000×1000 image, for example, the QM-based Mandelbrot application averaged 63.92 seconds, compared to 139.62 seconds for the standard PVM version in a heterogenous network of five workstations (comprised of Sun4's and an IBM RS/6000).

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Statement of Problem 2

1.3 Comparison with other Work 2

2 Implementation **4**

2.1 Using the Queue Manager 4

2.2 Fortran-77 Routines 8

2.3 Load information and QM Process Initiation 8

2.4 Granularity Issues in PVM Dynamic load-balancing 10

3 Benchmarks **12**

3.1 Methodology 12

3.2 Mandelbrot Benchmark Results 15

3.3 Conjugate Gradient Benchmark 21

3.4 Map Analysis Application 27

4 Conclusions **36**

Bibliography **37**

Appendices **39**

A QM Installation **40**

B Harmonic Mean Execution Times for Benchmarks **41**

C Statistical Formulas used to Compute Tables **44**

List of Tables

2.1 C Based User Routines for the QM 6

2.2 Fortran-77 User Routines for the QM 9

3.1 Machines used in QM benchmarks and their observed LINPACK benchmark ratings 14

3.2 Abbreviations for Benchmark Versions 15

3.3 Mandelbrot Average Task Granularities (Heterogeneous Network) 17

3.4 (Mandelbrot) Heterogeneous Network: Arithmetic Mean Execution Times (in seconds) and Sample Variances 18

3.5 (Mandelbrot) Homogeneous Network: Arithmetic Mean Execution Times (in seconds) and Sample Variances 18

3.6 Mandelbrot $n = 1000$: Observed Machine Performance (mean sec/task and megaflops/sec reported) 19

3.7 CG Task Granularities for Heterogeneous Network. 24

3.8 (CG) Heterogeneous Network: Arithmetic Mean Execution Times (in seconds) and sample variances 24

3.9 (CG) Homogeneous Network: Arithmetic Mean Execution Times (in seconds) and Sample Variances 25

3.10 Distribution of cluster sizes for three p values 28

3.11 Map Analysis Application Version Summary. $N_c \equiv$ total number of clusters in map, $N_w \equiv$ total number of machines in PVM network. 29

3.12 (Maps with $p = .1$) Homogeneous Network: Arithmetic Mean Execution Times (in seconds) 29

3.13 (Maps with $p = .3$) Homogeneous Network: Arithmetic Mean Execution Times (in seconds) 30

3.14 (Maps with p = .1) Heterogeneous Network: Arithmetic Mean Execution Times (in seconds) 30
3.15 (Maps with p = .3) Heterogeneous Network: Arithmetic Mean Execution Times (in seconds) 30
3.16 (Maps with $p = .3$) Homogeneous Network: Sample Variances . . 30
3.17 (Maps with $p = .1$) Homogeneous Network: Sample Variances . . 31
3.18 (Maps with $p = .1$) heterogeneous Network: Sample Variances . . 31
3.19 Map Analysis ($n = 768$, $p = .1$) : Observed Machine Performance 34
3.20 Map Analysis Application ($p = .1$) : Task Granularities. 35

B.1 (Mandelbrot) Heterogenous Network: Harmonic Mean Execution Times (in seconds) 41
B.2 (Mandelbrot) Homogeneous Network: Harmonic Mean Execution Times (in seconds) 41
B.3 (CG) Heterogenous Network: Harmonic Mean Execution Times (in seconds) 42
B.4 (CG) Homogeneous Network: Harmonic Mean Execution Times (in seconds) 42
B.5 (Maps with $p = .1$) Homogeneous Network: Harmonic Mean Execution Times (in seconds) 42
B.6 (Maps with $p = .3$) Homogeneous Network: Harmonic Mean Execution Times (in seconds) 43
B.7 (Maps with $p = .1$) Heterogeneous Network: Harmonic Mean Execution Times (in seconds) 43
B.8 (Maps with $p = .3$) Heterogeneous Network: Harmonic Mean Execution Times (in seconds) 43

C.1 Definitions of Statistical Formulas. n= Number of Samples, x_i=i-th elapsed wall-clock time sample. 44

List of Figures

2.1 QM Data Flow 7

3.1 The Mandelbrot set generated by the QMDLB version of the Mandelbrot Application on the heterogeneous network. The real axis runs north and south, while the imaginary axis runs east and west in this figure. 16

3.2 Mandelbrot Application, Heterogeneous Network 20

3.3 Mandelbrot Application, Homogeneous Network 20

3.4 CG Method on Poisson's Equation, Heterogeneous Network 26

3.5 CG Method on Poisson's Equation, Homogeneous Network 26

3.6 Map Application with $p = .1$, Homogeneous Network 32

3.7 Map Application with $p = .1$, Heterogeneous Network 32

3.8 Map Application with $p = .3$, Homogeneous Network 33

3.9 Map Application with $p = .3$, Heterogeneous Network 33

1. Introduction

In this thesis, a Queue Manager (QM) application for PVM ([BDGM91]) is presented. The QM can be used to effectively load balance PVM applications and, in many cases, improve run time performance. The motivation for the work is discussed below, along with other load-balancing applications. The use and implementation of the QM are discussed in Chapter 2. Performance results and an analysis of various types of load-balancing strategies are presented in Chapter 3. Finally, conclusions are presented in Chapter 4.

1.1. Motivation

The rapid increase in workstation performance in recent years has fueled interest in the development of loosely coupled parallel systems able to take advantage of the vast quantities of unused computing resources present in a typical scientific computing environment . Such systems offer power comparable to that of modern parallel machines such as the Intel i860 at a fraction of the cost. PVM is one such system that has been developed at the University of Tennessee and Oak Ridge National Lab (ORNL). PVM configures a cluster of workstations in a Local Area Network (LAN) as a distributed-memory MIMD (Multiple Instruction Multiple Data) virtual parallel computer.

In any distributed system, load-balancing among processors can be problematic. It is important that each processor receive work in proportion to its ability, since overloading one processor at the expense of another will increase run time and reduce efficiency. Load-balancing is especially difficult in a PVM network, however, for two reasons: (*i.*) PVM runs in a multi-user environment and (*ii.*) PVM supports heterogeneous virtual machines.

Since nodes in a PVM network are typically workstations utilized by other users in a laboratory or office situation, loads on the machines can vary dramatically. If, for example, the workstations' owner is just using the machine to send and receive electronic-mail, the machine load will be light, allowing it to accomplish much more additional work than an engineer's workstation running a complicated simulation. The difficulty, of course, lies in determining how much load each machine currently has, and how much it should be given to provide the fastest overall execution time for a PVM batch job.

This predicament is further complicated by the heterogeneous nature of PVM. Since it is capable of incorporating a wide variety of architectures of dramatically different capabilities into a single virtual machine, the load-balancing equation becomes a two variable one. Not only is the optimal task distribution a function of the loads of the machines in the network, it is also a function of the capabilities of those machines. Even within architecture classes, these differences can be quite startling. For example, the (unoptimized) LINPACK benchmark rating of a SUN 4/280 is about .43 megaflops/sec[1], while the rating of a SUN4 75/C (Sparc 2) is about 1.3 megaflops/sec. Note that these machines are in the same binary class. Differences can become even more pronounced when different architectures are incorporated into the same virtual machine. It is possible, for example, to have a SUN4 and a Cray in the same parallel virtual machine. These machines differ in their capabilities by multiple orders of magnitude.

1.2. Statement of Problem

Given the constraints of the PVM environment, develop a load-balancing tool for PVM which will compensate for differences in machine architecture and load, and thereby load balance PVM applications so that run time is minimized.

1.3. Comparison with other Work

Several packages exist for scheduling PVM tasks at the application level. DQS [Green92] and Condor [LiLM88] are two such batch-processing applications. Both

[1] millions of floating-point operations per second.

take advantage of unused machine resourses to schedule large batch jobs which may or may not run under PVM. Both applications also provide some sort of load-balancing at the *application* level, by monitoring the work load of the machines on which they are executing jobs, and by redistributing the work should the load become too heavy on a particular machine. Neither, however, provides any sort of load-balancing *within* the PVM application to evaluate the optimal host on which to initiate a PVM process, nor do they provide any sort of dynamic load-balancing of tasks within a PVM application.

LINDA [AhCG86], however, does provide a mechanism for dynamic load-balancing within an application, in the form of its *tuple space*. Like the QM, LINDA uses a *pool-of-tasks paradigm* in its approach to dynamic load-balancing. Tasks can be placed in the tuple space, then extracted in an asynchronous fashion by idle processors, which process the task and return the result to the tuple space. This process is very similar to the dynamic load balancing aspect of the QM. LINDA, however, provides no analog for the static process initiation function of the QM. The LINDA environment is geared toward dynamic load-balancing, which is often the least effective form of load-balancing for parallel machines with low communication bandwidth between nodes (such as PVM). The problems and perils associated with dynamic load-balancing in this environment are discussed in Chapter 3.

Jade [Rina92] is a shared-memory programming language implemented in both shared-memory and message passing environments, including PVM. The PVM implementation allows the user to program using *shared objects*, which are data structures potentially accessible to any node of the PVM network. Jade can also be used to implement dynamic load-balancing using shared-memory variables to synchronize the parallelism. Like LINDA, however, it does not incorporate any automated mechanism for assessing machine loads and capabilities, nor does it provide a mechanism for the initiation of PVM processes.

2. Implementation

2.1. Using the Queue Manager

The QM is a PVM process which is initiated just like any other PVM process on any of the machines in the PVM network. Since elapsed-time reduction relies upon an efficient QM, however, it is not advisable to run the QM on one of the *slower* machines in the network. For example, one would not want to initiate the QM on a SUN 3/260 workstation (having an observed LINPACK rating[1] of .46 megaflops/sec) and use it to load-balance a network of SUN Sparc 10 workstations (having an observed LINPACK benchmark rating of about 6.7 megaflops/sec). Machines suffering long network lag times should also be avoided since the QM is the focal point of network traffic within the virtual machine. The QM should be placed on the same machine as the PVM process that will be sending the majority of the tasks to the QM for completion, thus reducing execution time by making it unnecessary for the host to send tasks across the network to the QM.

The `qmanager` executable takes as an argument a *hostfile* similar to the hostfile used to start PVM. This `qmanager` hostfile contains machine names and the LINPACK megaflops/sec or other benchmark rating for that particular machine. The format of the file is

machine_name benchmark_rating.

While not all the machines in the hostfile must be used in the current PVM network, all machines in the network must have an entry in the hostfile.

When the `qmanager` binary has been invoked, functions from the user library

[1]These tests were compiled and run with level 2 compiler optimization. For the official LINPACK benchmark ratings, see [Dong92].

can then be used (see Table 2.1).

Generally the first function to be called when using the static load-balancing aspect of the QM is `qinitiate`. This function is similar to the standard PVM `initiate` function, with the exception that it attempts to initiate the component on the *optimum* machine, as determined by the loads and machine characteristics of the nodes in the network (see Section 1.3). The `qinitiate` function also contains an additional integer argument, an unused message type, that the QM can use to send the instance of the newly-created process back to the user program. The `qinitiate` function provides static load-balancing by yielding a faster PVM network than would normally result from the round-robin scheduling used by PVM `initiate`. In other words, the QM exploits lightly-loaded and higher-performance machines more thoroughly than standard PVM.

The remaining functions in Table 2.1 can be used to implement the dynamic load-balancing aspects of the QM. In a typical user program, for example, the host initiates one node on each machine using standard PVM `initiate` and sends tasks to the QM, which distributes them to worker nodes as those nodes become idle (finish their current task). In a cyclic fashion, these nodes send an idle message, accept and process a task, return the results, and send another idle message until all tasks are processed (see Figure 2.1). At this point, the worker nodes can be terminated by the user program.

To send tasks to the QM, a group of `qput` statements are used to place information in the send buffer. After the task data is placed in the buffer, the `qsnd` command is called to send the task to the QM. The arguments to `qsnd` are: (1) the `component` (executable) which will accept the task, (2) the type of the task (used to classify tasks into groups) and (3) the message type associated with the task when it is sent to an instance of `component` by the QM. The QM will queue the task if there are no idle processes or send it immediately to an idle process (if one exists) . The node program, in turn, receives a task from the QM via the `qrcv` command. The `qrcv` command receives as input a character array and a pointer to an integer and returns the name and PVM instance number of the task's source. The user can then extract the data from the buffer using PVM

Table 2.1: C Based User Routines for the QM

`int` **qinitiate**`(char *object_file, char *arch,int` **unused_msg_type**`)`	
-	instructs the QM to initiate a process and returns instance number (>= 0) if successful or < 0 if error. If `arch` is NULL, then the QM chooses an architecture. The **unused_msg_type** variable is any integer msg type greater than 0 not being used by the user. It is used by the QM to return the instance number of the process it initiates.
`int` **send_load**`()`	
-	gets and sends load and architecture information to the QM for the machine on which it is executed. Returns −1 on error.
`int` **qput[type]**`([type] *ptr, int num)`	
-	inserts **num** values beginning at **ptr** into the send buffer. Only used when sending a task to the QM. Returns −1 if out of memory. **[type]** must be `nint, nfloat, ndfloat, ncplx, ndcplx, string,` or `bytes.`
`int` **qrcv**`(char *component, int *instance, int msgtype)`	
-	receives a task of the indicated `msgtype` from the QM or any type if `msgtype`=−1. Other arguments are pointers to the memory location where the function will place the component and instance number of the task's source. (Synchronous) . Returns actual message type received.
`int` **qsnd**`(char *component, int task_type, int msgtype)`	
-	Sends task in send buffer to the QM, which will send it to the first available instance of `component`. `task_type` is a user-defined tag which can be used to identify the task when it is received by the target component. `msgtype` is the integer message type of the task message which will be sent by the QM to the execution component. Returns < 0 if error.
`int` **snd2task**`(char *component, int task_type, int msgtype)`	
-	Sends message in send buffer to all PVM processes named `component` processing tasks of the indicated `task_type`. The `msgtype` parameter is the type of the message which will be sent by the QM.
`int` **leaveq**`()`	
-	Sends a message to the QM instructing it to kill all the processes the QM has initiated. The QM then terminates.
`int` **killq**`()`	
-	Kills the QM and its load daemons without killing the processes the QM has initiated.
`int` **send_idle**`()`	
-	sends an idle message to the QM (requests another task).

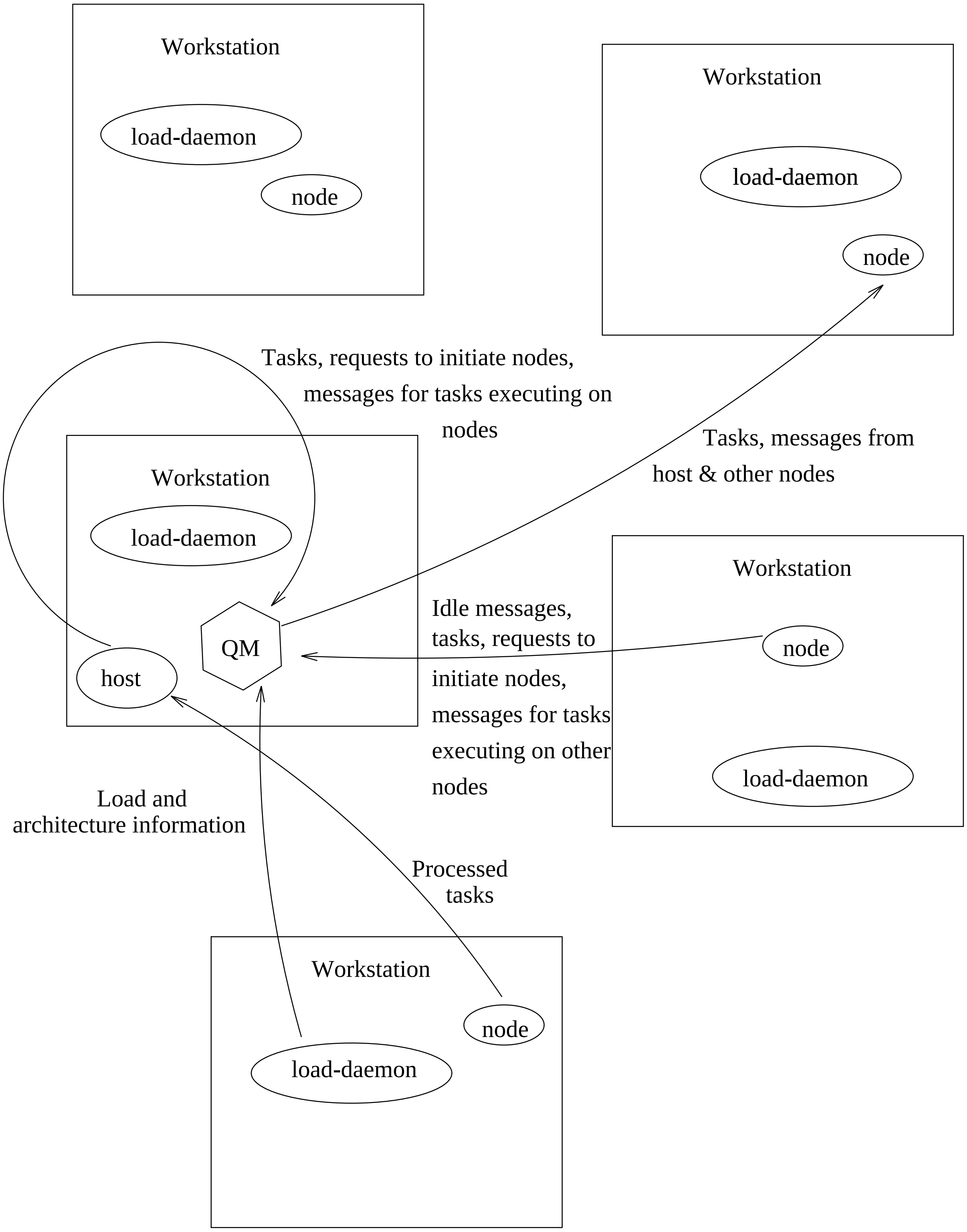

Figure 2.1: QM Data Flow

user library `get` calls, such as `getnint` and `getbytes`.

The `snd2task` function is used to pass messages between tasks. The message is sent to the QM, which will queue it if there are no tasks of that `task_type` executing, or forward it to all tasks of that `task_type`. A large amount of overhead is associated with this function, since it requires all the messages to be routed through the QM. It should therefore be used sparingly.

When the host is ready to terminate the worker nodes, it calls the function `leaveq()` which sends a terminate message to the QM. The QM then kills all the node processes created by `qinitiate` and exits. The user must terminate all nodes started with standard PVM `initiate`.

2.2. Fortran-77 Routines

The Fortran-77 routines in Table 2.2 (prefixed by an '`f`') provide a Fortran-77 interface to the QM. Each has an analogous function in the C user library, so, for example, the function `leaveq()` in C, is `fleaveq()` in Fortran-77.

The Fortran-to-C interface subroutines essentially pass their parameters to their C counterparts. They are accessed by linking in the `libqf2c.a` and `libf2c.a` library at compile time.

The QM Fortran-77 routines, like the original PVM routines, are defined as subroutines (as opposed to functions). Moreover, a null character \0 must be appended to the end of every string passed to a QM function call. Table 2.2 lists the names and argument lists of the supported Fortran-77 calls in QM version 1.0.

2.3. Load information and QM Process Initiation

Load information for each machine is sent automatically to the QM by load daemons invoked by the QM immediately after execution. Specifically, a floating-point number reflecting the average number of jobs in the machine's run queue over the last minute (also known as *load points*) is sent to the QM. This informa-

Table 2.2: Fortran-77 User Routines for the QM

int **fqinitiate**(object_file,arch,info,unused_msg_type,info)
int **fsend_load**()
int **fqput[type]**(array,num,info) - type must be nint, nfloat, ndfloat, ncplx, ndcplx, string, or bytes.
int **fqrcv**(component,instance,msgtype,info)
int **fqsnd**(component,msgtype,info)
int **fqsnd2task**(component,task_type,msgtype,info)
int **fleaveq**()
int **fkillq**()
int **fleaveq**()
int **fsend_idle**()

tion is obtained via a system call to the UNIX [2] `uptime` command, which gives the 1 min, 5 min and 15 min load averages on most UNIX systems. One notable exception are IBM RS/6000 architectures, which use the *current* number of jobs in the current run queue, rather than an average, since `uptime` does not return averages under AIX[3]. Load information is the single most architecture-dependent aspect of the QM implementation. Finding equivalent load information on some architectures is problematic.

On startup, the QM will wait for a specified amount of time to receive an initial load response from daemons executing on each machine in the network. At this point the load daemons *time-out* and are ignored by the QM. This delay time is determined at compile time by the constant `TIMEOUT` defined in the file `qmanager.h`. Load information is pivotal to the QM heuristic which determines on which machines the QM will initiate processes in its static load-balancing capacity. Consequently, the QM does not start distributing tasks or initiating processes until after the load information is received from each machine, or the machines have timed out, whichever comes first. The QM uses this information, in conjunction with available performance benchmarks for each machine, to determine on which host to initiate a PVM process when it receives `qinitiate` request. It does so by rank-ordering machines from highest to lowest in task scheduling priority according to the size of the ratio

[2]UNIX is a registered trademark of AT&T.
[3]AIX is a registered trademark of IBM.

$$\gamma = \frac{(\rho + \tau)}{\sigma}, \tag{2.1}$$

where ρ is the average number of jobs in the run queue for a machine, τ is the number of PVM process the QM has already initiated on that machine, and σ is an available measure of machine performance appropriate for the application the QM will load-balance. The LINPACK benchmark megaflops/sec measure would be an example of such a metric. The QM will initiate the next process in its queue on the machine with the smallest current γ ratio.

2.4. Granularity Issues in PVM Dynamic load-balancing

Granularity is defined by Golub and Van Loan [GoVL89] to be "the amount of computation that takes place in between synchronization points". Hence, the terms *coarse-grain* and *fine-grain* are used to describe, respectively, an application with a relatively large and small amount of computation between synchronization points. These are necessarily vague definitions, since what is considered *fine-grain* for a particular machine architecture may be considered *coarse-grain* for another. As noted by Jack Dongarra et al. in [Dong91], an increase in parallelism may come at the expense of finer granularity, and the optimal balance between the two depends on the architecture of the machine in question. For example, the original PVM dynamically load-balanced version of the map benchmark (discussed in Section 3.4) was a port of the *coarse-grain* version of a CM-5 implementation of the application, but low communication bandwidth forced the creation of an even *coarser-grain* PVM version. Thus a direct port of the CM-5 *coarse-grain* version became the *fine-grain* PVM version, and the new *coarser-grain* version of the PVM code became the *coarse-grain* PVM version.

Typically, dynamically load-balanced applications should have fine-grain task sizes, thus maximizing parallelism by minimizing the idle time of processors waiting for the last few tasks to be completed. In a PVM environment, however, this tendency toward fine-grained task size in dynamically load-balanced applications is compromised by typically low communication bandwidth in the PVM network. Hence, there exists an inherent bias toward coarse-grained applications in PVM. Therefore in any dynamically load-balanced PVM application, the tendency toward *fine-grain* task sizes must be balanced with the necessity of minimizing inter-machine communication if optimum execution time is to be realized. Unfortunately, producing an *ideal* task granularity is very difficult due to the high variability of some determining factors, such as network traffic, machine speed, and machine load.

3. Benchmarks

3.1. Methodology

Three application benchmarks were used to test the efficacy of the QM, and to determine which load-balancing strategies work best for the QM and PVM: (*i.*) a **C** Mandelbrot image generator, (*ii.*) a Fortran-77 Conjugate Gradient application, and (*iii.*) a **C** map analysis application. It should be noted that there are significant difficulties inherent in any attempt to benchmark a PVM application. Because of the multi-user distributed environment in which PVM operates, benchmark elapsed wall-clock times can have enormous sample variance[1]. It can therefore be quite difficult at times to determine if poor performance of a particular application is due to a problem with the application or to the effects of other users who are working on the same machines in the network. Consequently, interpretations of QM performance may be best explained in terms of trends rather than individual timings or their sample variance. While the QM may exhibit better overall performance, this does not imply all the QM benchmark elapsed-times exhibit better performance than their standard PVM analogs.

Additional problems with resetting the PVM *epoch* also made benchmarking difficult. An *epoch* in PVM 2.4 is a time period in which all PVM processes are running, and is separated from another *epoch* by a period when all PVM processes have exited. PVM assigns each *instance* of a particular executable a number, and the benchmark applications, like many PVM applications, use these numbers to determine message destinations. When PVM resets the epoch, all PVM processes have exited, and any further PVM executables which perform an `enroll` get instance numbers in sequence from 0 to $n-1$, where n is the number

[1]See Appendix C for definitions of sample variance, harmonic mean and arithmetic mean.

of executables enrolling in PVM. If the epoch fails to reset after all processes have exited, however, instance numbers of the enrolling executables will continue where those of the previous benchmark left off. This typically causes PVM applications to abort since any assumptions made about certain worker nodes (executables) having specific instance numbers are usually false. The epoch resetting problem posed significant difficulty for the simulations which accumulated elapsed-time data for the three applications. This difficulty was somewhat mitigated by placing time delays between application runs, but this solution was not viable for the map code. Moreover, since some of these benchmark suites run for 15 or more hours, it was not feasible to manually reset the epoch. Consequently some of the map code benchmarks had to be done in stages with restarts initiated if an abnormal termination occured. For example, if the map analysis simulation had accumulated data on the 256×256 map prior to its termination while processing a 512×512 map, the simulation would be restarted for the 512×512 map, but the results for the 256×256 map (and all previously benchmarked maps) would be preserved. This process preserved temporal locality of the data points for a particular map and problem size, so that different versions of the map application could be compared equitably.

For all three applications considered, benchmarks were obtained on two networks of machines, the first (homogeneous) network consisted of 10 SUN Sparc IPX machines, while the second (heterogeneous) network consisted of two SUN 4/280's, a Sparc 10, a Sparc 2, and an IBM RS/6000 Model 550 (see Table 3.1).

A number of problem sizes were examined for each application, and five timing samples were collected for each problem size. The scheduling of problem sizes for the Mandelbrot and CG applications was done in a random fashion. During each cycle of the benchmark program, a problem size was randomly chosen, and test runs for all versions of the application using that problem size were performed. The map code benchmark, however, used a pre-determined scheduling policy which manually alternated problem sizes. Timing information was collected using the elapsed wall-clock time of a system call invoking that particular application, and included overhead associated with making the system call.

Table 3.1: Machines used in QM benchmarks and their observed LINPACK benchmark ratings .

Machine	Architecture	megaflops/sec*
`duncan`	SUN Sparc 10	6.3
`austin`	IBM RS/6000 Model 550	17.5
`cetus`xy	SUN 4 IPX	3.0
`lego`	SUN 4/280	1.4
`galoob`	SUN 4/280	1.0
`berry`	SUN Sparc 2	3.0

*megaflops/sec : These tests were compiled and run with level 2 compiler optimization. They represent the timings for matrixes of order 100, double precision floating point. For the official LINPACK benchmark ratings, see [Dong92]

The heterogeneous network benchmarks were run during the day (9:00 a.m. to 5:00 p.m.), in a network environment that could be characterized as *loaded.* This enabled the QM to take advantage of differences in machine load as well as differences in machine speed. Due to machine usage constraints, however, the homogeneous benchmarks were run *after 10 p.m.* on a network of SUN IPX machines in a computer laboratory at the University of Tennessee. Such an environment could be categorized as comparatively *unloaded.* Together, these two environments demonstrate the approximate upper and lower bounds on QM performance, with the heterogeneous network and its exploitable differences in machine speed and load representing a more favorable environment for QM, and the homogeneous network with its lack of exploitable differences representing a less favorable environment. The mean excution times reported for the three benchmarks are arithmetic means, but harmonic means, which are less prone to be skewed by individual atypical numbers in the data set, are presented in Section B. The abbreviations for benchmark versions listed in Table 3.2 were used to label tables and graphs in a more compact and readable fashion. The distinctions between task granularities were only used in the map analysis application, which used two different task granularities in its benchmarks.

Table 3.2: Abbreviations for Benchmark Versions

Abbreviation	Version
QMDLB	QM with Dynamic Load-Balancing
QMSLB	QM with Static Load-Balancing
QMDLBF	QM with Dynamic Load-Balancing (Fine-grain)
QMDLBC	QM with Dynamic Load-Balancing (Coarse-grain)
SLB	Static Load-Balancing
OSLB	Original *PVM* with Static Load-Balancing
ODLBF	Original *PVM* with Dynamic Load-Balancing (Fine-grain)

3.2. Mandelbrot Benchmark Results

The Mandelbrot application generates the Mandelbrot set [Glei87] for an $m \times n$ image, given the x and y coordinates in the complex plane. Generating the image is computationally intensive (see Table 3.3), but the algorithm for generating the set is fairly simple. The Mandelbrot set is defined as the set of all points such that the iteration

$$z_{n+1} = z_n^2 + c \tag{3.1}$$

remains unbounded as $n \rightarrow \infty$, where the variable z has an initial value of 0 and c is the complex point being tested (see Figure 3.1).

The image is generated by testing all points within the x and y boundaries of the complex plane in this manner. The resolution of the image is determined by the m and n values which define the size of the image.

The original application utilized static load-balancing and a host-node computational model. The host decomposed the image into *tiles* and assigned each node processor the same number of equal-sized tiles. The nodes iterated over the complex points in the tile they were given, and returned the results to host, which re-assembled the tiles into an image for output to disk.

Two QM versions of this program were developed: one utilizing the (static) process initiation function of the QM (QMSLB), and the other utilizing the dynamic load-balancing aspects of QM (QMDLB). In the QMDLB version, the host sent all the tiles to the QM, which subsequently routed them to the worker nodes as they became idle. The workers then returned their processed tiles to the host,

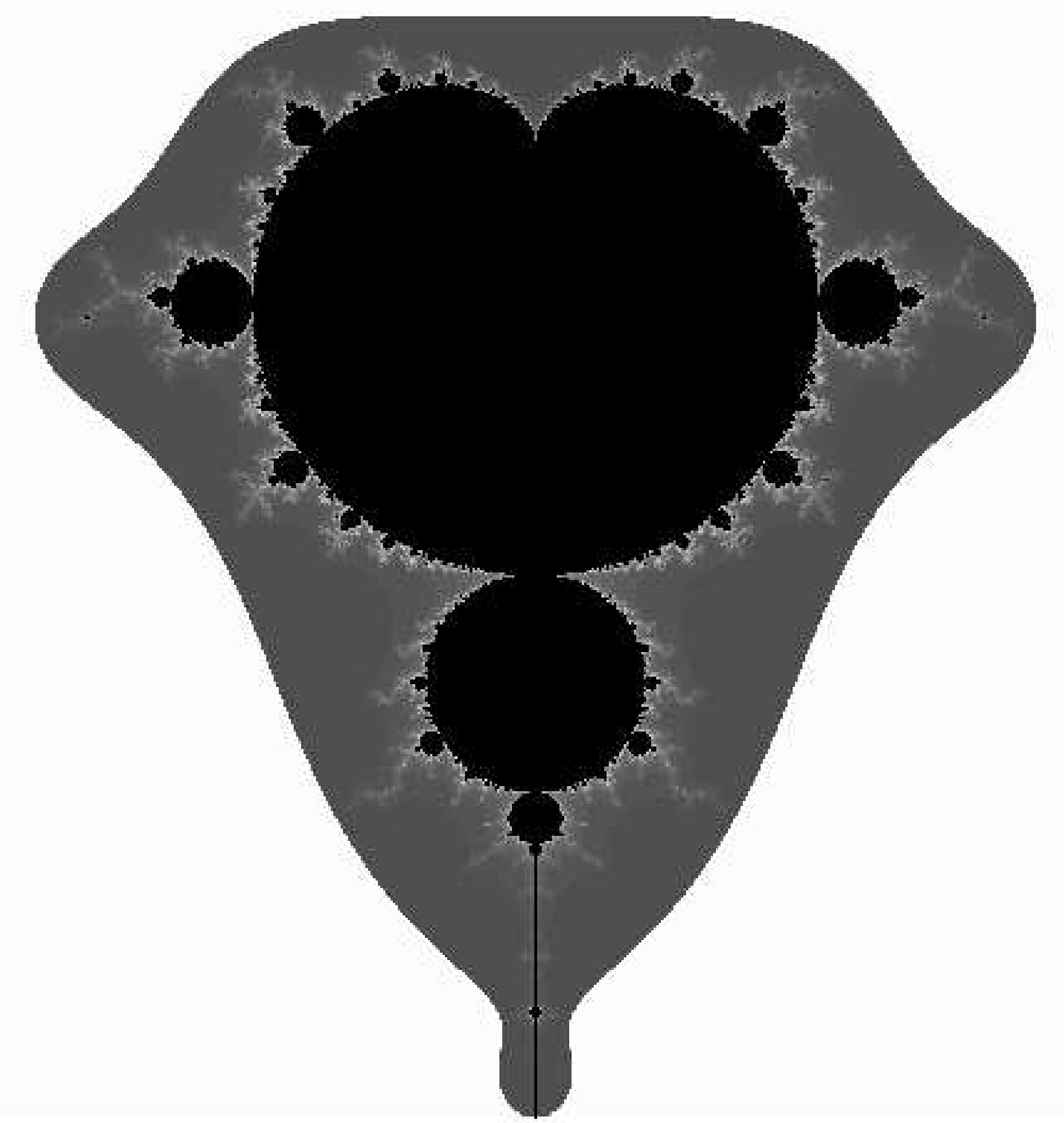

Figure 3.1: The Mandelbrot set generated by the QMDLB version of the Mandelbrot Application on the heterogeneous network. The real axis runs north and south, while the imaginary axis runs east and west in this figure.

which constructed the image and wrote it out to file. The QMSLB version, on the other hand, only used the QM to initiate processes on *optimal* hosts.

Table 3.3 illustrates the average task granularities for the QMDLB version and of the Mandelbrot benchmark. These numbers are averages (arithmetic means), because the actual number of floating-point operations involved in processing a task is dependent on the portion of the image which is processed in that particular task. The task sizes for the 1000×1000 image range from 38.4 megaflops to .73 megaflops, which constitutes a substantial difference in granularity.

Task granularity was constant as a *percentage* of the total work load, but the physical task sizes in terms of megaflops/task increased along with the problem size. PVM's bias toward coarse-grain applications was the primary factor in the decision to scale megaflops/task with task size. Keeping task size constant would have limited the task size to a maximum of the smallest problem size divided by the number of nodes in the network. Experiments performed on the map code indicated that such an approach to load-balancing yields poor elapsed time performance (see Section 3.4).

Table 3.3: Mandelbrot Average Task Granularities (Heterogeneous Network)

Image Size	megaflops/task
100×100	0.20
500×500	5.01
1000×1000	20.05

Clearly, the QMSLB version outperformed both the QMDLB and the OSLB versions for the heterogeneous network (see Table 3.4 and Figure 3.2). The QM-SLB arithmetic mean execution time for the largest test case was less than half that of the OSLB version (13.34 seconds versus 52.53 seconds). This performance gain can be attributed to the QM's ability to recognize performance characteristics (LINPACK megaflops/sec rate) and machine loads before assigning machines a proportional number of processes based on their ability to work. Unfortunately, performance of the QMDLB application was not nearly as good as that of the QMSLB version. The QMDLB elapsed-times were significantly higher than the QMSLB version, though it still outperformed the OSLB version. The process initiation decisions made by the QMSLB version improved load-balancing enough to reduce execution times, and did it without incurring the communication overhead associated with the QMDLB version of the application.

Table 3.4: (Mandelbrot) Heterogeneous Network: Arithmetic Mean Execution Times (in seconds) and Sample Variances

	OSLB		QMSLB		QMDLB	
Image Size	mean time	variance	mean time	variance	mean time	variance
100× 100	2.21	0.15	2.12	1.23	1.94	0.21
200 × 200	6.28	0.22	4.35	1.25	4.61	4.06
300 × 300	14.23	2.00	6.98	0.85	7.08	0.30
400 × 400	22.38	2.62	12.00	8.21	15.10	23.96
500 × 500	39.98	76.06	21.20	61.11	24.74	106.71
600 × 600	54.90	28.71	23.72	1.82	46.12	987.51
700 × 700	69.71	8.51	31.30	9.97	47.88	482.57
800 × 800	92.95	71.47	43.44	29.28	51.65	381.94
900 × 900	115.43	8.57	49.46	24.10	54.15	106.63
1000 × 1000	139.62	61.43	63.92	49.80	84.46	1574.13

The homogeneous benchmark results (see Table 3.5 and Figure 3.3) were slightly better or about the same as the OSLB version for most of the problem sizes. Differences in machine load may account for the QM's performance in this case.

Table 3.5: (Mandelbrot) Homogeneous Network: Arithmetic Mean Execution Times (in seconds) and Sample Variances

	OSLB		QMSLB		QMDLB	
Image Size	mean time	variance	mean time	variance	mean time	variance
100×100	2.26	0.01	2.12	0.01	1.83	0.01
200×200	6.47	0.34	6.40	0.34	3.35	2.63
300×300	9.11	9.69	9.32	6.45	7.04	9.62
400×400	8.76	0.81	9.16	1.22	8.21	1.18
500×500	10.94	0.54	9.49	0.62	10.70	1.22
600×600	13.05	0.53	14.69	0.36	13.94	4.89
700×700	19.03	1.58	15.05	1.96	17.54	7.34
800×800	24.57	1.39	18.47	0.95	26.04	16.37
900×900	32.29	5.13	25.33	25.36	29.41	10.14
1000×1000	41.48	11.15	31.05	28.55	37.79	34.04

The QM load-balanced versions, like the other versions, exhibited significant sample variance, especially the QMDLB version of the Mandelbrot application. Such large variances may be attributed to network response lags and memory swapping. Since all tasks are sent to the QM before being sent to the worker nodes, an extra leg is added to each task's journey from its originator to the worker node that processes it, hence making the QM more susceptible to slow network response. Like any other user process, the QM can be swapped out of memory by the virtual memory system. Such swapping impairs the ability of the QM to process worker requests in a timely manner, thereby degrading the performance of the user application. In an effort to compare assumptions made by the static load-balancing algorithm with the results of the dynamic load-balancing algorithm, megaflop/second ratings for machines in the heterogeneous network were accumulated in a typical test run of the QMDLB version of the Mandelbrot application (see Table 3.6). The megaflop/sec rating for the machines turned out fairly well, especially considering the fact that they include communication time as well as processing time for each task. More than likely, the good results are primarily a result of the structure of the application itself, since it consists of two tight loops around Equation 3.1.

Table 3.6: Mandelbrot $n = 1000$: Observed Machine Performance (mean sec/task and megaflops/sec reported)

Name	Type	Sec/task	Megaflops/sec	% of Total Tasks
`duncan`	Sun Sparc 10	4.26	3.62	21.43
`austin`	IBM RS/6000 M. 550	2.09	9.73	35.71
`berry`	Sun Sparc 2	10.86	3.54	7.14
`lego`	Sun 4/280	14.91	.51	14.29
`galoob`	Sun 4/280	26.45	.49	14.29

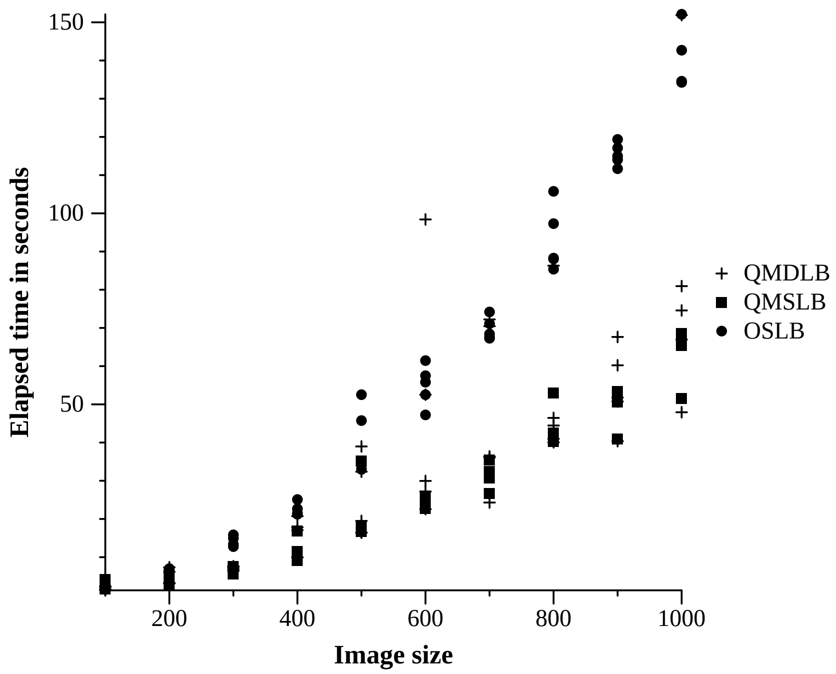

Figure 3.2: Mandelbrot Application, Heterogeneous Network

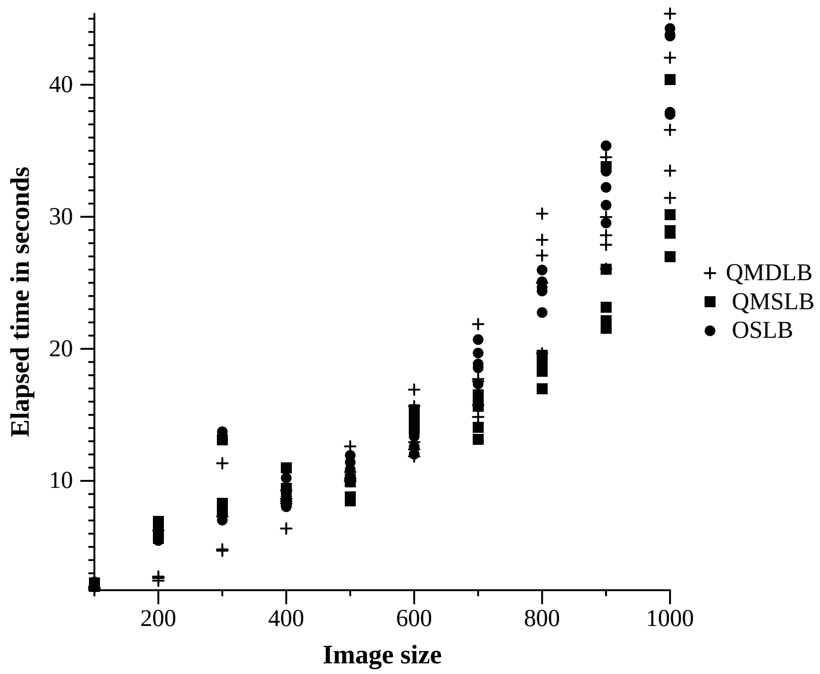

Figure 3.3: Mandelbrot Application, Homogeneous Network

With the exception of the Sparc2, the results of this experiment approximately paralleled the LINPACK rank ordering of the machines. It failed, however, to confirm the significant differences in *megaflop/sec* rates observed in test runs of the LINPACK benchmark. The *sec/task* ratings for `berry` and `galoob` are poor, but their *megaflops/sec* ratings do reflect the fact that they processed larger tasks than the other machines. The two machines with the highest observed LINPACK rating, however, did process the majority of the tasks. Originally, this test was intended to confirm or disprove the assumptions of the static load-balancing portion of the QM with respect to machine capability, but the significantly *improved* performance achieved by the static load-balancing algorithm, argues against such an approach. Network lag time seems to limit the ability of the fastest machines in the network to grab the available tasks, leading to over-utilization of the slower machines in the network and a greater overall elapsed-time for the QMDLB version.

3.3. Conjugate Gradient Benchmark

The Conjugate Gradient (CG) method [GoVL89] is an iterative method for solving a system of linear equations of the form

$$Ax = b. \tag{3.2}$$

At each step of the iteration, the algorithm chooses the direction vector p_k to be the closest vector to the residual vector r_k that is A-conjugate to all previous direction vectors. The residual r_k is then examined to determine if its 2-norm is less than a user supplied tolerance ϵ i.e., $(\| r_k \|_2 < \epsilon)$. Following [GoVL89], the Jacobi preconditioned CG algorithm is defined below. Let the residual r_k be defined as $r_k = b - Ax_k$ for the kth step of CG, and suppose M is the Jacobi preconditioning matrix composed of the diagonal elements of A.

Compute $r_0 = b - Ax_0$ for some initial guess x_0

for $k = 0, 1, ...$

solve $Mz_k = r_k$

if $k = 1$

$p_1 = z_1$

else

$\beta_k = r_{k-1}^T r_{k-1} / r_{k-2}^T r_{k-2}$

$p_k = z_{k-1} + \beta_k pk - 1$

$\alpha_k = r_k^T r_k / p_k^T$

endif

$x_k = x_{k-1} + \alpha_k p_k$

$r_k = r_{k-1} - \alpha_k A p_k$

if $\| r_k \|_2 < \epsilon$, stop

end

Like many PVM programs, the CG application uses a host/node computational mode. The responsibilities of the host, however, consist almost entirely of initiating the node processes, which do all of the *real* work. Node processes are initiated according to the parameters in the file `con.dat`. This file enables the user to specify the dimensions of a two-dimensional processor grid which will be used to solve Poisson's equation

$$u_{xx} + u_{yy} = f, \tag{3.3}$$

where f is a given function of x and y. If u_{ij} denotes the approximate solution at $x_i = ih$ and $y_j = jh$ with mesh spacing h on a rectangular domain Ω, then the finite difference analog of (3.3) is

$$(4u_{ij} - u_{ij+1} - u_{ij-1} - u_{i+ij} - u_{i-ij})/h^2 = f_{ij},\ i, j = 1, 2, ..., n, \tag{3.4}$$

where $f_{ij} = f(x_i, y_j)$.

As discussed in [GoVL89], (3.4) may be cast as an $n^2 \times n^2$ linear system of equations (3.2) where A is a symmetric positive definite (SPD) matrix having 5 diagonals (main diagonal of 4's, off diagonals of -1's). Unlike the Intel i860 hypercube version of this application, the PVM version does not require that n be a power of two. After the nodes are initiated, the unknowns are evenly distributed across the 2-dimensional processor grid, so that each node will approximate the solution at n^2/p interior points, where p is the number of nodes in the PVM network. Suppose the processors, P_k, from the 2-dimensional processor grid are ordered so that $k = (i-1) * p + j$, then a particular processor P_k is assigned unknowns according to its i, j position. Specifically, processor i_p, j_p is assigned unknowns with the coordinates i, j (from the computational grid for Ω) where

$$i = (i_p - 1) * n/P_k + 1, ..., i_p * n/P_k, \tag{3.5}$$

$$j = (j_p - 1) * n/P_k + 1, ..., i_p * n/P_k. \tag{3.6}$$

The nodes then generate the portions of the coefficient matrix corresponding to their unknowns, and CG is used to solve the resulting system of linear equations.

Because of the large amount of inter-node communication, a QMDLB version of the CG application was impractical. The large number of messages generated by the application would have had to have been routed through the QM, since only the QM knows to which processor a particular task was mapped. Forcing the QM to route all these messages would have almost certainly created a network bottleneck which would have limited the performance of the application. Consequently, only a QMSLB version of the CG benchmark was developed.

Task granularities for CG benchmarks corresponding to three different problem sizes are given in Table 3.7. These task granularities are listed *per/machine*, since the QM version of the application was statically load-balanced.

The performance of the QMSLB version of the CG application was excellent. In the homogeneous network, it performed as well as the OSLB version (QMSLB 32.59 sec versus OSLB 34.35 sec for the 130×130 grid) and substantially better

Table 3.7: CG Task Granularities for Heterogeneous Network.

n^2	megaflops/machine	Iterations
100	.006	10
3600	.575	30
16900	5.144	58

in the heterogeneous network (QMSLB 13.34 sec versus OSLB 52.53 sec for the 130×130 grid). A complete set of performance results are provided in Tables 3.8 through 3.9, and Figures 3.4 and 3.5. The excellent results are primarily due to the QM scheduling of a majority of the tasks on the IBM RS/6000 model 550, which is well suited to floating-point computation.

Table 3.8: (CG) Heterogeneous Network: Arithmetic Mean Execution Times (in seconds) and sample variances

	OSLB		QMSLB	
Grid Size	mean time	variance	mean time	variance
10	4.81	3.11	2.96	0.02
20	5.08	0.13	3.70	0.12
30	7.07	5.55	4.67	0.40
40	8.75	7.20	6.11	0.39
50	11.06	3.47	6.87	3.69
60	11.98	6.57	7.89	4.16
70	16.88	7.61	7.93	0.77
80	19.02	11.02	8.85	6.85
90	24.05	32.09	8.78	2.36
100	31.75	54.65	11.54	1.13
110	33.01	29.90	11.69	1.95
120	41.28	16.03	12.88	6.41
130	52.53	166.38	13.34	1.57

The sample variances for the QMSLB version were also substantially lower than those of the OSLB version. This may also be attributed to the QM's tendency to concentrate the majority of PVM processes on the IBM RS/6000, which helped reduce the influence of network traffic on the execution times of the benchmark runs. Moreover, the overall run time was reduced, which made it less likely that a substantial change in system load would occur after the scheduling decisions were made.

Table 3.9: (CG) Homogeneous Network: Arithmetic Mean Execution Times (in seconds) and Sample Variances

	OSLB		QMSLB	
Grid Size	mean time	variance	mean time	variance
10	3.17	0.03	3.05	0.01
20	3.92	0.10	3.90	0.02
30	5.08	0.13	5.91	3.63
40	6.11	0.02	6.41	0.18
50	10.90	73.22	10.57	23.15
60	8.84	0.13	8.98	0.11
70	10.85	0.05	11.53	2.47
80	13.95	4.18	13.03	0.04
90	16.41	5.26	17.06	5.14
100	18.80	0.02	19.04	0.06
110	22.22	0.09	24.62	5.95
120	27.66	6.30	26.92	0.36
130	34.35	10.99	32.59	0.90

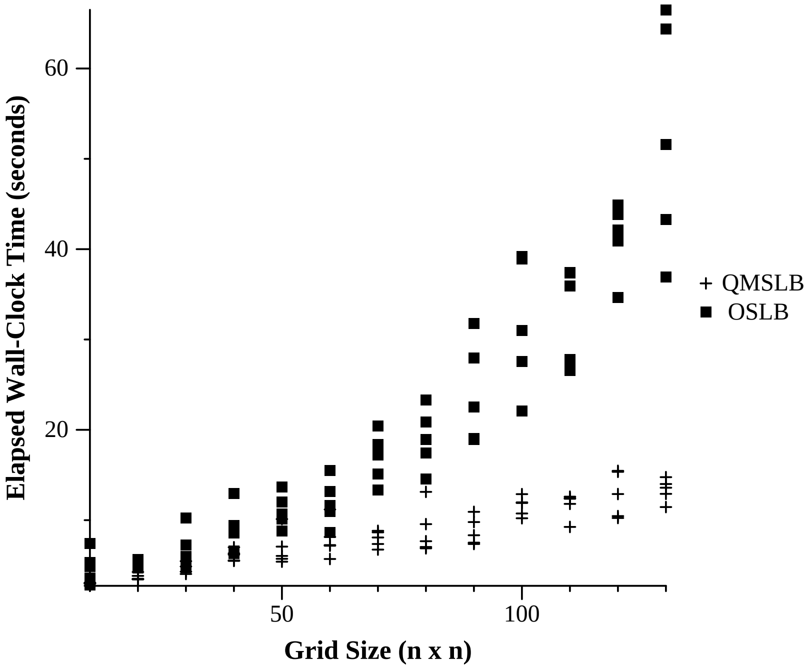

Figure 3.4: CG Method on Poisson's Equation, Heterogeneous Network

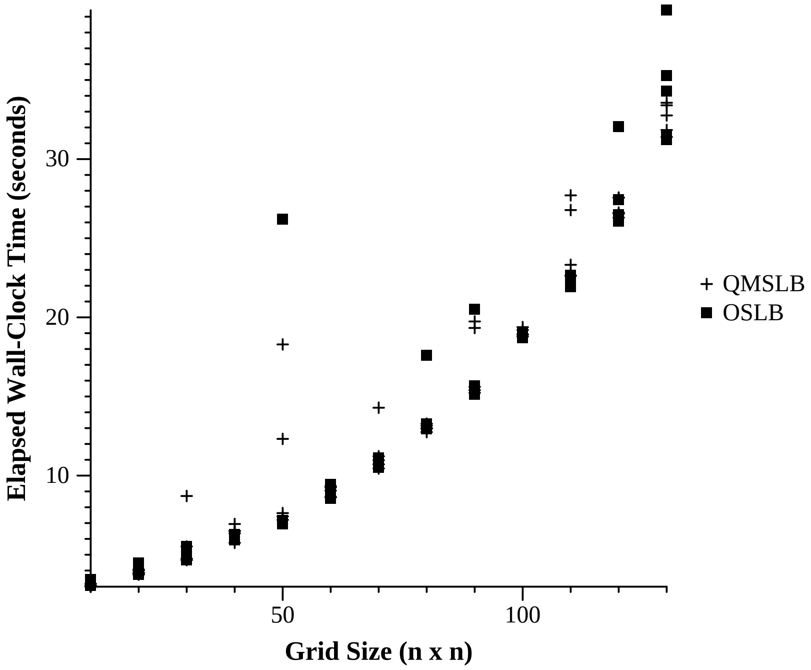

Figure 3.5: CG Method on Poisson's Equation, Homogeneous Network

3.4. Map Analysis Application

The map application benchmark was developed from a computer model used to assess habitat fragmentation and its ecological implications[BeCM93]. It involves the analysis of maps composed of binary pixels in which a 1 represents a *suitable* habitat, and a 0 an *unsuitable* habitat. The actual analysis is performed on clusters of 1's in the map to determine their mean square radius. The computations involved are: (*i.*) Identification of the clusters in the map done sequentially by the host program and (*ii.*) the determination of the mean square radius of each cluster done in parallel on the worker processors. The mean square radius R_s^2 is defined by

$$2R_s^2 = \sum_{i,j} \frac{|r_i - r_j|^2}{s^2}, \tag{3.7}$$

where r_i is the position of the i-th pixel in the cluster, and s is the total number of pixels in a cluster.

Benchmarks were collected on randomly-generated $n \times n$ maps, where n ranged from 64 to 768. Two different map densities were used for each map: $p = .1$ and $p = .3$, where p is the probability that any given pixel will be a 1.

Like the Mandelbrot applications, however, the map analysis code had task sizes of varying granularity, due to the variation in cluster size (see Table 3.10) [BeCM93]. The original PVM version (ODLBF) was ported from the *coarse-grain* version of the map code written in C and CMMD [TMC92] for the CM-5. No modifications were made to the task graph or task granularity for optimization under PVM. Both the PVM and CMMD versions were dynamically load-balanced, with each worker node in the network operating in a cyclic fashion: (*i.*) accepting a single cluster (task) from the host to process, (*ii.*) processing it, (*iii.*) sending an idle message to the host to solicit another cluster, until all the clusters in the map had been processed. A QMDLBF version of the application was also developed, in which the host program sends all the clusters to the QM for dynamic load-balancing. It quickly became apparent, however, that the communication costs associated with dynamically load balancing thousands of clusters severely degraded the performance of the QMDLBF and ODLBF versions of the map application. Consequently, a SLB version of the code was developed. The SLB version substantially outperformed both the ODLB version and the QMDLB version. In an effort to determine if dynamically load-balancing the application could produce *any* performance gain, a *coarser-grain* QM version of the code was developed (QMDLBC). THe QMDLBC version dynamically load balances tasks consisting of cluster *blocks* rather than individual clusters. Due to its coarser granularity, this version is referred to as the QM *coarse-grain* version (QMDLBC), and the previous dynamically load-balanced PVM versions are referred to as the *fine-grain* PVM versions (ODLBF and QMDLBF). Note that the fine-grain versions have the same task granularity as the original CM-5 *coarse-grain* version. A QMSLB version was also developed, but this version proved impractical due to the enormous memory requirements of the application. The QMSLB version of this code frequently attempted to put more than one PVM node on a machine, which ran the machine out of memory (`malloc`s failed) and caused the nodes to terminate abnormally. The four PVM versions of the map analysis application are listed in Table 3.11.

Table 3.10: Distribution of cluster sizes for three p values

p	Map Size	Size of Largest Cluster	No. of Clusters
0.1	64	8	325
	128	6	1305
	256	7	5227
	512	12	20917
0.3	64	25	534
	128	29	2157
	256	33	8484
	512	42	33891

The results of the map application benchmarks were similar to the results of the Mandelbrot application. In the homogeneous network, the QM performed slightly worse than the SLB version of the code due to the overhead of dynamic load-balancing in a network lacking the differences in architecture speed and load which make dynamic load balancing worthwhile (see Tables 3.12 and 3.13, Figures

Table 3.11: Map Analysis Application Version Summary. $N_c \equiv$ total number of clusters in map, $N_w \equiv$ total number of machines in PVM network.

Version	Approximate Granularity of task
ODLBF	1 cluster
QMDLBF	1 cluster
SLB	N_c/N_w
QMDLBC	$N_c/3N_w$

3.6 and 3.8). However, the QMDLB version (see Tables 3.14 and 3.15, Figures 3.7 and 3.9) performed much better than the SLB version in the heterogeneous network (543.63 sec versus 854.41 sec for the 768 $\times$ 768 map with p =.1). Both the fine-grain dynamically load-balanced versions of the application (ODLBF and QMDLBF) had very poor performance. The overhead associated with the large amount of message passing done by these two applications clearly outweighed any increase in parallelism accrued by their fine granularity. Because of their poor performance, results for the fine-grain versions were accumulated on only the three smallest problem sizes ($n = 64, 128, 256$) as an illustration of the perils of fine-grain dynamic load-balancing in PVM. Sample variances (see Tables 3.16–3.18) were significant for all versions, but were especially great for the QMDLB version. As previously discussed, this can most likely be attributed to the QM's sensitivity to changes in network traffic, and the possibility that it could be swapped out of memory at any time.

Table 3.12: (Maps with p = .1) Homogeneous Network: Arithmetic Mean Execution Times (in seconds)

Map Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	1.74	3.38	4.01	8.29
128×128	2.95	6.36	12.97	21.80
256×256	9.42	12.66	49.31	82.41
512×512	64.21	61.36	—	—
768×768	321.20	310.74	—	—

Table 3.13: (Maps with p = .3) Homogeneous Network: Arithmetic Mean Execution Times (in seconds)

Map Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	1.74	2.09	5.59	9.44
128×128	3.12	6.70	19.96	34.08
256×256	22.67	24.70	85.00	137.88
512×512	191.24	150.87	—	—
768×768	780.38	767.08	—	—

Table 3.14: (Maps with p = .1) Heterogeneous Network: Arithmetic Mean Execution Times (in seconds)

Map Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	1.73	1.84	4.62	7.99
128×128	2.25	2.25	13.71	21.66
256×256	11.46	17.12	63.56	100.81
512×512	121.49	193.52	—	—
768×768	543.63	854.41	—	—

Table 3.15: (Maps with p = .3) Heterogeneous Network: Arithmetic Mean Execution Times (in seconds)

Maps Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	1.65	1.54	10.21	18.02
128×128	3.53	5.20	22.79	43.08
256×256	23.20	44.72	101.93	164.72
512×512	441.01	687.54	—	—
768×768	2319.18	3595.59	—	—

Table 3.16: (Maps with p = .3) Homogeneous Network: Sample Variances

Map Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	0.04	1.84	27.12	0.78
128×128	0.02	0.05	3.98	0.01
256×256	1.69	11.74	0.38	2.10
512×512	750.97	2.26	—	—
768×768	3442.98	2137.25	—	—

Table 3.17: (Maps with $p = .1$) Homogeneous Network: Sample Variances

Map Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	0.06	6.72	0.07	27.12
128×128	0.57	0.52	1.03	3.98
256×256	0.14	0.05	0.38	0.38
512×512	6.42	6.23	—	—
768×768	122.32	3399.20	—	—

Table 3.18: (Maps with $p = .1$) heterogeneous Network: Sample Variances

Map Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	1.11	3.36	1.97	1.97
128×128	0.38	0.01	1.71	1.71
256×256	9.10	1.52	16.91	16.91
512×512	148.95	203.40	—	—
768×768	8138.74	1696.71	—	—

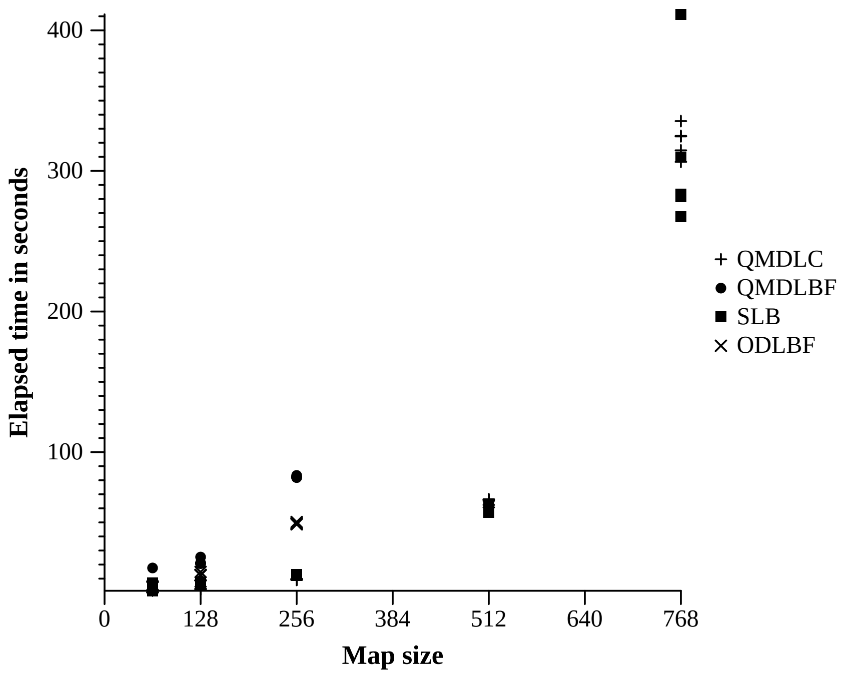

Figure 3.6: Map Application with $p = .1$, Homogeneous Network

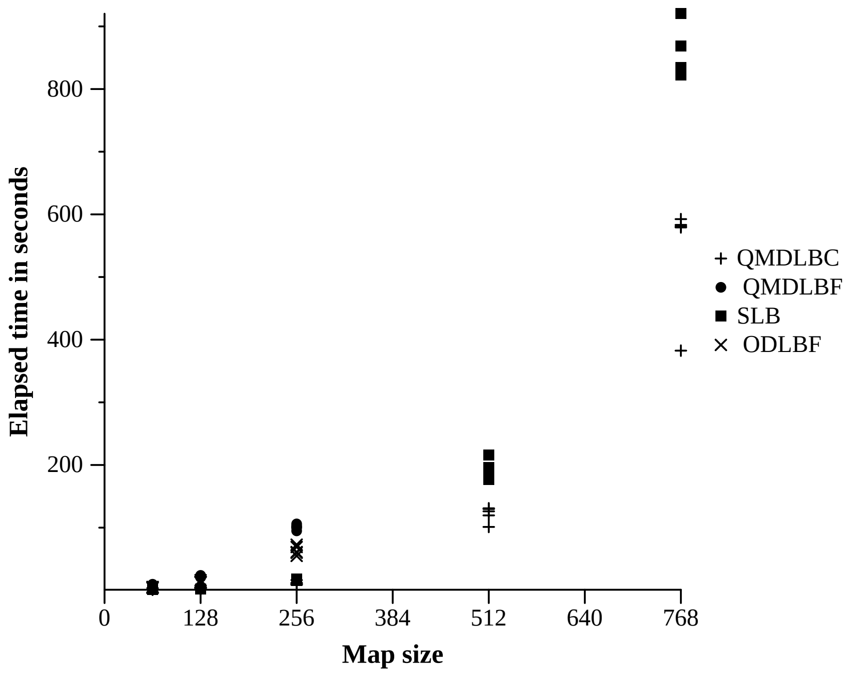

Figure 3.7: Map Application with $p = .1$, Heterogeneous Network

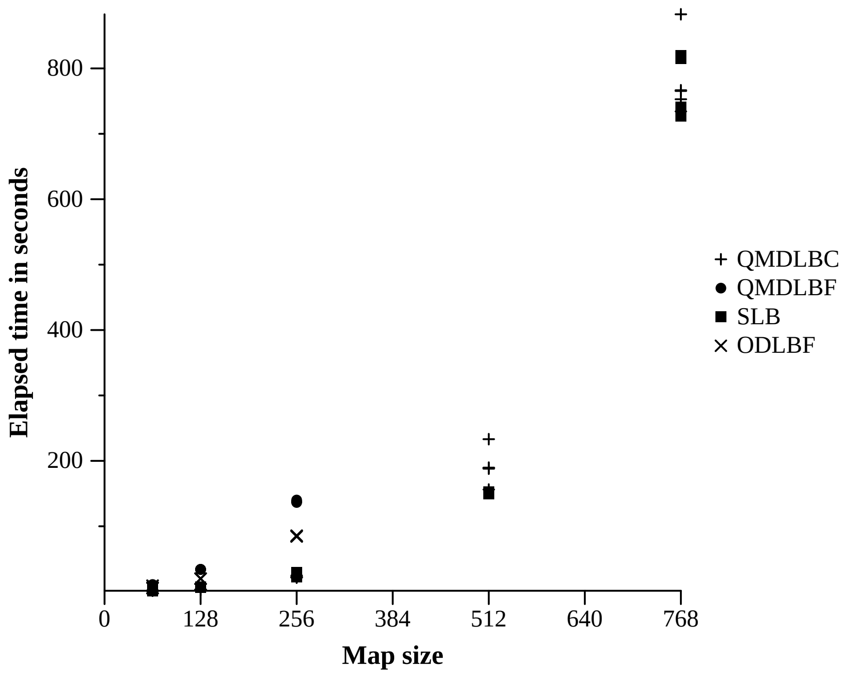

Figure 3.8: Map Application with $p = .3$, Homogeneous Network

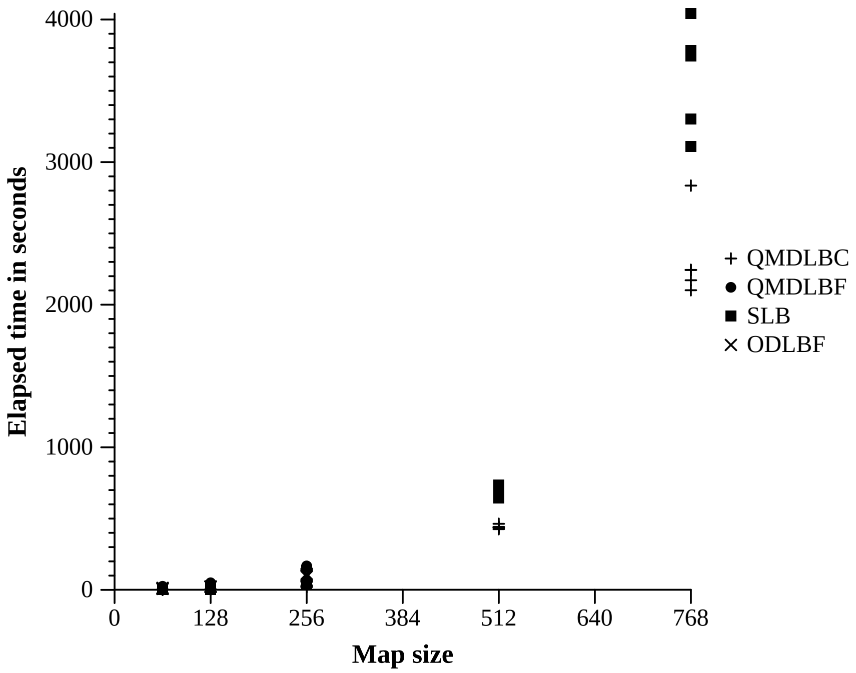

Figure 3.9: Map Application with $p = .3$, Heterogeneous Network

The relatively poor megaflops/sec performance of the map application (see Table 3.19) may be best explained by the memory usage associated with this application. The task granularities in Table 3.20 may be misleading, since they do not reflect actual memory usage. The large memory requirement of this application may cause caused a significant amount of paging on the machines in the PVM network. In fact, each node in the network dynamically allocated 4.8 megabytes of memory for the 768 map size, for example. Moreover, unlike the Mandelbrot application, the map analysis application references a significant number of non-contiguous memory locations which may further contribute to the paging problem. The *rank ordering* of the machines in the network, however, was consistent with the LINPACK benchmark megaflops/sec rating, which further validates the use of this benchmark to determine the relative speeds of computers by the static load-balancing function of the QM.

Table 3.19: Map Analysis ($n = 768,\ \ p = .1$) : Observed Machine Performance

Machine Type	Sec/task	Megaflops/sec	% of Total Tasks
Sun Sparc 10	91.80	.0001100	26.67
IBM RS/6000 Model 550	60.12	.0363980	33.33
Sun Sparc 2	100.85	.0000968	20.00
Sun 4/280	265.52	.0000373	6.67
Sun 4/280	265.18	.0000388	13.33

Table 3.20: Map Analysis Application ($p = .1$) : Task Granularities.

Map Size	flops/task
64×64	96.47
256×256	1125
768×768	10306

4. Conclusions

In the heterogeneous network the QM was able to significantly improve execution times by better utilizing lightly-loaded and faster machines. Moreover, the performance of the QM in the homogeneous network also indicated the overhead associated with the QM was reasonably low. The QM seems to have been successful in improving elapsed run times through load balancing, which was its original aim.

Admittedly, the Mandelbrot and the map benchmarks call into question the practicality of the dynamic load-balancing paradigm when a QMSLB implementation is feasible. Clearly, in the Mandelbrot benchmark, the QMSLB version outperformed the QMDLB version by a large margin. Judging by the excellent performance of the SLB version of the map code, it is possible that a QMSLB version of the map application would perform better than the QMDLB version. The unreliability of Ethernet bandwidth seems to contribute significantly to sample variance as well as increased execution times in the QMDLB versions of the benchmarks. A network can be an extremely complicated environment, with the possibility for a large number of defects/anomalies at any given moment. In such an environment, a clear difference between architecture speeds and/or loads must exist between machines in the PVM network for dynamic load-balancing to be viable.

Bibliography

[AhCG86] S. Ahuja, N. Carriero, D. Gelernter. *Linda and Friends*, IEEE Computer 19(8) 26-34, 1986.

[BeCM93] M. Berry, J. Comisky, and K. Minser. *Parallel Map Analysis on 2-D Grids.* Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing. 312–319. Norfolk, Virginia, March 1993.

[BDGM91] A. Begeulin, J. Dongarra, G. Geist, R. Mancheck, and V. Sunderam. *A User's Guide to PVM.* Technical report CS-91-136, University of Tennessee, July 1991.

[Dong92] J. Dongarra. *Performance of Various Computers Using Standard Linear Equations Software.* Technical report, Oak Ridge National Laboratory, ORNL, June 1992.

[Dong91] J. Dongarra et al. *Solving Linear Systems on Vector and Shared Memory Machines.* Society for Industrial and Applied Mathematics, Philadelphia 1991.

[Glei87] J. Gleick. *Chaos: Making a New Science* Penguin Books, New York 1987.

[GoVL89] G. Golub and C. Van Loan *Matrix Computations* Johns Hopkins University Press, Baltimore 1989.

[Green92] T. Green *Distributed Computing At SCRI* Supertimes, Fall 1992.

[LiLM88] M. Litzkow, M. Livny, and M. W. Mutka, *Condor - A Hunter of Idle Workstations.* Proc. of the 8th International Conference on Distributed Computing Systems, San Jose, California, June 1988.

[Rina92] Martin C. Rinard et. al *Heterogeneous Parallel Programming in Jade.* IEEE Computer 245-256, 1992.

[TMC92] *CMMD Reference Manual* Version 2.0. Thinking Machines Corporation, Cambridge, 1992.

Appendices

A. QM Installation

Installation of the QM is similar to the installation of PVM 2.4. The QM directory should be created as a subdirectory under the PVM directory, and will contain the QM source code, as well as several architecture directories. First, a `get_load` function will need to be written (or copied) and placed in the file `getload.c`. This function must return a floating point number representing the average number of jobs in the run queue over the last minute (the 1 minute load point average). Several of these are available under the name `getload_<ARCH>.c`, and can be copied to `getload.c` if your architecture is one of these. To compile the QM for a particular architecture, move to (or create) the directory corresponding to the desired architecture (i.e. `SUN4, RIOS, SUN3`), then modify the *ARCH* in the `makefile` to reflect the new architecture. Of course, if you just created the directory, you will need to copy the makefile from another directory into the current one before you can modify it. Next, type `make` and four exectuables will be created, `qmanager`, `loadd`, `killq`, `leaveq`, and a user library called `libpvmq.a` which contains the object code for the user call-able QM functions. The user library will remain in that architecture directory, but the two executables (the qmanager and a load daemon) will be moved automatically to the `../../ARCH` directory for the architecure indicated by makefile. Architecture independent make `aimk` can also be used to compile the QM.

At this point, the QM has been installed and is ready to be used. To compile a qmanaged program, the QM object files must be linked when the user program is compiled. This is done the same way the PVM user library is linked. The file `libpvmq.a` is included on the cc line, along with the standard `libpvm.a` library.

B. Harmonic Mean Execution Times for Benchmarks

Table B.1: (Mandelbrot) Heterogenous Network: Harmonic Mean Execution Times (in seconds)

Image Size	OSLB	QMSLB	QMDLB
100×100	2.65	1.94	1.85
200×200	6.25	4.11	4.02
300×300	14.12	6.88	7.05
400×400	22.29	11.53	13.74
500×500	38.61	19.64	21.78
600×600	54.47	23.66	34.79
700×700	69.62	31.03	40.19
800×800	92.37	42.98	47.63
900×900	115.37	49.02	52.52
1000×1000	139.28	63.20	73.37

Table B.2: (Mandelbrot) Homogeneous Network: Harmonic Mean Execution Times (in seconds)

Image Size	OSLB	QMSLB	QMDLB
100×100	2.26	2.12	1.83
200×200	6.42	6.36	2.96
300×300	8.50	8.91	6.02
400×400	8.69	9.06	8.08
500×500	10.91	9.44	10.61
600×600	13.02	14.67	13.68
700×700	19.96	14.94	17.23
800×800	24.52	18.43	25.47
900×900	32.17	24.66	29.15
1000×1000	41.26	30.44	37.08

Table B.3: (CG) Heterogenous Network: Harmonic Mean Execution Times (in seconds)

Grid size	OSLB	QMSLB
10×10	4.31	2.95
20×20	5.06	3.68
30×30	6.53	4.61
40×40	8.17	6.06
50×50	10.81	6.53
60×60	11.53	7.52
70×70	16.51	7.85
80×80	18.54	8.35
90×90	23.05	8.58
100×100	30.29	11.46
110×110	32.06	11.54
120×120	40.94	12.48
130×130	49.93	13.24

Table B.4: (CG) Homogeneous Network: Harmonic Mean Execution Times (in seconds)

Grid size	OSLB	QMSLB
10×10	3.17	3.05
20×20	3.90	3.90
30×30	5.06	5.55
40×40	6.11	6.38
50×50	8.29	9.25
60×60	8.83	8.97
70×70	10.84	11.38
80×80	13.75	13.03
90×90	16.20	16.83
100×100	18.80	19.04
110×110	22.21	24.43
120×120	27.50	26.91
130×130	34.11	32.57

Table B.5: (Maps with $p = .1$) Homogeneous Network: Harmonic Mean Execution Times (in seconds)

Map Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	1.74	2.16	3.99	6.87
128×128	2.83	6.30	12.91	21.67
256×256	9.41	12.66	49.31	82.41
512×512	64.13	61.28	—	—
768×768	320.89	303.46	—	—

Table B.6: (Maps with p = .3) Homogeneous Network: Harmonic Mean Execution Times (in seconds)

Map Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	1.73	1.71	5.58	9.38
128×128	3.12	6.70	19.96	34.08
256×256	22.61	24.34	84.99	137.86
512×512	188.19	150.86	—	—
768×768	777.13	764.90	—	—

Table B.7: (Maps with p = .1) Heterogeneous Network: Harmonic Mean Execution Times (in seconds)

Map Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	1.42	1.20	4.52	7.79
128×128	2.16	2.25	13.60	21.60
256×256	10.92	17.04	62.91	100.68
512×512	120.40	192.71	—	—
768×768	528.26	852.88	—	—

Table B.8: (Maps with p = .3) Heterogeneous Network: Harmonic Mean Execution Times (in seconds)

Map Size	QMDLBC	SLB	ODLBF	QMDLBF
64×64	1.58	1.42	9.55	16.98
128×128	3.49	4.78	22.77	42.78
256×256	22.95	44.69	101.51	164.67
512×512	440.67	686.36	—	—
768×768	2293.16	3562.69	—	—

C. Statistical Formulas used to Compute Tables

Table C.1: Definitions of Statistical Formulas. n= Number of Samples, x_i=i-th elapsed wall-clock time sample.

Term	Definition
arithmetic mean	$\mu = \frac{\sum_{i=1}^{n} x_i}{n}$
harmonic mean	$\mu_h = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$
sample variance	$s^2 = \frac{\sum_{i=1}^{n} (x_i - \mu)^2}{(n-1)}$

VITA

Douglas Sept was born in Pocatello, Idaho on April 17, 1969. He graduated from Pocatello High School in 1987 and received the Bachelor of Science degree in Computer Science from Trinity University in May 1991. In August 1991, he entered the Computer Science program at the University of Tennessee and was awarded the Master of Science degree in Computer Science in August 1993.