IOWA STATE UNIVERSITY
Digital Repository

Retrospective Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

1995

Monitorable network and CPU load statistics and their application to scheduling

Trevor Ethan Meyer
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

 Part of the Computer Sciences Commons

Recommended Citation
Meyer, Trevor Ethan, "Monitorable network and CPU load statistics and their application to scheduling " (1995). *Retrospective Theses and Dissertations*. 11071.
https://lib.dr.iastate.edu/rtd/11071

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

www.manaraa.com

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Monitorable network and CPU load statistics and their application to scheduling

by

Trevor Ethan Meyer

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements of the Degree of

DOCTOR OF PHILOSOPHY

Department: Electrical and Computer Engineering
Major: Computer Engineering

Approved:

Signature was redacted for privacy.

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1995

Copyright © Trevor Ethan Meyer, 1995. All rights reserved.

UMI Number: 9610972

UMI Microform 9610972
Copyright 1996, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI

300 North Zeeb Road
Ann Arbor, MI 48103

TABLE OF CONTENTS

1. INTRODUCTION

Nothing is more certain in science and engineering than the trend that what seems a powerful computer system today will prove woefully inadequate for tomorrow's problems. Although processor speeds steadily increase, and memory and storage capacity continuously grow, application software always grows faster, taxing hardware technology. One way to gain processing power without waiting for CPU technology advances is to apply more than one standard processor to a problem [44][20]. This solution is particularly attractive in the context of some of the most computationally-intensive physics problems, which neatly divide into largely independent pieces.

Unfortunately, commercial multiprocessors and parallel computers are prohibitively expensive for all but the largest research facilities. Shared supercomputing centers alleviate the problem somewhat, but computer time is still expensive and must be carefully rationed. As an alternate solution, currently gaining popularity, researchers are using existing networks of workstations to attack their difficult computation tasks. These workstations are treated as processors within a multiprocessor, using the interconnection LAN to pass parameters and data among running processes. Although the LAN architecture is not optimized for this use, existing networks in many facilities provide for high performance computing at almost no cost. All that is required is support software to schedule tasks on the workstations and handle data exchange between them [55].

Much work is ongoing in the development and evaluation of intelligent scheduling schemes for workstation networks [14][1][29][54]. These schemes typically involve the measurement of system load, and the use of the resulting measurements to find suitable machines for task placement. What the literature lacks, however, is any description of the methods used to measure load and their effectiveness. Not one of innumerable papers examined describing load monitoring schedulers provided an explanation of the methods used to measure load or of the typical profile of the load measures obtained. Even more surprising, none provided any quantitative results justifying the use of the load measures chosen; it is simply assumed that a measure such as load average provides insight into the run time of an application program. This

may be the case; to be certain, however, a quantitative, experimental study is required to precisely determine the relationship between the load measure in question and the target application(s). Such a study would provide insight into the affect of a given rise in load on run time, and on the magnitude of change in load required before program run time is significantly altered.

This dissertation describes such a study. Specifically, two software-implemented CPU load measures, and one software-implemented network load measure, were implemented under the PVM distributed computing environment for parallel processing on workstation networks. The relationship between load, as given by these measures, and the runtime of an example application was then examined through the use of extensive experimental test runs, recording the application run time and the three load values. In addition, the behavior of the three load measures in response to artificial system loading was examined, and insights gained into the behavior of the load measures themselves and their ability to reflect system conditions explained. The results of the experiments are then examined in the context of scheduling. Specifically, the role which load plays in common scheduling algorithms is detailed, and the application of the above gained knowledge within this context is expounded upon in detail.

1.1 Problem Statement

This dissertation describes research with the following goals: (1) Develop CPU and network load measures which may be easily implemented in software, and create utilities to record these measures unobtrusively on a running PVM system. (2) Develop a test PVM application program which mimics the behavior of a typical, computation-bound multiple task program; the behavior of this program may be modified through runtime inputs, varying total amount of work performed and percent of work spent in communication. (3) Record runtimes for the test application under varying load conditions, both CPU and network, and derive a relationship between load and run time; this relationship might be used to predict the run time of future runs of the program, given load. (4) Examine the behavior of the load measures under varying load conditions, to determine their significance and identify weaknesses and strengths. (5) Examine network segments characteristics, based on load measures, for the purpose of

identifying high traffic network segments unobtrusively. (6) Search for methods, based on CPU and network load measures, to automatically locate the limiting machine in a PVM configuration, so that it may be removed during future application runs. (7) Determine ways in which the information learned in (3), (4), (5), and (6) may be used during the implementation of a scheduler for compute-intensive programs on workstation networks; specifically, examine the role of run time prediction and load monitoring in common scheduling algorithms.

1.2 Contribution of this research

This research fills a void in the current literature by experimentally examining the role of system loading in parallel processing on workstation networks. In particular, the commonly used load measure provided by the UNIX kernel, load average, is analyzed in detail to determine its value as a measure of actual system load. Although load average is frequently quoted as a measure of CPU load, such usage is seldom justified. This study provides detailed insight into the relationship between load average and the execution time of a computation-intensive program. In addition, the role of network traffic, both on communicating and non-communicating tasks, is considered. Finally, load profiles are studied, with consideration both for typical values over time during normal system usage, and behavior during excessive loading conditions.

The typical use for load measures is during task scheduling, and this research focuses on the application of the experimental results toward common scheduling algorithms. In particular, the information policy of a scheduler commonly monitors system load. In the conclusion section of this dissertation, the applicability of the experimental load measures toward the information policy of a scheduler is discussed, as is the usage of load for application run time prediction; run time prediction allows the use of scheduling algorithms based on total run time of submitted tasks, algorithms which normally require external user-supplied information. The ability to predict run time makes many common algorithms more attractive for implementation.

Network load is also applicable to task scheduling, but is less studied even than CPU load. Due to the nature of network load, is monitoring and usage is complex. This study considers a simple means of identifying busy network segments, so that they may be avoided

during scheduling of communicating tasks. The method given for monitoring network traffic is unobtrusive and straightforward to implement. As with the other load measures described in this work, no special equipment of OS modifications are required for implementation.

Although these experiments were performed on one particular network with a single application, the technique developed for determining load-run time relationships may be applied to any workstation network system with any computation-intensive application program. The importance of the results, therefore, lie in the technique used to derive the numeric values, as opposed to the values themselves. The deliberate avoidance of system-dependent techniques, such as use of internal kernel data, allows the processes described in this dissertation to be applied to any workstation network with any number of processors.

1.3 Research Overview

The goal of this research was to discover the relationship between load, as measured through several software techniques, and runtime for a typical parallel PVM application. Both CPU and network load were to be measured, and their affects measured for both programs with no communication, and programs with varying levels of communication. If a relationship was found, that relationship was to be applied toward the problem of process scheduling on networks of workstations.

The research was performed through the use of experiments performed on a network of workstations over a period of a year. A test application program was written to mimic the behavior of typical parallel, computation-intensive program, whose behavior could be easily modified through choice of runtime startup parameters. These parameters allow the variation of computation-communication ratio, communication patterns between processes, and distribution of computation and communication operations. Additional programs were then written to monitor loading on a network of workstations; one program measures CPU loading on each machine, while another measures network load on the network segment(s) connecting the machines. Each experiment consisted of repeated executions of the test application program while the load monitors recorded loading conditions. The resultant data results contain application runtimes and the loads measured during those runs.

The data from each experiment was preprocessed, then submitted to regression analysis, the results of which were, in some cases, further processed. Regression revealed correlation between runtime and one measure of CPU load, load average. It also revealed very close correlation between runtime and communication-computation ratio, measured in units of time, for one or more of the participating processes. No correlation was seen between network load, measured as round trip packet delay, and runtime, even for communicating processes. However, network segments were found to be classifiable by three characteristics, derivable from the network load delay measure.

Finally, the above results were applied to the problem of parallel process scheduling on networks of workstations. The correlation between load average and runtime are used to predict relative execution times for a program, and to predict absolute execution time given minimal past knowledge of the program's behavior under load. The network characteristics are used to locate and avoid busy network segments. And the correlation between communication-computation time ratio and runtime are used to locate limiting machines in a network and to aid in runtime prediction.

1.4 Organization of Dissertation

The remainder of this dissertation provides background information, surveys the current literature, and discusses the above experiments and results in detail. Chapter 2 defines workstation networks as used in this research, including ways in which workstations may be used for parallel processing. Chapter 3 details the problem of process scheduling, and lists popular scheduling algorithms and techniques for improving algorithm performance. Software written to support parallel processing and scheduling is described in Chapter 4, in particular PVM, the software used in this research. In Chapter 5, the system load models assumed for this research are given, and in Chapter 6, the software programs used for the experiments are detailed. Finally, Chapter 7 gives the results of the data analysis, and Chapter 8 provides conclusions, focusing on the application of the results to the problem of scheduling.

2. PARALLEL PROCESSING ON WORKSTATION NETWORKS

As workstation computers drop in price, large networks of workstations become commonplace in both industry and academia. The number of machines involved varies from a few to hundreds, all interconnected by a local area network, such as ethernet. The machines themselves are multitasking, usually running the UNIX operating system. And though their CPU power compares to that of mainframes less than a decade ago, these workstations are generally considered a single-user resource. For that reason, a collection of workstations represents a large, mostly untapped computing resource, with machines largely idle while their owners are away, or even while in use by their owners, engaged in low CPU-usage tasks such as reading email. The question naturally arises: can these many idle CPU cycles be captured somehow to perform useful work?

2.1 Types of Workstation Networks

There are two basic types of workstation networks: homogeneous networks and heterogeneous networks, with three basic structures: general purpose networks, dedicated farms, and wide-area networks. All consist of a set of computers connected by a communication medium. The computers may be PCs, UNIX boxes, or giant supercomputers; most common are "workstation class" computers, somewhere between PCs and mini-computers, but no computer types are excluded. The network may be local area (LAN) or wide area (WAN), and may be based on ethernet, token ring, FDDI, or some other technology. The important trait shared by all workstation networks is that they support multiple computers, any one of which may communicate with any other.

A *homogeneous* network contains machines that are all the same "type." The definition of "type" varies, but at minimum it indicates object-code compatibility. Frequently, it also implies that the computing environment, including filesystem, is identical at every machine. The critical property of a homogeneous network is that a given program may be executed, without modification, on any of the machines in the network.

A *heterogeneous* network is one that is not homogeneous; it contains computers of varying architectures, so that a program will not execute on every machine in the network without being recompiled. Sometimes, a heterogeneous network is really a homogeneous collection of small computers connected to one or two special architecture machines, such as parallel processors or supercomputers. The special computers are included to process compute-intensive jobs especially suited to their unique architectures, while the many small computers perform support processing and provide a user-interface to the system.

As mentioned above, computer networks, whether homo- or heterogeneous, are structured in one of three ways. This structure is determined by the purpose to which the computers in the network, and the network as a whole, are dedicated.

Within a *general purpose* network, the computers are available for use in a variety of tasks, typically with console and remote logins possible. This is the most common type of workstation network, the type that has given rise to so much research lately. The normal workload of a general purpose network does not include parallel or batch processing; rather, these tasks are performed while machines on the network are idle, in an attempt to use the excess computing power. First and foremost, however, the computers are dedicated to the individual users.

A *workstation farm*, on the other hand, is a collection of computers networked together for the sole purpose of parallel or batch processing (see below). Frequently, machines within a workstation farm are only accessible remotely, without individual consoles. Viewed as a computation resource, the computers in the farm are accessed through a front end, in a manner similar to most supercomputers, and are targeted at long-running, computationally intensive jobs. Workstation farms are not nearly as common as general purpose networks, and are typically found only in research institutions.

Wide area networks are networks that span more than one institution. A common reason for implementing a wide area network is to include special resources not available locally, such as a supercomputer. In a sense, any local computer network that is attached to the Internet is part of a large wide area network. However, in the sense used here, machines at the remote site must be accessible to the local users; two networks that are simply connected to exchange email do not fall under this category.

The work described in this document focuses on a homogeneous, general purpose network. A homogeneous network was chosen for convenience; the results discussed in later sections apply equally well to a heterogeneous network, and PVM makes a transition between the two almost invisible. A general purpose network was chosen for two reasons: (1) it is much more common than workstation farms and wide area networks, and (2) it presents many challenges not applicable to workstation farms. The architecture of the specific workstation network used in this study is detailed in Section 4.4.

2.2 Using the Idle Cycles

Remote execution facilities supported by UNIX and other operating systems allow the "idle" workstation cycles to be used unobtrusively, by farming work out to idle workstations for background execution. With OS network support, a single workstation can start a program running on any machine in the network, send it necessary runtime parameters and collect results upon completion. Before examining such possibilities, however, it is important to define what is meant by "idle cycles" or "idle workstations."

Qualitatively, an *idle* machine is a machine whose computing resources are underutilized. Quantitatively, one might measure idleness in two ways: 1) a machine is idle when its owner is absent (there is no console activity) or 2) a machine is idle when its load average falls below some threshold.

The first definition is important as it relates to the "good neighbor" policy, which itself may prove crucial in gaining the use of other people's workstations for background processing. By considering a workstation with an active console busy, one ensures that a background task will not degrade workstation performance while the workstation is being used by its owner, where the "owner" is the individual sitting at the workstation console. If workstation owners are assured that background processing will not interfere with their work, they are more likely to allow use of their computer while it is idle. Generally, workstation activity is defined as keyboard input or mouse movement. On method in use currently [7] is to wait for a set time period after the keyboard and mouse become inactive. Once the time period has expired, the workstation is considered idle and one or more background jobs may be started. These jobs are

periodically checkpointed, saving the work completed so far. When keyboard or mouse activity resumes, the running background jobs are stopped for a set waiting period. If the workstation is not idle again within that period, the jobs are killed and the most recent checkpoint file is transferred to another machine where the running jobs are reconstructed and resumed. This technique ensures unobtrusiveness while minimizing lost work when background jobs are removed.

An owner-oriented definition of idleness tends to lead to lower overall efficiency, however, because common console activity, such as editing files or reading email, underutilizes the CPU. Measured in terms of load average (see below), the load on a machine with a single user at the console is typically very small [31]. A more efficient definition of idleness may be derived from a count of the average number of user jobs awaiting execution over some short time period, known as the load average. This count provides a measure of the relative busyness of the workstation, with a value of 1 or more indicating that the processor is completely utilized. Using load average, an idle workstation is defined as a workstation whose load average is consistently under 1 over some predefined time period. The distance under 1, ranging all the way down to 0, then provides a continuum of idleness values. Thus, a scheduling entity with access to multiple idle machines can find the most idle machine when starting a new job; with a user-oriented idleness definition, only a Boolean-valued measure is available, and the scheduler has no way of knowing when a workstation will become busy again. On the downside, a workstation owner who is editing a file on a workstation with load average 0.05 may complain if a background job suddenly appears in the system, raising the load average to 1 or more. If the owner permanently removes the workstation from further consideration for background processing, the quality of the idleness measure, relative to that machine, is immaterial.

Once idleness is defined, software may be written which will accept jobs from users and then hunt for idle workstations on which to run them. This software may be implemented at the OS or the user level. The foundation for such a system is straightforward. A queue manager collects submitted jobs while monitoring available workstations. Depending on implementation, the manager either waits for a workstation to become idle, or chooses the most idle workstation, and sends the next job in the queue to that machine. The job may be executed and communicated with using the network primitives built into the underlying OS. While the job executes, a "stub"

program may be maintained on the submitting machine, providing the user with the illusion that the job is running locally. When the job completes, the manager collects results and ships them back to the source machine. A number of examples of this type of system are given in Section 4.3.

2.3 The Workstation Network Multiprocessor

An extension of the above ideas, currently being investigated by a number of researchers [11][43], is to use the workstations as individual processors within a multiprocessor. A program exhibiting task-level parallelism could then be executed on the workstation network by dividing the program into a group of processes and then executing a process on each participating machine. Processes may exchange data across the LAN that connects the computers, using TCP/IP or whatever transport protocol is supplied by the OS. With slight modifications, the queue manager described above could handle these parallel programs just like a series of unrelated jobs.

A large number of experimental software systems for distributed parallel processing on workstation networks are currently available. Many of these are briefly described by L. Turcotte [55], and a few are discussed in Sections 4.2 and 4.3. For the most part, these existing packages provide the framework for parallel processing described above, but little more; obtaining satisfactory performance with a given algorithm relies on the ingenuity of the programmer. A number of problems arise that must be addressed, some by the application programmers and some by the distributed OS designers. Some of the more important ones are briefly described below.

Algorithm suitability: Not all parallel algorithms will execute equally well on a multiprocessor whose underlying hardware consists of workstations connected by a LAN. Such a system is typically characterized as a collection of powerful processors connected by a costly communication bus. The high cost is due not only to the inherent speed of the LAN itself, but also to the fact that all processors must share a single channel (though a unique solution to this problem, using ethernet, is given by Tavangarian, et al. [53]). Thus, algorithms with a high ratio

of computation to communication are better suited for such a system. In cases where executable files must be transferred to a target machine over the LAN, and results transferred back, overhead can also be a factor, leading to restriction of the system usage to relatively long-running programs. More will be said about algorithm suitability in Section 8.7.2.

Scheduling: When the workstations in the network are not dedicated to parallel processing, intelligent process scheduling becomes important, particularly when long-running jobs are involved. In a non-dedicated system, the parallel distributed OS kernel has no control over load introduced by outside users. Since the execution time of a parallel program is determined by its slowest component, a high load on a single machine can drastically reduce overall performance. The longer the job, the greater the probability that one or more involved machines will become busy at some time during execution. The solution to this problem is not a simple one, but an outline is easy to develop: load statistics external to the parallel jobs may be collected to gauge to busyness of each workstation, and a load balancing algorithm may be implemented to avoid overloaded machines when starting remote jobs, and possibly move running jobs off machines that become overloaded. These issues will be discussed in more detail in Sections 3.2 and 4.2.

Obtrusiveness: In the case of a non-dedicated workstation network, a parallel program must not interfere with users currently on the network. The solution, as outlined above, is to use only idle workstations. However, finding idle workstation may itself disturb the users on non-idle workstations, if not implemented correctly. In addition, monitoring load statistics and moving processes during load balancing operations may cause noticeable degradation in performance on involved machines. If general purpose machines are to be used for parallel processing, these interference problems must be avoided. Some aspects are interference are discussed in Sections 3.2 and 5.3.

3. PROCESS SCHEDULING

As mentioned earlier, process scheduling is an important part of parallel computing on workstation networks. To understand the challenges of process scheduling on workstation networks, it is important to understand the scheduling problem itself and the components of a scheduler. This chapter describes the scheduling problem, and discusses some optimizations and their applicability to general purpose workstation networks.

3.1 The Scheduling Problem

The most challenging aspect of parallel processing on workstation networks is process scheduling. Put concisely, the scheduling problem is this: given a set of possibly interrelated tasks, what task-processor assignment gives the "best" overall performance. Best is put in quotes because there is more than one measure of the goodness of an execution schedule; the measure chosen is dependent upon the way in which the system is used. Typically, a good schedule satisfies one of the following goals: provide most efficient processor utilization, provide highest program throughput, or provide best response time. These three goals are not necessarily compatible. The first goal, efficient utilization, is satisfied if all processors are kept busy; this is best accomplished by scheduling long-running jobs that use all available processors. The second goal, high throughput, is satisfied if the largest number of programs are completed per unit time; high throughput is typically attained by scheduling short-running processes in favor of long-running processes. The last goal, best response time, is satisfied if every submitted program completes in the shortest possible amount of time; this goal may be attained by executing several programs at once, in order of submission, each using a subset of available processors.

The scheduling problem is defined by Casavant [9] as a resource management resource problem. The scheduler is a resource, which manages a collection of processors, or computing resources. In a typical implementation, the scheduler is a part of the operating system kernel which maintains a queue of submitted program jobs. The jobs, in turn, consist of one or more processes, multiple processes representing parallelized jobs. Through kernel data structures, the

scheduler maintains a picture of the available system resources. This picture could be simple, consisting merely of a list of available processors, or complex, consisting of numerous, frequently updated system loading statistics. During a scheduling event, the scheduler takes the most suitable job from the queue and sends it to the most appropriate processor or processors. Determining when a scheduling event occurs, which job is scheduled, and what processor(s) receive the job is the duty of the scheduling algorithm.

Scheduling algorithms may be placed in a treelike taxonomy. At the root of the tree are two broad classes of scheduling policies, local and global. A *local* policy applies to the scheduling of processes within a single processor, and is the domain of the local operating system kernel on a given machine. A global *policy* applies to the scheduling of processes within a group of processors, and as such is directly applicable to multiprocessors, including workstation networks. From this point forward, the reader should assume all policies discussed are global policies.

A global policy is said to be static or dynamic. A *static* scheduling algorithm determines the processor on which a task will execute at the time the task's object code is linked by the compiler, and the processor-task assignments are written into the executable file. Much work has been done on the static scheduling problem, and it is better understood than dynamic scheduling; some examples are given in [60][59][47]. Static scheduling assumes that the topology of the multiprocessor system is itself static, and well understood at link time. Because this assumption is not realistic for a workstation network, static schemes are not as useful in this type of system as the alternative, dynamic scheduling algorithms.

By contrast, a *dynamic* scheduling algorithm chooses the destination processor for a given task at run time, based on current information about the processor system. This information must be updated periodically by the operating system and, as mentioned above, may be as simple as a list of machines or as complex as a detailed set of loading statistics.

Dynamic policies are further divided into distributed and centralized algorithms. A *distributed* algorithm is executed by a collection of processors, possibly by all processors in the system, or by a single processor within each of a number of processor groups. A *centralized* algorithm is executed by a single, master processor. In a workstation network, a single processor scheduling authority has the advantage of simplicity; however, when communication costs are

high, as they often are on a shared bus network like ethernet, a distributed authority may give better performance. Performance of a centralized authority on such a system will suffer due to overhead; in order to gather information about the system, the authority must poll all machines, generating excess network traffic in the process. The Rediflow computer [36][33][34] represents a unique solution to the polling problem is discussed in the context of load balancing (see Section 3.3.5). This solution, used in the design of a multiprocessor/operating system called "Rediflow," illustrates the effectiveness of a fully-distributed scheme in which processors need to communicate only with their immediate neighbors.

During scheduling in a distributed system, processors may function noncooperatively or cooperatively. In a *noncooperative* distributed system, each processor acts independently, making decisions based on local factors only. These decisions may have a negative overall affect on the system. In a *cooperative* system, all processors work together and decisions are made based on both local and global conditions. Generally, a noncooperative algorithm is easier to implement but may yield lower overall system throughput than a cooperative scheme.

Because a cooperative scheme requires multiple processors to work together, achieving an optimal scheduling solution may be difficult, particularly when more than a few processors are involved. To obtain an *optimal* schedule, all information that is relevant to the scheduling problem must be known, and a solution found based on this information. In many cases, the optimal solution process is known to be NP-complete [22], the required solution time growing at a larger-than-polynomial rate with the number of processors. Even with modest numbers of processors, the calculation time for an optimal solution becomes prohibitive. In these cases, a suboptimal solution is necessary.

A *suboptimal* scheduling solution may be approximate or heuristic. During an *approximate* solution process, the optimal solution procedure is applied to an artificially limited solution space. The solution time is thus reduced, but the results are not guaranteed to be optimal. Even so, in many cases an approximate method will produce a schedule as good or nearly as good as an exact, optimal method; the difficulty lies in deciding how to reduce the solution space.

On the other hand, a heuristic solution process uses a trial-and-error approach incorporating factors that are known to affect the system, but whose affects are difficult to

quantize. An example heuristic given by Casavant is the clustering of tasks that must communicate frequently on a single processor; when communication cost is high, such a solution improves performance by exploiting communication locality (see Section 3.2.3). In general, heuristic methods are based on intuition and experimentation rather than provably correct procedures.

This taxonomy serves to classify scheduling algorithms and to allow comparison between them; it does not attempt to gauge the performance of any one algorithm. A number of specific scheduling algorithms are compared and contrasted by Levis and El-Rewini [35]. These and other algorithms are described in Section 3.3, and examined in the context of workstation network multiprocessors in Section 8.7.

3.2 Scheduling Improvement Techniques

There are several common techniques, often applied in a heuristic manner, which will improve execution schedules on multiprocessor systems, including workstation networks. The most common are load balancing, load sharing, preemptive migration, and locality management. Additionally, in the case of workstation networks, the "good neighbor" policy can improve performance in a different sense, by encouraging users to donate their machines to the system. During the design of a scheduling algorithm, each of these techniques should be carefully considered, as their impact on performance could be large.

3.2.1 Load balancing and load sharing

Though frequently used interchangeably, there is a subtle difference between load balancing and load sharing. A *load balancing* algorithm attempts to distribute work so that all processors have identical workloads at any given time [12]. A *load sharing* algorithm, on the other hand, merely ensures that no processor is idle while jobs are queued [56]. The general goal of each is the same, however: to maximize throughput.

An overview of the load balancing/load sharing problem, henceforth referred to as the load leveling problem, is given by Shivaratri, et. al. [48]. Issues that must be considered in the design of a load balancing algorithm are discussed, as are the basic components of the algorithm

itself. These components will be examined here, with particular attention paid to the case of the workstation network multiprocessor.

To understand load balancing algorithms, one must first understand the concept of load. The *load* at a given processor is usually modeled as a queue of tasks. A newly arriving task is inserted at one end of the queue, while the opposite end supplies tasks for execution. An important first step in designing a load leveling algorithm is the choice of a suitable loading measure. This measure, called the *load index*, is a value that can be calculated from known information about the system and that indicates the relative performance of a task if sent for execution on the measured processor. In the case of the workstation network, the load index may include such factors as processor queue length, local network utilization, distance between communicating tasks, and average background load. This latter factor, as distinguished from individual machine loads, is a system-wide traffic pattern that varies in a known way with the time of day and/or day of week; it is discussed further in Section 8.7.3.

3.2.1.1 Types of load leveling algorithms

As with general scheduling algorithms, load leveling algorithms may be categorized in a tree-like manner. At the top level, the load leveling authority may be centralized or decentralized. A centralized authority resides on a single processor, while a decentralized authority is spread among multiple processors. In choosing between the two, the same issues must be considered as were discussed in the context of general scheduling algorithms: communication overhead versus complexity. A centralized algorithm is simpler to implement but must gather information from throughout the network, possibly incurring high communication overhead, while a decentralized algorithm eliminates some information gathering but necessitates a more complex implementation.

Load leveling algorithms can also be classified as static or dynamic, though these terms should not be confused with static and dynamic scheduling. In *static* load leveling, jobs are located without regard to current system state; a static algorithm uses a priori knowledge about the system configuration to make its decision. As with static scheduling, this method will fail if network topology is mutable, as is the case with workstation networks. A *dynamic* load leveling algorithm uses system state information to place tasks, and can thus adapt to changing system

parameters. A dynamic algorithm can route tasks away from highly loaded processors, and can often deal with overloaded network segments and even failed workstations [13].

A special type of dynamic algorithm is the *adaptive dynamic* algorithm. This type of algorithm not only monitors system state, but also modifies its underlying placement algorithm based on this state. For example, an adaptive dynamic algorithm might switch to a static type placement policy when system usage is below a certain threshold.

For dynamic algorithms, task transfer may be nonpreemptive or preemptive. A *nonpreemptive* task transfer policy is one in which a task must execute to completion at the processor on which it is started. A *preemptive* task transfer policy allows an executing task to be stopped and moved to a new processor, without losing work completed. Usually, the overhead associated with preemptive transfers is large, since the state of the transferred task must be saved, sent over the network to the destination machine, and then rebuilt on that machine. An additional complication arises when transferring one of a group of communicating processes; the other members of the group must be able to locate the moved process, or their messages must be forwarded by the OS. Preemptive transfers will be examined in more detail in Section 3.2.2.

3.2.1.2 Components of a load leveling algorithm

Any load leveling algorithm may be divided into four parts which work together during scheduling [48]: the transfer policy, the selection policy, the location policy, and the information policy. During a load leveling operation, these policies determine when a task will be transferred, where the task is taken from, where the task is transferred to, and what system information is considered when making the above decisions. Note that scheduling with load leveling typically involves two steps: initial placement of tasks, and redistribution of tasks when a processor becomes over- or under-loaded. The load leveling algorithm operates during the second of these steps; the first step is the duty of the scheduling algorithm.

The first component, the *transfer policy*, determines when a given processor will participate in a task transfer. The policy is activated when a state change occurs that causes the processor to become over- or under-loaded. Typically, such a change follows the arrival of a new process or the completion of a running process. At that time, the transfer policy examines the current load and decides if a transfer, either into or out of the processor, should take place.

Commonly, one or more threshold values are consulted to make this determination. Such a policy is called a *threshold* policy. The threshold value may be a constant, set by a system administrator, or it may be a fluctuating value maintained by the scheduler itself. Usually two thresholds are used, one to define over-loading, one to define under-loading, with a dead space in between.

The second component, the *selection policy*, chooses the task to be transferred once the transfer policy determines that a transfer must take place. This selected task may be a task waiting in the processor queue, as in nonpreemptive load leveling, or a currently executing task, as in preemptive load leveling. In addition, the selection policy may consider other factors when choosing a task to move, such as specialized hardware requirements of the task, or communication/data locality considerations (see Section 3.2.3). If no eligible task is located, the transfer is terminated.

The third component, the *location policy*, finds a partner processor for the transfer. If the transfer policy has chosen an over-loaded processor, the location policy should find an under-loaded processor, and vice versa. Polling is commonly used to locate the partner in decentralized load leveling schemes; in such systems, the other processors may be polled randomly, or in some set order, or simultaneously through a broadcast. On the other hand, a centralized load leveler will have the necessary information on hand, an can thus make a decision immediately. In the latter case, the processor making the decision is called the *coordinator*.

The last component, the one on which the bulk of this research concentrates, is the *information policy*. It is arguably the most important of the four components. The information policy determines what data is gathered by the load leveler, how often it is updated, and how it is used to make scheduling decisions. A given information policy may be classified as one of three types: demand-driven, periodic, and state-change-driven.

In a *demand-driven* policy, information is amassed only when the transfer policy determines that a task transfer should take place. If the processor that triggered the transfer is over-loaded, the policy is said to be *sender-initiated*. If the processor that triggered the transfer is under-loaded, the policy is *receiver-initiated*. A third type of demand-driven policy, the *symmetrically-driven* policy, may be activated by either an over-loaded or an under-loaded processor.

With a *periodic* policy, information is collected according to a regular schedule. Such a policy may be centralized, with a single processor amassing data through polling of all other processors, or decentralized, with all processors collecting local information. Notice that a periodic policy, if not implemented carefully, could generate significant overhead. Polling rates and information exchange between processors must be adjusted carefully to avoid overloading communication channels or bogging down the processors themselves. This is particularly true of workstation networks, which have no special channels dedicated to scheduling information exchange, and whose communication costs are often high. Possible solutions for these networks are to collect information only within small groups of machines, or within subnets, and to vary the collection rate based on current network loading levels.

The last type of information policy, the *state-change-driven* policy, operates when a major change has occurred in the system. Each processor that has undergone a significant state-change then broadcasts its new state. What qualifies as a "significant" change must be determined experimentally. One example might be a processor that has emptied its task queue. As in periodic policies, state-change-policies must be designed carefully to avoid thrashing or excess network traffic.

In all three cases, the most critical decisions in designing an information policy are 1) what system statistics are relevant, and 2) how often should the statistic database be updated. Monitoring the wrong statistics, obviously, will lead to poor transfer decisions. Even given the best statistics, updating too often will *generate* excess traffic in the system and may lead to thrashing, while updating too seldom could force decisions based on stale data. The information policy will be considered in greater detail in Chapter 8.

3.2.1.3 Importance of load leveling

As mentioned above, load leveling algorithms correct flaws in task distributions generated by the scheduler. One might be tempted to think, then, that a good scheduler obviates the need for load leveling. In theory, this is true; in practice, design of such a scheduler, at least for workstation networks, is impossible. There are two primary reasons for this impossibility. The first reason: except for the case of dedicated workstation farms, the scheduler for a system such as PVM does not have control over all processes entering the network, but only PVM jobs;

users may login and start other types of programs at any time, unbalancing the workload. The second reason: a perfect or near perfect schedule necessitates certain a priori knowledge about the jobs to be scheduled, such as runtime, communication rate, and in-memory size; this information simply is not available in realistic systems, be they workstation networks or standard multiprocessors. Thus, a load leveler can be an important part of the scheduler on these types of systems, since it can compensate, to some extent, for these scheduling deficiencies.

As a final note, many workers seem to feel that load balancing or load sharing is a universal first step toward improved performance. There are cases, however, when locality problems will cause performance drops when loads are leveled (see Section 3.2.3). One such case is a workload including frequent interprocess communications; as the processes are spread out among processors, numerous process message exchanges lead to network congestion and thus performance degradation [21][40].

3.2.2 Preemptive task transfer

An important choice to be made when designing a scheduling algorithm is whether to support preemptive task transfer, often called simply "task migration." As mentioned in the context of load leveling algorithms, a *preemptive* task transfer is one in which a currently executing task is stopped, moved to a new processor, and restarted. Generally, it is assumed that some or all of the work performed by the processes before the transfer is preserved. In other words, simply killing a process on one machine and reexecuting it on another is not considered a preemptive transfer.

Preemptive task transfer is important in many systems, especially those based upon workstation networks, because it allows a poor load distribution to be corrected. Consider a PVM parallel program with processes running on five workstations, when a user logs into one of the workstations and begins a computationally intensive operation. The PVM scheduler, even if heavily optimized, has no way of knowing in advance if a user is going to log into a workstation sometime in the future (although time-of-day and day-of-week usage patterns may provide a clue [31]. Because a parallel program executes only as fast as its slowest part, the resulting heavy load at one workstation will delay completion of the entire program. The solution is a load

leveling algorithm that can detect the rise in activity at a given machine and preemptively migrate its PVM processes to other, less busy processors.

Unfortunately, a preemptive migration is often quite costly. A running task must be stopped, its memory image and CPU registers saved, and the resulting data file transferred to another processor and rebuilt. In addition, any incoming messages must be rerouted to the new processor, and open files must be likewise transferred. On a dedicated multiprocessor machine, some of this cost may be eased by special hardware and system software support. However, on a network of workstations running UNIX, the transfer will be quite difficult.

Experiments have been performed in an attempt to determine if task migration provides enough benefit to offset its costs. Sprite [17], Charlotte [2], Amoeba [42], and Condor [37] (see Section 4.3) all support migration, and their designers and users claim reasonable performance. However, a study by Eager et al. [19] found that the benefits of task migration in load sharing are minimal, with only slight improvements over far simpler, nonpreemptive algorithms. Eager concluded that task migration could offer, at best, modest improvements in performance for systems under extreme conditions. On the other hand, performance gains due to migration will vary greatly with the program's execution environment, and Eager's "extreme conditions" may take on a variety of meanings. Within a workstation network, best suited for long-running, computationally intensive tasks, a process might run for hours or even days. If even a 20% gain in execution speed might be had by moving that process to a different machine, the time saved overall could be substantial. In such a scenario, even a migration cost of several minutes would be negligible.

Another benefit of migration, in the context of workstation networks, is its support for the "good neighbor policy." This is one of the primary reasons stated for implementing migration within the Condor system [7]. The *good neighbor policy* states that running tasks will be evicted from a workstation when that workstation's owner returns. Task migration mechanisms provide the ability to move these tasks to other machines without losing work completed; only the capability of activity detection at the workstation console is required to complete the policy.

Several software products have been developed which support preemptive task transfer on workstation networks. These include the Charlotte distributed operating system and the Condor batch processor (see also Section 4.3). The latter is especially notable as it implements

migration between workstations running the UNIX OS without assistance from the UNIX kernel [38]. This disproves the popular notion that task migration requires kernel support. On the other hand, not all processes are eligible for migration under Condor; existence of child processes or files open for both reading and writing prevent a task from being moved. Charlotte can migrate any process, but only with kernel support.

3.2.3 Locality management

A scheduling factor tied closely to load leveling is locality management. It comes in two forms: data locality and communication locality. In *data locality management*, processes are scheduled so that they are physically close to the data they access; a process opening a file would be placed on the machine whose local disk contains the file. In *communication locality management*, A process is scheduled so that it is physically close to other processes with which it plans to communicate. Notice that, unlike other scheduling considerations discussed so far, locality management requires some information about process behavior. In a system which does not support preemptive task transfer, taking advantage of locality would only be possible if submitted jobs included information about their communication and data requirements. A system with preemptive transfer, on the other hand, could monitor running jobs and then move them once enough information was collected.

Locality management might be thought of as network load leveling, as opposed to CPU load leveling discussed earlier; communication is spread out over the system, localized within each machine. By grouping communicating processes together and data-accessing processes with their data, use of the shared network resource is reduced significantly. At the same time, CPU load leveling is frequently compromised, as processes are bunched together on a few machines. Thus, locality management and CPU load leveling tend to oppose each other, and for best performance, some balance must be found between them [40]. For example, grouping five heavily communicating processes together on a single machine will drastically reduce the communication latency between the processes, but it may also overload the machine, killing CPU performance. Finding a compromise between load leveling and locality considerations requires determining at which point the CPU loading due to the clustering of processes balances the network loading due to the distribution of processes.

A number of workers have examined locality management on distributed systems [40][41][30][10]. Solutions generally take on one of three forms: migrate processes toward data, migrate data toward processes, or allow the programmer to specify a data-process match function. In the latter case, special constructs are typically added to a parallel language to allow the programmer to specify data/process distributions, either through directives or hints to the compiler. This programmer-dependent solution takes advantage of the programmer's knowledge of program behavior, which is not available to the scheduler. However, compiler directives or hints place an often unwanted burden on the programmer, and tend to reduce the portability of the code.

In the past, research has concentrated on load leveling at the expense of locality. In [40], it is claimed that locality management is actually the more important factor in scheduling, and that CPU load leveling should be considered secondary. Whether this is true for a given system depends heavily on the system itself; a fast network connecting slow CPUs favors load leveling, while the reverse favors locality management. Thus, the best solution will remain technology dependent, and must be decided on a case-by-case basis.

3.3 Scheduling Algorithms

It is instructive to examine some representative scheduling algorithms, in order to highlight their similarities and differences. A number of algorithms described in the literature are discussed briefly below. Notice, in particular, the assumptions made by each of the algorithms; in some cases the assumptions are irreconcilable with a real-world system. Other algorithms are suitable for tightly-coupled, dedicated multiprocessors, but fail when applied to a workstation network multiprocessor as described previously.

The algorithms described below may be loosely grouped into four categories: list schedulers, queuing schedulers, module-clustering schedulers, and threshold schedulers. Note that the category into which a given algorithm is placed is somewhat subjective, and some algorithms may fit equally well into multiple categories. The grouping used here was chosen based on the usage in the referenced papers. At the end of the section, some additional algorithms that do not fit comfortably into any of the above categories are briefly detailed.

3.3.1 List schedulers

One of the most common types of scheduling policy is the list scheduler [22]. A list scheduler assigns each waiting task a priority. When a processor becomes available, it is given the waiting task with the highest priority. The primary difference between list scheduling algorithms is the manner in which the priorities are assigned. Thus, a list scheduling policy is defined primarily by its selection policy.

A number of priority assignment schemes are described in [22]. One technique is to represent the program by an acyclic directed graph called a task graph. The task graph highlights dependencies between the tasks, and thus simplifies the derivation of task execution order requirements. Priorities may then be assigned based on the level in the graph at which a task appears, and on its number of successors; tasks near the bottom of the graph with the fewest successors are given high priorities. Another graph-based scheme assigns priority based on the longest path from a given task to a task with no successors. The path length may be calculated with or without considering communication costs. A third graph-based list scheduler, described by Lewis [35], uses the critical path through the graph to prioritize tasks.

For the most part, the above algorithms ignore communication costs between tasks. These costs may be incorporated into the placement policy. Instead of simply sending the highest priority task to the next available processor, the scheduler may attempt to group communicating tasks together on the same processor, if one task is the successor of the other. Alternatively, communicating tasks may be duplicated on each participating processor. Another way to reduce the effect of communication cost overall is to schedule additional tasks into delay slots caused by communicating tasks that must wait for an incoming message; in this way, the processor is not idle during the delay. Of course, all the above modifications require a priori information about the communication behavior of the tasks in question, information that is often not available.

3.3.2 Queuing schedulers

Although most schedulers maintain a queue of some sort, the class of "queuing" schedulers contains those schedulers which distribute tasks based on arrival/service rates/probabilities derived from queuing theory. These systems generally maintain one or more

arrival queues and one or more departure queues. Tasks are assumed to arrive at a specified average rate and be serviced at each server at a rate determined by the server. For a given server, the calculated service rate may be considered fixed, or may be based on fluctuating system parameters.

Two queuing schedulers with variations are described by Chow [13]. The first scheduler is non-deterministic; an arriving task is routed to a server processor based entirely upon a set of probabilities, with the sum of the probabilities for all servers equaling 1. The policy is non-deterministic in the sense that the target processor is chosen without regard to the properties of the arriving task. On the other hand, the probability that a task is sent to any given server processor may be based on processor state information, such as loading. Thus, this non-deterministic scheduler fits the workstation network multiprocessor model well.

The other scheduler is a deterministic scheduler, which routes tasks to servers based on a criterion function in addition to predetermined probabilities. The criterion function is calculated using statistics related to both the currently running and target tasks. Three criterion functions are described. The first uses a minimum response time policy, in which the job is sent to the server with the smallest expected turnaround time for the target application; calculating such a time requires knowledge of the current load on the server and the expected execution load caused by the target task. The second uses a minimum system time policy, in which the job is sent to the server which will complete all submitted jobs, including the current target job, in the minimum total time; to calculate total service time for all jobs, information regarding the behavior of all involved tasks must be known in advance. The third criterion function is calculated using a maximum total throughput policy, sending an arriving task to a server so that the total system throughput is maximized until the next task arrives; as with the first criterion function, this calculation requires knowledge of current loads and target application behavior.

A number of queuing schedulers which, for the most part, ignore both system statistics and task behavior, are listed by Wang [56]. These schedulers model the system as containing two groups of processors, sources and servers, both of which may have queues. A task is generated at a source processor and serviced at a server processor. Note that a given processor may serve as both a source and a server. Since these algorithms assume that all tasks are unrelated, they are best suited for parallel jobs with little or no interprocess communication.

A total of ten algorithms are given in the paper. All are static, in the sense that their placement decisions ignore system load information. Brief descriptions of the ten algorithms follow. (1) Source processors are grouped, and each group is served by a single server processor. (2) Server processors are grouped, and each group serves one source; tasks are distributed evenly among the servers in the group. (3) Tasks from each source are randomly distributed among all servers. (4) Each server takes a task from a randomly selected source. (5) Sources give tasks to all servers, selecting the individual servers in a round robin fashion. (6) Servers take tasks from all sources, selecting individual sources in a round robin fashion. (7) When a source has a task, it selects the server with the shortest task queue. (8) An available server takes its next task from the source with the longest queue. (9) Jobs are processed by the servers in the order in which they arrive from the sources. (10) Servers select the shortest tasks first. Note that the last policy requires a priori information about each task execution length.

3.3.3 Module-clustering schedulers

A module fusing, or clustering, scheduler makes placement decisions based on expected interprocess communication. Obviously, such a scheduler needs advance information about task behavior, information that is often not available and/or is unpredictable. In simplest terms, a module clustering scheduler tries to group tasks that communicate frequently onto one processor, so that the communication is eliminated.

In [21], three variations of the module-clustering scheduler are given. In the first algorithm, all possible task pairs are examined, and the one with the highest communication load is considered for clustering. If these tasks can be combined, they become a single task and the algorithm continues. When no more combinable task pairs are found, the remaining tasks are sent to the available processors in a round robin fashion.

The second variation is similar to the first, except that a distance value is calculated between every possible pair of tasks. The distance between two tasks is based upon the ratio of the communication volume within the task pair to the communication volume between the pair and all other tasks. Task pairs with a large ratio value are considered "close" and are eligible for clustering. Again, the process continues until no more task pairs may be fused.

The last algorithm is a modification of the first. One disadvantage to the first algorithm is that the number of tasks produced does not necessarily correspond to the number of available processors. The third algorithm solves this weakness by iterating through multiple passes of the clustering procedure. Tasks are fist clustered to minimize communication between clusters. The clusters are then distributed to available processors and the resulting processor loads examined. If processors are under or over-loaded, the clusters are reformulated to smooth load irregularities.

3.3.4 Threshold schedulers

Threshold scheduling algorithms distribute tasks naively and then use preemptive migration to balance the resulting load distribution. As implied by the name, threshold schedulers use a threshold policy to determine when a task must be moved. During a scheduler pass, the number of tasks on each processor is compared against a low- and high-water mark. If the number is below the low-water mark, the node is considered under-loaded. If the number is above the high-water mark, the node is considered over-loaded. The processor is then examined as a source or destination, respectively, of a migration event.

Three threshold algorithms, which differ in the location policy used to place a migrating task, are compared by Eager, et. al. [18]. In the first algorithm, the target processor is chosen at random, without regard to its current task load. In the second algorithm, a target candidate processor is chosen at random, but the task is only transferred if it will not put the target over its high-water mark; if the transfer would overload the target, a new target is randomly chosen. In the third algorithm, a subset of processors is chosen at random, and the task is sent to the processor in the subset which has the smallest task queue.

It is interesting to note that, when the performance of these three algorithms was compared, it was determined that algorithm three offers very little improvement over algorithm two, even though it attempts to choose the least loaded processor within some subset of processors, while algorithm two simply avoids the most heavily loaded processors.

3.3.5 Miscellaneous algorithms

Several schedulers which do not fit comfortably into any of the above categories are given by [56]. They are called Sequential, Diffusion, and Contract Bidding schedulers.

A sequential scheduler actually consists of a group of independent schedulers, one for each source processor, which act sequentially. At any given moment, the acting scheduler is determined by the possession of a token. The token-bearing scheduler then looks for processing resources on behalf of its processor. When it is finished, the token is passed to the next scheduler. The use of a token ensures that two schedulers will not place tasks on a processor simultaneously, thus possibly overloading it. Each individual scheduler may use one of the algorithms described previously.

A diffusion scheduler is a distributed scheduler which operates within a small neighborhood surrounding every source node. Tasks are preemptively migrated away from loaded processors, thus smoothing the task distribution over time. This is the type of scheduler used on the Rediflow system, described in [36].

In a contract bidding scheduler, a processor with a task to process requests bids from possible server processors. The servers respond with bids for work based on their current load and their ability to process the given task. The source processor then picks the server with the most attractive bid. A variation of this system allows idle servers to request bids for work from source processors.

3.4 Scheduling On Workstation Networks

Scheduling for general purpose workstation network multiprocessors is a special case of general scheduling, with the following constraints: (1) network use is costly, and so should be minimized, (2) workstation users should not be disturbed by background jobs, and (3) load may be introduced externally to the system, and so must be measured in some way. The last of these constraints has been the least studied, and is thus the focus of the work described in this document. Many experiments have been performed wherein processes are placed on workstations based on some local load measure, some of which are described in Chapter 4.

However, little analysis of the value of these load measure in reflecting actual machine load have been made. Put another way, one must ask, how useful is this measure in predicting the performance of a given task run on this machine? The work described here attempts to begin a quantitative examination of loading data and program execution time, to determine whether certain load measures are related to runtime of an application, and, if so, how?

4. SOFTWARE FOR WORKSTATION MULTIPROCESSING

To use the idle cycles on a workstation network for computation, some additional software must be added to the operating system to handle the mechanics of launching remote processes and coordinating communication between them. This chapter describes one such package, PVM which is being used in this research, along with some third party extensions to PVM. The chapter also covers briefly a number of similar packages which display one or more desirable features that PVM lacks.

4.1 Parallel Virtual Machine (PVM)

Parallel Virtual Machine, or PVM, is a popular software package designed to allow exploitation of networks of workstations for parallel processing [27][16][26][24][25]. It was developed jointly at Oak Ridge National Laboratory, University of Tennessee, and Emory University, and is currently available free through anonymous ftp. Because it is easy to use and its source code is readily available, PVM has become one of the most popular workstation parallel processing packages among research institutions.

PVM implements a message-passing distributed memory parallel-processing model. Programs use PVM subroutines to spawn remote processes, pass messages, and perform synchronization. The system supports heterogeneous workstation networks, allowing programs to take advantage of special purpose architectures, such as parallel processors and supercomputers, in addition to ordinary workstations. All process control is handled through a console process on the initiating workstation. In the latest version of the software, this console has an X-window interface which graphically displays program progress and message passing. In addition, an add-on package, called HeNCE (Heterogeneous Network Computing Environment) is available which extends the PVM environment with graphical program construction and debugging tools [5][6][3][4].

The PVM system consists of two parts: the OS kernel and the subroutine library. The OS kernel handles PVM program execution, starting processes, sending and receiving messages, and

interacting with the user through the console. The subroutine library contains C and FORTRAN routines which allow the programmer to access the PVM system. When developing a PVM program, the programmer codes using the PVM subroutines to implement a coarse-grained, task-level parallelism; frequently, a master-slave architecture is used, with separate master and slave program modules. The program modules are then compiled normally and linked with the PVM library. To execute the program, the user first starts the PVM console process, then adds each remote machine to the system. The master program module can then be run from the shell command line; the PVM OS locates the necessary slave object modules and starts them on the appropriate machines. Currently, remote machines are chosen for execution in a simple, round-robin fashion.

The set of participating workstations in a given program run is called the PVM *configuration*. Machines may be added to and removed from the configuration dynamically by an executing PVM application. This capability allows the programmer to build simple load balancing algorithms into parallel applications, a desirable quality since the PVM OS contains no scheduling optimization or load balancing code. Lately, other workers have added scheduling and load balancing code to the PVM OS kernel itself [8]. Work is also ongoing in the area of task migration and checkpointing, two other capabilities that native PVM lacks. Checkpointing, in particular, is important, as, currently, a workstation crash will lock up a PVM application unless it is carefully coded, causing all work completed before the crash to be lost.

One problem faced by PVM users is objection by workstation owners to the use of their machines because of performance degradation while parallel background tasks are running. When a dedicated workstation farm is available, of course, such problems do not arise, but many installations do not have the resources to dedicate a group of workstations entirely to parallel processing. Unfortunately, native PVM does not support the "good neighbor" policy implemented by systems such as CONDOR [7] and Sprite [17]. In such a policy, a background task executing on a workstation is terminated, frozen, or moved whenever the workstation console is active; processing only occurs while the workstation is "idle." In the case of CONDOR, program progress is regularly checkpointed; when workstation console activity is detected, the current running task is immediately killed. The latest checkpoint file then allows

the program to continue on a new workstation, or on the current workstation if it becomes idle again. This capability has been added to PVM recently by outside researchers [8].

Obtaining good performance from PVM requires that, (1) a program is suitable for the system, and, (2) a configuration is used that contains relatively idle workstations. Typically, workstation networks are best suited for long-running, computation-intensive programs which perform little or no inter-process communication. As one would logically expect, these programs perform best on lightly utilized workstations connected by quiet networks. The terms long-running, computation-intensive, lightly utilized, and quiet are seldom quantitatively defined, however, in literature discussing workstation parallel processing. One of the aims of this work is to better define the qualities that make a program suitable for PVM, and to examine the load measuring techniques and resulting values that indicate the suitability of a given workstation.

4.2 Schedulers for PVM

As has been mentioned several times, obtaining good performance during parallel processing on non-dedicated workstation networks requires an intelligent scheduler, something that PVM lacks. In its current implementation, scheduling is performed in a round robin fashion, using the list of hosts in the configuration. No attempt is made by PVM to check load factors on the various machines, nor to place or move tasks so that interference with other workstation users is minimized. Still, PVM has become popular among researchers, probably because it is easy to use, easy to install, and costs nothing. Therefore, there has, of late, been a flurry of work in the area of PVM scheduling improvements.

Like most software packages in existence for more than a few years, PVM has gone through a number of changes since the initial, developer's internal version. During its evolution, a number of features have been added or improved, while many have been dropped. For example, the original, pre-release version of PVM had a number of experimental features that are not found in the current incarnation [27]. These features include fault tolerance, shared memory support, post-mortem debugging, and, most importantly, dynamic load balancing. In response to inquiries regarding the removal of these useful features, the PVM developers explained that they proved unstable.

Within the last two years, three PVM schedulers have become available. Each of them attempts to improve performance by adding intelligence to the scheduling algorithm used in PVM. The first, DQS, and second, QM, are both queue managers which accept PVM tasks and then find machines for execution. The third is actually a trio of modified PVM versions, called MPVM, UPVM, and ADM, which support load balancing in various forms. In addition, a scheduler is currently under development which uses the load statistics returned by the Utopia distributed OS.

4.2.1 DQS

DQS implements a queue to which users submit jobs along with execution requirements, such as machine architecture and minimum desired computing power [28]. The requests may be hard or soft; a *hard* request must be met before the submitted program is executed, while a *soft* request need only be met if resources are available. DQS then submits the jobs to computers in the network, choosing machines based on requests and availability. The overall goal is to maximize resource usage.

DQS contains support for multiple-process PVM programs, and is thus able to serve as a scheduler for PVM [45]. The user submits the job to DQS in the normal manner, but sets a flag which identifies the job as a PVM program. DQS automatically sets up the PVM environment and farms the tasks out to available workstations. Unfortunately, DQS has no facility for monitoring load on machines, other than the number of DQS jobs in each queue. Also, DQS cannot move a job off a machine once it is started, though a modified version called DNQS allows users to set the availability status of individual workstations [55].

4.2.2 QM

QM is a queue manager for PVM developed at the University of Tennessee, Knoxville, which attempts to address some of the shortcomings of DQS [46]. When choosing a target workstation for a given task, QM uses two pieces of information about the workstation: inherent speed and current load. The inherent speed is provided by the user in the form of a benchmark rating whose value rises with machine speed; the actual benchmark used is immaterial, as long as it runs on all involved machines, though a benchmark similar to the target PVM application, if

available, is recommended by QM's author. The current load value is obtained by QM through the UNIX *uptime* command; it is simply the 1 minute load average calculated by the UNIX kernel.

When choosing a machine for execution of a job, QM uses load average, benchmark rating, and number of currently running PVM jobs to rank each available machine in order of available computing power. The *available computing power* of a given machine is calculated as the ratio of total load to speed, where total load is the sum of the load average and the number of running PVM jobs, and speed is the user-supplied benchmark rating. The resulting metric will drop with rising computing power. Notice that, since PVM jobs are themselves user-level UNIX processes and are thus reflected in the load average value, they are given extra weight in this calculation. No explanation for this is given in the QM documentation. Also, no data are given relating available computing power on a given machine to application program performance, though comparisons between runtimes with and without QM for several applications are given, with QM demonstrating improved performance over straight PVM.

4.2.3 MPVM, UPVM, and ADM

Three migrateable versions of PVM have been developed at the Oregon Graduate Institute of Science & Technology, called MPVM, UPVM and ADM. Each supports process migration for load redistribution and obtrusiveness avoidance, each through different implementation philosophies.

MPVM: MPVM allows transparent preemptive migration of standard PVM processes. Migration events are initiated by a global scheduler which has dominion over the entire configuration. In MPVM, the migrateable entities are standard UNIX processes; like Condor, MPVM saves a running program's state space so that it may be reconstructed on another machine. Only two restrictions are imposed: a process cannot be migrated while it is executing in the PVM runtime library, and a process cannot migrate to a machine with an architecture differing from that of the machine on which the process was started.

UPVM: Like MPVM, UPVM supports transparent migration of running PVM processes. In UPVM, however, a single PVM process is subdivided into multiple thread-like entities which may be migrated independently. This multi-threading allows for a finer resolution during load balancing, and thus a better balance. From a user perspective, a thread, or ULP (user level process) is similar to a process; ULPs exchange data only through message passing. However, message passing between ULPs belonging to the same process is optimized and thus less costly than message passing between separate processes. Migration of ULPs is also controlled by a global scheduling entity.

ADM: The ADM approach to migration is data driven rather than process driven, sacrificing transparency for speed. During a migration event, data shuffling occurs on a global scale, with all active processes participating. Unfortunately, the migration strategy must be determined during coding, and leads to a rather complex finite-state machine structure within each ADM program. Preliminary results indicate that the complexity of ADM programs grows rapidly with program size, placing a large burden on the programmer.

4.2.4 Utopia

The need to crunch numbers for large astrophysics Monte Carlo problems led to the development of a PVM scheduler at the University of Toronto [58]. This scheduler runs under the Utopia distributed OS and uses load statistics accumulated by Utopia to choose target machines for PVM applications. Specifically, a machine is considered viable if two conditions are met: (1) the load level on the machine is below some threshold, and (2) no other PVM process of the same name is currently running. Among the viable machines, the one with the lowest load is chosen. This scheduler was designed to steal idle cycles from a large, undergraduate computing lab, and the developers report parallel speedups of 45 using 60 machines, with few complaints from workstation users. Utopia is discussed in greater detail in Section 4.3.1.

4.3 Other Software Packages

There are currently available a large number of software packages that allow use of "idle" time on workstation networks. These can be broadly divided into three classifications: batch systems, distributed operating systems, and programming toolkits or languages. What they all have in common is the ability to execute programs on remote workstations connected by a LAN. Some representative packages are described below, and many more are listed in [55]. Note that most of these packages will fit into more than one category, and thus the groupings below are somewhat artificial. For example, PVM can be considered both a distributed OS and a programming toolkit.

4.3.1 Batch processors

Batch processing workstation network packages allow the user to submit a list of programs to execute on machines in the network. The batch processor then determines the best machine for the job, starts the program on that machine, and collects results when it finishes. Some batch systems also check for idleness and move or stop programs when a workstation's owner returns. Condor is probably the best known of these packages, and is representative of the group. Other batch systems include Utopia and DQS.

4.3.1.1 Condor

Condor was developed at the University of Wisconsin, and provides for the unobtrusive use of idle workstation cycles to run single-process application programs [37][38]. Jobs are submitted to a queue. Condor monitors the available workstations, and when it finds one that is "idle," the next job in the queue is sent to that workstation for execution. In this case, idleness is determined by keyboard and mouse activity; if the mouse and keyboard are inactive for a certain period of time, the workstation is considered idle.

While a Condor job is running, it is periodically checkpointed. The checkpointing process includes saving a copy of the program's local memory, register contents, and open file information. Condor uses this information to build an executable file which, when run, will start the checkpointed job from the last checkpoint, as if it never stopped. Within Condor,

checkpointing is used for two purposes. One, to provide immunity to workstation failures; when a workstation goes down, the job can be continued from the last checkpoint with minimal lost work. Two, to implement the "good neighbor" policy; when workstation keyboard or mouse activity is detected, the job is killed and restarted in another location using the last checkpoint file. Unfortunately, not all programs can be checkpointed. Programs that consist of multiple processes, as through a *fork* or *exec* call, and programs that read and write to the same file will not work under Condor.

4.3.1.2 Utopia

The Utopia load sharing facility, also called LSF, was developed jointly by the University of Toronto and Platform Computing Corporation [61]. It provides invisible, remote execution of programs on large, heterogeneous workstation networks without program modification. Utopia supports both standard jobs and parallel programs written with a programming toolkit, such as PVM or Linda (see Section 4.3.3).

Utopia divides the network into clusters, providing centralized control within each cluster and distributed control across clusters. A cluster may contain machines of differing architectures, and Utopia supports the use of special machines, such as parallel processors and supercomputers. However, the filespace must be distributed; Utopia assumes an identical filesystem on every machine.

A number of statistics, such as CPU queue length, free memory, number of logins, and swap space, are maintained by the Utopia OS kernel. When a job is submitted for execution, these statistics are used to locate a suitable target machine. Utopia does not support task migration, however, so once a job is placed, it must complete execution on that machine. Also, no checkpointing is provided; Utopia is thus vulnerable to workstation crashes.

4.3.1.3 DQS

Developed at Florida State University, DQS (Distributed Queuing System) accepts jobs from any workstation on a network, queues the jobs, then executes them on other available workstations. Jobs may be standard programs or parallel PVM applications. DQS supports heterogeneous networks, allowing users to specify resource requirements when submitting jobs.

DQS then schedules jobs in one of two ways: either in the submission order, or in the order that resource requests can be satisfied.

DQS views the workstation network in terms of two resources, queues and groups. A queue is associated with a specific machine, and operates in a FIFO manner; a job submitted to a queue waits until the associated machine is free for execution. A group represents a set of queues on machines of a single architecture. When a job is submitted to a group, it is sent to the first queue that becomes available. This strategy thus implements a simple load leveling policy.

A similar system, called DNQS (Distributed Network Queuing System) was also developed at Florida State and later rewritten at McGill University. It specifically addresses security problems, and includes provisions for workstation owners to add or remove their computers from the computer pool. The McGill version removes some technical limitations in the original, FSU, version.

4.3.2 Distributed operating systems

Most of the software packages providing support for processing on networks of workstations can be considered operating systems, in the sense that they form an interface between the user and the hardware, allowing submission of jobs, monitoring of progress, and return of results. However, a full-featured operating system, such as UNIX, provides a number of additional services, including a file system, access to I/O peripherals, and a programming interface to OS services. The distributed operating systems described in this section are full-featured operating systems whose underlying hardware is a network of workstations rather than a single machine. Additionally, these systems provide facilities for parallel processing across the workstations in the network.

4.3.2.1 Clouds

Clouds is an object-oriented operating system running on networks of Sun-3 workstations [15]. Under Clouds, workstation resources are divided into three categories: computation servers, which perform processing; data servers, which store data objects; and user workstations, which provide an interface between users and the OS. However, the categories are

not mutually exclusive; a given workstation may function as both a computation and a data server.

All user programs, system services, file storage, and I/O are viewed as objects, a unification of data and code. To run a program or request a system service, the OS invokes one of these objects. Execution then takes the form of a thread, decoupled from the objects themselves, which traces a path of execution from one object to the next. Physically, objects are implemented as disk files, and are thus non-volatile. This implementation allows recovery from system crashes since an object's data area provides a snapshot of thread execution before the crash.

4.3.2.2 Charlotte

The Charlotte operating system, developed at the University of Wisconsin, was designed specifically for the purpose of experimentation with load balancing and migration policies (for details on load migration, see Section 3.2.2; for load balancing, see Section 3.2.1). It is a message-passing OS, designed to run on a group of VAX 11/750 computers connected by a token ring [2]. A user running programs on the system sees the collection of computers as a single machine. Where a submitted job actually executes is completely irrelevant and invisible to the user.

The necessity for process migration is determined through the collection of system statistics. Both the statistics collected and the collection policy itself are easily varied during experiments. Three types of statistics may be collected: machine load statistics, including number of processes, CPU load, network load, and number of communication links; individual process statistics, including age, CPU utilization, and communication rate; and machine-network link statistics, including number of packets sent and received. These statistics are updated periodically, with a tunable period. They are also updated after major system state changes.

The most significant aspect of Charlotte's design is the elimination of locality dependence in executing processes. Location independence is simplified somewhat by the homogeneous nature of the supported hardware. However, interprocess communication, necessary for parallel processing and fully supported by Charlotte, often significantly complicates process migration. Under Charlotte, either of two methods may be used to handle

message routing to moving processes: the sending process may search for the receiving process, or the sending process may send to a global message router, which locates the receiver. In either case, Charlotte ensures that communicating processes are frozen during the transfer so that a migration event does not occur in the midst of a message transfer. The method used, and many parameters involved in the process are all easily varied during experiments.

4.3.2.3 Amoeba

Amoeba is another object oriented distributed OS, developed at the Free University and Center for Mathematics and Computer Science in Amsterdam [42]. Like Clouds, it divides hardware resources into categories, including pool processors, which perform computation, workstations, which provide an interface to the user, and servers, which supply file storage and other specialized tasks. Amoeba is aimed primarily at workstation farms, in which groups of diskless, terminaless computers serve solely as processors for the system pool.

Like the other distributed OSs discussed, Amoeba hides the topology of the system from the user; the user sees a single, unified computer. To this system the user may submit single process jobs, or multiple-process parallel jobs. In the later case, each process may end up executing on a different processor in the pool. Amoeba handles communication between processors and user-process interaction so that the user and the processes need not know where a given process is actually running.

Process scheduling is performed under the assumption that there are always idle processors available in the pool. This assumption, that each user has available 10 to 100 free processors, is based on drops in hardware prices during recent years. Thus, the scheduler simply looks for the next free processor, and sends the current process there. If all processors are busy, the scheduler begins a queue for each processor. This round-robin scheduling technique is similar to that used by PVM. However, in a PVM system, workstations are not dedicated to the system, and thus may be running unrelated programs that are out of the PVM kernel's domain of control. In a workstation farm environment, such as Amoeba, this problem does not exist.

4.3.2.4 Sprite

Like Charlotte, Sprite is designed to provide transparent process migration and to support experimentation with load balancing policies [17]. The Sprite user sees a system similar to 4.3 BSD UNIX. However, Sprite unifies a nondedicated group of workstations connected by a network into a single logical processor. A program executed under Sprite appears to run on the machine at which it was started. In reality, however, it moves to the least busy workstation and executes there.

Process migration occurs automatically under two conditions: when a workstation's owner returns (when console activity is detected), or when a UNIX exec system call is made (when a new process is started). Beyond these cases, a user may cause a process to be migrated from one machine to another through a system call. The Sprite kernel provides a list of currently idle workstations, on which a transfer may be based. Although such a policy must be implemented at the user process level, the policy can be tuned to an individual algorithm, with possibly many different policies in effect at any one time, as opposed to a single, system-wide policy.

4.3.3 Parallel languages and programming toolkits

This last category includes both completely new languages, designed for the soul purpose of parallel computing, and software libraries, providing subroutines and extensions that add parallel processing primitives to existing languages. PVM, a toolkit, has already been discussed. Many others are described briefly in [55]. Two systems are examined here: Linda, a parallel programming model added to existing languages such as C or FORTRAN, and COOL, a parallel language supporting a form of load balancing.

4.3.3.1 Linda

Originally developed at Yale University, Linda refers to a set of instructions added to an existing programming language to support use of a shared memory multiprocessor [57]. Network Linda extends this support to networks of workstations, though the shared-memory model is ill-

fitting to such systems. Linda hides details of the system architecture from the programmer, but imposes a somewhat unusual computation model on the program.

Parallelism is supported in Linda through the use of "tuple space," a conceptual shared memory that multiple processes use to exchange information. Linda adds to an existing language commands to put, take and read items from tuple space. Details of the physical implementation of tuple space are hidden from the program; tuple space may exist in actual shared memory, or it may be distributed across many separate computers. In either case, locating a requested item in tuple space is handled by Linda automatically. In a workstation network, this process could be lengthy, and thus network Linda programmers are advised to keep tuple space access to a minimum.

4.3.3.2 COOL

Though designed for standard multiprocessors, COOL is a language designed to express the type of task-level parallelism that maps well to workstation networks [10]. Data items in COOL are represented by objects which exist in memory and may migrate from processor to processor during program execution. A COOL program performs a form of load balancing by distributing data objects evenly among the processors, and executing processes on the processor nearest the required objects.

Since load balancing is performed at the data level, it is dependent on the programmer to provide an intelligent data distribution. In addition, the programmer supplies information, in the form of "affinity hints," to the system to help it locate processes. These hints allow the programmer to associate processes with specific objects, send processes to specific processors, or migrate objects between processes. Currently, only affinity hints are used by the runtime scheduler to locate tasks for execution. However, current research is examining the use of load statistics to locate idle processors and distribute tasks accordingly.

4.4 Current Research Directions

The above software packages demonstrate the variety of solutions so far proposed toward the problem of efficiently using idle workstation cycles. As may be apparent from the descriptions, most of these solutions are heuristic in nature. While several systems use machine

and network statistics to make process placement decisions, little quantitative justification for the choice of specific statistics, nor the way in which they are used, is given. An analysis of program behavior relative to several common load measures will serve to determine how useful the measures are, or whether they are useful at all. The work that follows attempts to perform such an analysis. Although a homogeneous, general purpose workstation network running PVM is used, the results are general enough to apply to most of the systems described. The load statistics chosen are those that may be easily monitored at the user process level, with no special hardware or OS kernel modifications. Thus, they should prove portable to most systems. In addition, the application program whose performance is monitored, though implemented as a PVM parallel program, runs as a collection of standard UNIX processes, and thus results will apply to any UNIX process, and to processes running under many other operating systems.

5. THE LOAD MODEL

During scheduling, some simplified model of the system must be used when predicting which target machine will provide the best performance while executing the current program. For this work, each machine on the network is modeled as a "CPU box" with three parameters characterizing processing performance: CPU speed, CPU load average, and CPU benchmark performance. Interconnecting these boxes is a complex of wires, one wire for each PVM configuration recognized by the scheduler, each wire with a network delay value associated with it. It is hypothesized that all system variables are reflected sufficiently within the parameters of this model to efficiently schedule program execution. Furthermore, factors not well represented within the above parameters are either unimportant for scheduling the target application type, or are unmonitorable within the software environment.

5.1 The "CPU box" Model

As stated above, each workstation on the network is modeled as a black box whose behavior is characterized by three parameters: the CPU speed, the CPU load average, and the CPU benchmark rating. No distinction is made between workstation architecture types; it is assumed that a program can be executed equally well on any participating machine. In general, such an omission does not disallow heterogeneity within the PVM configuration, as long as an executable file compiled for each type of machine is available to the scheduler. On the other hand, the model ignores special support for unusual architecture types, such as supercomputers and multiprocessors. Use of these machines is supported, even encouraged by the PVM developers [25], but is probably uncommon in most settings and is thus considered beyond the scope of this work.

5.1.1 CPU speed

The CPU speed is a relative performance measure of the processing power of a CPU, without respect to load. That is, the CPU speed is constant for a given type of processor, and is

roughly equivalent to the benchmark ratings currently used to compare computer performance. Of course, the speed measure discussed here is vastly simplified over standard benchmark programs, mostly to minimize loading of the system due to the benchmarking process. On the other hand, more accurate results might be had by increasing the complexity of the speed measurement process; since it need only be run once per machine, interference is not such an important issue. However, the accuracy of the method described here has proved adequate for the task.

The speed measurement is implemented by a floating-point vector-multiply operation. Floating-point operations were chosen because of the floating-point-intensive applications commonly targeted for execution on multiprocessors, particularly workstation network multiprocessors. An integer version of the measurement routine could be easily added, if desired. The speed rating is obtained by timing the multiply operation using the system CPU timer; the result obtained is the number of CPU clock ticks spent executing the code. Note that a clock tick may vary in size across different machines, and thus must be converted to some machine independent form. The inherent speed of the CPU, then, is inversely proportional to the number of clock ticks required to execute the code.

In practice, on the DECStation 2100, 3100 and 5000 models, speed results were consistent between runs, with a typical variation of +/- 4.3% on the 5000s, 2.9% on the 3100s, and 2.2% on the 2100s. In all cases, this variation amounted to half a clock tick out of 11 to 22 total ticks for the test. It is likely that increasing the execution time for the measurement code would yield even greater consistency, since the low to high value range in all cases was a single clock tick. As desired, system loading had no affect on the CPU speed calculations.

5.1.2 CPU "benchmark" rating

The CPU benchmark rating is one of two measures of CPU loading; the "benchmark" is used to measure the relative process load on the workstation. Thus, "benchmark" is a misleading term, since it normally implies a measure of inherent processor capacity, as described above. However, the term was used in this case because the code used to acquire this load measurement is almost identical to the benchmark code used to measure speed.

The benchmark load measurement code consists of the same floating-point vector-multiply code as described above. Since floating-point math is often processed by an FPU separate from the main CPU, it might be argued that such a test only measures the FPU processing load. However, since the CPU and FPU are always processing the same application, the load between the two is equivalent for these purposes. That is, either integer or floating point operations can be used to determine the relative size of the computational time slice given the program, and thus the relative total load on the machine.

The difference between the speed rating process and the benchmark load measurement lies in the timing. When measuring load, both the CPU and the wall clock timers are employed. The CPU clock gives a measure of the absolute amount of CPU time required for a task, while the wall clock timer gives that actual amount of time that elapses during execution of the task. Thus, if the processor has no other tasks, the CPU time will equal the elapsed wall clock time. If the CPU has two tasks, and splits its time evenly between them, then the elapsed wall clock time will equal twice the CPU time.

The actual load measure used is the ratio of wall time to CPU time. In theory, this ratio can never be less than one. In practice, the resolution of the timer is such that during low load periods, the recorded wall clock time may slightly exceed the CPU time. One may ask how the CPU and wall clock times can differ by fractions of a clock-tick; this is an artifact of the timing process. The wall clock timer has a finer resolution than the CPU timer, measuring in microseconds rather than clock ticks. When the microsecond timings are converted to clock ticks, fraction values usually result.

Thus, the accuracy of this load measure during periods of light loading is questionable. Increasing the size of the timed code block improves accuracy but increases obtrusiveness; the execution of the benchmark code itself causes an increase in system load. Since slight variations in loading on a lightly loaded system are not liable to greatly affect program execution time, the benchmark was kept small and used primarily to detect periods of high loading.

5.1.3 CPU load average

One of the most commonly used measures of system loading is the load average statistic, maintained by the UNIX kernel and returned by the *uptime* command. This statistic is calculated

as the average length of the ready-to-run queue, measured over one, ten, and fifteen minute intervals. Since the ready-to-run queue on a UNIX system contains only preempted jobs, the length of the queue gives an indication of how busy the system is; if many processes are waiting for CPU time, the system is obviously busy. Note that the ready-to-run queue does not contain jobs waiting for I/O (see below) such as shells or editors waiting for user input, telnet sessions waiting for incoming network packets, or sleeping daemon processes.

It is noted in [39] that the load average can be misleading for three reasons. One, the calculation of load average does not take into account process scheduling priority, set by the *nice* command; thus, a low priority process running in the background and consuming little CPU time is counted equally with a computation-intensive foreground job, even if the foreground task gets 90% of the available CPU time. Two, real-time applications under SV R4 are counted equally with regular tasks, even through real-time tasks consume a greater percentage of CPU time. And three, jobs waiting for disk I/O are considered ready-to-run, even if the disk I/O involves an NFS mounted disk and is thus occurring over a network. In the first two cases, the load average will be artificially low, even though the system might be quite busy and thus quite sluggish. In the third case, the load average will be artificially high, since jobs waiting for disk I/O are not actually consuming CPU resources, particularly if they are waiting for a slow network connection.

The one, ten, and fifteen minute load average values may be obtained in two ways: through the *uptime* command, or through internal kernel data structures. The first solution was used for the PVM load monitor program because to keep it portable; reading values from kernel data structures is system dependent, and usually requires special runtime permissions. Use of the *uptime* command, on the other hand, is somewhat crude and slow, but will work on any UNIX system.

5.2 The Network Model

The network model topology differs somewhat from the actual topology of the modeled network, due to the difficulties of characterizing load on a network. From a program's point of view, the network provides a direct path between the two communicating entities, and associated

with that path is an average delay value. This delay is the length of time required for a message to traverse the path, and it is an average because individual delay values may vary wildly from moment to moment. The variation is due to the bursty nature to network traffic, and to the collision resolution methods of the TCP/IP protocol suite [50]. For purposes of PVM program behavior analysis, the network is thus modeled as a series of discreet connections between groups of machines; each connection corresponds to a distinct PVM configuration. Thus, for a group of four machines, eleven differing network connections are possible, one for each distinct combination of two or more machines. An average communication delay value is then determined experimentally and associated with each of these paths, thus characterizing the network as seen by programs running on those four machines.

5.2.1 Network delay

The network load calculation is necessarily crude; this is due to the fact that the network is only easily accessed from a user program through the top of the TCP stack. Due to the volume of traffic on a typical ethernet segment, only a dedicated workstation with its interface set in promiscuous mode is capable of counting all packets on the network. Typically, network load is monitored by special devices whose only purpose is to maintain network statistics. One goal of this study, however, is to avoid special purpose hardware and operating system modifications; all load statistics, including network load, are obtained through user level, easily-ported software.

The network load statistic is based on communication delay. The monitor program regularly sends a single-packet message among the participating workstations, timing the travel time. Note that the delay value thus measured includes not only delay due to traffic on the network itself, but also processing time entering and leaving each workstation. Such inclusions are deemed valid since, from a program standpoint, only the delay from the time a message leaves its sender to the time it arrives at its receiver is important; the source of the delay is irrelevant.

The resulting values are "snapshots" of network traffic, which may then be averaged over time to produce a measure analogous to the UNIX load average. The nature of network traffic, however, leads to delay values which fluctuate wildly from moment to moment. Likely, much of this delay variation is due to the exponential backoff strategy used during packet

collisions [49][52]; at any moment, it is impossible to predict how many processes may be waiting to retransmit due to a collision. On the other hand, process creation and movement into the ready queue are frequently dependent on human response time, which is slow relative to CPU performance. Therefore, is unlikely that, at any given moment, a large number of ready-to-run processes will suddenly appear.

5.3 Monitoring load for program performance prediction

In a system characterized by one of more load values, some means must be found to measure those values if they are to be used for program performance prediction. Determining the effective CPU load on a computer, for example, is not a simple task; usually even the operating system kernel data structures lack the relevant information. The cause of this complexity is, of course, due to the many factors which contribute to loading on a system, whether at the CPU or the network level. To make matters worse, the mere act of attempting to measure load may increase the load itself. Not only will this degrade the quality of the measurement, but it may adversely affect the performance of the system as a whole.

There are two basic techniques for measuring any quantity that cannot be measured directly. One is to measure a related quantity that is affected in a known way by the quantity of interest. The other is to perform an experiment, and determine the quantity based on the outcome of the experiment. The latter technique is useful when measuring network loading, while both may be applied to the measurement of CPU load.

5.3.1 Network load measurement and performance prediction

Network load is not an easy thing to quantify. Usually, it is expressed as a percentage of total available bandwidth, which brings to mind images of a pipe partially filled with water; as the water level in the pipe rises, so it nears full capacity. However, this analogy is misleading, at least in the case of ethernet, the most common LAN in use today. An ethernet line is a shared bus, and as such can only be used by a single computer at a time. Thus, at any given moment, the bus is either in use (full) or idle (empty); there are no levels in between. A measurement in terms

of total bandwidth is thus only meaningful over some time interval, as the percentage of time the network is in use; at any given moment, such a value is meaningless.

One way to measure usage of such a shared resource is to measure the length of the queues formed by entities, in this case processes, waiting for access. As discussed further below, this method is applicable for measuring load on local shared resources, such as a CPU, but generally fails for a global shared resource such as a network segment. The problem arises due to the fact that there is no single queue; rather, there are queues on every machine attached to the network. Thus, measuring load in terms of queue length would require polling every machine with access to the network, including any remote machines that might try to communicate with machines on the local segment. Even if there were a single, easily accessed network queue on every machine--which there generally is not--the task of polling so many machines would likely contribute significantly to the traffic that was being measured.

The purpose to which a load measure is being put sometimes suggests a solution. For example, dedicated computers may be attached to a network segment to count and even examine every packet that appears on the network segment. These "network sniffers" can generate statistics regarding network traffic patterns and utilization levels which may be put to many uses, such as finding overloaded segments or calculating performance. In many cases, though, dedicating a machine to network monitoring is not feasible, or at least not convenient.

If one is interested in determining the affect of network load on the performance of a running program which uses the network in some way, a simpler solution suggests itself. Consider two processes, running on different machines, which must communicate. The network load will have little affect on runtime until a message is sent from one process to the other. A this point, the processes, as a whole, will see a certain delay due to the network, as the time between when the message is sent to the time it is received. Although this delay may be due to a number of factors, such as subnet load, local machine load, and router and bridge delays, only the actual length of the delay is important to the performance of the pair of processes. For predicting program performance, therefore, one need only be interested in the process-to-process communication latency. This value may be easily measured with another pair of processes, running on the same machines, timing brief message transmission times. As long as the

frequency and size of timing transmissions are kept low, very minimal network interference will occur. This is an example of an experimental measurement method.

5.3.2 CPU load measurement and performance prediction

CPU load is easier to conceptualize than network load, at least within a multitasking operating system such as UNIX. In general, a CPU divides its time among a set of currently running processes. Each process may be in one of three states: running, waiting to run, and sleeping. Only a single process is running at any given time; all others are either waiting to run or sleeping. A sleeping process is either waiting for I/O or waiting for a timer to expire; until one or the other happens, a sleeping process does not compete for CPU resources. Therefore, the load on the CPU at any given time can be calculated as the number of processes that are waiting to run, or, stated another way, the length of the ready-to-run queue. This is an example of a measurement made by measuring the affect of the desired quantity on another quantity; we measure the CPU load by watching the length of the run queue.

Unfortunately, the run queue, part of the operating system kernel's internal data structures, is not easily accessed without kernel modifications. However, UNIX and other operating systems provide a time average of run queue length called the *load average*. This load average seems to provide the load measure desired, but there are a few important considerations. First, depending on the OS, a mere count of the number of waiting processes does not take into account the relative priority of those processes. UNIX, for example, typically runs interactive user jobs at a higher priority than background tasks. Thus, a load consisting of a process run by a user sitting at the console, and a load consisting of a background process started remotely, will yield the same load average even though the actual load on the CPU is different. This is true even if the same process is involved in either case.

Second, the load average does not reflect differing CPU speeds, a serious deficiency if load average is being used to predict program performance. Two different CPUs may have identical run queue lengths, but if one processor is twice as fast as the other, one would expect a performance advantage running programs on that processor. One solution is to scale the load average values, based on the relative CPU speeds of the machines under consideration. In this

manner, a CPU power metric may be derived, giving an indication of the relative CPU computing power available at each machine.

Other hardware-related factors affect program performance, and thus loading, in less tangible ways. Examples are amount of free memory, math coprocessor speed, memory architecture, and disk speed. The impact each of these factors has will be dependent on the application being executed. A very large application will show poor performance on a machine with little free memory, while a small application may run unaffected. Including these factors in the loading measurement, or CPU power metric, is thus difficult. Research in this area is being performed on the Charlotte distributed OS [2].

It is also possible to measure relative CPU load using the experimental method, by timing the execution of a small program on every machine. The UNIX operating system allows a program to be timed in two ways: wall clock time and CPU time. The wall clock time is the actual time elapsed between the program's launch and its termination. The CPU time is the amount of time spent by the CPU executing the program's code. The difference between these values is a sum of the time spent by the program sleeping and waiting in the ready-to-run queue. If the code is written to avoid sleeping--no I/O operations or sleep calls--the ratio of CPU time to wall clock time gives a measure of the percentage of time the program waited for the CPU, and thus a measure of the busyness of the CPU.

This method has drawbacks, however. If the program is small, it may frequently run to completion on lightly loaded systems without being preempted. Thus, calculated load values may be unreliable when load is low. Increasing the run time of the program will increase sensitivity but will also add to the total load, thus possibly reducing the performance of the applications running on the machine being monitored. Also, hardware-factors mentioned above could again lead to misleading results; a small benchmarking program may run well on a machine with little available memory, while a large application spends most of its time swapped to disk.

6. EXPERIMENTAL PROGRAMS

Data for this research was collected through the use of a dummy parallel application program and several load monitor programs. The application program was written to mimic the various behaviors of real parallel applications, and to allow the user to adjust certain parameters such as the ratio of communication operations to computation operations, and the length of each operation. The load monitor programs were written to collect the relevant statistics as unobtrusively as possible. This chapter discusses the data collection strategy in more detail and describes each of the relevant programs.

6.1 Overview of the Experimental Procedure

The goal of this research was to discover the quantitative relationship between system load, as measured using several techniques, and application program runtime; this relationship could then be used to create a program scheduler for the given system, or systems of its type. In this case, the work focuses on workstation networks running coarse-grained multiprocessor applications. However, similar techniques could be used in other situations.

Data was gathered by executing a series of programs, in various combinations, on a small network of workstations. One of the programs is a CPU load monitor, which attempts to gather data regarding local load on a machine. Another program gathers network load information for the network connecting the workstations. A third program is a PVM application, designed to mimic the behavior of a typical parallel program, performing computation and communication and recording various statistics as it runs. In addition, a program to generate CPU and network load was written, as a way to gauge the response of the load monitor programs.

6.1.1 The workstation network

The workstation network used consists of four DECstation 5000 computers, paired onto separate ethernet segments. These segments were then joined through a third segment (see

Figure 6.1). For each experiment, between 1 and 4 of these machines were used. Most experiments were run on sets of 1, 2, 3 and 4 machines, with the same machines used for every experiment with a given number of machines--i.e. all 2-mcahine experiments used the same 2 machines. For a given experiment, some combination of the above mentioned programs was run on each machines, and the resulting data recorded.

6.1.2 The programs

The CPU load monitor program consists of a master task and a group of slave tasks. When executed on a workstation network, the master program starts a slave on every machine in the network. Each slave records CPU load local to the machine it is running on, and periodically reports back to the master, which records all incoming data in a file. The program runs for a specified length of time. When it terminates, the produced file contains a list of load values on each machine being monitored.

The network load monitor program is similar to the CPU load monitor, except that it only records a single load value, the round-trip network delay. It also consists of a master task and multiple slave tasks, one on each machine. The slave tasks pass a message between them, the initiating slave recording the time it takes for the message to pass through every machine on the network. The resulting time is returned to the master, which records it in a file. The network monitor also runs to a specified period of time, producing a list of network load values.

The PVM test application program is structured identically to the load monitor programs: it consists of a master task and multiple slave tasks. The slave tasks perform the simulated workload, some number of computation and communication operations, while the master program times each slave's execution. When the program is started, various parameters may be specified to determine the amount of communication and computation it will perform, and the pattern in which it will be performed. It is claimed that, in this manner, the test application may mimic the behavior of many real applications, by simulating the amount of work performed, and the relative proportion of communication and computation. When the test application terminates, it records the runtime of each slave and the ratio of communication to computation performed, measured in time.

6.1.3 Experiments

The PVM test application program is meant to mimic a user application program. The goal of this research is to discover how the execution time for a typical application is affected by load. Thus, the primary aim of the experiments was to time the execution of an application, while simultaneously recording the system load. Then, execution times may be correlated with the measured load values. It was assumed that system load was such that the test application was not the primary loading agent.

To perform a typical experiment, the CPU and network load monitors were started on some subset of the workstation network, and allowed to run for a few minutes recording "typical" load values. The test application was then started, with parameters specified to mimic a specific type of real application, such as pure-computation, 10% communication, etc. The three programs were allowed to run for several hours, during which load values and application runtimes were recorded. At the end of the experiment, the load values were matched with corresponding application runtimes, using timestamps recorded in the files, and the resulting data processed. For more details on data processing steps, see Chapter 7.

The above description applies to the bulk of the data taken. As a supplement, some loading data sets were recorded without the test application running, to gather typical "background" load values for the network. In these cases, one or both of the load monitors were started on the workstations and allowed to run for a number of hours, recording measured values a file. In a few experiments, a small load generating program was also run, to gauge the response of the load monitors.

6.1.4 Organization of Chapter 6

The remainder of chapter 6 begins with an overview of the data collection strategy, followed by detailed descriptions of each of the programs used. First the CPU load monitor is described, followed by the network load monitor, the PVM test application, and finally the load generator. Program descriptions include definition of all runtime parameters, listing of output values, and execution flow in both master and slave tasks.

6.2 The Data Collection Strategy

The conclusions of this study are based on data collected over a period of time on a set of workstations within the Project Vincent computer system. These machines are grouped into two sets, located in two buildings, Coover and Durham. The buildings are attached to different ethernet segments, and are connected to each other through a router. Figure 6.1 shows a diagram of the network.

Data collection took two forms: collection of load statistics on an "idle" system, with the purpose of determining typical "background" behavior, and collection of load statistics during controlled execution of a parallel application, with the purpose of discovering the affect of current load on application execution time. Note that, in this case only, "idle" refers to a system with no running PVM applications (except the monitor programs); the workstations may be busy with other, non-PVM jobs.

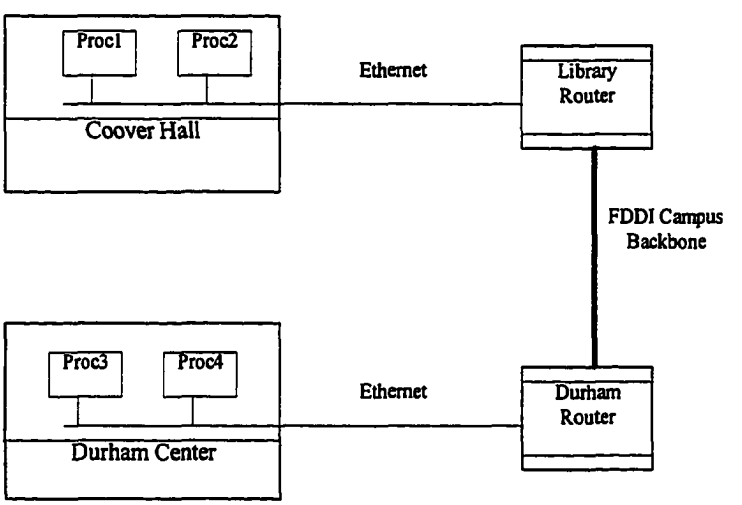

Figure 6.1 Topology of network used in experiments.

The task of load monitoring is assumed by two PVM applications, one which monitors load information local to each machine, and one which monitors load on the network connecting the machines. These programs are written to be as unobtrusive as possible, spending most of

their time asleep. During a data collection session, one or both of these programs run continuously, their output directed into temporary files. The output may be later viewed or processed. The total run time is set by the user; during runs without running PVM applications, a run length of four hours was commonly used.

The role of application program was filled by the PVM test application program, described in detail below. This program performs a user-specified workload, consisting of both computation and communication operations, by dividing the work evenly among a user-specified number of slave programs. The slave programs execute in parallel on the machines in the PVM configuration. A typical experiment is performed by choosing the desired amount of work, both of computation and communication, starting the load monitor programs on the machines in the configuration, and then executing the test application multiple times. At the end of each test application run, the total run time, along with other statistics returned by the application, is stored in a file. When the experiment is over, the user has three files, one with application run times, and two with load statistics recorded during those runs. Note that the total run time for such an experiment is variable, as it depends upon the runtime of each application execution; therefore, the total load monitor time must be estimated. In cases when the load monitors terminated before the test application finished the desired number of runs, the experiment was repeated. In the end, each experiment contains approximately the same number of test application runs.

In most cases, experiments were performed during the afternoon or evening hours, when typical usage is high (see [31]). When typical background usage patterns are important to the results, runs were made during the normal school semester. When only relative performance between runs was important, some experiments were performed during break, when the system is lightly loaded. For details on individual data sets, see Section 7.1.

6.3 The CPU Loading Monitor

The PVM-based CPU loading monitor software consists of two PVM programs: the master program, cpu_mon, and the slave program, cpu_slave. During execution, the master program starts a slave on each machine to be monitored, then receives and formats data from the

slaves, printing it to the screen (stdout). Execution continues for a user-specified period of time, after which the master program kills each slave and then exits. During a long run, data is captured in a file using I/O redirection.

6.3.1 The master program

The master program takes as parameters the run time and the monitor interval, where the run time is the total number of minutes to monitor the system, and the monitor interval is the number of seconds in between slave reports. These values may be specified on the command line; if they are omitted, the program will prompt the user to enter them. Once the parameters are obtained, the master outputs the current time and date and then starts a slave task on each machine in the current configuration. If that operation is successful, the monitor interval is sent in a message broadcast to all slave tasks. The master program then waits for a confirmation message from every slave, containing the machine name and CPU speed (see below) of the slave processor. The machine names are output along with the corresponding speeds.

Once all machine speeds are received, the master program enters a loop, waiting for CPU load reports from the slaves. It remains in this loop until the monitor time, specified by the user, expires. Each time a CPU load report is received, the master outputs the current time, the name of the reporting machine, and the load values (see below). The master waits for a report from each machine within a given interval before accepting reports for the next interval; this groups the reports for a given interval together in the output stream and thus simplifies processing. It will, however, cause the monitor program to lock up if a report is lost, as happens when a machine crashes. The UDP version of the load monitor, described below, fixes this shortcoming.

At the end of each monitor interval, the master program checks the timer, and if the monitor time has expired, it kills the slaves tasks and exits, outputing the message "terminated normally."

6.3.2 The slave program

A slave task is started on each machine in the PVM configuration. The slave begins by obtaining the process ids of itself and its parent (the master program). It then waits for a message

from the master giving it the monitor interval. This value, specified by the user, is the number of seconds between reports by the slave. It specifies the amount of time the slave sleeps between reports.

Once the monitor interval is received, the slave performs a simple benchmarking operation to determine the speed of the local CPU. This benchmark consists of the repeated pointwise multiplication of two vectors. The vectors contain 200 values of the C type *float*. A value in the source code called CLOOPS specifies the number of times the vectors are multiplied during a single benchmark. This value must be large enough so that the time for the operation exceeds by an order of magnitude or more the resolution of the timer. Currently, the UNIX system timer *times* is used. This timer returns the number of CPU seconds spent executing the code up until the timer call. By calling it before and after execution of the benchmark loop, one obtains the number of CPU seconds required to execute the benchmark. That value, in turn, gives an estimate of the relative speed of the processor, compared to other processors running the same benchmark. Each slave runs the benchmark a number of times specified in the source code by the constant SPDAVGS, currently equal to 5. The resulting timings are averaged and the result is sent to the master program. Thus, the "speed" value is really inversely proportional to the inherent speed of the CPU, since larger values indicate slower CPUs. The resulting values were found to be consistent for each type of machine architecture, to within 3% over 20 runs.

When the benchmark is complete, the slave transmits the benchmark result and the hostname of the local machine to the master program, then enters the monitor loop. This loop is infinite; the slave runs until killed by the master. Within the loop, the slave uses two methods to determine the current load on the CPU. The first method consists of the execution of a benchmark code block similar to that described above. The second consists of reading the "load average" value maintained by the UNIX kernel and returned by the *uptime* command. This latter value is an average over a time interval of the length of the ready-to-run queue. It is thus a measure of the average number of user processes waiting for CPU time during the specified time interval [39].

The first method of measuring load consists of timing the execution of a vector multiply operation. The same vectors discussed above are multiplied together a number of times specified by the constant CLOOPS. Again, the UNIX timer routines are used to obtain the execution time

for this code block. Three time values are returned. The first, CPU time, is the amount of CPU time spent executing the user level code (the multiply operations). The second, SYS time, is the amount of time spent executing UNIX kernel code in behalf of the user level code, and is usually 0. The third, wall clock time, is the amount of time that elapsed from start to finish of execution of the user level code. In theory, if the system is lightly loaded, the sum of the SYS and the CPU times should approach the elapsed wall clock time. As the system grows busy, more time will be spent by the benchmark code waiting for execution, and thus the elapsed wall clock time will exceed the CPU and SYS time sum. The ratio of wall time to CPU plus SYS time is thus the basis for the load measure. This ratio will approach 1 when the system is lightly loaded; it will grow as the system becomes heavily loaded.

The second load measure returned by the slave is the system load average, maintained internally by the UNIX kernel. This load measure is the average number of jobs waiting for execution in the ready-to-run queue, taken over 1, 5 and 15 minute periods. All three values are returned by the UNIX *uptime* command, and in unprivileged mode, the slave code uses this command to get the load average value. Specifically, the *system* system call is issued, causing the execution of the *uptime* command with its output directed into a file in the /tmp directory. The slave then reads this file to obtain the 1 minute load average value. In privileged mode, the slave may read the load average directly from kernel memory. However, privileged mode requires that the code be compiled by the root user, which was not generally possible. The load average of a typical machine varies from 0.0 to 2.0, but on a heavily loaded computer the load average may reach 15.0 or more.

After obtaining the benchmark timings and the 1 minute load average, the slave sends them along with its machine name to the master program. The slave then sleeps for the time period specified by the report interval received from the master. This cycle repeats until the slave is killed by the master.

6.4 The Network Loading Monitor

The PVM-based network load monitor consists of two programs, the master program called net_mon and the slave program called net_slave. Upon execution, the master program

starts an instance of the slave program on each machine in the current PVM configuration. The slaves work together to determine the round-trip delay for a message passed in a circular manner among the machines in the configuration. The "lead" slave then sends that value to the master program, and all slaves sleep until the next measurement. The master program formats and displays the data, which may be captured into a file using I/O redirection.

6.4.1 The master program

The master program begins by parsing the command line for user-specified parameters. If the values are not found on the command line, the user is prompted to enter each parameter. The first parameter, runtime, specifies the total amount of the time that the network monitor will execute; when this time expires, the master program kills the slaves and exits. The second parameter, monitor interval, is the number of seconds between slave reports; the slaves will sleep for this period of time after each network delay measurement.

When all parameter values are obtained, the master starts the slave program on each machine in the PVM configuration. It then sends the slaves the report interval specified by the user, the number of slaves, and a list of slave process id numbers. The slaves respond with their machine names, which the master program copies into a table, storing them in the same order as their corresponding process ids are stored in the process id table (return by the PVM *spawn* command).

Once the machine names are obtained, the master enters a loop, waiting for reports from the "lead" slave and printing the results to the screen (stdout). The master remains in the loop until the user specified runtime expires, at which time the master kills all slave processes and exits. Upon exit, "terminated normally" is printed to the screen.

6.4.2 The slave program

The slave program begins by obtaining the process id of itself and its parent, the master program. It then waits to receive the following information from the master program: the monitor interval, the number of slaves, and the slave process id list. The monitor interval is the number of seconds that the slave will sleep between reports to the master. The number of slaves

is used to determine the size of the slave id list. And the slave id list is used to chose a "lead" slave (described below), and to order the slaves for message passing.

The slave then searches the process id list for its process id; the offset of its id in the list becomes the slave's "index." The index determines the order in which messages will be passed among the slaves. The slave with index 0 is designated the "lead" slave, and is responsible for starting each measurement sequence, timing the round trip delay, and reporting the results to the master. The other slaves simply wait for a message from the slave with index one lower than their own, and then relay the message to the slave with index one higher than their own. The one exception is the slave with the highest index, which relays any received message back to the lead slave.

After determining its index, the slave gets the hostname of the local machine and sends it to the master. It then enters an infinite loop, in which it participates in a delay timing, as described above, followed by a sleep period of length equal to the report interval passed in from the master. This cycle is repeated indefinitely, until the slave is killed by the master program.

Within the loop, the slave checks its index to determine whether it is the lead slave. If it is the lead slave, it sends an empty array of time stamps to the slave with index 1, then waits to receive a message from the slave with the highest index. When that message arrives, the lead slave calculates the amount of time elapsed since the message was sent, and sends that time value to the master. It then sleeps until the next report time arrives.

If the slave is not the lead slave, it waits for a message to arrive from the slave whose process id is one less than its own id. When that message arrives, the slave gets the current time and inserts it into the empty time stamp array send by the lead slave. The array is then sent to the slave with the next highest process id. Once the message is sent, the slave sleeps until the next report time arrives.

6.5 The PVM-Based Test Application

Like the monitor programs, the test application program consists of two parts, a master program and a slave program. During execution, the master starts a user-specified number of slaves, and the slaves perform a combination of computation and communication operations. The

total number of operations performed, and the ratio of communication to computation operations, is set by the user. Upon completion, the total execution time is returned, along with the ratio of time spent communicating to time spent computing. By setting the communication to computation operation ratio, total operation count, and communication pattern appropriately, the test application will mimic the behavior of many real-world applications.

6.5.1 The master program

The master program, called ccmaster, begins by parsing the command line, looking for parameter values. If these values have not been supplied by the user, the user is prompted to enter them. A total of eight parameter values determine the behavior of the test application. Proper selection of these values allows the test application to model the behavior of many real-world applications. The parameters are, in order: number of processes, number of computations, computation length, number of communications, number of packets, multiplier, communication pattern, and workload distribution. Each of these variables is described below.

Number of processes: The number of processes is the number of slave instances to be started on the configuration. Unlike the monitor programs, which automatically start one and only one slave on each participating machine, the test application allows an arbitrary number of slaves to be executed. The resulting slave programs are scheduled amongst the available processors using the standard PVM round-robin scheduling.

Number of computations: The number of computations is the number of computation operations to be performed in each "unit" of work. Each computation operation is a call to the *compute* subroutine within the slave, described with the slave program below. A "unit" of work consists of the user-specified number of computation operations, followed by a user-specified number of communication operations.

Computation length: The computation length specifies the length of each computation operation in one of two ways: either by specifying the length of wall clock time, in milliseconds, to compute, or by specifying the number of computation loops (described with the slave

program, below) within each compute operation. If the value is positive, it is interpreted by the master as a time length. If the value is negative, its absolute value is interpreted as a number of compute loops. Note that when the time length of a compute operation is specified, the actual number of compute loops performed during each compute operation will vary, and vice versa.

Number of communications: The number of communications determines the number of communication operations which make up each "unit" of work. Each communicate operation is a call to the communicate subroutine within the slave, described with the slave program, below. Since each communication involves another slave task, which serves as a receiving task, a single communication operation generally involves both sending a message and receiving one. The only exception is communication in the random pattern (see communication pattern, below).

Number of packets: This parameter specifies the number of data blocks that are transmitted during each communicate operation. The size of each block is determined by a compile time constant, MESS_LEN, within the slave program. The value of MESS_LEN is chosen close to the maximum packet size for ethernet networks. Thus, the number of blocks is also the number of packets sent during each communicate operation.

Multiplier: The total number of "units" of work performed during one run of the test application is determined by the multiplier parameter. Thus, if the user specifies 2 compute operations and 3 communicate operations, with a multiplier of 10, a total of (2+3)x10 = 50 operations will be performed.

Communication pattern: The communication paths used by the slaves during each communication operation are determined by the communication pattern parameter. Three choices are available: circular, pairwise, or random. During circular communication, slaves are organized by process id into a circle and each slave passes a message to its clockwise neighbor. During pairwise communication, slaves are paired by process id, and each slave exchanges messages with its partner; with an odd number of slaves, the last slave communicates with itself. During random communication, the master program randomly chooses a receiving partner for

each slave, then sends a list of message paths to each slave; the slaves use the list both to pick their receiver, and to determine how many incoming messages from other slaves to receive.

Workload distribution: The final parameter, workload distribution, specifies the interleaving of communication and computation operations. There are two available values: distributed and lumped. During distributed communication, communication and computation operations are performed during each work unit, computation first, followed by communication; the pattern is repeated a number of times specified by the multiplier. During lumped communication, all computation operations are performed first, followed by all communication operations; the number of each operation performed is determined by the user specified number (above) multiplied by the multiplier. In both cases, the total number of each type of operation performed is the same.

After parsing the input parameters, the master program outputs to the screen (stdout) the current time and date, along with the user-specified parameters. It then performs some processing on the parameters, starts the main program execution timer, and spawns the number of slaves specified by the user. It sends relevant parameter values to each slave, and, if necessary, calculates random partners for slave communication. Then it waits for each slave to return its "results." In addition to dummy values, the results include the communication-to-computation time ratios, the number of messages received, and the number of computation loops performed per compute operation. This latter value is relevant if the computation operation length if specified by time rather than operation count (see above). Results are printed to the screen as each slave replies.

Once all slaves have reported, the master program stops the main program execution timer and calculates the total time for program execution. It then prints the value to the screen and terminates.

6.5.2 The slave program

The slave program, called ccslave, represents the workhorse of the test application; it performs the actual computations, returning results to the waiting master. These computations

are implemented with two subroutines: the compute subroutine and the communicate subroutine. These subroutines are described next, followed by a description of the slave program itself.

6.5.2.1 The compute subroutine

Each slave computation operation is implemented by a call to the compute subroutine. Parameters to the routine include the computation length value chosen by the user and two dummy operands of C type *float* to be processed. Returned by the subroutine are the result of the computation and the number of loops performed (explained below).

The computation consists of a multiply and sum operation; the two dummy operands are multiplied and added to an accumulator. If the user specified a time-based operation length, the multiply-sum step is repeated until the timer expires, and the total number of loops performed is returned to the calling program. If the user specified a loop count-based operation length, that number of multiply-sum operations is performed, and the loop count is returned. Each multiply-sum is performed with C types of *double float* and the final sum is returned to the calling program.

6.5.2.2 The communicate subroutine

Each slave communication operation is implemented by a call to the communicate subroutine. Parameters to the routine include a list of slave process ids, the index of the receiving partner in the process id list, the data block to send, the number of transmissions per message, and the number of messages to receive. This latter parameter is normally 1 except during random communication patterns, in which case a given slave may serve as receiving partner for more than one transmitting slave. Returned by the subroutine is the number of messages received.

A communication operation consists of the transmission of one message, and then the receipt of all incoming messages. Each message, in turn, is made up of one or more data block transmissions; the total number is determined by the user-specified number of packets, sent to the slave by the master. The subroutine returns the actual number of messages received, which will always be equal to the number specified by the input parameters, since the subroutine uses a blocking receive; if the correct number of messages is not received, the subroutine will never return.

6.5.2.3 The main slave program

The slave program begins by retrieving the process ids of itself and its parent (the master program). It then gets the hostname of the local machine and waits to receive its runtime parameters from the master. When the parameters are received, the slave searches the slave id list for its own id; the resulting offset becomes the index for the slave. This index then determines the slave's position in the communication pattern.

Two random numbers of C type *double float* are then generated to serve as dummy operands to the compute subroutine, and an array of random integers of C type *long int* is created to form the body of the messages sent during communication. The size of this array is defined by the constant MESS_LEN, and is set to a value close to the maximum packet size on the network. Thus, each transmission takes up one full packet, and each message consists of a number of packets equal to the number of transmissions specified by the user.

Once the compute and communicate operands have been initialized, the slave enters one of two loops, dependent on the workload distribution parameter. For a distributed workload, the loop counter is set to the multiplier parameter, and during each pass through the loop, a sequence of computation and communication operations occur. For the lumped workload, the loop is broken into two pieces, the first of which performs all the computations while the second performs all communications. In either case, communication and computation times are tracked, and the totals are used to calculate the ratio of communication time to computation time. Also, ratios of communication and computation times to total time are derived.The ratios are then sent to the master program along with the total number of computation loops in the last compute operation, and the slave terminates.

6.6 The UDP-Based Monitor Programs

Two weaknesses in the PVM-based monitor programs revealed themselves as the programs were used. The first problem arises because of the configuration of the workstations on the Project Vincent network; rsh and rexec are not allowed on many public workstations. This restriction was implemented to reduce outside traffic which interferes with a user sitting at the

console. Unfortunately, because PVM is built upon these remote UNIX commands, the PVM-based monitors could not be run on many machines.

The second problem was workstation crashes, which caused the monitor master program to lock up waiting for messages from slaves on the crashed machines. The locking problem could possibly be fixed, to some extent, with non-blocking reads, but there is no easy way to restart the lost slaves and PVM daemon processes once the crashed workstation comes back up.

To solve these problems, a monitor program was written using UDP for communication, with slave programs installed as daemon processes started automatically at boot time. The master then listens to a UDP port and accepts any message that appears there. Any number of slaves may be writing to that port at a given time. This arrangement allows long, trouble free monitor periods. Unfortunately, the current implementation allows passive monitoring only; there is no UDP equivalent of the test application program.

6.6.1 The master program

The master program, called cpu_serv, consists of two processes, the recorder process and the reflector process. The recorder process, the main body of the master program, listens for messages from daemons and outputs them to the screen (stdout). The reflector process, forked by the master program during initialization, listens for slave timing messages and sends them back to the transmitting slave. This message reflecting will be described in detail during discussion of the slave process, below.

After forking the reflector process, the master program (now the recorder process) opens a UDP datagram socket and binds to it the well-known port defined by the constant SERV_UDP_PORT. The reflector process similarly opens a socket and binds to it the port number SERV_UDP_PORT + 1. Both processes then enter endless loops, waiting for messages to arrive from the slaves. When the recorder receives a message, it checks the message type to determine what data is contained, formats the data, and prints it to the screen. Three message types are recognized. Message type 1 is a greeting message from a new slave, while message type 2 indicates a standard data transmission. Message type 3 is a special message from the exit program (see below) telling the master to terminate.

When the reflector receives a message, it reads the message header to determine the network address of the sender, then sends the message back to the sender. It performs no processing on the messages.

When the recorder process receives an exit message, it kills the reflector process and exits. Notice, however, that all slave (daemon) processes continue to run, unaware that the master has exited. When a new master program is started sometime in the future, it will begin receiving the slave messages just as the previous one did.

6.6.2 The slave program

The slave program, called cpud, is a UNIX daemon process that is started on a participating workstation during bootup. The slave begins by parsing the command line for parameters, which include the report interval, the server (master program) IP address, and the server port number. The report interval, specifying the number of seconds between transmissions to the master program, must be specified; default values for server address and port, hard-coded into the slave program, are used if overriding values are not given on the command line.

After processing parameters, the slave generates an array of random numbers of C type *double*, for use during benchmark timing. It then converts itself from user-level process to daemon process by changing its parent to process 0, changing its default directory to root, and closing all open file descriptors. It then gets its process id and hostname, and opens two UDP datagram sockets. The first socket is used to send reports to the master program recorder process, the second to perform round-trip timings with the master program reflector process.

When the sockets are created and bound to local addresses, the slave sends a welcome message to the recorder process, turns on the alarm signal handler, and begins load monitoring. It remains in the load monitor loop until killed; the master program currently has no means of deactivating a slave.

Inside the load monitor loop, three tasks are performed: the load average is obtained from the kernel memory, a benchmark timing is performed, and the round-trip communication time to the master program is recorded. These three load measures are then transmitted to the master program recording process, after which the slave sleeps for the report interval, then returns to the start of the loop.

The benchmark timings and load averages are obtained in the exact same manner as in the PVM-based monitor slaves. To perform round-trip timings, however, requires additional code, due to the nature of the datagram service. Because UDP is an unreliable service, the slave has no way of knowing whether the master program is currently running. This uncertainty has no consequence in regard to load reports to the recorder process; the packets are simply lost in the network. However, during a round-trip timing, the slave must have a communication partner on a non-local machine. Because the slave has no way of contacting other slaves (their port addresses are not known), the communication partner is implemented by the master program reflector process, which listens at a well-known port on the server machine. However, if the master program is not running, a slave will hang waiting for a reply. This reply will never arrive, even if the master program is restarted, because the original reply request message is lost in the network. Similarly, the master program has no way of contacting the slaves to "wake them up," because their port numbers are not known.

The solve this problem, the UNIX alarm signal service is used. After the round-trip timing message is sent to the (possibly nonexistent) reflector process, the slaves sets its alarm timer for ten seconds and then goes to sleep, waiting for the message to return. If the message returns, the alarm is canceled and the slave calculates round-trip time normally and reports it to the master program. If, however, the alarm goes off, indicating that the packet was lost or the master program was not running, the slave sets the round-trip time to 0 and reports to the master program. A value of 0 was used in case a packet is lost while the master program is actually running; the impossible round-trip time can be recognized during further data processing and discarded. On the off chance that a "lost" timing packet somehow survives one or more report intervals and returns during a subsequent timing operation, the timing messages are numbered. Any out-of-sequence message is discarded.

6.6.3 The exit program

Both the master and the slave programs were designed to run indefinitely, mostly for simplicity of design. However, data buffers are flushed only if the master program terminates normally; if it is killed, recently received load data will be lost. To properly terminate the master program at any time, the exit program was written to send a special message to the master

program's well-known port. As described above, upon receiving this message, the master program terminates. Since the exit message, implemented with UDP datagrams, may be lost, multiple copies are sent. The number of copies is set by the constant NUMSENT.

6.7 The Load Generator Program

In order to provide a wider range of loading conditions within the workstation network, and to ease calibration of the load monitor programs described above, a load generator program was written. The load generator is a PVM application which allows the user to artificially raise the system CPU and network loads by introducing a user-specified amount of work. The load generator consists of a master program and three slave programs. The master program accepts parameters from the user and then starts up the appropriate slave programs. The slave programs generate the actual load on the participating workstations.

6.7.1 The master program

The master program, called generate, begins by parsing the command line parameters. If a single parameter is given, command line help is printed and the program exits. If less than the required eight parameters are given, the program ignores the command line and prompts the user for the necessary parameters. Otherwise, the following parameters are read from the command line: number of communication slaves, number of messages per burst, length of pause between bursts, number of computation slaves, number of computation passes per burst, length of pause between computations, and total run time. Each of these values is described below.

Number of communication slaves: The master program generates network traffic by starting up one or more communication slaves on each workstation, specified by this parameter. As will be explained below, two slaves are actually started for every communication slave specified, a sender and a receiver. Obviously, the more slaves on each machine, the greater the amount of traffic generated.

Number of messages per burst: The communication slaves generate bursts of transmissions, separated by quiet periods. This parameter gives the number of individual PVM messages that are sent during each burst, where each message is approximately 1000 bytes. The value for this parameter must be chosen carefully, as too much message traffic will lock up the PVM daemon. Experiments show that four slaves per machine sending bursts of 30 messages put the daemon near its upper limit; four slaves sending 40 messages crashes PVM.

Length of pause between bursts: This parameter indicates the number of seconds to sleep between transmission bursts. The value should be at least 1 to avoid overloading the PVM daemon.

Number of computation slaves: The master program generates CPU load by starting one or more computation slaves on each workstation, specified by this parameter. The larger the number of slaves on a workstation, the greater the load on the workstation's CPU.

Number of computation passes per burst: The computation slaves generate load by performing two vector multiply operations, one with a vector of C type *double* and one with a vector of C type *long*. This parameter specifies the number of times the vector multiplies are performed before the slave pauses. On the test machines, lightly loaded DECstation 5000/25s, a value of 2000 raised the load average from near 0.1 into the range 1.5-2.5.

Length of pause between bursts: This parameter gives the number of seconds each slave should pause between computation bursts.

Total run time: The total run time is the number of seconds the generate program will run before killing the slaves and exiting. If the specified value is 1 or more, the master program will run for that length of time. If the specified value is 0, the master program will prompt the user to type "q" to quit, and then run until "q" is typed. In either case, the master program spends most of its time asleep once the slaves are running.

Once parameters have been parsed, the master program requests the number of machines in the current configuration from the PVM kernel, then uses that number to calculate the total number of each type of slave. A random number seed is then generated for each slave by reading the microsecond counter returned by the UNIX system call *gettimeofday*. This method assumes that each call to *gettimeofday*, along with the assignment of the resulting value to the seed array, takes longer than a microsecond. If such is not the case, multiple slaves will receive identical seeds. An alternate solution is to have each slave generate its own seed, using a similar method, and counting on the startup delays for the slaves to desynchronize the timer calls.

Once the seeds are generated, the master starts up the network load generation slaves. For each slave specified by the user, the master starts a sender and receiver slave program, called gen_trans and gen_recv, respectively. It then transmits pertinent parameter values to the slaves. When that transmission is complete, it starts up the computation slaves and sends parameter values to them.

With all slaves started, the master program's work is complete, and it waits to terminate. If the user specified a run time of 0, the master prompts for input from the user, and terminates when a key is pressed, killing all slaves and exiting. If the user specified a non-zero run time, the master gets the current time in seconds with gettimeofday, adds the run time to the value to get the stop time, then enters a loop. Within the loop, it checks the current time, compares it to the stop time, and if the stop time has been exceeded, exits, as described above. Otherwise, it sleeps for 1 second and repeats the process.

6.7.2 The network load generator slave programs

The network load generator consists of two slave programs, a sending slave, called gen_trans, and a receiving slave, called gen_recv. Operation is straightforward; the sending slave chooses a receiving slave, then periodically sends that receiver a burst of messages. The receiver simply discards the messages. In between bursts, the sending slave sleeps for a period of time specified by the user. In order to prevent synchronization of multiple sending slaves running on the same machine, a random startup delay time is calculated by each slave, which then sleeps for that period of time before sending the first burst.

6.7.2.1 The sender, gen_trans

The sending slave begins by requesting from the PVM kernel its process id and the process id of the master process. It then waits to receive its startup parameters from the master. Included in these parameters are the sender slave process id table, the receiver slave process id table, and a table of random number seeds, of C type *long*. The slave locates its process id in the sender slave id table, and saves the offset into the table as its index value.

The slave must now calculate the process id of a suitable receiver slave. The only requirements in choosing a receiver are that the receiver be on a different machine and that it not be chosen by any other sender slave. These requirements are satisfied by taking advantage of PVM's round-robin scheduling technique. Assuming the total number of sending slaves is an integral multiple of the number of machines, a sender and receiver which share the same index value must be located on the same machine. Therefore, if the number of machines in the configuration is greater than one, the sender can be sure that the receiver whose index is one greater than its own must be located on a different machine. It can then obtain the process id of the receiver by adding one to its index and using the resulting value to index into the receiver slave process id table.

After finding the receiver process id, the slave uses its index to pick a random number seed from the array of seeds sent by the master. It uses this seed to start the random number generator *drand48*. The random number generator is then used to get a random value between 0 and 10, representing a startup delay in seconds; the slave sleeps for this amount of time before its first transmission. The purpose of the delay is to prevent synchronization of multiple slaves which are running on a single machine. By desynchronizing the slaves, the load they generate is spread more evenly across their execution time, and the probability of overloading the PVM daemon process is reduced.

Upon waking from the startup delay, the slave enters an endless loop, in which it will remain until killed by the exiting master program. Within the loop, the slave repeatedly constructs a message of size 1000 bytes and sends the message a number of times determined by the input parameters from the user. The slave then sleeps for the inter-burst pause time specified by the user, and returns to the top of the loop.

6.7.2.2 The receiver, gen_recv

The operation of the receiver slave processes is trivially simple; it enters and endless loop in which it waits for a message from any sender. Upon receipt of a message, it unpacks it and then waits for the next message. It terminates when it is killed by the exiting master program.

6.7.3 The computation load generator slave program

The computation load generator program begins by requesting the process id of itself and its parent, the master program. It then waits for receipt of startup parameters from the master program. These parameters include an array of random number seeds and the list of slave process ids. The slave locates its process id in the id list, using the offset of that id in the list as its index value. It then picks a random number seed from the seed array, using its index value, and seeds the random number generator, *drand48*. From the random number generator, it generates a random value between 0 and 10, and sleeps for that number of seconds. This startup pause is intended to desynchronize multiple slaves running on the same machine, spreading their combined load more evenly.

After waking from its startup pause, the slave generates four vectors of random values, two of C type *double* and two of C type *long*. It then enters an endless loop, in which it performs a vector multiply of each pair of vectors, a number of times specified by the user. After the multiplies, the slave sleeps for a period of time given by the user, and then returns to the top of the loop. The slave remains in the loop until killed by the exiting master program.

7. EXPERIMENTAL RESULTS

Various experiments were performed on a small PVM configuration using the programs described in the previous chapter. The current chapter details each experiment and then presents the resulting data. Conclusions regarding the general form of the data are also presented, along with attempts to explain any anomalies present. Final conclusions, however, are reserved for the next chapter.

7.1 Overview of Data Sets, Data Analysis, and Results

The experimental programs described in the previous chapter were run in various combinations to produce a series of data files. Each experimental run, typically lasting a day, produced one or more files of data, including application runtimes, communication-computation ratios, and system loads. All entries in a given file are timestamped, so that they may be correlated with corresponding entries in the other associated files from the experiment. From these files were derived the "raw" data sets, the "cooked" data sets, the regression plots and other analysis tools, and the final results of the experiment.

7.1.1 The raw data sets

The data recorded from the experiment consists of one or more files, each containing either a list of load values or application runtimes. In ease further processing of this data, the files were parsed to produce a single data set containing all information relevant to a particular experiment, called a "raw" data set. Typically, timestamps associated with each entry in a given file were read, and matched against timestamps in the other files associated with an experiment. Loads and runtimes with the same timestamp were then written out as a data record in the new, raw data set. Only minor processing, such as averaging loads during a given test application run, was performed on the data.

7.1.2 The cooked data sets

The cooked data sets were derived from the raw data sets through further processing. Typically, a cooked data set was generated for the purpose of performing a specific analysis, such as the relationship between highest CPU load average and test application runtime. To generate a cooked data set, the parent raw data set is parsed, and the relevant data removed, processed, and written into the new set. Processing consisted of operations such as finding the machine with the maximum load during a given time period, finding the runtime of the last slave task to terminate, etc. Note that a single raw data set may be processed in several different ways, producing multiple cooked data sets.

7.1.3 The data set analysis

Each cooked data set was then analyzed using one or more of the following techniques: regression analysis, scatter plotting, histogram generation, statistical averaging, and visual inspection. In many cases, multiple techniques were applied, each to the new data resulting from the previous.

7.1.4 The results

The results of the analysis of the cooked data sets are divided into two categories, experimental results and final conclusions. Experimental results include interesting features or correlation's within a data set, and hypothesized explanations. Preconceptions and expectations regarding the outcome of a given experiment are also discussed, in light of the results. Chapter 7 concentrates on experimental results. Final conclusions, including how experimental results may be used together for purposes of scheduling, are reserved for Chapter 8.

7.1.5 Organization of Chapter 7

Chapter 7 begins with a description of the raw data sets used in this work, including a list of the programs that generated the data, the runtime parameters for the programs, and the resulting data values recorded in the set. Next is a description of the cooked data sets, organized by the raw data set from which they were derived. Finally, the analysis procedure performed on

each cooked data set, and the results, are given. Relevant plots are provided, and analysis steps are described in detail.

7.2 Description of Raw Data Sets

Four groups of experimental data were taken following the procedure described in Section 6.2. Each group in turn consists of the results of multiple experiments, with each experiment represented by a file(s) recording the outputs of one or more of the programs described in Chapter 6. Generally, each experiment is identified uniquely by the values of a set of parameters to the experiment. In some cases, though, identical experiments were run multiple times to help distinguish between salient features of the data and noise. The individual groups of experiments are described briefly below; for a detailed explanation of the programs involved, see Chapter 6. For all data sets, time and date of execution for each program run are recorded in the experiment output file, as are the names of the machines involved.

7.2.1.1 Application runtime data set

This data set records PVM application behavior on a group of workstations during typical loading conditions. The PVM sample application, network and CPU load monitor programs were run on groups of 1 to 4 workstations, the output of each program recorded in a file. The report period for the load monitors was set to 25 seconds, and the PVM application was run with the following parameters: one process per machine, 50 loops per computation, and 4 packets per message, using a circular, distributed communication pattern. The other application parameters, number of operations per unit work, and total number of iterations performed, were varied with each run.

The numbers of computations and communications per unit of work were varied to give a unit ratio of communication to computation of 0.0, 0.1, 0.2 or 0.3. Since limiting the total number of units was necessary to reduce total runtime, and since fractional units are not possible, the total number of units performed, including both communication and computation, varied slightly, depending on the desired unit-to-unit ratio. For a ratio of 0.0, 52 units of computation were performed; for a ratio of 0.1, 55 units were performed, 50 of computation and 5 of

communication; for a ratio of 0.2, 60 units were performed, 50 of computation and 10 of communication; and for a ratio of 0.3, 52 units were performed, 40 of computation and 12 of communication. Thus, the total number of operations performed per run of the application varied slightly, depending on the desired communication to computation unit ratio.

The numbers of units given above were performed on each pass through the main loop of the application. The total number of loops performed, also called iterations, per run was set to be one of 40, 80, 120 or 160. Since it was desired that the total length of time covered during each experiment be relatively constant, the total number of application runs in an experiment decreases as the number of iterations increases. This decrease in number is due to the fact that, for example, a program performing 80 iterations has double the work of a program performing 40 iterations, and thus can be expected to take roughly twice as long, all other factors being equal.

The following data were recorded for each run in an experiment output file: total runtime for the application, actual time ratio of communication to computation (Cm/Cp) performed by each processor for the application, load averages for each processor during the run, benchmark load measures for each processor during the run, and network delay during the run amongst processors participating in the computation.

The primary purpose of this experiment group was to discover the relationship between the various load statistics and the run time of the application. Given three parameters, and four possible values per parameter, it can be seen that a total of 64 experiments were performed, yielding 64 data files. All experiments were performed using the same pool of four DECstation 5000s.

7.2.1.2 Monitor resolution data set

The output of the CPU and load monitor programs were recorded in the absence of the PVM application, in an attempt to examine characteristics of the system during normal operating conditions. Each experiment contains the output of the monitors using a recording interval of 5, 10, 15, or 20 seconds. In addition, each experiment was run three times, yielding a total of 12 experiment output files.

The following data were recorded in each experiment output file: CPU load averages for each processor, benchmark load measures for each processor, and network delay among the processors.

The primary purpose of this data was to determined the behavior of the load statistics over time, and to discover what effect the sampling interval has one the resultant patterns. All experiments were performed using the same four DECstation 5000s.

7.2.1.3 Network segment delay data set

This data set contains the delays as measured by the network load monitor when run on one of three different network segments. Two of the segments are within buildings (Coover and Durham), while the third segment connects the two buildings. For each segment, three identical experiments were run, each experiment consisting of execution of the network load monitor on two DECstation 5000s within the segment. The monitor period was 25 seconds.

The following data were recorded in each experiment output file: Network delay between the two machines within the segment.

The purpose of this data was to discover whether individual network segments could be identified based entirely on the network load as measured by the network load monitor.

7.2.1.4 Artificial load generation data set

As a second means of examining the relation between program execution and load measures, the load monitor programs were run and their output collected while the load generator application was run on the same machines. The artificial load generation data set contains the output of the monitor programs along with the load level produced by each run of the load generator. During each experiment, periods of generated load were interspersed with periods of load generator silence, so that background and generated load levels could be easily compared. All experiments were performed on a single pair of DECstation 5000s, with 1 second pauses between load generation bursts, and longer periods between generation runs (see below).

The CPU load generator was run for periods of 5 minutes, with interrun periods of 5 minutes. Each experiment was characterized by the number of load generating processes per machine, and a multiplier that determines the number of loops performed by each process during

each computation burst. Experiments were run with from 1 to 3 processes per machine, with each process generating load bursts in the range of 100 to 8000 loops per burst. Each experiment records 20 different burst lengths, in the range 100 to 2000 multiplied by a multiplier between 1 and 4. Thus, an experiment with a multiplier of 1 generates loads of 100, 200, 300, ... , 2000 loops while a multiplier of 3 generates loads of 300, 600, 900, ... , 6000 loops. The load monitor interval was set to 25 seconds.

Each CPU load generator experiment output file contains the following information: Load average and benchmark load measures on both processors, and the CPU load level, measured in loops per burst, being generated by the load generator at any given time.

The network load generator was run for periods of 2 minutes, with interrun periods of 5 minutes. Each experiment was characterized by the number of load generating processes per machine, and a multiplier that determines the number of transmissions performed by each process during each communication burst. Experiments were run with from 1 to 3 processes per machine, with each process generating load bursts in the range of 10 to 120 transmissions per burst. Each experiment records 4 different burst sizes, in the range 10 to 40 multiplied by a multiplier between 1 and 4 (a run with bursts of 160 transmissions was not made because of message buffering problems in PVM). Both CPU and network load monitors were run, with monitor intervals of 30 and 15 seconds, respectively.

Each network load generator experiment output file contains the following information: Load average and benchmark measures for both processors, the network delay between the processors, and the network load level, measured in transmissions per burst, being generated by the load generator at any given time.

7.3 Data Processing Steps

The processing of the data generated by the experiments described above progressed in two stages. During the first stage, the raw data generated by the various programs, described in the previous section, was parsed and processed to generate new files that could be read into Excel. During the parsing, fields of interest were extracted from the collection of raw files, rudimentary processing such as averaging or thresholding was performed, and the final results

were written to new files. The result files are then collected into "cooked" data sets, with the files in a given data set differing from each other only in the values of certain runtime parameters, such as number of processors or number of iterations. The various cooked data sets used in this work are described below; included in each description are the parameters that identify each file in the set, and the parsing/processing performed on the raw data to generate the cooked files.

During the second stage of processing, the cooked data files were read into Excel and statistically processed. Processing included scatter plotting, histogram plotting, and trend analysis. The processing steps performed on each cooked data set, along with the results, will be described in more detail in the next section

7.3.1 Cooked data sets derived from the application runtime raw data set

The following parameters identify the individual files contained within cooked data sets derived from the application runtime raw data set: number of processors (1, 2, 3, or 4), number of processing iterations (40, 80, 120, or 160), and communication to computation unit ratio (0.0, 0.1, 0.2, or 0.3). Note that data sets using Cm/Cp include only unit ratios of 0.1, 0.2, and 0.3.

7.3.1.1 Load response data sets

Each entry in the files in these data sets contains a test application runtime or communication to computation time ratio (Cm/Cp) value, an associated network delay value, and one or more CPU load values. Three cooked data sets were generated by pairing PVM test application runtimes with network delay and either load average or the benchmark load measures. Similarly, four data sets were generated using Cm/Cp instead of runtime. The seven resultant combinations are listed below (each includes network delay):

- Set 1: Runtime versus worst load average.
- Set 2: Cm/Cp on machine with worst load average versus worst load average.
- Set 3: Cm/Cp on all machines versus load average on all machines.
- Set 4: Runtime versus load average on the last machine to respond.
- Set 5: Runtime versus lowest benchmark .
- Set 6: Cm/Cp on machine with lowest benchmark versus lowest benchmark.

- Set 7: Cm/Cp on all machines versus benchmark on all machines.

The raw data files were parsed as follows. Load values for a given run of the test application were calculated as the average of the values returned by the monitor programs while the application ran; timestamps allowed these averages to be calculated. Worst load average was chosen as the highest load average on the machines in the configuration during a particular run (for the above data, all machines had the same CPU speed). Similarly, lowest benchmark was the smallest average benchmark value associated with a run. Finally, the last machine to respond was the last machine to return its Cm/Cp slave data to the application master program during a given run.

7.3.1.2 Speedup data sets

Each entry in the files in these data sets contains a test application runtime, an associated network delay value, a CPU load value, and the number of processors for the run. To generate an individual file, test application runs with all parameters identical except number of processors (1, 2, 3, or 4) were combined, with number of processors added as a new field in the data. Two of the sets contain CPU loads measured by benchmark measurements, and two contain CPU load data measured by load averages. In two of the data sets, all runs for a given number of processors were recorded, while in the other two data sets, only an average of all runs was recorded. The four resultant combinations are listed below (each includes network delay):

- Set 1: Speedup with benchmark loads, all values listed.
- Set 2: Speedup with benchmark loads, average values listed.
- Set 3: Speedup with load average, all values listed.
- Set 4: Speedup with load average, average values listed.

The raw data was parsed to generate the speedup data sets as follows. Runtime versus load data was generated as described under "Load response data set," above. Then, for each iteration value (40, 80, 120, 160) and each communication to computation unit ratio (0.0, 0.1,

0.2, 0.3), four files corresponding to runs with 1, 2, 3 and 4 processors were combined into one file, with the number of processors added as a new field in the data.

7.3.1.3 Ratio versus runtime data set

Each entry in the files in this data set contains a test application runtime and the associated Cm/Cp value on each of the participating processors. Number of iterations (40, 80, 120, 160) and communication to computation unit ratios (0.1, 0.2, 0.3) were varied to create the individual files in the data set.

The raw data was parsed to generate the ratio versus runtime files as follows. The application runtime and associated communication to computation time ratios were read from the raw data.

7.3.1.4 Histogram data set

The files in these data sets contain histograms. There are two data sets. Set 1 contains histograms of Cm/Cp values, and set 2 contains histograms of application run times.

The raw data was parsed to generate the histogram files as follows. A set of evenly spaced ranges, with minimum and maximum values sufficient to contain all the points in the data set, were created, and the number of runtimes or Cm/Cp values falling into each range counted. Each entry in a histogram file contains the upper end of a range and the number of points falling in that range.

7.3.2 Cooked data sets derived from the monitor resolution raw data set

The following parameters identify the individual files contained within cooked data sets derived from the monitor resolution raw data set: monitor time interval (5, 10, 15, or 20 seconds) and experiment number (1, 2 or 3).

7.3.2.1 Load versus time data sets

Each entry in the files in these data sets contains a time stamp, starting at 0, and a load value. Set 1 contains network delay values, set 2 contains load average values, and set 3 contains benchmark measure values. One file was generated in each data set for each of the monitor

periods (5, 10 , 15, 20) and each of the three runs, giving a total of 12 files in each data set. Note that the four monitor period data files represent four distinct runs; the four resolutions are not four samples of the same time interval.

The raw data was parsed to generate the load versus time files as follows. The first time stamp in the file, representing the start of monitor program execution, was read and saved. Load values and associated time stamps were then read from the raw files and written to the cooked files, with the initial time stamp subtracted from the current time stamp to give the time elapsed since the start of execution.

7.3.3 Cooked data sets derived from the network segment delay raw data set

The following parameters identify the individual files contained within cooked data sets derived from the network segment delay raw data set: network segment monitored (Coover, Durham, or Coover-Durham) and experiment number (1, 2 or 3).

7.3.3.1 Network delay versus time

Each entry in the files in this data set contains a time stamp, starting at 0, and a network delay value. One file was generated for each of the three runs in each of the three segments (Coover, Durham, Coover-Durham), giving a total of 9 files in this data set.

The raw data was parsed to generate the load versus time files as follows. The first time stamp in the file, representing the start of network monitor program execution, was read and saved. Network delay values and associated time stamps were then read from the raw files and written to the cooked files, with the initial time stamp subtracted from the current time stamp to give the time elapsed since the start of execution.

7.3.4 Cooked data sets derived from the artificial load generation raw data set

The following parameters identify the individual files contained within cooked data sets derived from the artificial load generation raw data set: number of load generating slaves per machine (1, 2, 3, or 4) and the load multiply factor (1, 2, 3, or 4)

7.3.4.1 Artificial load

Each entry in the files in these data sets contains a time increment, a generated load level, and one or more measured load values. The generated and measured load types define each data set as follows:

- Set 1: Generated: network load. Measured: network delay, load average.
- Set 2: Generated: network load. Measured: network delay, benchmark load measure.
- Set 3: Generated: CPU load. Measured: load average.
- Set 4: Generated: CPU load. Measured: benchmark load measure.

The raw data was parsed to generate the generated load versus time files as follows. The first time stamp in the file, representing the start of the load monitor program execution, was read and saved. Network delay and CPU load values and associated time stamps were then read from the raw files and written to the cooked files, with the initial time stamp subtracted from the current time stamp to give the time elapsed since the start of execution. The load level parameter input into the load generator was also written into the file, with 0 written during periods of generator inactivity. The generated load values were then scaled into the range of actual loads measured for plotting purposes. The actual numeric values of these generated load levels are meaningless; a CPU load generation level 0f 2.0 does not correspond to a load average of 2.0. The generated load values were included to indicate relative load generation levels only.

7.4 Experimental Results: Application Run Time Data Sets

The application run time data sets contain results of PVM test application runs with various parameter settings on various PVM configurations, as described previously. They may be broken into four groups: load response, speedup, ratio versus runtime, and histogram. Results in each group are given in the following subsections.

7.4.1 Cooked data set used: load response set 1 -- load average data

7.4.1.1 Description of analysis procedure

PVM test application run times were scatter plotted against the worst load average among the machines in the configuration during the run interval. The worst load average was chosen by averaging all 1-minute load averages (taken from the OS) on each machine during a given test application run, and then taking the highest average. Linear regression was then performed on the data, and the slope of the regression line and the quality of fit, R^2, recorded. The slopes and R^2 values were plotted against the three variable parameters of the data set--number of processors, number of iterations, and communication-computation unit ratio--and linear regression performed if the data appeared to exhibit a trend.

7.4.1.2 Results

Figure 7.1 shows a histogram of the fitness values, R^2, for the trendlines fit to the run versus load data. Of the 64 plots, 50, or 78%, showed a fitness value above 0.1, while 36, or 56%, showed a fitness above 0.2, 14, or 22% showed a fitness value above 0.5 and 6, or 10%, showed a fitness value above 0.7. Figures 7.2, 7.3, and 7.4 show typical plots with fitness values of 0.01, 0.15, and 0.42, respectively. A clear trend can be seen in Figure 7.3 even though a fitness value of 0.15 may sound low. In fact, trending of the data points generally becomes visible to the eye between fitness values of 0.1 to 0.2. Plots with fitness values below 0.1 probably contain a number of runs whose execution time was affected by unusually heavy network activity, although other, unknown factors may also contribute. More will be said about network traffic later in this chapter.

Figures 7.5, 7.6, and 7.7 show the fitness values plotted versus number of processors, number of iterations, and communication-computation unit ratio, respectively. Examination of the plots shows the fitness values to be evenly distributed in all cases; these variables appear to have little affect on the tendency of the data points to lie on trends.

Figures 7.8, 7.9, and 7.10 show the trendline slope values plotted verses the same three variables. Were the load measure perfect, the slope would represent the relationship between run time and load, or the load line for the system. Thus, trends in slope values when plotted against

other, outside factors, may represent the effect those factors have on the quality of the load measure.

The plot of slope versus number of processors again shows an even distribution of points. When plotted against number of iterations, however, a trend toward smaller slopes as iteration count rises is visible; a linear regression on this data gives a regression line with a fitness factor of 0.24. Viewing the data, it is apparent that the range of slope values also tends to shrink as number of iterations rises. This relationship implies that longer running programs may behave more consistently in their response to load than short running programs. However, if that is the case, it is surprising that the fitness factors for the trendlines discussed above do not appear to rise with number of iterations.

A slight upward trend is seen when slope is plotted against communication-computation unit ratio. A linear regression on this plot produces a trendline with a fitness factor of 0.04, which is rather low. Intuitively, increasing the percentage of communication should increase the spread of program runtime values, and thus blur any relation with system load average. However, the trend in this data is not clear enough to draw any strong conclusions.

The notion of the existence of an ideal, unique relationship between program run time and load average, reflected in the slopes of the trendlines fit to such data, may be further tested by plotting slope against trendline fitness. The assumption here is that the better a group of data fit a certain trendline, the closer that line is to the ideal load line for the system; if the data closely fit a trendline other than the "ideal" line, then either the proximity of the data to the second trendline is a complete coincidence, or the ideal line, by definition, does not exist. The chances of a coincidental fit drop rapidly as the number of points fit rises.

A plot of slope versus fitness for the trendlines in the run versus load data is shown in Figure 7.11. A trend is apparent, producing a rise in slope value as fitness rises. A regression analysis of this data produces a trendline with a fit of 0.27. As expected, all negative slope values correspond to small fitness values; it would be surprising if the run time of any program dropped as system load rose. Even for fairly high fitness values, however, a considerable range of slope values exist. For example, trendlines with a fitness near 0.8 show slopes ranging from 0.01 to 0.04. This result implies that, if a single load line exists, its slope falls somewhere in this range.

More likely, however, multiple lines exist, with slopes falling within this range. One challenge, then, is to learn which load line a given program is likely to follow.

Two outlier points may be noted in the plot, at fitness values of 0.5 to 0.6. The plots corresponding to these points are shown in Figures 7.12 and 7.13. It is interesting to note two similarities between these plots: (1) they both correspond to runs with 40 iterations, and (2) they both contain a small number of outlier points themselves. More will be said below about outlier points in run versus load plots; for the moment notice that removing those points would significantly reduce the slopes of the trendlines. The small number of iterations, on the other hand, has already been shown to produce a wider range of slope values, with the values typically larger, than runs with a greater number of iterations. If these points are considered questionable for the above reasons, and are thus removed from the original plot, a new regression analysis gives a fitness factor of 0.31. This new plot is shown in Figure 7.14.

As mentioned above, several of the run time versus load average plots contain outlier points that diverge radically from the primary trend in the data. These points typically correspond to runs with small execution times during high load averages, or unusually long runs during periods of light load. Out of 64 plots, 13 contain one notable outlier and 5 contain 2 outliers. Also, most plots with low trend fitness values contain more than two outliers, though defining an outlier from a trend is difficult if there is little trend to begin with. The plot in Figure 7.2 is an example of such a plot. An example of a plot with a clear trend and 2 outliers is shown in Figure 7.3. More will be said about outliers in general in Chapter 8.

It is apparent that outliers are caused by system phenomena not reflected in the load average. In the case of high load averages with low runtimes, the load average has been artificially raised by some occurrence within the system that did not significantly affect program run time. In the case of high run times during low loading periods, some system event not reflected in the load average is reducing program performance. The possible causes of outlier points in run versus load plots will be examined again in Chapter 8.

Since the outlier points lie far outside the data trend, they will have an effect on the trendline, most likely reducing the fitness. To verify this conclusion, outliers were removed from the 18 plots mentioned above and regression analysis performed again. The ratio of new to old fitness was then calculated for each plot; such a ratio will be greater than 1 if removal of the

point improved the fitness of the trendline. A histogram of the resulting ratios is shown in Figure 7.15. As can be seen, in 12 out of 18 plots, removal of outliers improved the fitness of the trendline. Figure 7.16 shows the new trendline fit to the data from Figure 7.3, with the two outlier points removed.

To determine a possible cause for the presence of outlier points, the number of points was plotted versus the three variables within the data set: number of processors, number of iterations, and communication-computation unit ratio. These plots are shown in figures 7.17, 7.18 and 7.19, respectively. The plots against iterations and unit ratio show no conclusive trends; there is a distinct upward trend in the plot against number of processors, however. It is apparent, then, that adding processors to a program run tends to increase the likelihood of outlier points. That result is not surprising. If an outlier point is caused by unusual behavior on a given machine, adding machines increases the probability that, during a run, at least one machine will exhibit that behavior. In that light, however, it is surprising that an upward trend is not seen in the plot against number of iterations, since increasing iteration increases the length of time the program is running, and thus the chance that a participating machine will cause an outlier point. In fact, the largest number of outliers occurs in the runs with intermediate numbers of iterations.

Ratios of new to old fitness values were plotted against the same three variables, to see if any of the variables had an affect on the tendency for removal of an outlier point to improve the fit of a trend. No trends were seen in these plots, however.

One explanation for the presence of an outlier point is a congested network causing system load in the form of network delays. Such a load might effect runtime while not affecting load average, and vice versa; the details of this possible relationship will be discussed in the next chapter. To test this hypothesis, the net delay values corresponding to runs that generated outlier points were plotted versus the background network load during that run. If high network load levels contributed to outlier points, and if the network delay is an accurate measure of network load, one would expect to see unusually high network delays corresponding to outlier points. The original plot is shown in Figure 7.20. Three large network delays, with values of 27.223, 28.332, and 4.438 seconds are apparent, though most of the delays values are so much smaller that they are hard to discern when plotted with these three high values included. A second version of the plot, Figure 7.21, with the three high points removed, better shows the distribution of the

remaining points. Most of the values are below 400 milliseconds, and many are close to the background level, which was typically between 20 and 40 milliseconds. The network delays, then, do not obviously support the hypothesis outlined above.

7.4.2 Cooked data set used: load response set 1 -- network delay data

7.4.2.1 Description of analysis procedure

The PVM test application run times were also plotted versus the measured network delay during the run. As with load average, the network delay for a run was calculated as the average of all network delays recorded by the monitor program during the run. Since a given configuration is only associated with a single network delay, the delay through the network path connecting all the machines, no further processing was necessary.

7.4.2.2 Results

Two typical plots of run time versus network delay are shown in Figures 7.22 and 7.23. These plots both reveal an "L" shaped distribution of data points, with one arm of the "L" lying close to the run time axis, the other arm lying close to the network delay axis, and the majority of the points concentrated near the juncture of the arms. This distribution is, in fact, exactly what would be expected if run time was independent of network delay and the system, during the run, was lightly loaded. The reasoning behind this conclusion is as follows. During multiple runs on a lightly loaded system, most run times will fall near a single, typical value; this is the run time of the program when the interference from other system loading elements is minimum. A few run times, on the other hand, will be longer; they correspond to runs during brief periods of increased loads, possibly caused by a user logging onto the console or compiling a program. Similarly, most network delay values will fall near a minimum, background value (see Section 7.5 for more details on typical network delay values), with a minority of values spread out for an order of magnitude or two, caused by bursts of network activity on the local segment or segments. Taken together, a given program run time will most likely be associated with a minimal network delay, since the minimal delays are most common. Likewise, a given network delay will most likely be associated with a minimal program run time, since the minimal run times are most common.

These associations leads to the "L" shaped plots seen, with most points in the minimal region of the plot, and the rest scattered near the axis--minimal in one dimension, and unusually high in the other.

In a system that is more heavily loaded, the program run times are likely to be spread more widely across the range of run times, since a given run has a greater chance of being slowed by other system activity. In this case, most points will be scattered across the length of the run time axis, with a few points above that axis, corresponding to runs during heavy network loads. Two such plots are seen in Figures 7.24 and 7.25. This pattern and the "L" pattern typify almost every run-time-versus-network-delay data file collected.

7.4.3 Cooked data set used: load response set 5

7.4.3.1 Description of analysis procedure

PVM test application run times were scatter plotted against the corresponding CPU load as measured using the benchmark load measure. The benchmark load for a given program run on a given machine was calculated as the average of all the benchmark loads returned by the load monitor program during the test application run. One of these load values was calculated for each machine in the configuration during the run. The lowest benchmark load measure, representing the slowest or most heavily loaded machine, was then used for the plot. A linear regression analysis was performed on each scatter plot, and the slope and fitness value, R^2, of the resulting trendline was recorded.

7.4.3.2 Results

Trends were not obviously visible in most of the plots, and the fitness values of the trendlines were typically much lower than those in the load average plots described previously. Figure 7.26 shows a histogram of the R^2 fitness values. Out of 64 plots, 14, or 22% have a fitness greater than 0.1, 9, or 14% are greater than 0.2, 3, or 5% are greater than 0.5, and none are greater than 0.6.

To look for trends in fitness values, plots were made versus the three data set parameters: number of processors, number of iterations, and communication to computation unit ratio. Figure

7.27 shows the plot of fitness versus number of processors. An upward trend is seen in the data, with fitness values falling and the spread of fitness values shrinking as the number of processors rises. This result is not surprising, since an increase in the number of processors also increases the affect of communication, and thus of network load, on the run time. Since network load is not reflected in the benchmark load measure, any affect it might have on run time will place a data point outside the trendline. A regression performed on this plot gives a downward trendline with a fitness of 0.20. Figure 7.28 shows the percent of fitness values above 0.1, 0.2 and 0.5 plotted against number of processors, graphically illustrating the above conclusions.

The plot of fitness values versus number of iterations showed no trends in the data. If system irregularities are causing the lack of clear trends in runtime data, longer run times are not sufficient to cancel out their effects.

Figure 7.29 shows the plot of fitness values versus communication-computation unit ratio. With the exception of two points for the case of 0.3, and ignoring the ratio of 0.0, the spread of the fitness values shrinks noticeably as unit ratio rises. This trend is likely due to the same cause as the similar trend in the plot against number of processors: as unit ratio rises, run time depends more on communication and thus on network delays. As for unit ratio 0.0, it is quite surprising that such low fitness values are associated with these runs, since, with no communication, run time should be totally dependent on local load, which the benchmark load value is meant to measure. One possible explanation for this apparent anomaly is that the trend on this plot is, in fact, an artifact caused by a small sample space; further runs might cause the supposed shrink in point spread to vanish. This explanation is bolstered by the generally low fitness values of the trendlines in this data. As fitness drops below 0.1, trendline position is greatly effected by small changes in data point positions, and slope and fitness values have little meaning.

Slopes of the trendlines were also plotted against the three data set parameters. The plot of slope versus number of processors is shown in Figure 7.30. A downward trend in slope magnitudes, and a shrinking of point spread, is visible as number of processors rises. Performing a regression analysis yields a trendline with a positive slope and a fitness value of 0.11. As will be seen shortly, small magnitude negative and all positive slopes are actually associated with plots having low trendline fitness values. Thus, it is likely that raising the number of processors

is reducing the ability of the benchmark load measure to reflect actual CPU load, at least as it effects program run time.

No trend was seen in the plots of slope versus number of iterations or unit ratio.

To determine the relationship between slope and trend fitness, the plot shown in Figure 7.31 was generated. Notice that, although the trendline fitness is small in most cases, almost all the slopes are negative, as would be expected. Were the trendlines largely random, an even distribution of negative and positive slopes would be expected. Thus, there is a relationship between run time and benchmark measure. However, the relationship appears weak, and is diluted by other system factors that affect program run time.

Examining this plot, positive slopes are associated only with the smallest fitness values, as is expected. For trendlines with high fitness values, slopes appear to cluster around -0.005. There are not many data points in this region, however, so any conclusions made with this data are questionable.

7.4.4 Cooked data set used: load response set 2

7.4.4.1 Description of analysis procedure

Communication to computation time ratios (Cm/Cp) were scatter plotted against the load average on the most heavily loaded machine. Linear regression analysis was then performed on each plot, and the slopes and fitness (R^2) of the resulting trendlines recorded. Worst load average values were calculated as described under "load response set 1 -- load average data," and the value of Cm/Cp used was that returned by the machine with the worst load.

7.4.4.2 Results

Visible trends were largely absent from the data; a typical plot is shown in Figure 7.32. A histogram of the fitness values of the trendlines fitted to the data is shown in Figure 7.33; as can be seen, fitness values were typically low. Out of a total of 42 plots, 11, or 26% are above 0.1, 7, or 17% are above 0.2, 1, or 2% are above 0.5, and none are above 0.6.

Fitness values were plotted against the three variables in the data set: number of processors, number of iterations, and communication to computation unit ratio. The only trend

was seen in the plot versus number of processors, shown in Figure 7.34. The fit to the trendline is poor, though an increase in the spread of values, and in general an increase in average value, is visible as the number of processors rise. This increase may be due to the few, exceptionally high points, however, and the apparent overall trend is likely a coincidence.

Plots of slope values versus the same three variable were also made. Plots against iterations and unit ratio showed no trends in the data. A plot versus number of processors is shown in Figure 7.35, and displays a downward trend in slope values. Regression performed on this data yields a fitness value of 0.15. It is interesting to notice that roughly half the points have a positive slope and the other half have a negative slope, with the slopes tending toward negative values as the number of processors rises. This may indicate that the trends are largely random, reflecting no real trend in the data; if that were the case, however, one would expect the sign of the slopes to be more evenly distributed within each PVM configuration (1, 2, 3 or 4 processors). The other possibility is that the sign of the relationship between Cm/Cp and load really does reverse as the number of processors rises. Considering that computation time will rise as load increases, an overall downward trend in Cm/Cp would be expected. This is indeed what is seen as number of processors rises. On a single processor, on the other hand, Cm/Cp will rise with load because the "communication" time is composed entirely of processing overhead, since no actual communication takes place. The case of two processors appears intermediate between the two. For a two processor system, load may effect communication time significantly, since all communication must either enter or leave the heavily loaded processor; with three or more processors, only a fraction of the total communication travels through a single processor. The effect of load on communication time for a two processors system may be enough to reverse the sign of the trendline slope in some cases, thus explaining the trend seen in Figure 7.35.

A plot of slope values versus trendline fitness is shown in Figure 7.36. Visible in this plot is a downward trend in slope magnitudes as fitness drops. Note also that the trendlines with positive slopes tend to have lower fitness values; all the high fitness values correspond to negative slopes. Also note two outlier points, with fitness values of 0.29 and 0.07 and large positive slopes. With these points included, linear regression on this plot yields a trendline with a fitness value of 0.11. With the outliers removed, the new trendline has a fitness of 0.47. This second, modified plot is shown in Figure 7.37. These outlier points were likely caused by large

delays that occur occasionally on the network (see Section 7.4.1 for details on network delay behavior).

7.4.5 Cooked data set used: load response set 3

7.4.5.1 Description of analysis procedure

Scatter plots were made of Cm/Cp values versus the corresponding loads on all processors, rather than just the most loaded processor, as was discussed previously. Scatter plots were also made of Cm/Cp values versus network delay; in this case, a given network delay value is paired with a Cm/Cp value from each processor.

7.4.5.2 Results

Results were similar to those described for Load Response Set 2, above: visible trends were largely absent. Typical plots are seen in Figures 7.38 and 7.39. These two plots illustrate the only notable feature of the data, a tendency for the points to spread as the number of iterations rises. This trend was often especially notable when moving from 120 to 160 iterations. A lowering of process priority as program run time increases, a feature of the UNIX scheduler, could be responsible for this trend, though speedup plots showed a doubling of iteration count to roughly double execution time, even with unit ratios of 0.3. Perhaps a more probable explanation is that the addition of the fourth processor, the same machine in every case, caused the point spread; the fourth machine was used as a file server and tended to be much busier than the other three machine used in these runs.

Plots of Cm/Cp versus network load were also made. The same tendency of point spread to increase with increasing number of iterations was seen, though no trends following network delay values were detected.

7.4.6 Cooked data set used: speedup data set 4

7.4.6.1 Description of analysis procedure

For a given number of iterations (40, 80, 120, or 160) and a given communication to computation unit ratio (0.0, 0.1, 0.2, or 0.3) a plot of average test application run time versus

number of processors (1, 2, 3, or 4) was made. These plots provide inverse speedup curves for various problem sizes and communication percentages.

7.4.6.2 Results

The resulting curves looked about as expected. Figures 7.40 - 7.43 show the speedup curves for 40 iterations at each of the different communication percentages. Likewise, Figures 7.44 - 7.47 show the same set of curves for 160 iterations. Note that in the ideal case, for a run time of x on 1 processor, the 2 processor case should show a run time of $0.5x$ and the 4 processor case should show a run time of $0.25x$.

Examining the plots, it is apparent that this application on this system did not display ideal behavior. The amount of departure from the ideal increased as the amount of communication rose, and decreased as the number of iterations rose. Both behaviors are expected. The decrease in performance as unit ratio rises is due to the increase in communication overhead, which in not parallelizeable. Thus, as communication percentage rises, a greater portion of the program execution is spent in serial code. Similarly, an increase in performance with increasing problem size is due to the opposite effect; as computation problem size grows, so does the fraction of the code that is parallelizeable.

An unexpected feature of the plots is seen in Figures 7.42 and 7.43, where the character of the curve changes drastically as unit ratio increases from 0.2 to 0.3. This effect appeared to some extent at every value of iteration count, but it is most pronounced at 40 and diminishes as iteration count is increased. Examining the run times in the 40 iteration plots, it is apparent that, as the unit ratio rose from 0.2 to 0.3, execution time actually dropped significantly. The same is true of the 1 processor runs at 160 iterations, though for larger numbers of processors, the run time did increase slightly with unit ratio. The cause of this phenomena is likely a combination of three factors. (1) Processors 3 and 4 are on a high traffic network segment. Thus, introducing processor 3 reduces total computation load but increases average network delay for the configuration. Adding processor 4 then decreases total computation again, without adding additional network delay. (2) A greater total number of operations are performed during one loop of a 0.2 communication to computation unit ratio application run than are performed during a 0.3 unit ratio loop. This difference, as mentioned earlier, is due to the fact that fractional operations

are not implementable. As a side affect, runs at 0.3 unit ratio tend to be shorter than runs at 0.2 unit ratio when network load is light. (3) The processes running on 4 processor configurations may be small enough to gain a scheduling advantage over the processes running on 3 processor configurations; the UNIX scheduler decreases the priority of longer running processes.

In terms of speedup, 40 and 80 iteration runs all showed an increase in execution time with 4 processors. These runs typically lasted 120 seconds or less on one processor. Runs with 120 iterations saw some benefit at 4 processors, and had run times around 170 seconds. Runs with 160 iterations, at around 215 seconds, saw greater benefit. These times suggest a threshold in program size for parallelization; a program whose parallelizeable workload runs in less than 170 seconds probably will not benefit from more than three processors. Similar thresholds could be derived for greater number of processors, by increasing the number of iterations in the test application runs.

7.4.7 Cooked data set used: ratio versus runtime data set

7.4.7.1 Description of analysis procedure

The communication to computation time ratio (Cm/Cp) returned by each processor at the end of an application run were scatter plotted against the total execution time for that run.

7.4.7.2 Results

For the 1 processor case, the points appear to be randomly scattered across the plot. Since no actual communication occurs on a single processor run, the ratio Cm/Cp has little meaning, and thus random-looking plots are no surprise.

Somewhat surprising, however, are the strong correlations seen in the 2, 3, and 4 processor plots. Figures 7.38 and 7.39 show typical examples of the resultant plots for 2 and 4 processors, respectively. As is apparent in the plots, a strong relationship exists on most processors between Cm/Cp and total runtime. That there should be some relationship is expected, since the sum Cm + Cp on the slowest machine is equal to the run time. However, in most plots, the relationship between the two is strong on every machine. Considering that the

environment, including load average and local network congestion, varied greatly among the four machines involved, one would expect at least one of the machines to vary off the trend.

In general, since each run had the same amount of work to do, and each machine performed raw computation at the same speed, a rise in Cm/Cp should correlate with a rise in total execution time. Thus, on a given plot, the machine with the highest Cm/Cp curve should determine program execution time, and thus that curve must have a positive slope. Examination of all data showed that the highest curve does in fact have a positive slope in every case. Machines with Cm/Cp values consistently below this curve may follow any pattern, since their completion time is irrelevant to total program execution time. In fact, the other curves usually were either flat, or exhibited the same slope as the determining curve. Examples of both are seen in the above figures. A flat curve indicates that the computation and communication took consistent periods of time on every run, which is probably an indication of a lightly loaded machine. On the other hand, multiple curves with similar slopes may indicate network congestion with slowed communication to a similar degree on every machine. Consider Figure 7.49. Since the four machines involved were on two different subnets (see Section 6.1 for more details) the congestion probably existed on the trunk that connected the subnets. The machines receiving messages across the trunk were directly affected; the machines sending messages out onto the trunk were indirectly affected through increased TCP/IP stack processing due to network collisions. Similarly, Figure 7.50 shows a case where two machines, Proc1 and Proc2, lie on the same run time-determining trend, while the other two machines produce scattered points below. Since Proc1 and Proc2 shared a subnet, this is likely a case where local network traffic affected two processors and ultimately determined runtime, while fluctuating load conditions blurred the usual trend on the other two processors, one of which served as a heavily-loaded file server.

Figure 7.51 shows an interesting case, in which the runtime during fast runs was determined by the Proc2 curve, while slower runs were determined by the Proc1 curve. The fact that the Proc2 curve is almost flat indicates this machine maintained an even load throughout of test period. Proc1, on the other hand, had a fluctuating load which, at its lowest, dropped below the load on Proc2 but, at its highest, rose significantly above Proc2's load.

Another unusual plot is shown in Figure 7.52. Careful examination of this plot reveals two trend curves with each processor occupying parts of each curve. This phenomena is almost certainly due to network loading. It is likely that these two machines had similar local loads with a high network load between them. When messages were exchanged, one or the other message (but not both) would get delayed, and thus delay the receiving machine. The delays must have been fairly consistent to merge two sets of processors points into a single linear trend. In fact, these machines were located on the Coover subnet, which showed the most consistent network behavior of any of the three segments examined (see Section 7.5).

7.4.8 Cooked data set used: histogram data set

7.4.8.1 Description of analysis procedure

For each number of processors (1, 2, 3, or 4) and each iteration count (40, 80, 120, or 160), a histogram was plotted showing the distribution of run times for communication to computation unit ratios of 0.0, 0.1, 0.2 and 0.3. The histograms for the four unit ratios were combined to make comparison of the runtimes easier.

7.4.8.2 Results

Two typical 4 processor histograms are shown in Figure 7.53, for 40 iterations, and Figure 7.54, for 160 iterations. Each histogram shows the distribution of run times of all 4 processor PVM test applications runs for the given number of iterations (40 or 160) at each of the four unit ratios (0.0, 0.1, 0.2, and 0.3). In examining these histograms, it is important to keep in mind that the number of operations making up one unit of work is not the same for each of the unit ratios; for 0.0, 52 operations are performed per unit of work, for 0.1, 55, for 0.2, 60, and for 0.3, 52. Thus, even ignoring any difference between computation and communication units, a run at a unit ratio of 0.2 will perform more operations than a run at 0.0 with the same number of iterations. See Section 7.1, Description of Raw Data Sets, for details.

The histogram of the 40 iteration runs gives the following order of execution times, from low to high: 0.3, 0.0, 0.1, 0.2. As the 0.0 and 0.3 case perform the smallest number of operations per unit of work, it is perhaps not surprising that their execution times are smaller than the 0.1

and 0.2 cases. However, one would expect the 0.0 case to out-perform the 0.3 cases, due to the large difference in the amount of communication that is performed. The fact that 0.3 appears to run the best is likely due to the small overall problem size; true algorithm run times are being distorted by process startup overhead.

If the above argument is correct, one would expect to see 0.0 outperform 0.3 as the number of iterations rises. This is indeed the case in the 160 iteration plot. In fact, 0.3 performs the worst of the four, due to the large amount of communication performed. In this plot, the order of execution times, from low to high, is 0.0, 0.1, 0.2 and 0.3. In many of the other histograms, from 2 to 4 processors and 40 to 160 iterations, 0.3 outperforms 0.2, probably because the difference in work unit size overrides the difference in communication percentage.

The most important fact to glean from these histograms is the wide distribution of run times for any given unit ratio. This spread is particularly noticeable in Figure 7.53, in the 0.0 run; 0.0 run times are spread across nearly the entire range of run times in the plot. Intuitively, one would expect the 0.0 case to be least variable, since it involves only computation. Examination of the corresponding run time versus load average plot, shown in Figure 7.54, reveals a regression line fit of 0.5. The wide range in load averages, rising up to 2.5, is likely the cause of the wide spread of run times; during this particular experiment, the program responded to high loads, and that response is reflected in the histogram. Runs made with higher unit ratios happened to fall on lightly loaded days. For example, the run time versus load average plot for a unit ratio of 0.2, shown in Figure 7.55, reveals a much smaller spread in load average, and thus more closely clustered run times.

As a final note, Figure 7.54 reveals that the most common run times for unit ratios of 0.1, 0.2 and 0.3 all fall within 10 seconds of each other, and the vast majority of the runs for these three cases are within 30 seconds of each other. A similar phenomena is seen in Figure 7.56. Compensation for the larger work unit size in the 0.2 case will spread the execution times out somewhat, though in many cases, execution time for the 0.2 case is actually slightly longer than the 0.1 and 0.3 cases; an example is shown in Figure 7.57. In this latter case, compensation for unit size differences will bring the execution times closer together. In short, for the example application considered in this study, the difference in run times between unit ratios of 0.1, 0.2, and 0.3 is small and unpredictable.

7.5 Experimental Results: Monitor Resolution Data Sets

The monitor resolution data sets contain the output of the load monitor programs running without the PVM test application, using various report different intervals. They may be subdivided into two groups: CPU load response and network load response. Results in each group are given in the following subsections.

7.5.1 Cooked data set used: load versus time set 1

7.5.1.1 Description of analysis procedure

Network delay values were plotted versus time for monitor sample intervals of 5, 10, 15, and 20 seconds. The plots were then examined for the purpose of determining how the increasing monitor resolution effected the appearance of the plots; would the abrupt spikes be resolved into a continuous range of hills and valleys?

7.5.1.2 Results

The load monitor was run three times for each of the same intervals. Figures 7.59 through 7.62 show the resulting plots from one set of runs. It is immediately obvious that the character of the plots does not change as the monitor interval is reduced. In fact, there is no obvious change in the plot form; the plots appear to have the fractal characteristic of looking the same at any resolution. The same results were obtained with the plots from the other two experiments. Note that the apparent increase in spike width in the 10 and 15 second interval plots is a artifact of the decreased resolution; a single spike occupies a larger width on the time axis. Also, it is just coincidence that there are no spikes in the 20 second interval plot.

The explanation for this behavior will be discussed in more detail in Section 7.5. In short, packets injected into a congested network either get through or collide. Packets that get through see a fairly constant propagation delay value regardless of congestion. Packets that don't get through are transmitted again after waiting for some backoff period, which is usually long compared to the propagation delay. The low level background data points, derived as the mode

of the data, represent the packets that got through. The spikes represent packets that collided and had to be retransmitted. As the number of collisions rises, so does the number of spikes. As long as network congestion is fairly low, most of the packets will get through without colliding. Thus, at any resolution the plots show only isolated spikes, as only isolated packets are delayed due to collision.

7.5.2 Cooked data set used: load versus time set 2

7.5.2.1 Description of analysis procedure

The load average values were plotted versus time for monitor intervals of 5, 10, 15, and 20 seconds.

7.5.2.2 Results

One set of plots, for a single processor, is shown in Figures 7.63 through 7.66. Similar plots were made for the other three processors in the configuration. An increase in detail may be seen as the monitor interval shrinks. The load rises and falls over time, with large and small scale variations visible in the 5 second interval plot, but only large scale variations visible in the 20 second plot. For the purpose of predicting program behavior on a system, only relatively large scale variations are of interest.

It is interesting to note the periodic nature of the background load in the plots. This is especially visible in the 10 second plot. This is probably due to periodic system activity, such as cron operations or daemon processes.

The load remained fairly low during the monitor periods on the machine shown in the plots, though a large crest lasting several minutes is visible at the end of the 15 second plot. This peak would likely impact the performance of a program running on this machine noticeably. A similar peak on another processor is shown magnified in Figure 7.67.

7.6 Experimental Results: Network Segment Delay Data Set

7.6.1.1 Description of analysis procedure

Network delay values were plotted against time increment, and a histogram of the delay values was produced.

7.6.1.2 Results

Three different physical network segments were examined, called Coover, Durham, and Coover-Durham. The Coover and Durham segments were located in separate buildings and were separated from the rest of the network by bridges(??). The Coover-Durham segment is actually the multi-segment path that connects the Coover and Durham segments (see Section 6.1). Plots of typical delay profiles are shown in Figures 7.67, 7.68 and 7.69, with the corresponding histograms in Figures 7.70, 7.71 and 7.72.

It is apparent through examination of the histograms alone that the characteristics of these three segments are distinct. Two additional monitor periods taken on each segment verified these conclusions; the monitor profile for a given segment is characteristic for that segment. This profile characteristic may be specified by three parameters: background level, spike frequency, and spike magnitude.

The background level is calculated as the mode of the network delay data, and typically accounts for the vast majority of the data points taken on a given network segment. This value is represented by the flat bottom line on the plots. As may be seen in the histograms, almost all the data points taken are at or near the background. This behavior is typical of network load as measured by the load monitor program. On the three Coover runs, the background delay level measured was 19, 19 and 19 milliseconds. The three Durham runs yielded backgrounds of 7, 11, and 11. The three runs between Coover and Durham produced backgrounds of 15, 15, and 15.

Deviations from the background take the form of sharp, usually single-point-wide spikes. These spikes are frequently orders of magnitude above the background level. If a spike is defined as any point higher than 1 order of magnitude above the background, the spike frequency for a segment may be calculated as the number of spikes divided by the total number of readings taken. For the set of plots above, these values are, Coover: 0.5%, Durham: 1.2% , and Coover-

Durham: 3.1%. Considering a second set of data for each segment yields the following fractions: Coover: 0%, Durham: 1.0%, and Coover-Durham: 2.4%.

The third parameter classifying network behavior is typical spike amplitude. This value is harder to quantify, though examination of the plots in Figures 7.67, 7.68, and 7.69 reveals vastly different heights in the maximum spikes. In the Coover segment, the highest peak is near 300 milliseconds; in the Durham segment, the highest delay is 8000 milliseconds, and in the Coover-Durham segment it is 40000 milliseconds. This last value is somewhat extreme: examination of all three runs on each segment indicate typical Coover values around 300 milliseconds or less, while Durham and Coover-Durham values tend to fall in the 3000 to 8000 millisecond range.

It is apparent from looking at these plots that network delay does not measure network loading in the same manner that load average measures CPU load. Load average provides a load measure that varies slowly over time, without sudden jumps; thus, given a single point, one can estimate the load in the vicinity of that point. The network delay, on the other hand, remains almost constant except for large, single point deviations. Thus, a single network delay value provides no information about the delay values to either side of that point. Part of this difference between the measures is due to the fact that network delay is an instantaneous value, while load average is a time average. However, even if network delay values were averaged over time, the resulting curve would still contain large, sudden jumps at each spike.

The explanation for this behavior lies in the nature of network load. Loading on a LAN typically manifests itself in the number of collisions during a certain time period. Measuring network delay gives a distorted picture of this load by showing its effect on a single packet. The problem arises because a packet injected into a loaded network either traverses the segment in a fairly constant amount of time--the propagation delay of a signal on the network wire--or collides and does not traverse the network at all. The collision resolution algorithm used on TCP/IP networks then causes the transmitter to back off for a period of time that is large compared to the propagation delay of a signal on the wire. Therefore, the delay values measured are usually equal to the background (propagation delay) value, with occasional spikes caused by collisions. An increase in load does not raise the background level, but increases the frequency and size of the spikes.

7.7 Experimental Results: Artificial Load Generation Data Sets

The artificial load generation data sets contain the output of the load monitor programs running without the PVM test application, while the system is artificially loaded. The load is generated by one or more slave programs running on each machine, generating network traffic and computational load. The data sets may be subdivided into two groups: load set 1, representing artificial network load, and load set 2, representing artificial CPU load. Results in each group are given in the following subsections.

7.7.1 Cooked data set used: artificial load set 1

7.7.1.1 Description of analysis procedure

Network delay and load average data were plotted versus time. The artificial load level, measured in operations performed, was scaled into the same range as the measured load values and plotted with them so that relative artificial load levels could be more easily correlated with measured load.

7.7.1.2 Results

A typical plot of network delay versus time during artificial network load generation is shown in Figure 7.74. For this particular plot, two slave processes generated network traffic with a multiplier of 1 (see artificial load cooked data set description in Section 7.2.4 for more details). The measured network delay responds readily to the generated load; peaks are seen during each loading period. Interestingly, the network delay values form multi-point peaks rather than single-point spikes as seen while monitoring normal system load. Apparently, the artificial load was large enough to delay multiple packets. Still, even at the highest loading levels, with three slave processes and a multiplier of 4, the peaks tend to be narrow, sometimes falling to background levels while the artificial load is still present. Figure 7.75 shows this latter case; although a larger percentage of packets are being blocked, some still get through without delay. However, this

level of loading slowed system response noticeably, and caused an elevated load average, as shown in Figure 7.76.

The main reason for gathering this data was to check the underlying hypothesis that network delay values do reflect, in some form, network congestion. The data showed this hypothesis to be true, and also demonstrated that even the moderate load levels represented in Figure 7.74 were atypical of the system during normal operation. They may not be atypical, however, of more heavily used systems.

7.7.2 Cooked data set used: artificial load set 3

7.7.2.1 Description of analysis procedure

Load average data was plotted versus time. The artificial load level, measured in operations performed, was scaled into the same range as the measured load values and plotted with them so that relative artificial load levels could be more easily correlated with measured load.

7.7.2.2 Results

Load average responded readily to the artificially generated load, as may be seen in a section of one of the plots shown in Figure 7.77. This plot corresponds to a single slave process with a multiplier of 2 (see artificial load cooked data set description in Section 7.2.4 for details). The load average tracks the generated load level almost perfectly; the delay in fall times is due to the fact that load average is an average of load over the last minute.

This data demonstrates that load average does rise with the computation load on the system, as expected.

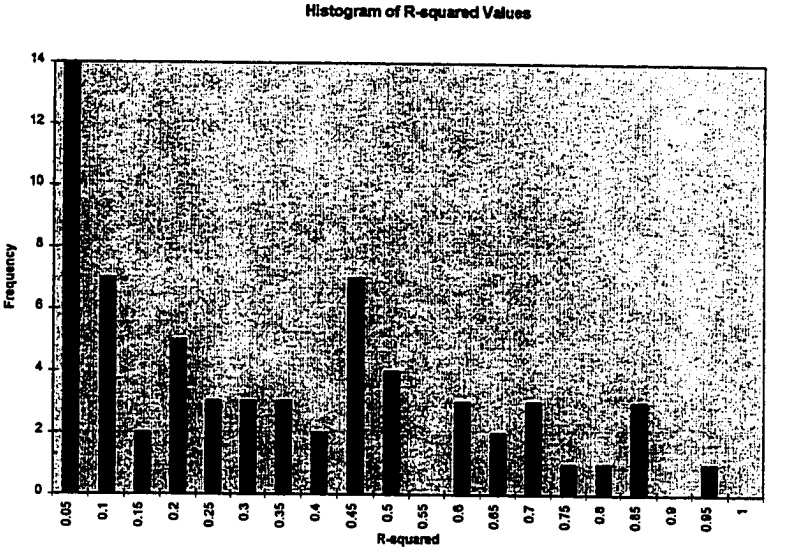

Figure 7.1 Fitness histogram for runtime versus load average plots.

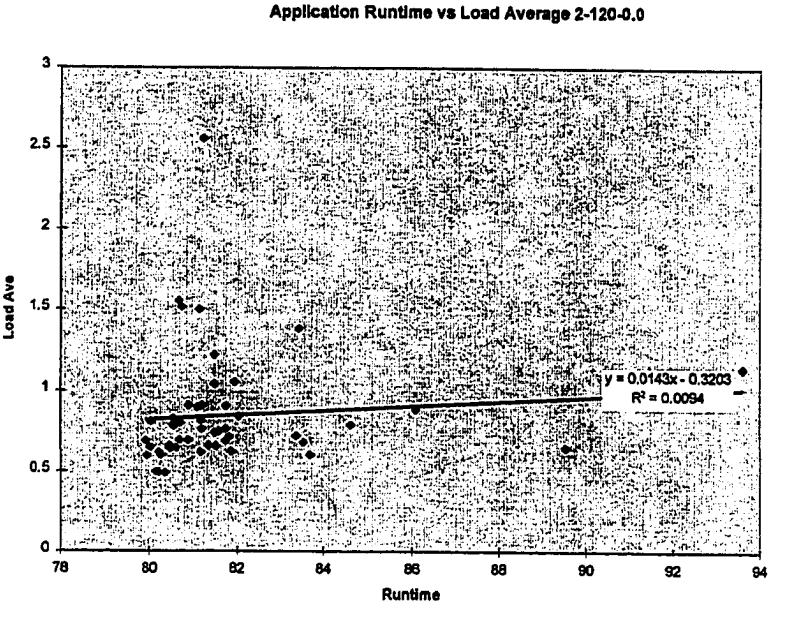

Figure 7.2 Plot of runtime versus load average, showing poorly fitting trendline.

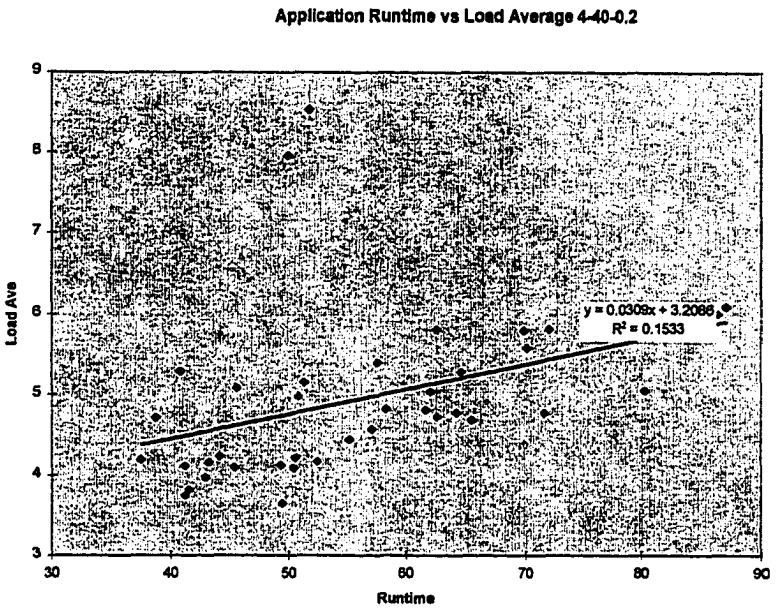

Figure 7.3 Plot of runtime versus load average, showing trendline with moderate fit.

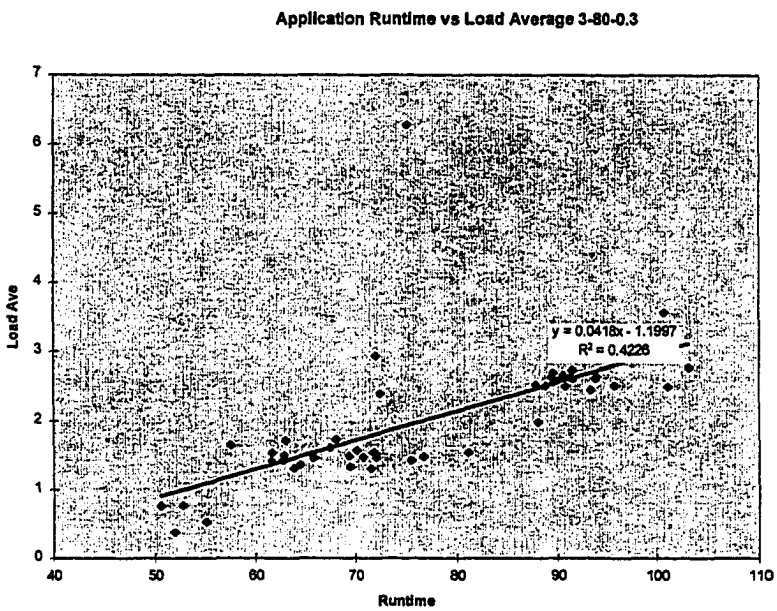

Figure 7.4 Plot of runtime versus load average, showing trendline with good fit.

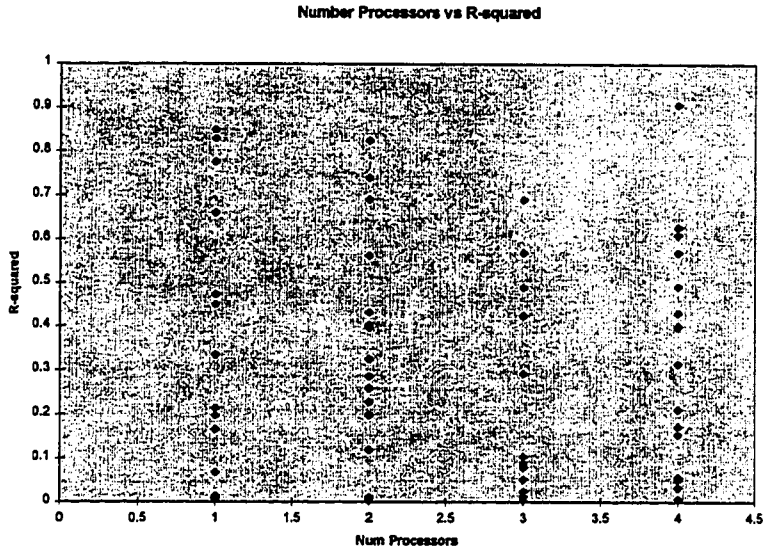

Figure 7.5 Plot showing relation between trendline fit and number of processors for load average plots.

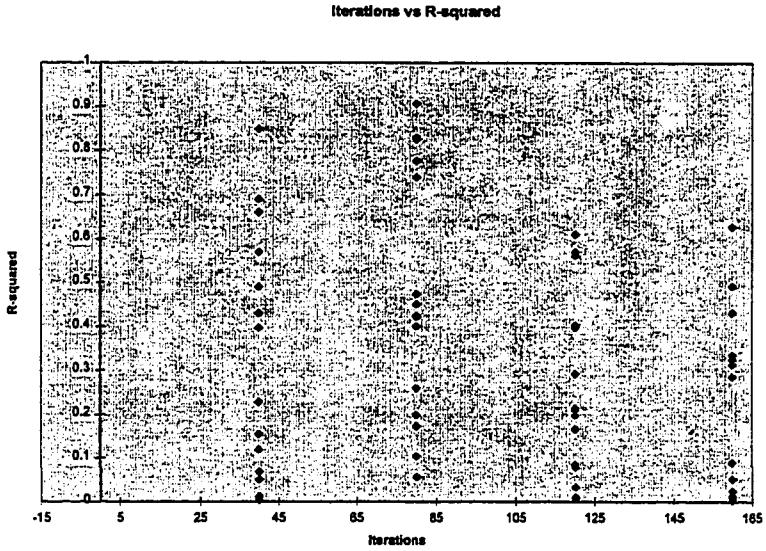

Figure 7.6 Plot showing relation between trendline fit and iterations for load average plots.

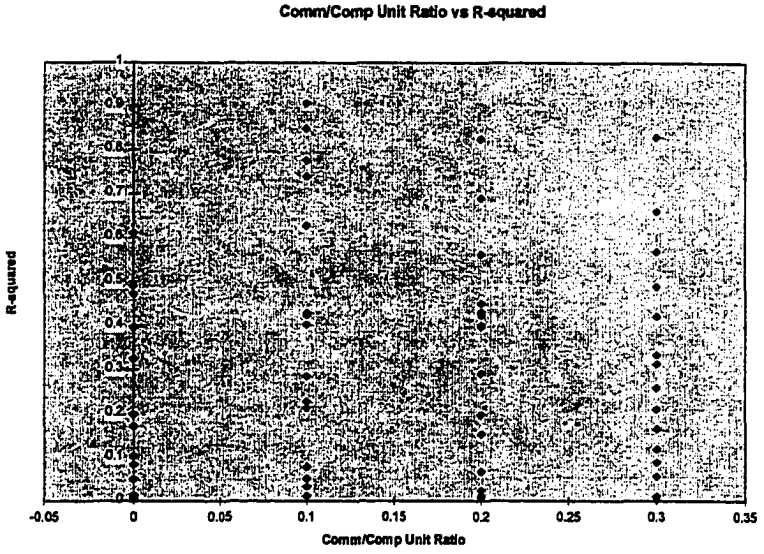

Figure 7.7 Plot showing relation between trendline fit and comm/comp unit ratio for load average plots.

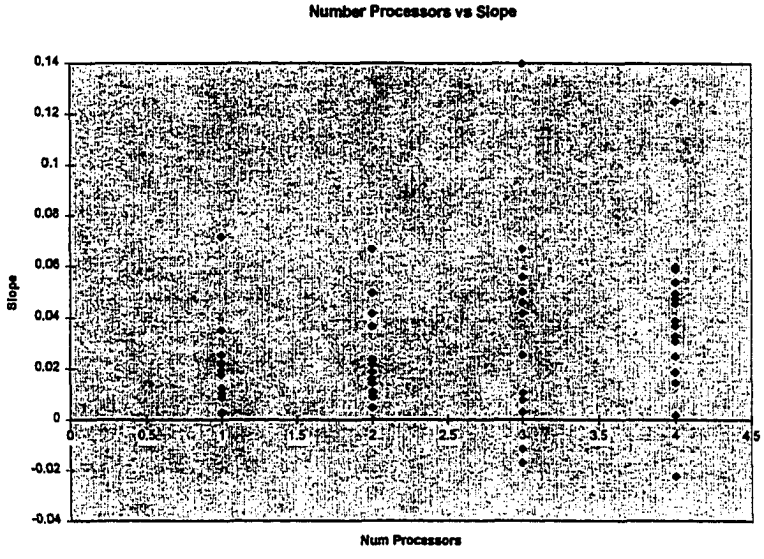

Figure 7.8 Plot showing relation between trendline slope and number of processors for load average plots.

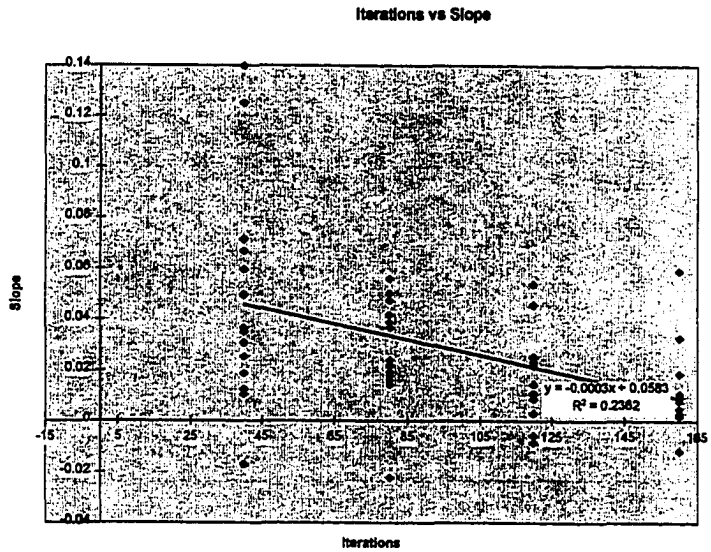

Figure 7.9 Plot showing relation between trendline slope and iterations for load average plots.

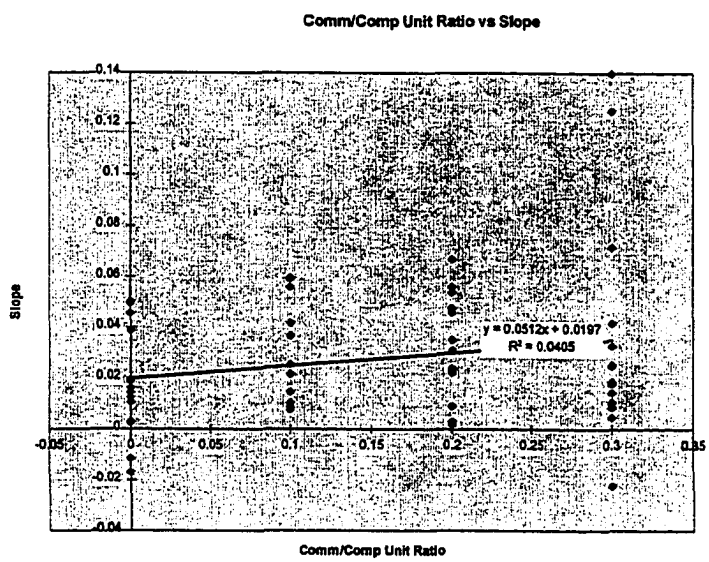

Figure 7.10 Plot showing relation between trendline slope and comm/comp unit ratio for load average plots.

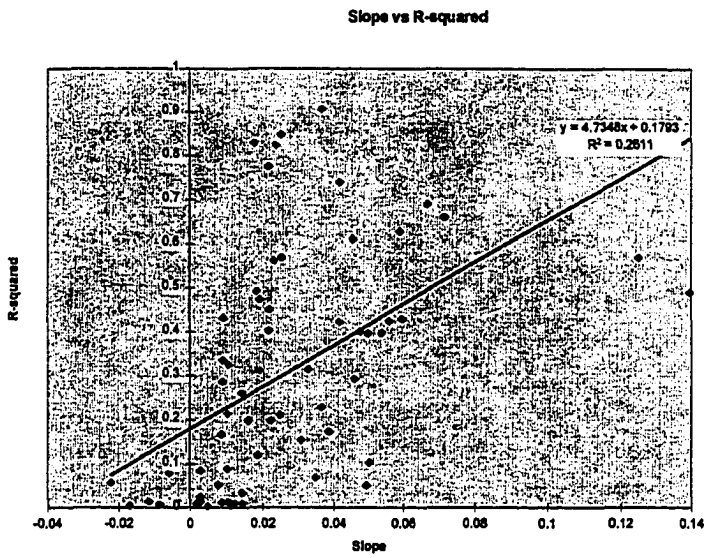

Figure 7.11 Plot of trendline fitness versus slope for load average plots.

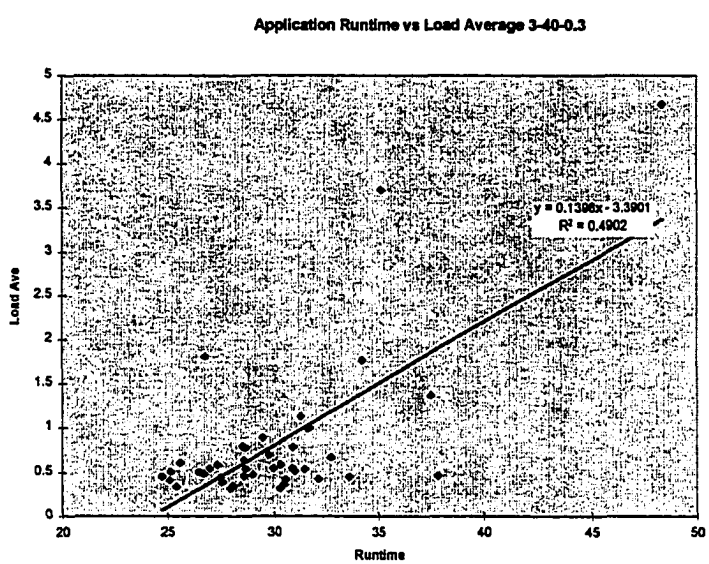

Figure 7.12 Plot of runtime versus load average corresponding to outlier point in Figure 7.11, at R-squared approximately 0.6.

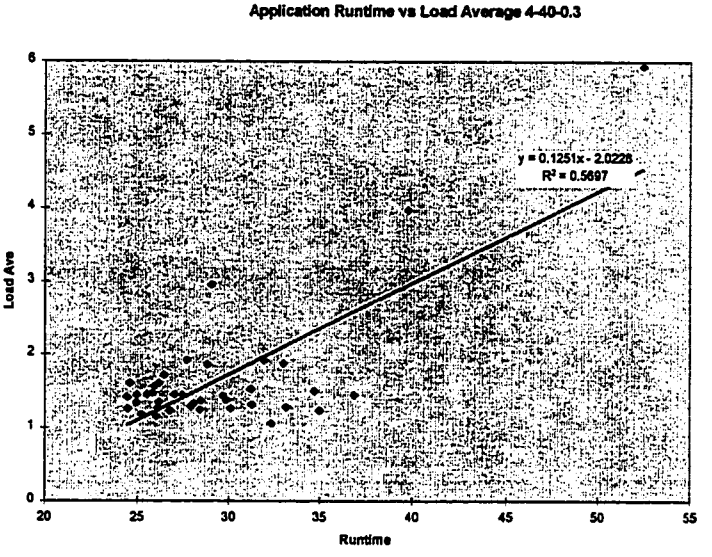

Figure 7.13 Plot of runtime versus load average corresponding to outlier point in Figure 7.11, at R-squared approximately 0.5.

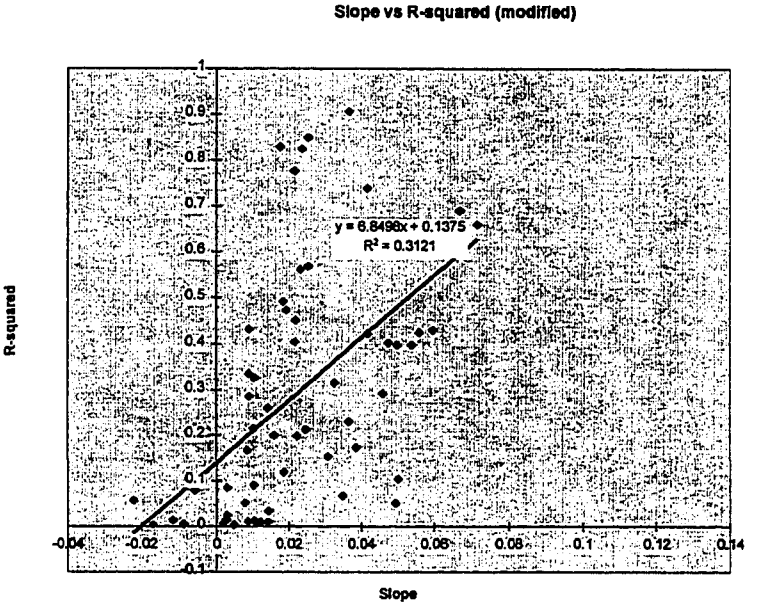

Figure 7.14 Plot of trendlines fitness versus slope for load average plots, with outliers removed and new regression performed.

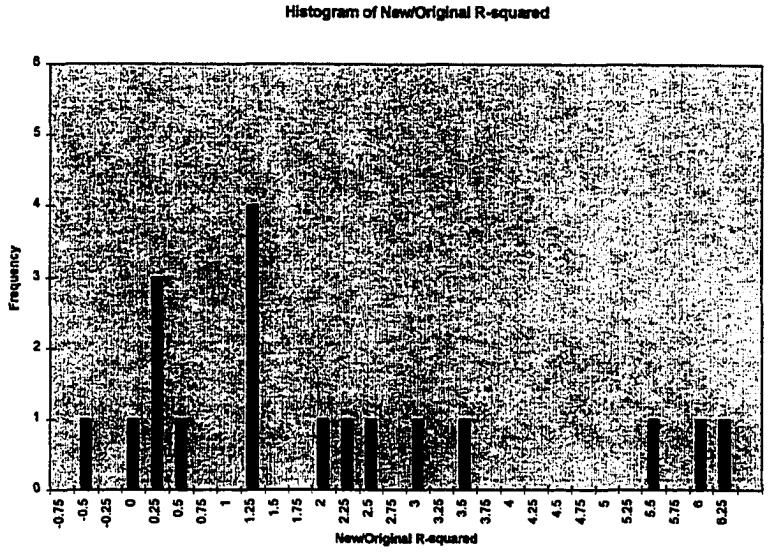

Figure 7.15 Histogram showing ratios of new to old fitness values after outlier point removal. A value above 1.0 indicates improved fit after outlier removal.

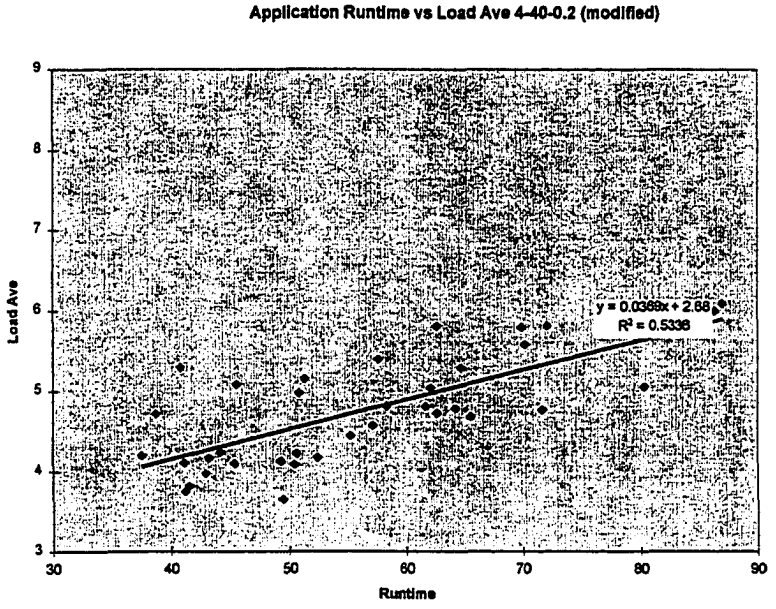

Figure 7.16 Plot shown in Figure 7.3 with new regression after outlier points were removed.

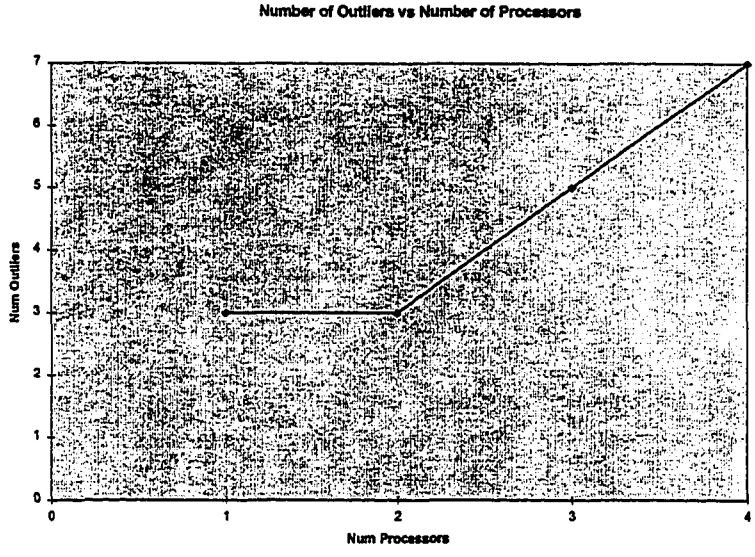

Figure 7.17 Plot showing rise in number of outliers with number of processors, for load average plots.

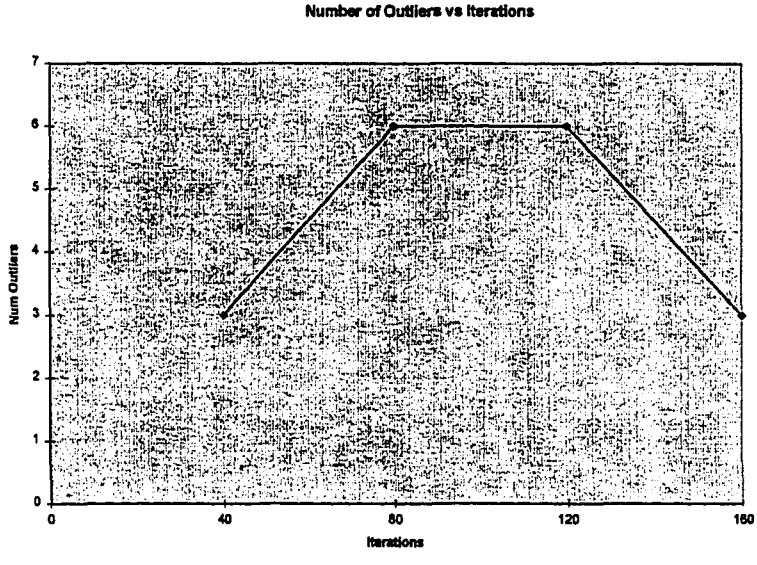

Figure 7.18 Plot showing relation between number of outlier points and number of iterations, for load average plots.

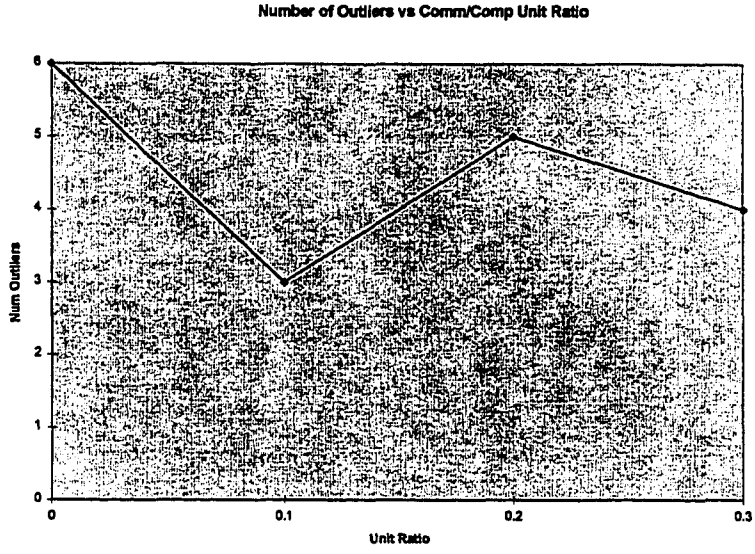

Figure 7.19 Plot showing relation between number of outlier points and comm/comp unit ratio, for load average plots.

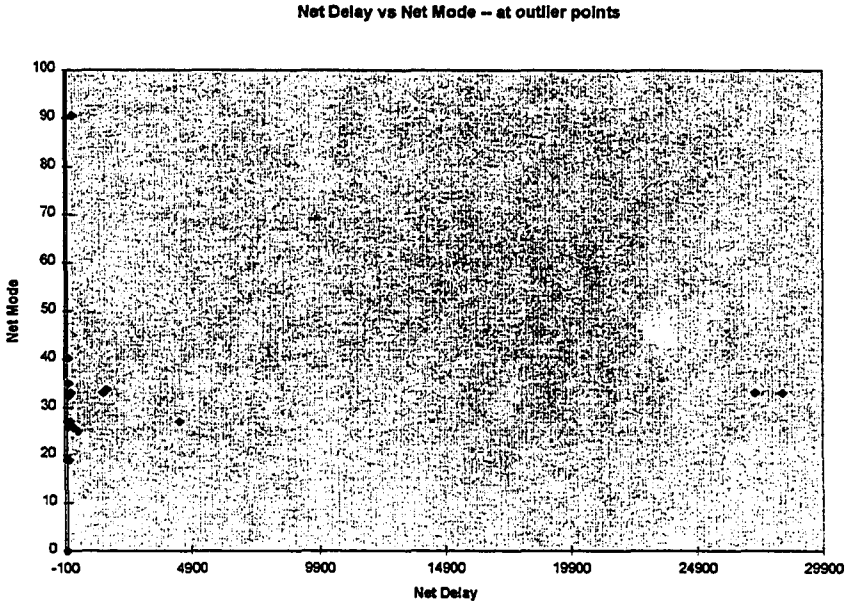

Figure 7.20 Network delay at outlier points versus typical network delay.

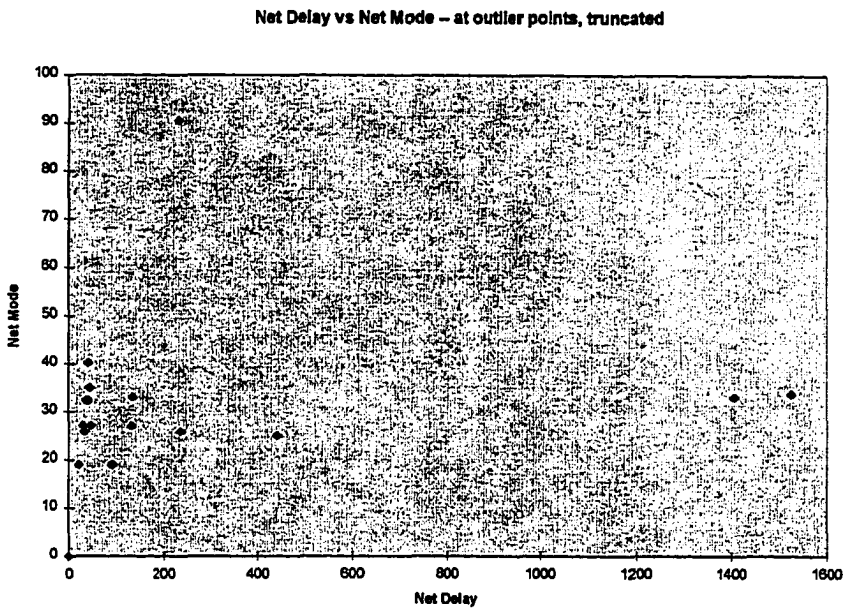

Figure 7.21 Plot of network delay at outliers versus typical network delay, showing lower values only.

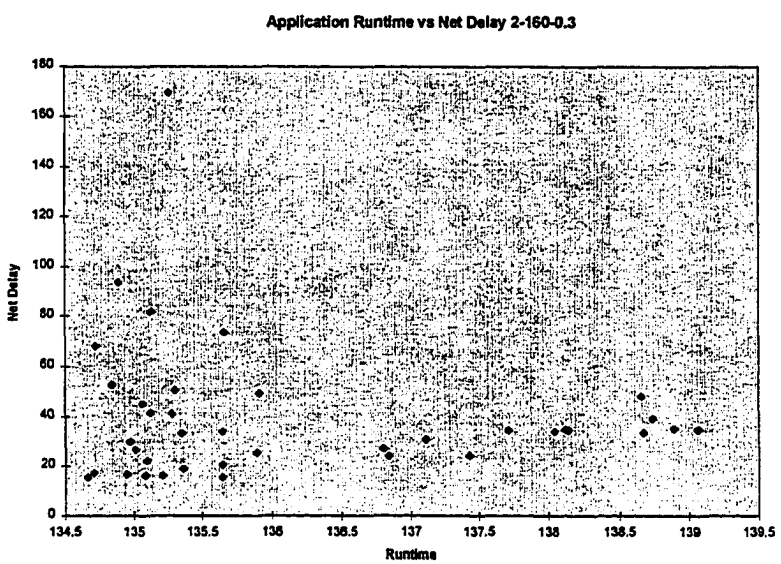

Figure 7.22 Plot of runtime versus network delay, showing typical "L" configuration.

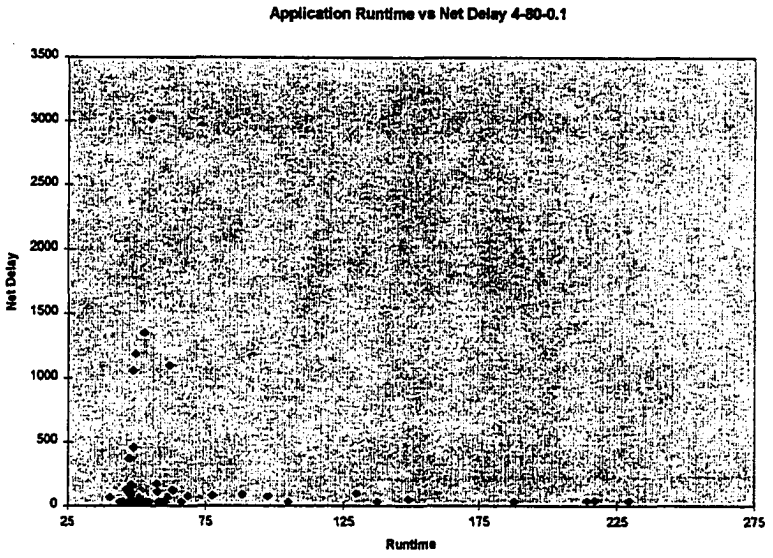

Figure 7.23 Another plot of runtime versus network delay. "L" shape is more pronounced here.

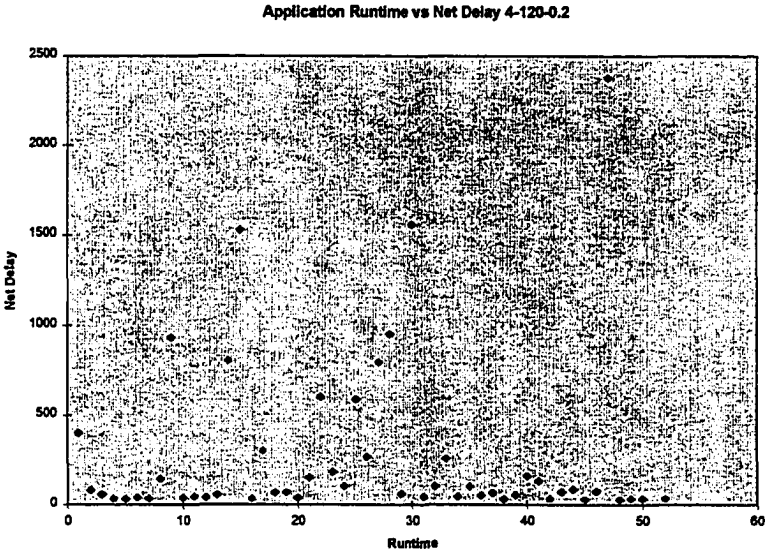

Figure 7.24 Plot of runtime versus network delay with a wider distribution of points, probably due to increased local load.

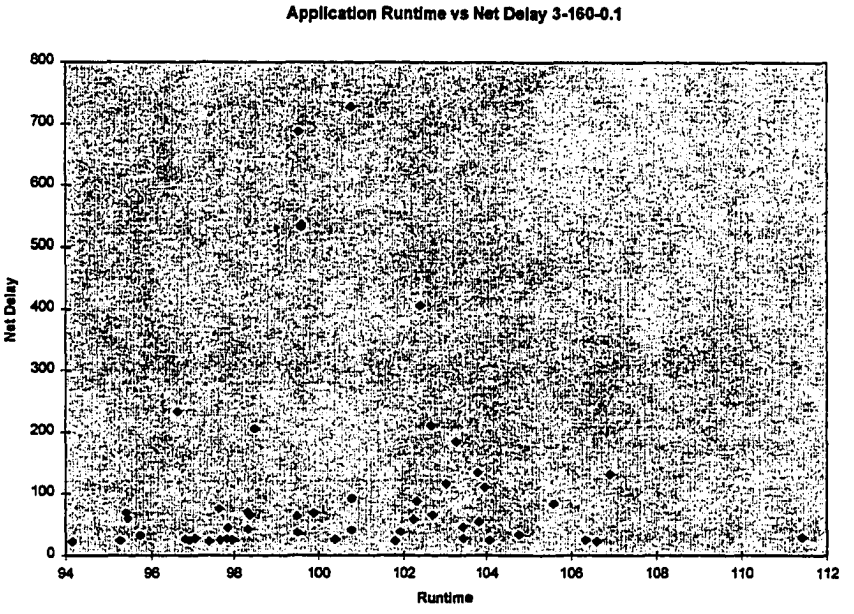

Figure 7.25 Another plot showing runtime versus network delay under increased local load.

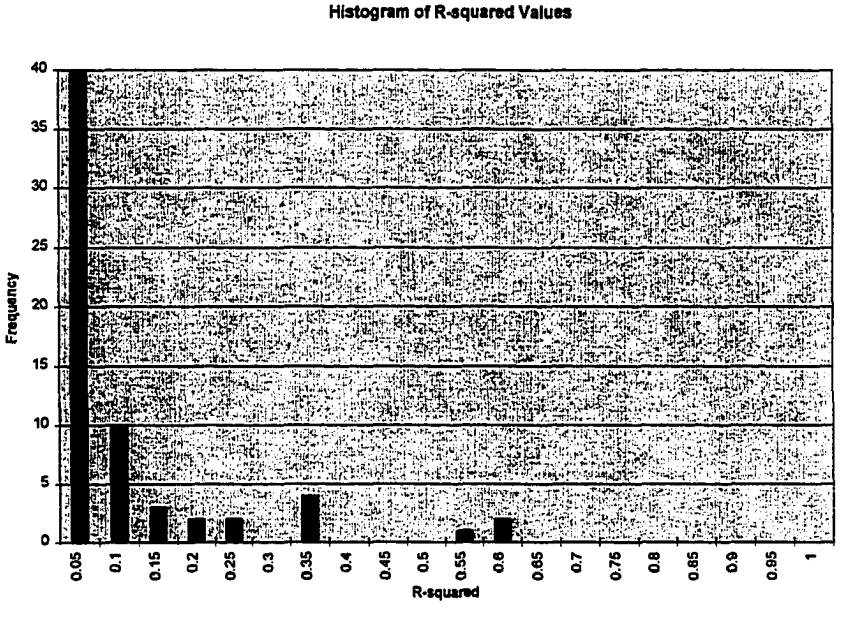

Figure 7.26 Histogram of trendline fit values for benchmark load measure plots.

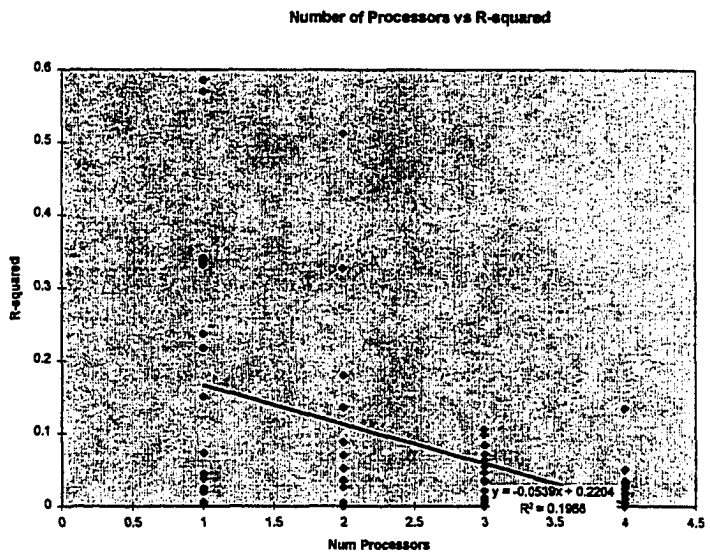

Figure 7.27 Plot showing relation between trendline fitness and number of processors, for benchmark load plots.

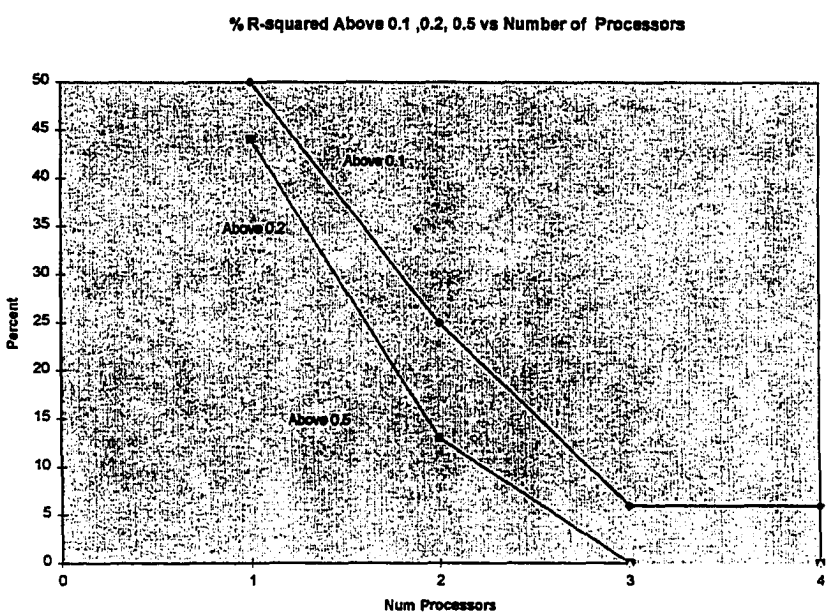

Figure 7.28 Percentage of trendlines fitness values above thresholds of 0.1, 0.2 and 0.5, for benchmark load plots.

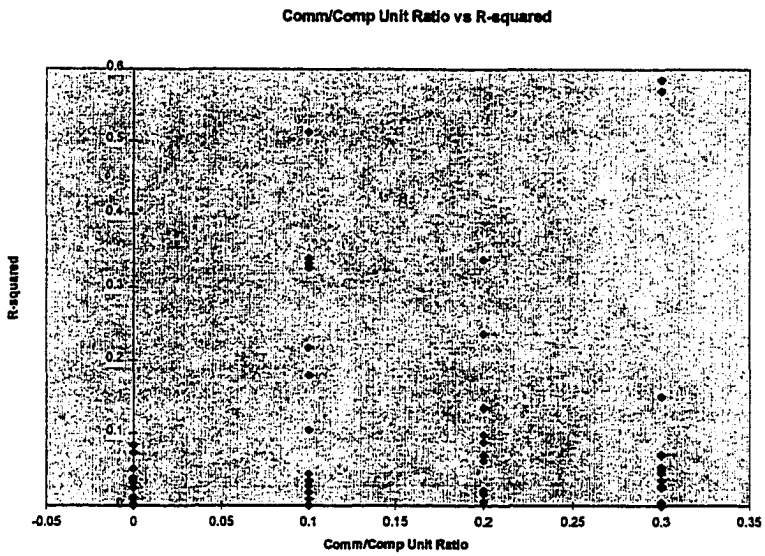

Figure 7.29 Plot showing relation between trendline fit and Comm/Comp unit ratio, for benchmark load plots.

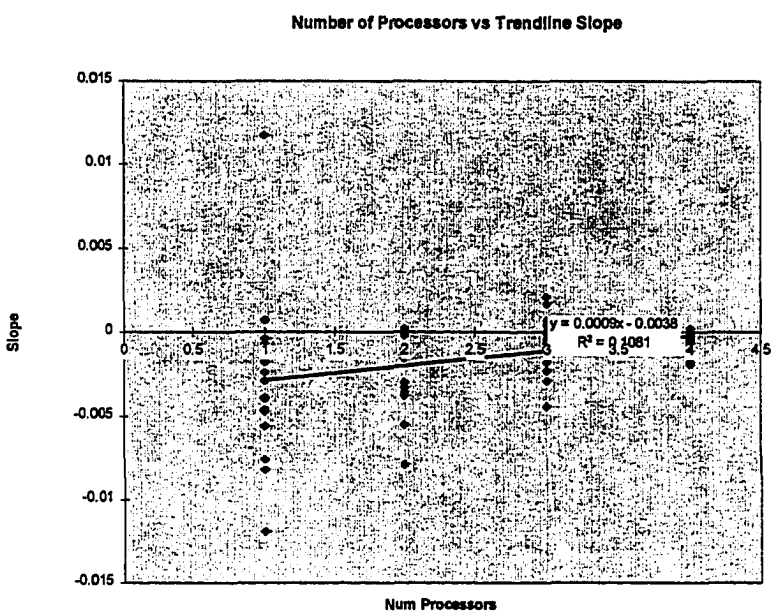

Figure 7.30 Plot showing relation between trendline slope and number of processors, for benchmark load plots.

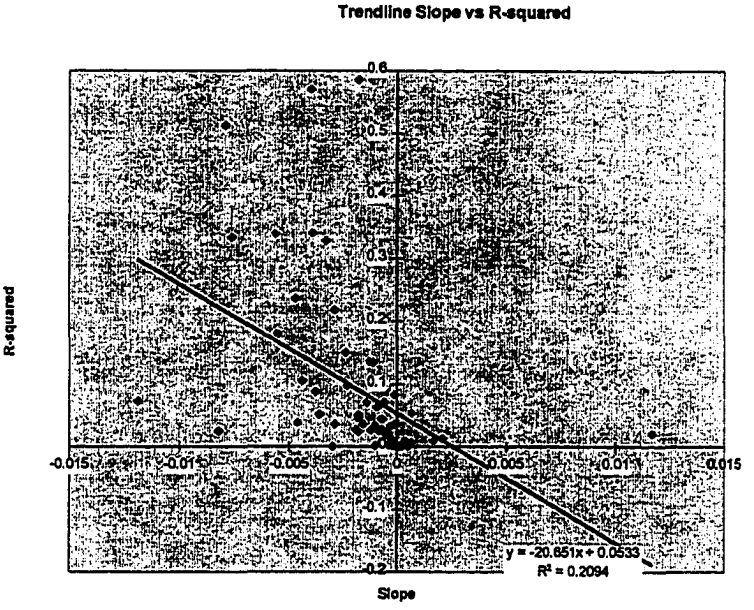

Figure 7.31 Plot of trendline fit versus slope, for benchmark load plots.

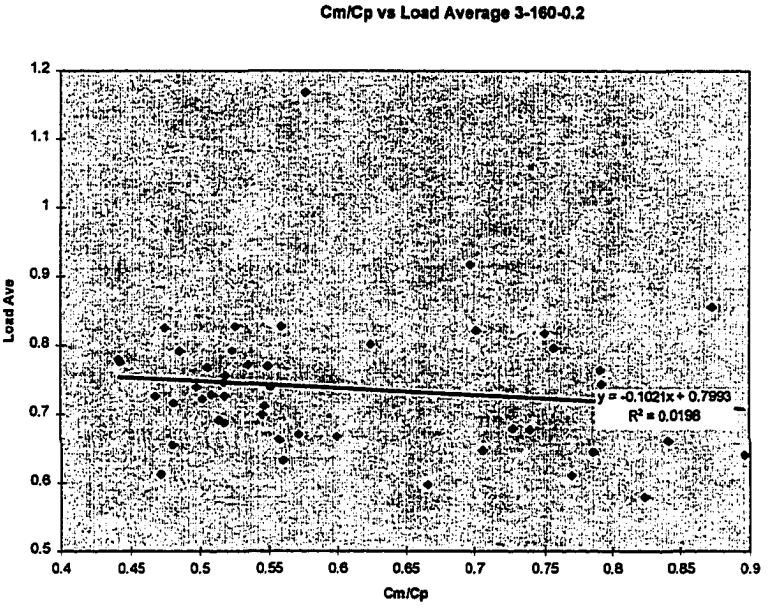

Figure 7.32 Plot of Comm/Comp time ratio versus worst load average.

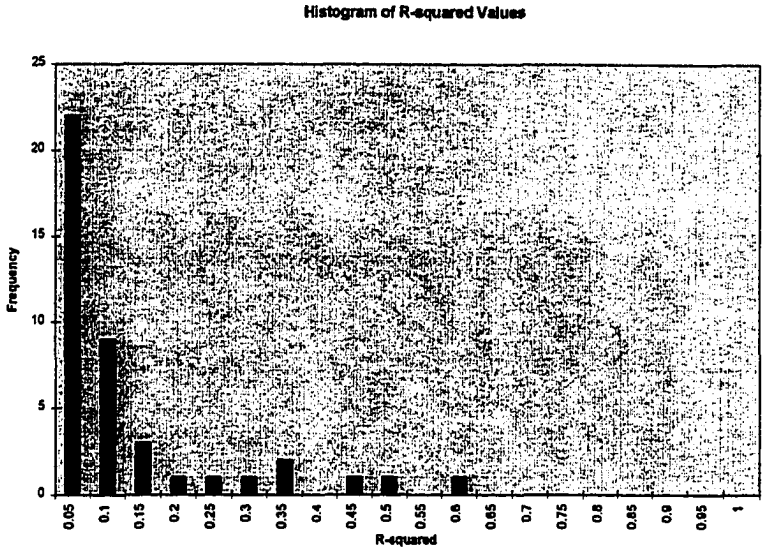

Figure 7.33 Histogram of trendline fit values for plots of comm/comp time ratio versus worst load average.

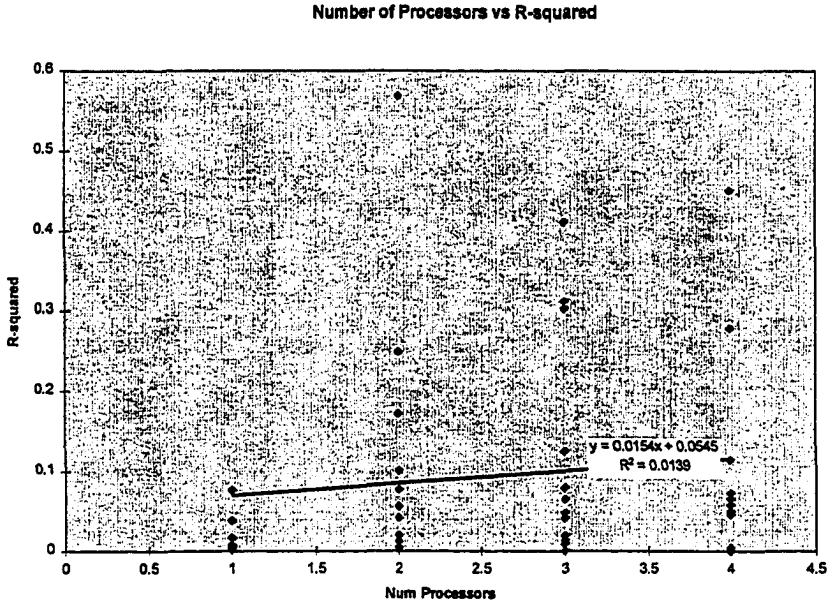

Figure 7.34 Plot showing relation between trendline fit and number of processors, for comm/comp time ratio plots.

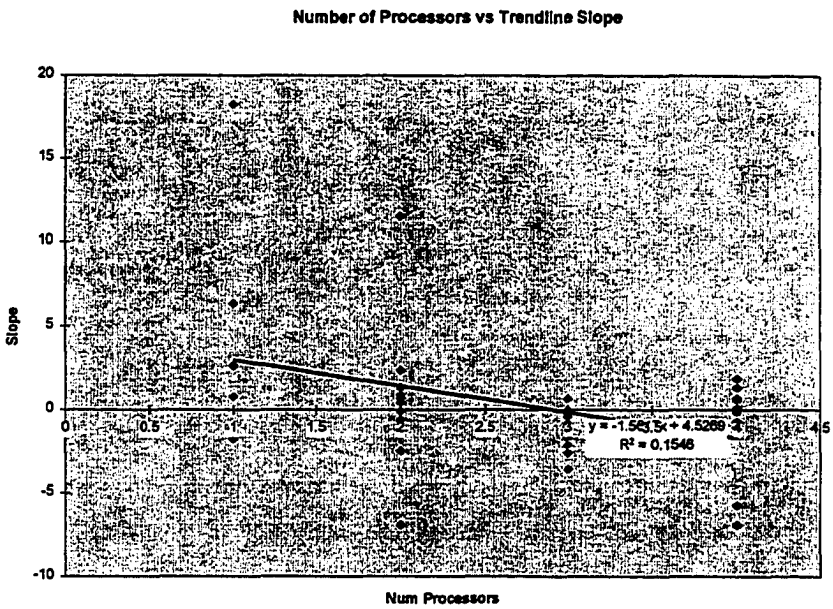

Figure 7.35 Plot of trendline plot versus number of processors, for comm/comp time ratio plots.

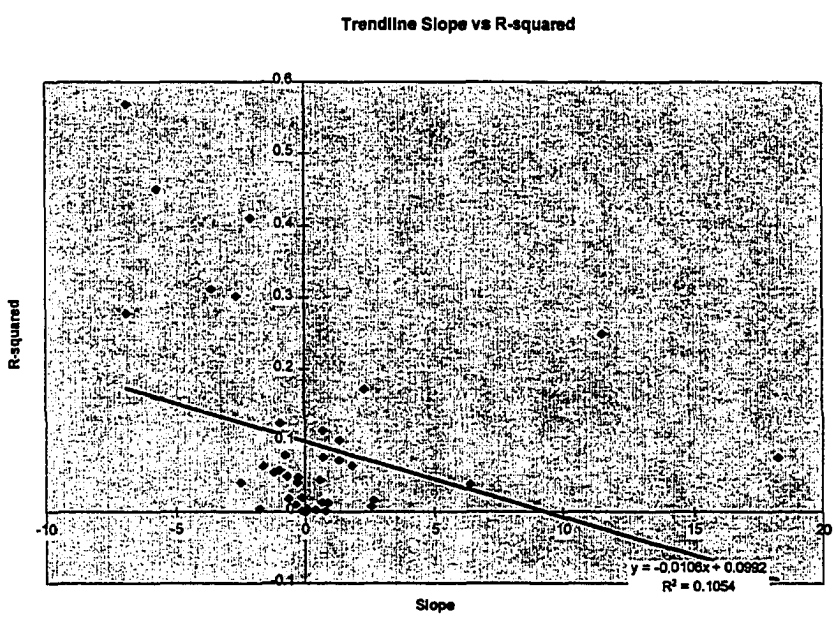

Figure 7.36 Plot of trendline fit versus slope, for comm/comp time ratio plots.

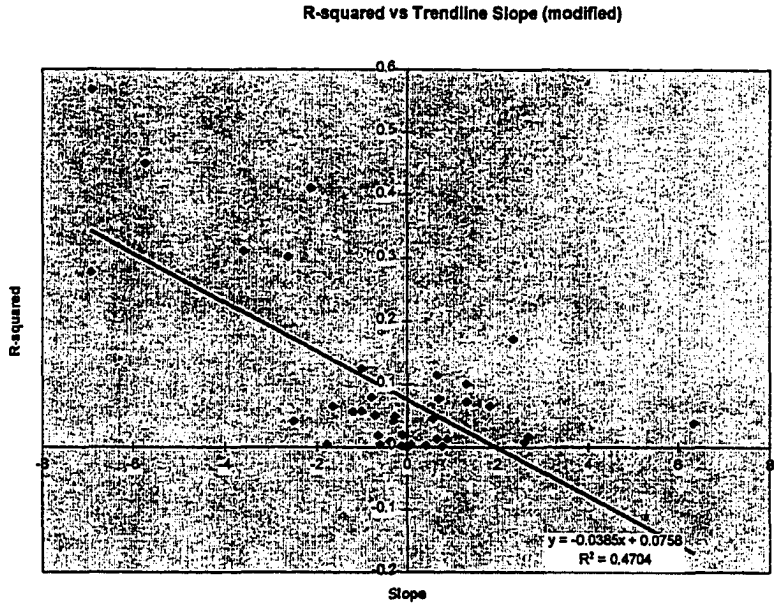

Figure 7.37 Same plot as above, with new regression after removal of two outlier points.

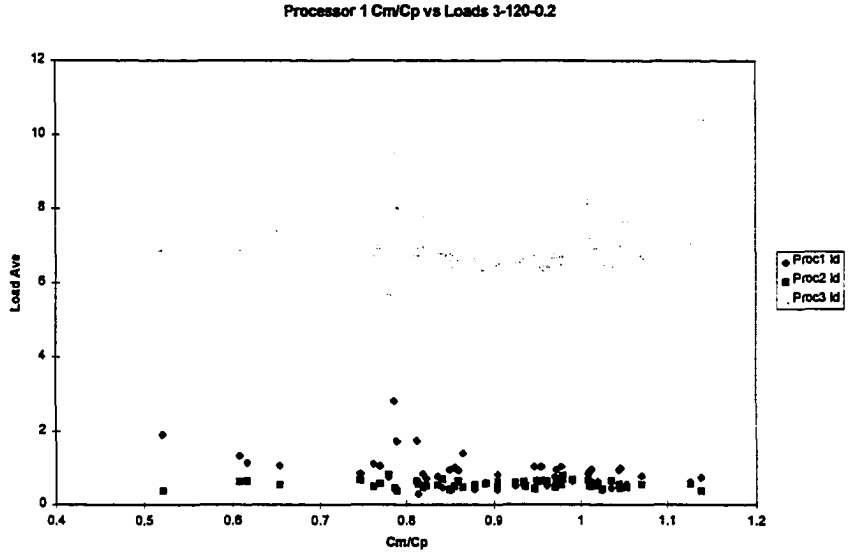

Figure 7.38 Plot showing load average versus comm/comp time ratio during a three processor run, showing processor 3 dominant and points clustered.

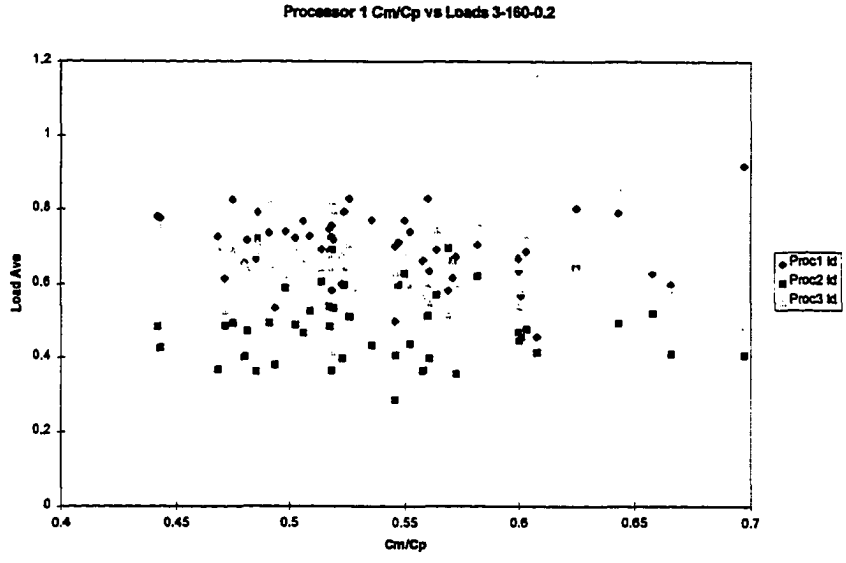

Figure 7.39 Plot showing load average versus comm/comp time ratio during three processor run, with no single processor dominant and points widely spread.

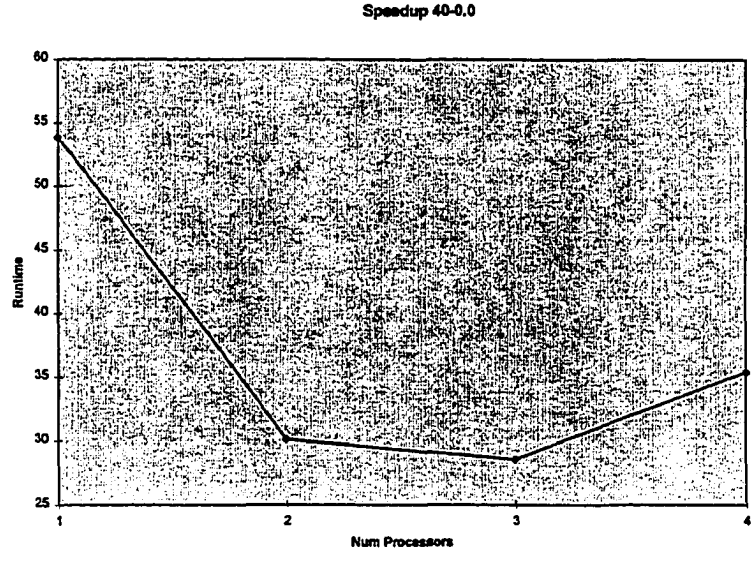

Figure 7.40 Plot showing speedup for program run with 40 iterations and unit ratio of 0.0.

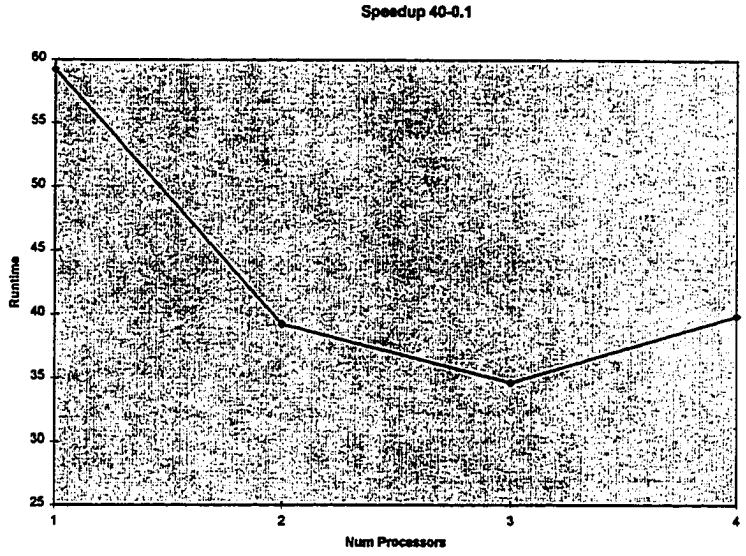

Figure 7.41 Plot showing speedup for program run with 40 iterations and unit ratio of 0.1.

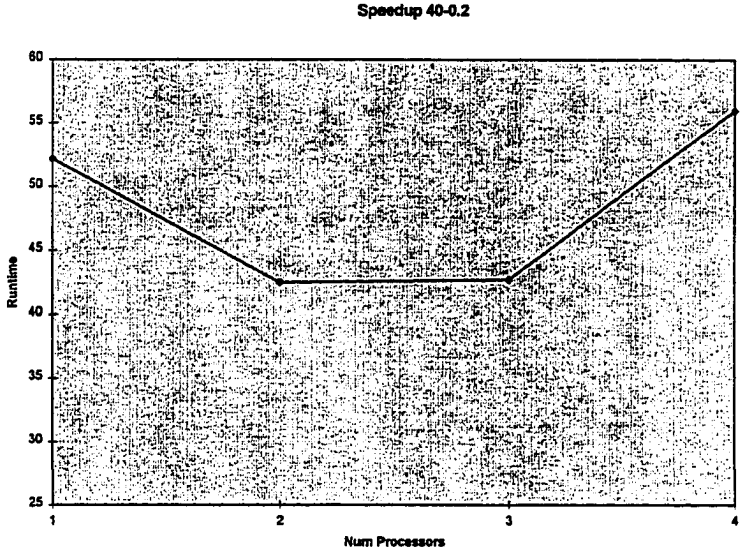

Figure 7.42 Plot showing speedup for program run with 40 iterations and unit ratio of 0.2.

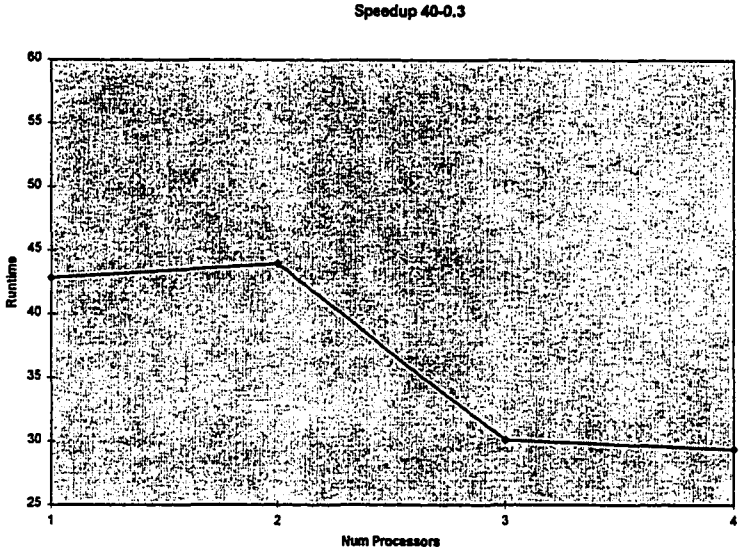

Figure 7.43 Plot showing speedup for program run with 40 iterations and unit ratio of 0.3.

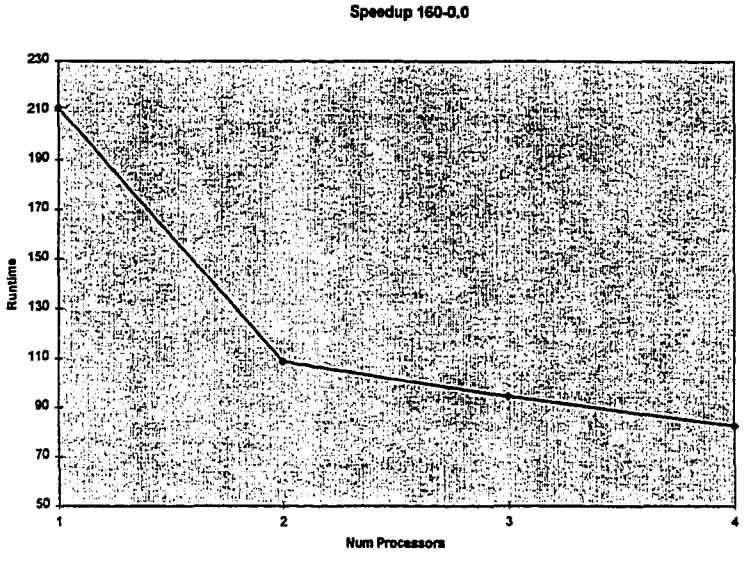

Figure 7.44 Plot showing speedup for program run with 160 iterations and unit ratio of 0.0.

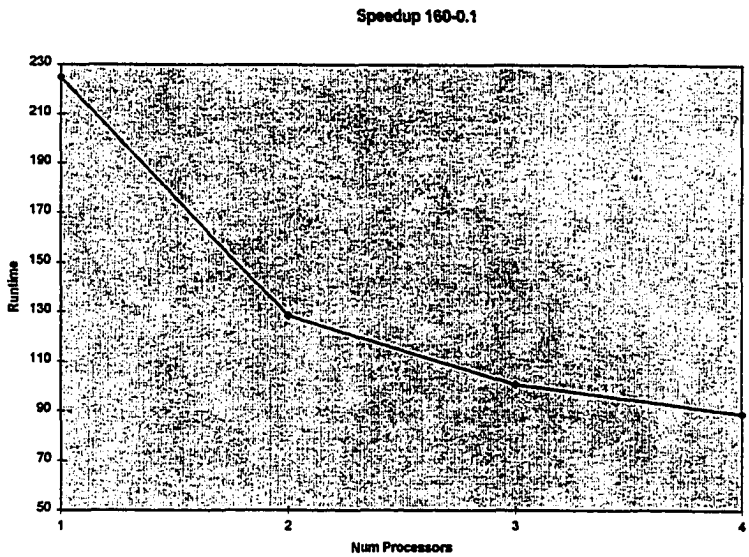

Figure 7.45 Plot showing speedup for program run with 160 iterations and unit ratio of 0.1.

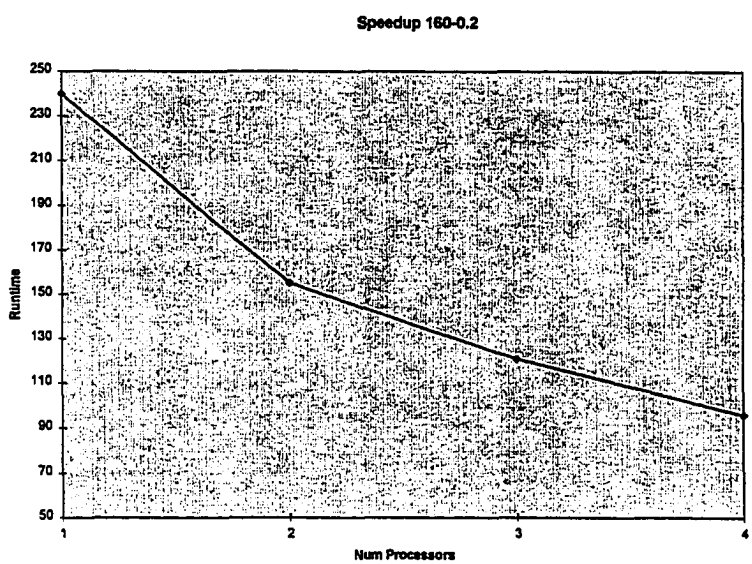

Figure 7.46 Plot showing speedup for program run with 160 iterations and unit ratio of 0.2.

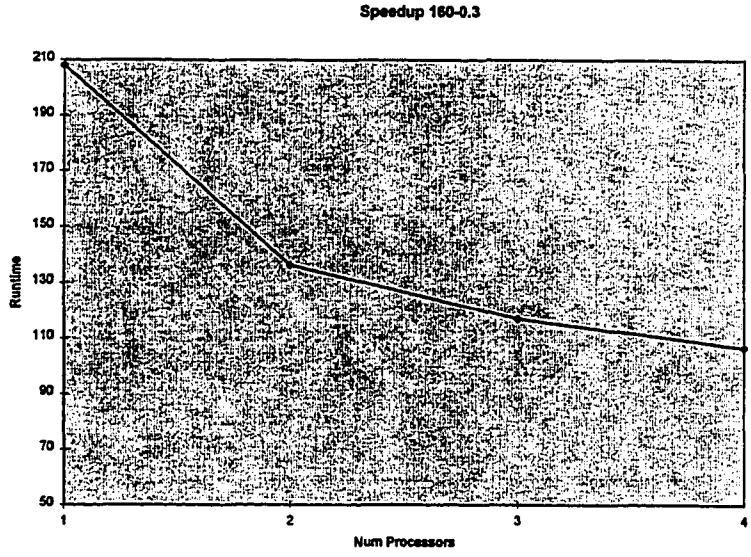

Figure 7.47 Plot showing speedup for program run with 160 iterations and unit ratio of 0.3.

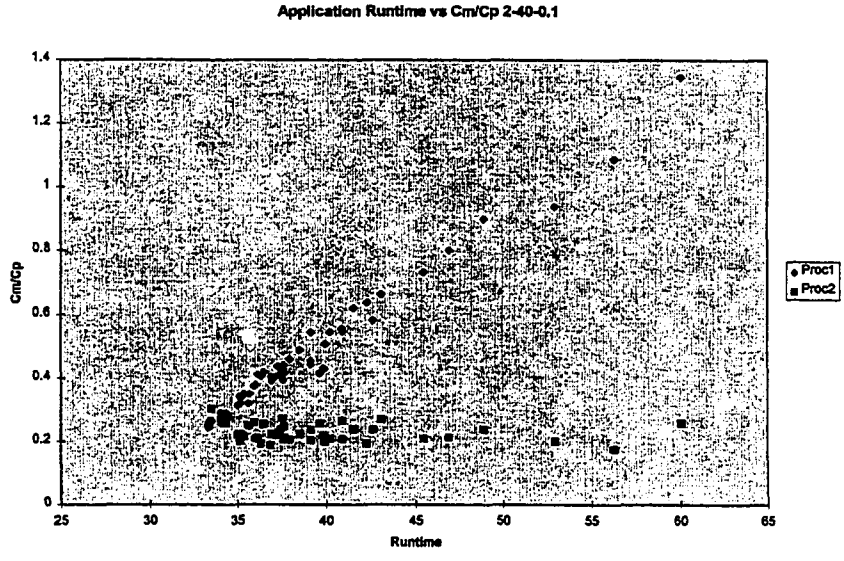

Figure 7.48 Typical plot of comm/comp time ratio versus application runtime, for two processors.

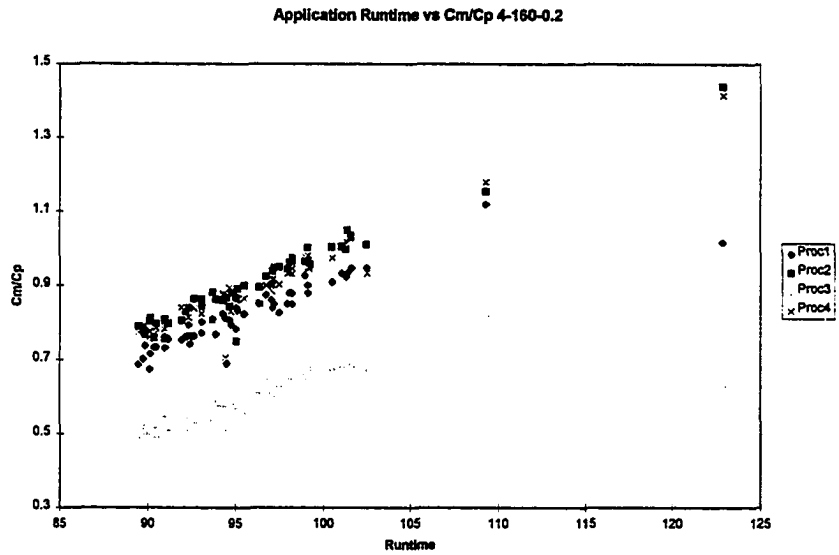

Figure 7.49 Plot of comm/comp time ratio for four processors, showing well defined trends for each processor.

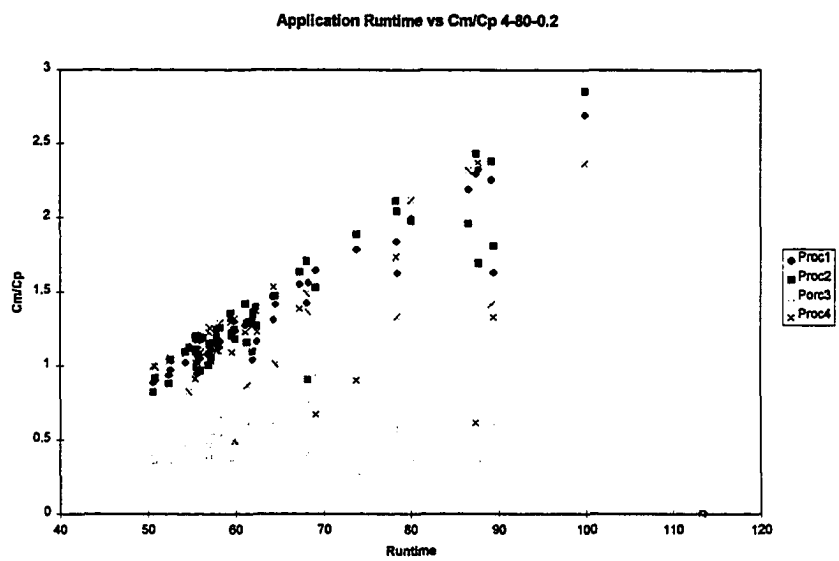

Figure 7.50 Plot of comm/comp time ratio on four processors, showing clustering on two processors and spreading on two processors.

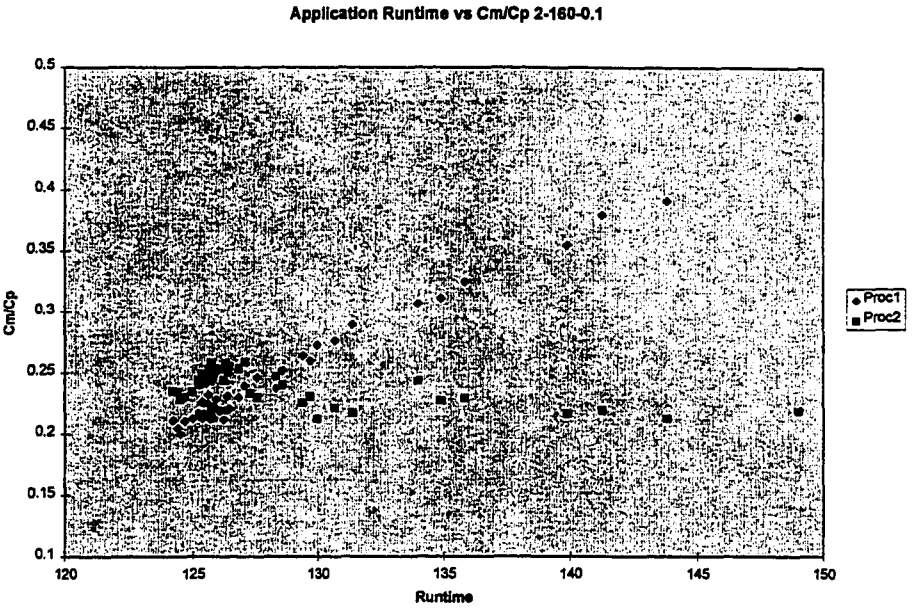

Figure 7.51 Crossing comm/comp time ratio trends.

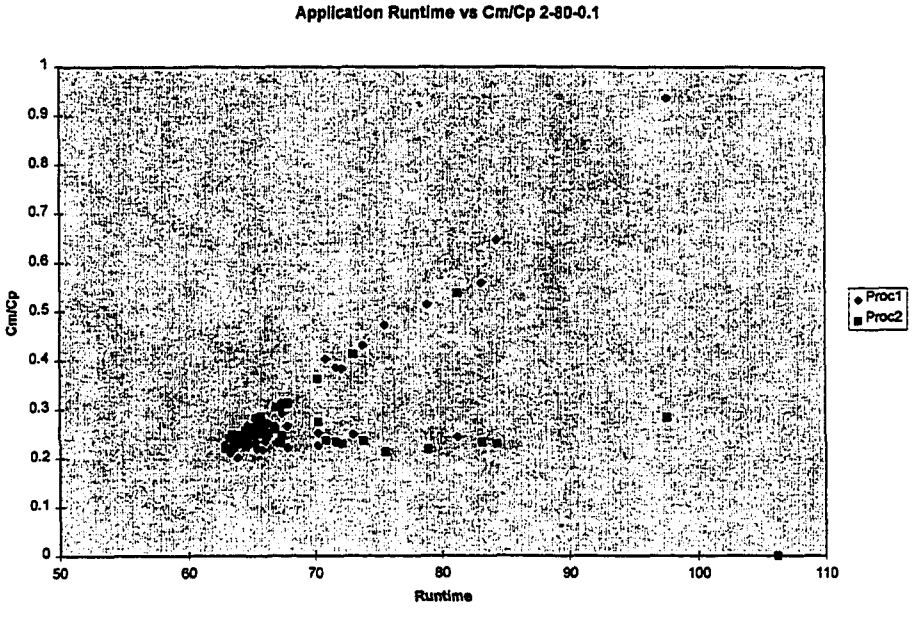

Figure 7.52 Intermingled comm/comp ratio trends on two processors.

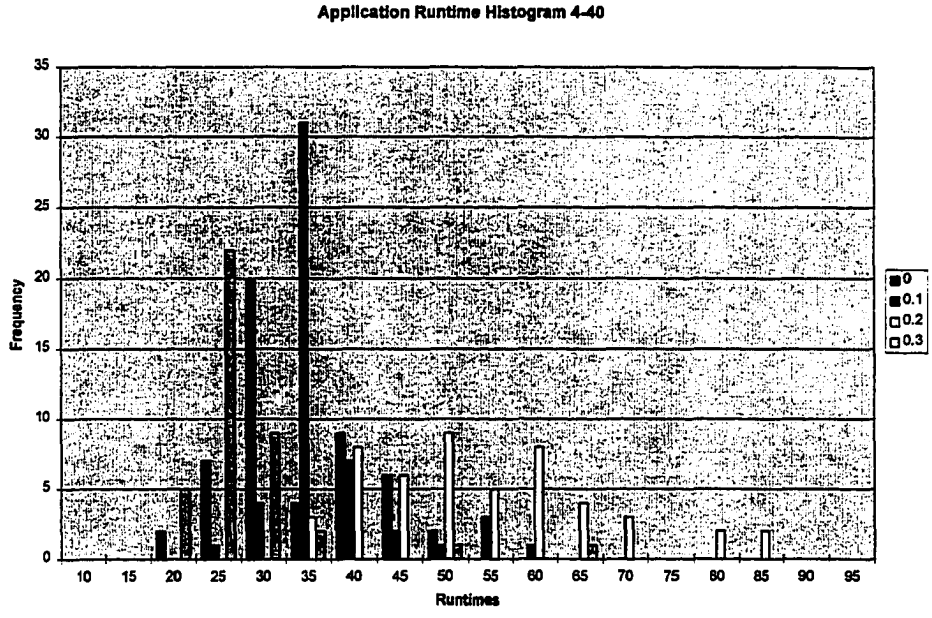

Figure 7.53 Histogram of program runtimes on four processors, with 40 iterations.

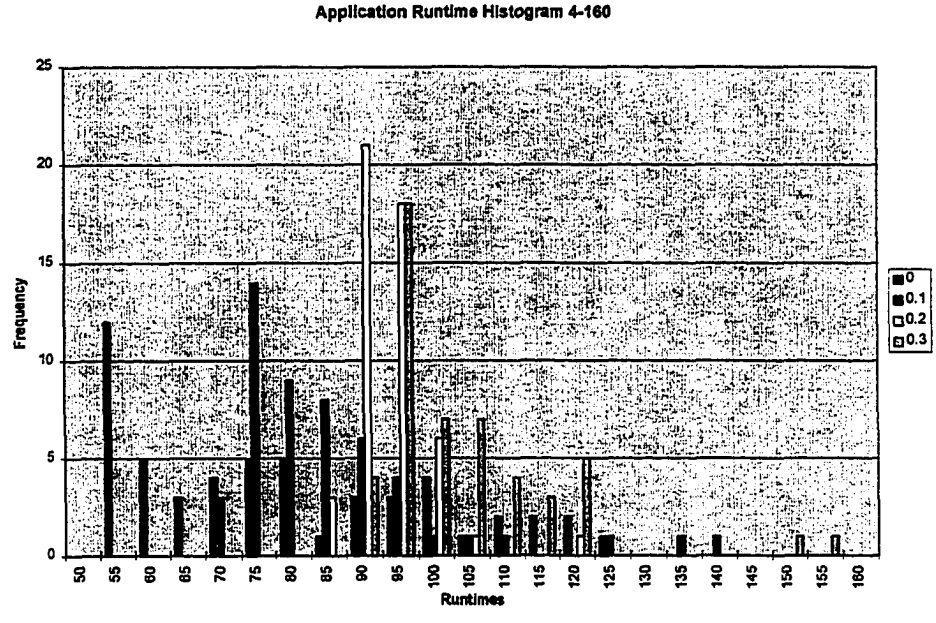

Figure 7.54 Histogram of runtimes for 4 processors, 160 iterations.

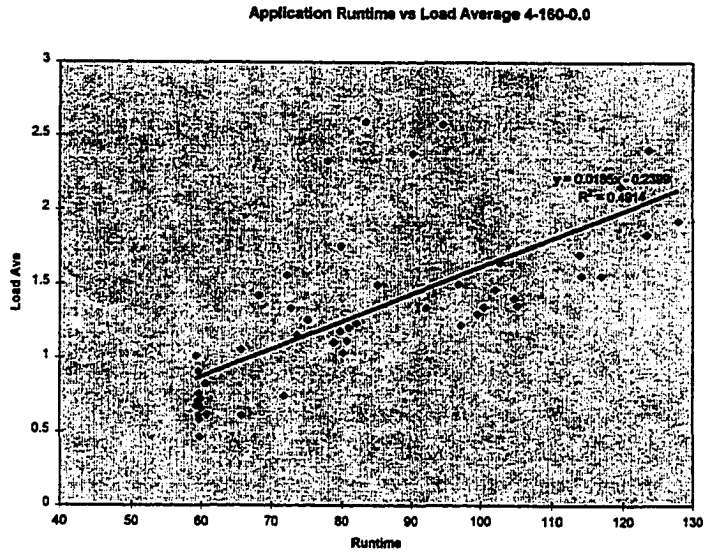

Figure 7.55 Plot of runtime versus load average for run in Figure 7.54 with comm/comp unit ratio of 0.0.

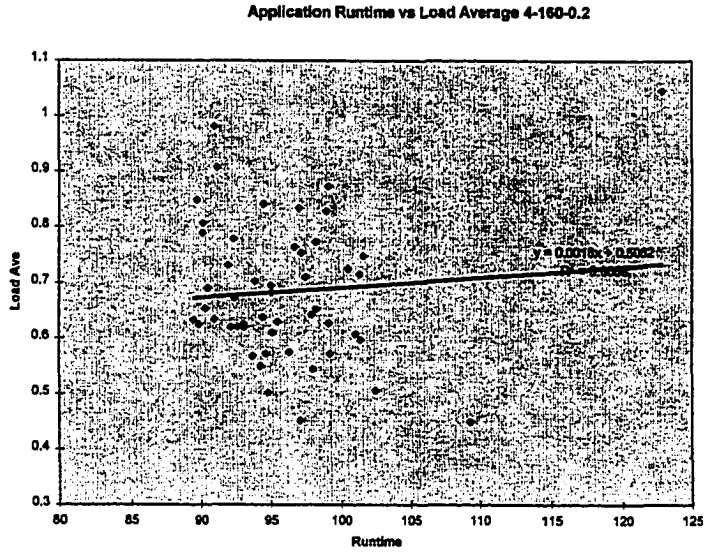

Figure 7.56 Plot of runtime versus load average for run in Figure 7.54 with comm/comp unit ratio of 0.2.

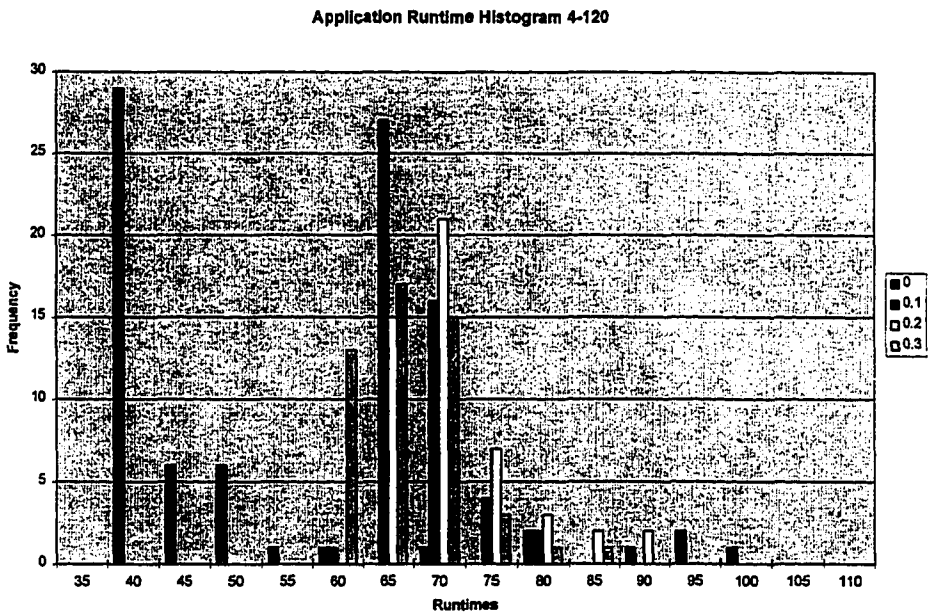

Figure 7.57 Histogram of runtimes, showing clustering of runs with communication.

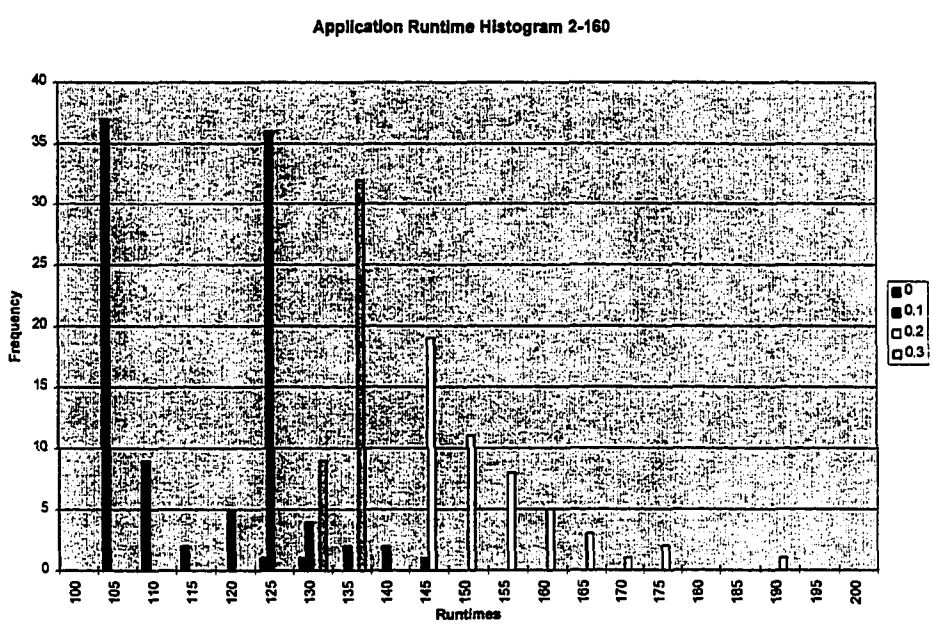

Figure 7.58 Histogram of runtimes, showing seperation of runs with comm/comp unit ratios of 0.2.

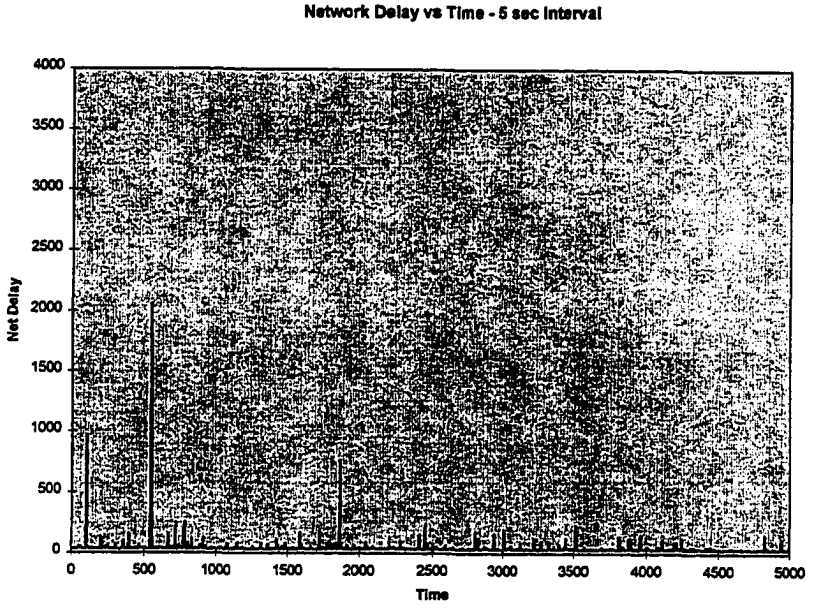

Figure 7.59 Plot of network delays sampled every 5 seconds.

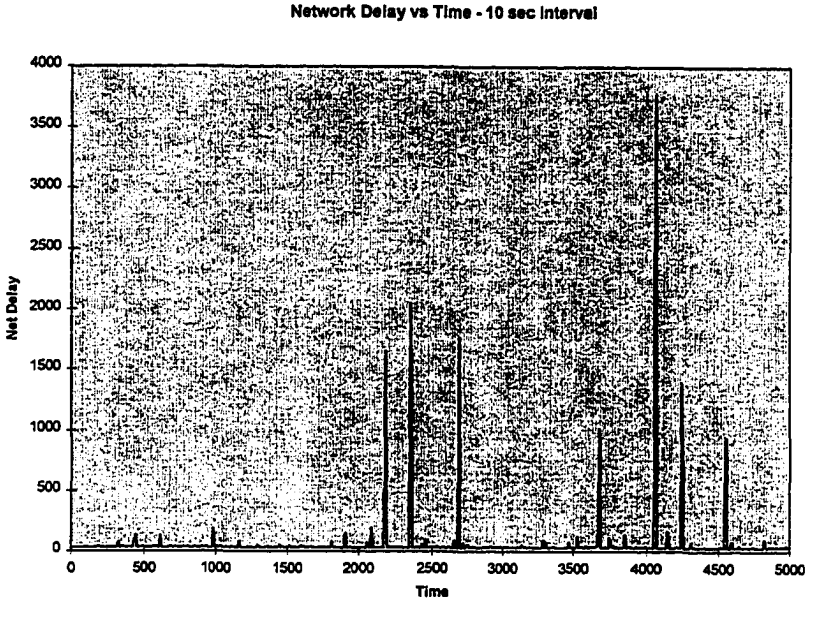

Figure 7.60 Plot of network delays sampled every 10 seconds.

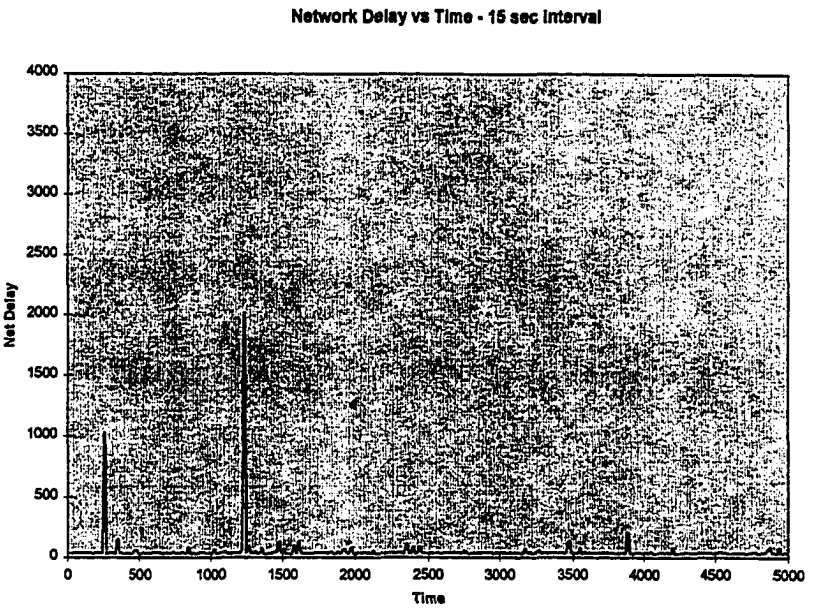

Figure 7.61 Plot of network delays sampled every 15 seconds.

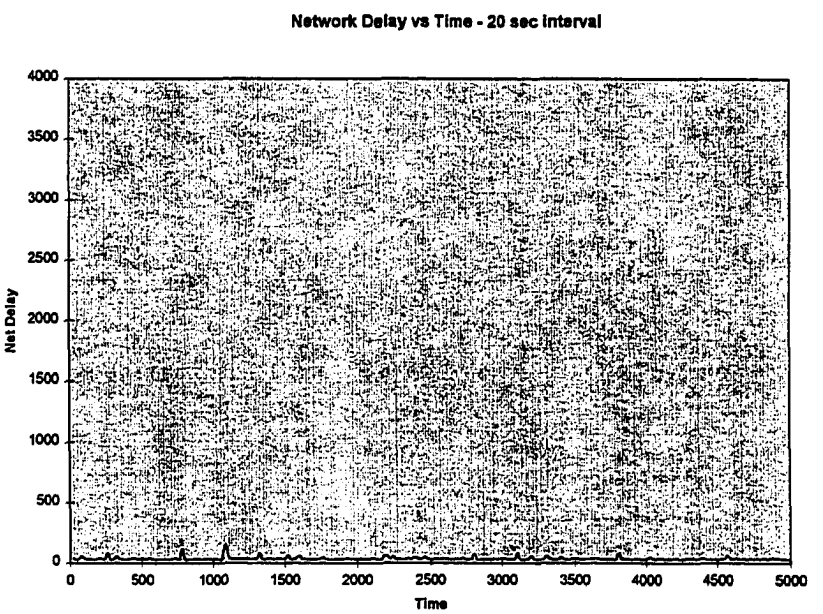

Figure 7.62 Plot of network delays sampled every 20 seconds.

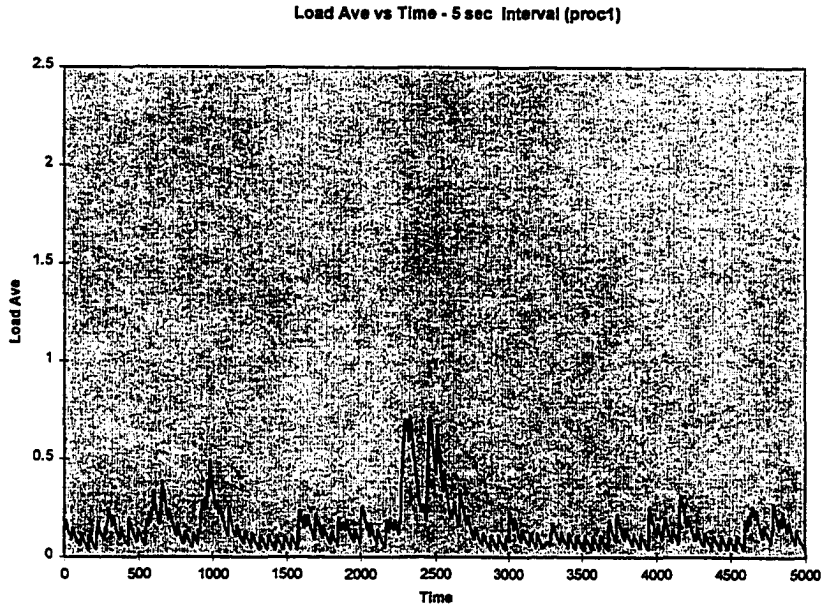

Figure 7.63 Plot of load average sampled every 5 seconds.

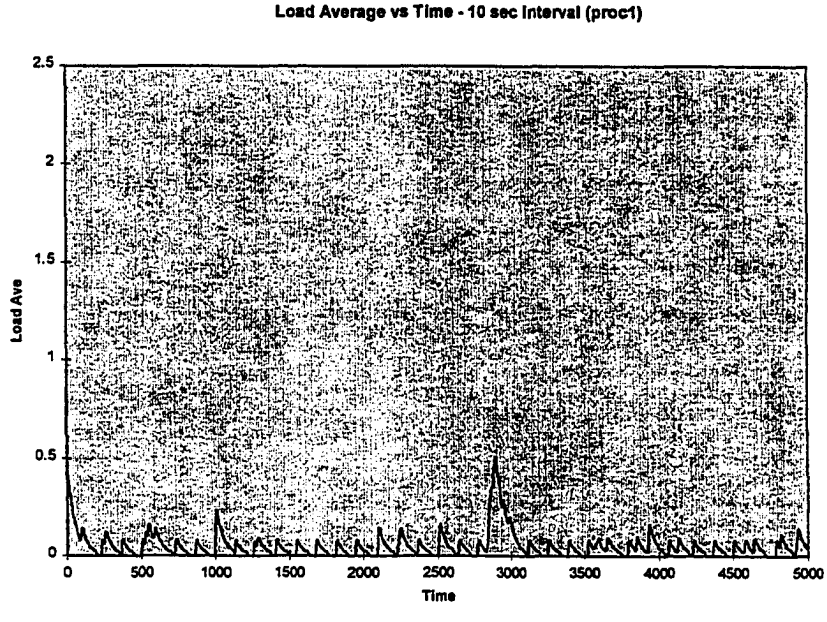

Figure 7.64 Plot of load average sampled every 10 seconds.

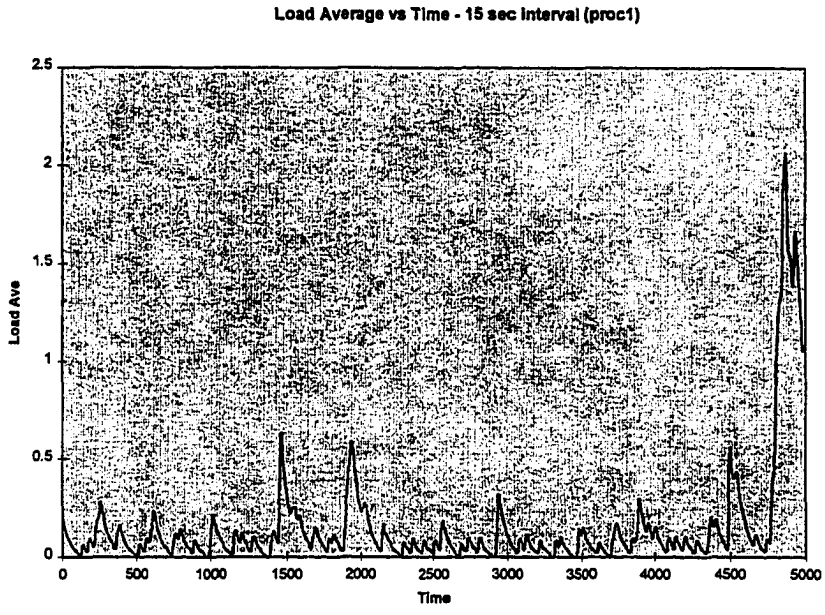

Figure 7.65 Plot of load average sampled every 15 seconds.

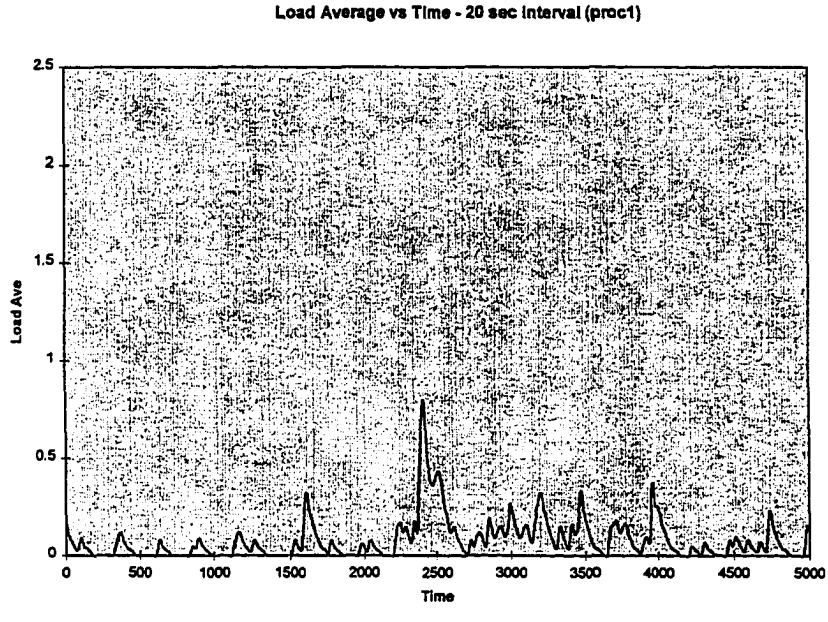

Figure 7.66 Plot of load average sampled every 20 seconds.

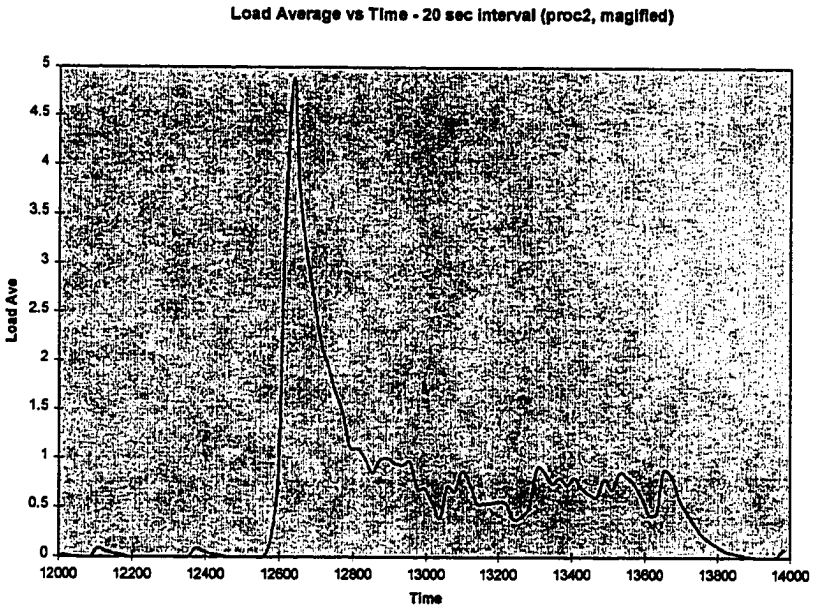

Figure 7.67 Plot of load average sampled every 20 seconds, magnified relative to previous plots.

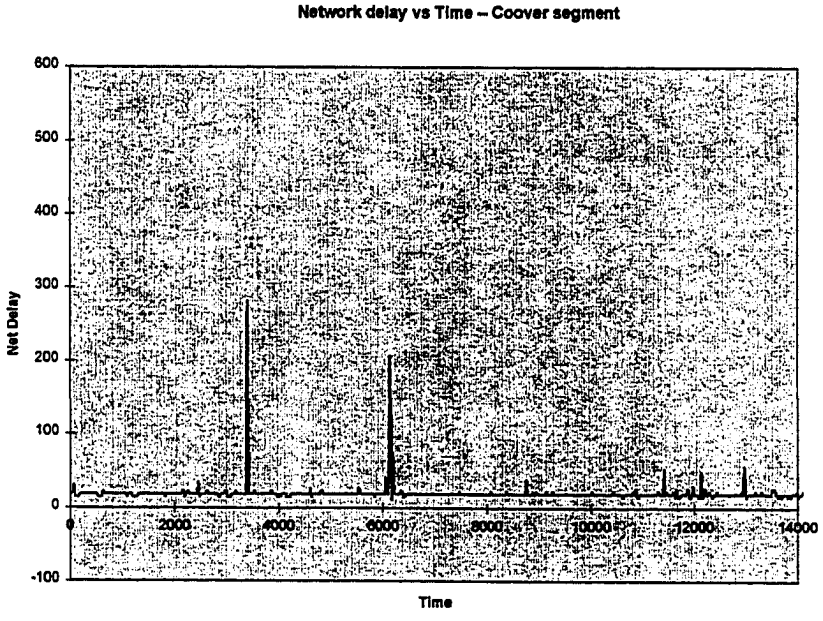

Figure 7.68 Plot of network delays on the Coover network segment.

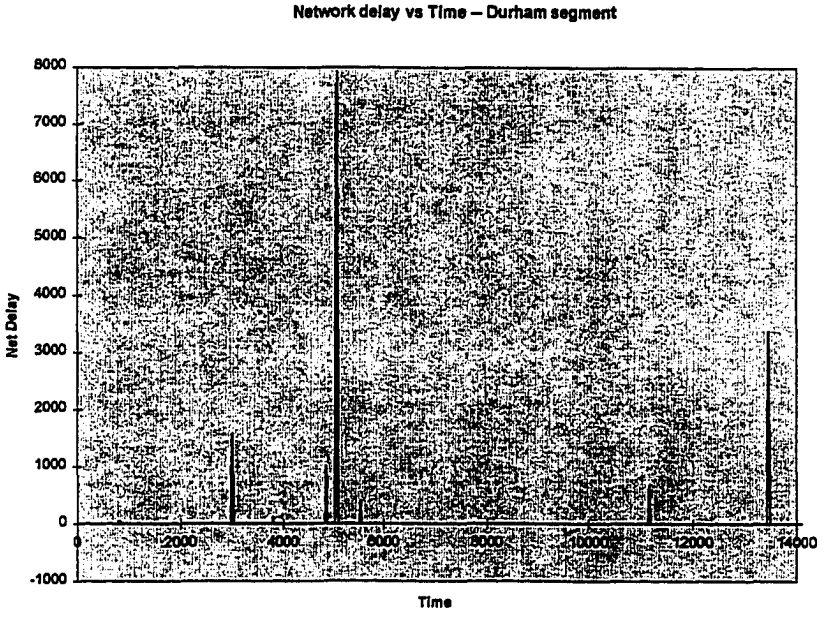

Figure 7.69 Plot of network delays on the Durham network segment.

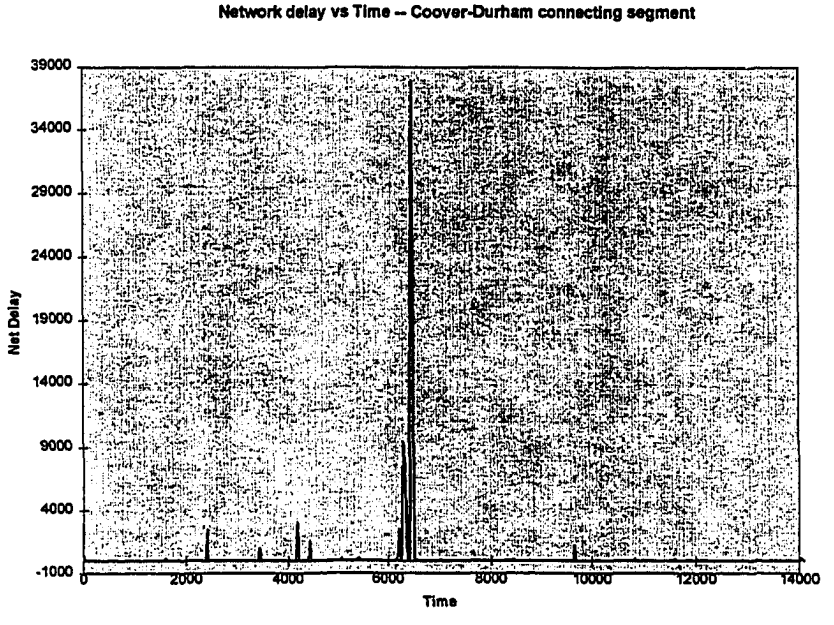

Figure 7.70 Plot of network delays on the segment connecting Coover and Durham.

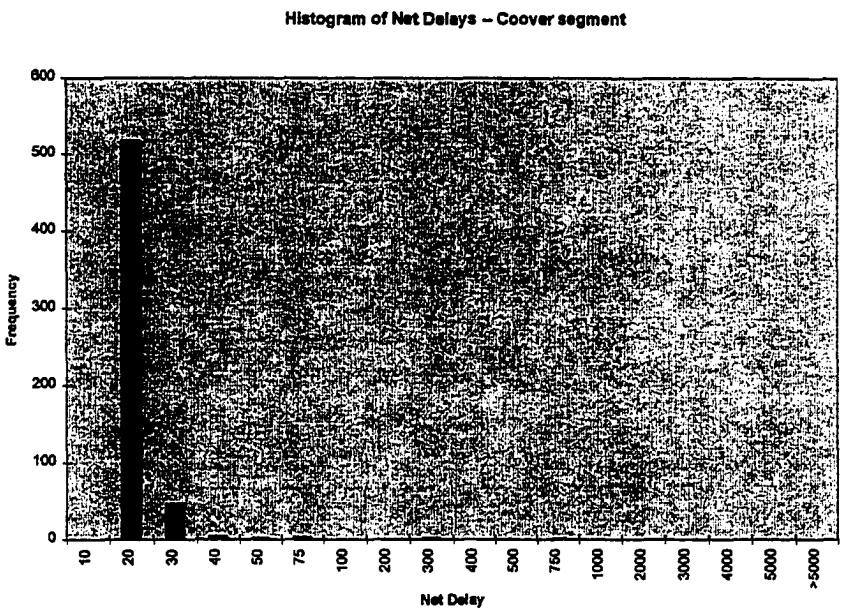

Figure 7.71 Histogram of network delays on the Coover network segment.

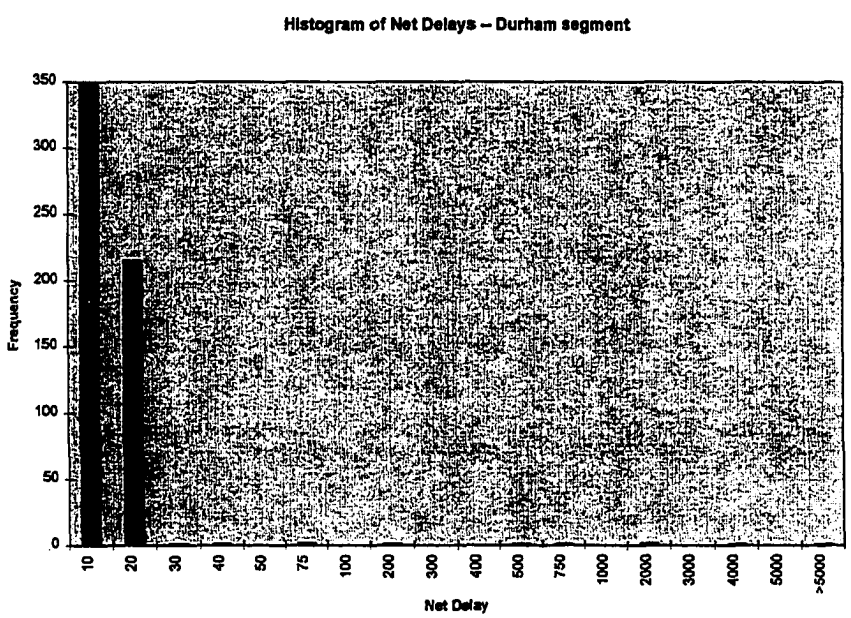

Figure 7.72 Histogram of network delays on the Durham network segment.

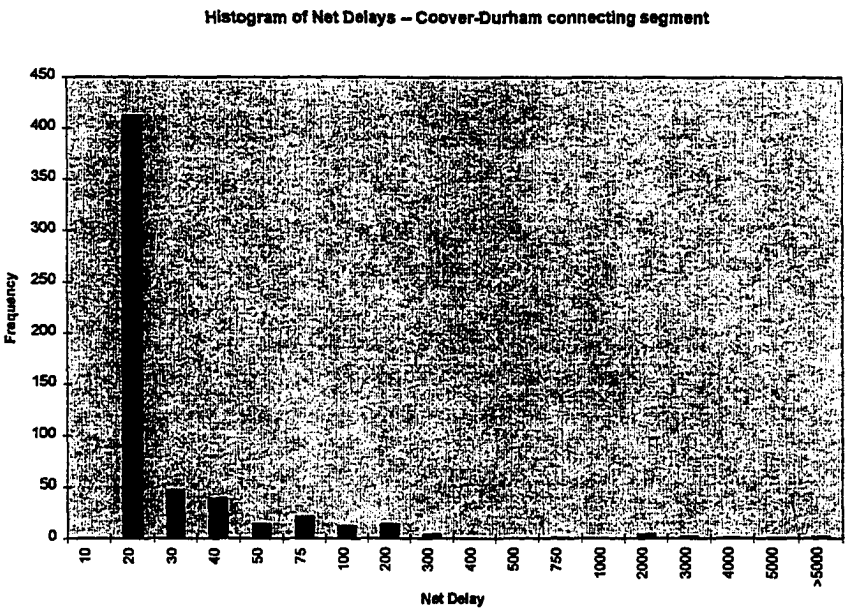

Figure 7.73 Histogram of network delays on the Coover-Durham connecting segment.

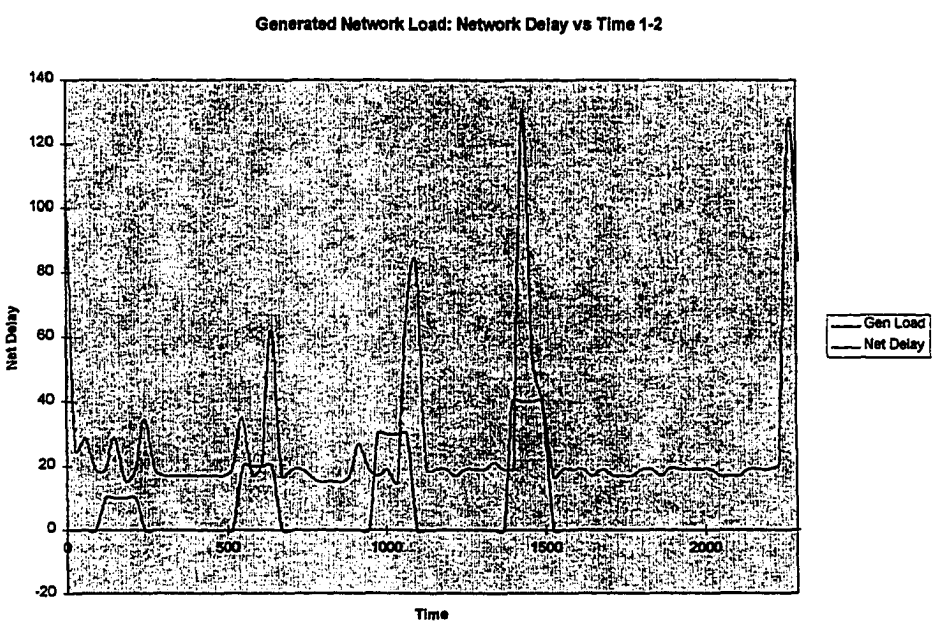

Figure 7.74 Plot of network delays under light artifical loading.

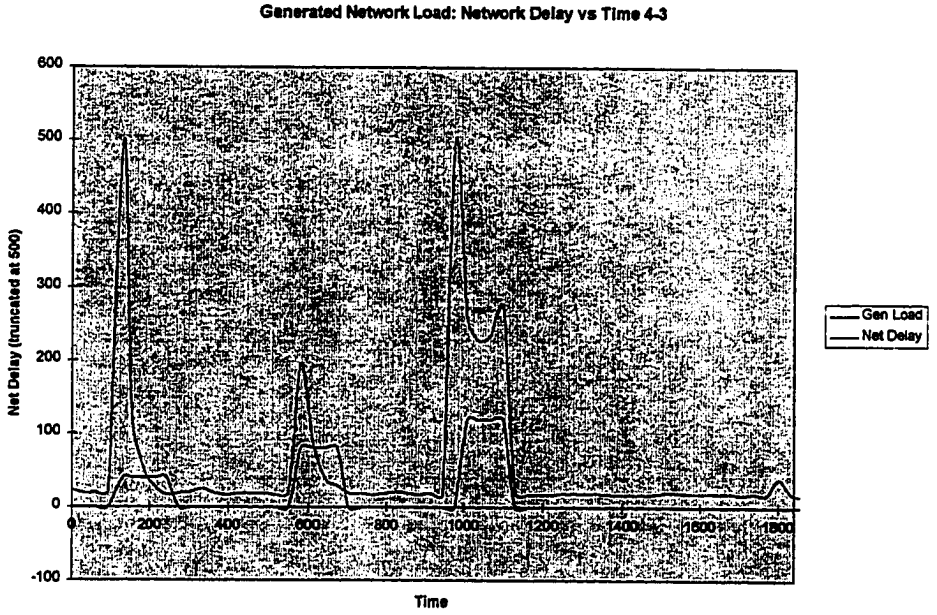

Figure 7.75 Plot of network delays under heavy artificial loading.

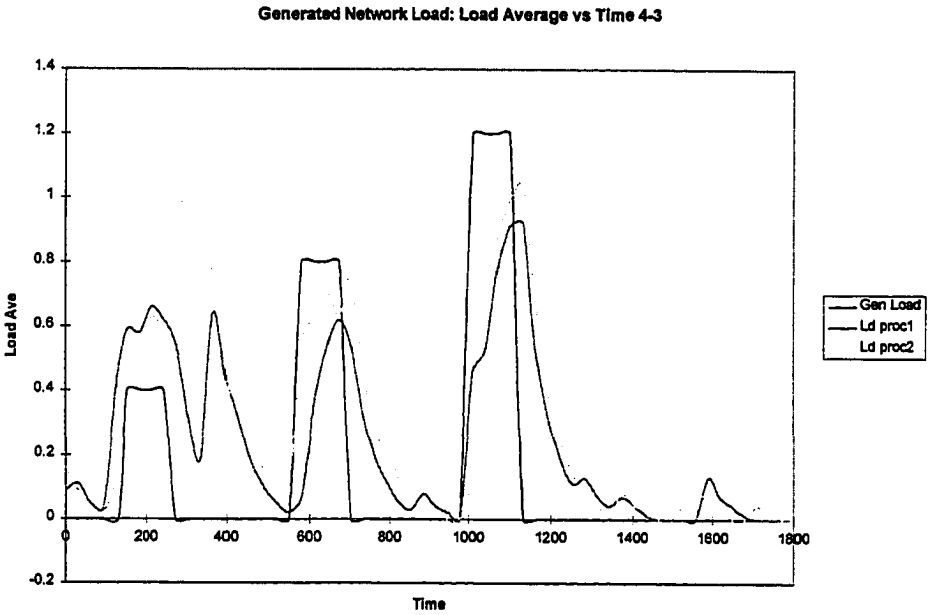

Figure 7.76 Plot of load averages on two processors during heavy artifical network loading.

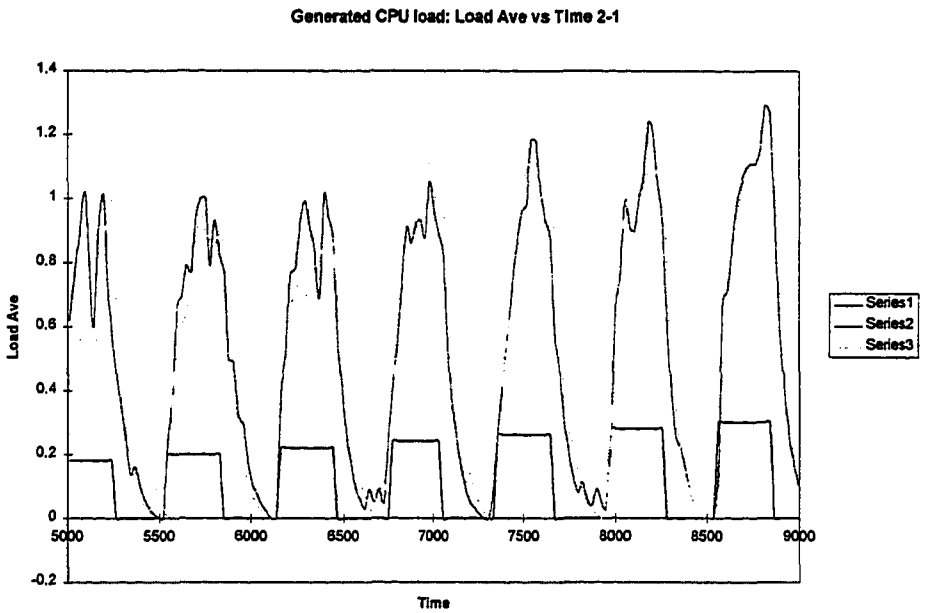

Figure 7.77 Plot of load averages during artifical CPU loading.

8. CONCLUSIONS

On a multiprocessor system in which nothing is known in advance about submitted tasks, a scheduler has limited options when placing tasks. It may monitor loads on the available processors and network segments, and use that information to choose suitable targets for a task. Many of the other algorithmic optimizations discussed while summarizing scheduling algorithms, however, require a priori information about program behavior which is not generally available in a real system. How, for example, will a scheduler know which of a set of submitted executable files will run for the shortest period of time?

It is common to see system load described in terms of an abstract quantity that rises smoothly and consistently as the processor and network utilization rises. Many workers equate this quantity with the "load average" value maintained by the UNIX kernel. As is apparent from the experiments described in the last chapter, however, the relationship between load average and submitted task performance is not a simple one. Looking at plots of program execution time versus load average, it is not obvious whether the cost of monitoring the load average value is worth the scheduling payoff, particularly when reports such as that of [18] reveal little value in load adaptive scheduling algorithms.

This work does not attempt to suggest a specific scheduling algorithm for any particular system. Rather, it provides techniques whereby system load may be monitored and processed to produce information useful to scheduling. In particular, the run time of compute-bound parallel programs may be predicted through the use of technique given in Chapter 7, and busy network segments and processors may be identified and avoided. Since the techniques are not system- or application program-dependent, they may be applied to workstation networks in general. In terms of scheduling, this work provides the load and execution time inputs needed by many common algorithms, allowing those algorithms to be implemented on real systems.

This chapter examines the results from Chapter 7 in the context of the scheduling problem of parallel tasks with little communication running on networks of workstations. Although PVM was used in this study, the results are not reliant in any way on PVM, and may be

applied to the other workstation network distributed computing packages described earlier in this document.

8.1 Overview of Conclusions

Three aspects of the results described in Chapter 7 are considered important to the purpose of task scheduling on networks of workstations. These are, (1) test application runtime shows correlation with load average measures on the target machines, (2) network segments show consistent properties through which different physical segments may be identified, and (3) communication-computation ratio measured using time units shows tight correlation with total application runtime. These three points will be briefly discussed below, and considered in detail in the remainder of this chapter.

8.1.1 Correlation between load average and application runtime

As intuitively expected, plots of test application runtime against load average on the most loaded machine showed correlation, and could be characterized by the slope of the resulting regression line. This slope represents the relation between local load and runtime; once the y-intercept of the line is established for a given program using a given set of input parameters, times for future runs of the program may be predicted given only the load average during the run. Knowledge of program runtime proves important to many scheduling algorithms, and is generally not available in real world systems. Because the workstation network multiprocessor is generally targeted toward frequently executed, long-running computation-bound scientific applications, any scheduler that can establish the relation between an application and load average would be ideally suited for such a system. Such a scheduler would "learn" the characteristics of its frequent applications, and could then use many algorithms normally useless to systems with no runtime prediction capability.

8.1.2 Network segment characteristics

Network round-trip packet delay proved useless in predicting execution time for tasks communicating over the network. However, the delay values themselves, when gathered over a

time span of a few hours, proved characteristic of the measured segment, and consistent enough to allow identification of the segment based entirely on network delay histograms. Three parameters were found to identify a given network segment: (1) mode of the delay values, called the "background" delay value here, (2) frequency of long delay "spikes" in the data, and (3) relative magnitude of these delay spikes. The background level represents typical propagation delay through the network, while spikes represent packets delayed several orders of magnitude above the background delay due to collisions. Busier networks produce a higher frequency of spikes, and spikes that represent longer delays, than quiet networks. Because these three characteristics proved relatively constant for a given network segment, a scheduler could measure these characteristics periodically, and use the resulting profiles to keep communicating tasks off busy segments. Even non-communicating tasks benefit from quiet networks, since excess traffic may generate processing delays due to CPU time spent processing in the TCP/IP stack. Although the data must be gathered for a period of hours to generate an accurate profile, the profile remains stable over periods of days or weeks, and thus the measurements need be repeated only infrequently by the scheduler.

8.1.3 Communication-computation time ratio correlation with application runtime

An unexpected result of the data analysis involving communication-computation time ratios were their tight correlation with runtime for the application. The time ratios are returned from each running task, and typically one or more of the tasks produced time ratios that show a strong trend when plotted against runtime. Since the application runtime is actually the runtime of the task on the slowest processor, it is claimed that the time ratios which correlate best with runtime are those returned by the slowest processor. This relationship provides two possibilities in light of the scheduling problem. (1) The time ratios produced by a dummy application running periodically in the background might be sampled regularly and used to identify the limiting machine in a group of machines. (2) Time ratios sampled from tasks involved in a running application program might be sampled periodically to predict the total runtime for that program. Both possibilities are important, the first because it provides means of avoiding busy machines during task scheduling, and the second because it allows another method of predicting execution time for a submitted program.

8.1.4 Organization of Chapter 8

The remainder of Chapter 8 presents conclusions based on the results from Chapter 7 in more detail. Section 8.2 discusses software portability issues, the driving force behind the selection of the load measurement techniques. Section 8.3 presents the first of the three important conclusions summarized above, the relation between application runtime and load average. In Section 8.4, network characteristics are described, the second of the three results important to scheduling. The load measure based on a benchmark approach is discussed in Section 8.5, and the reason for its failure examined, while in Section 8.6 the relationship between runtime and communication-computation ratio, summarized above, is detailed. Section 8.7 briefly discusses parallelism, Section 8.8 details the applicability of the data analysis results to the design of a process scheduler, while Section 8.9 presents limitations in the described techniques. Finally, Section 8.10 provides future research directions.

8.2 Portability of Load Measuring Schemes

Before examining the applicability of the previously described load measures to scheduling, it is important to address the portability of load monitor programs. In considering the results given in the last chapter, it is not difficult to imagine load measures that would prove more accurate than the ones given. For example, CPU load could be monitored by measuring the average percent of time a user program spends waiting for execution in the ready-to-run queue. Similarly, network statistics such as percent utilization and packets per minute could be obtained from special hardware such as network monitors and some routers.

However, one of the goals of this research was to examine load statistics that are easily applied to any system. Without this constraint, one might modify the OS code until a perfect load measure was obtained, then find that the resulting measure could not easily be applied to any other system. The load statistics described earlier in this work are easily calculated on any workstation network. Thus, it is argued, though the experiments were performed on a network of DECstation 5000s running ULTRIX, and the numeric results sensitive to the specific configuration used, the techniques are general, and may be applied with little modification to any network of contemporary workstations.

8.3 Load Average and Its Relationship to Application Run Time

As long as a program spends most of its time computing, as opposed to communicating, the measure of CPU loading will prove most important to the prediction of overall runtime for the program. This is not to say that communication plays an unimportant role, but rather that the role it will play in a given situation is not easily predicted, at least not with the network load measure discussed here (see Section 7.5). Therefore, it is wise to concentrate on predicting computation time, and leave communication time as a necessary noise level imposed on the results. Separate methods, discussed later, will provide means of minimizing that noise.

The plots of total run time versus load average presented in this work, and the many additional plots generated but not reproduced here, may be approached in two ways: analytically, through examination of the regression line slope and fitness values, and heuristically, through visual examination of the plots and any apparent trends they might have. In fact, both methods were employed with this data, and were found to be consistent. Trendline fitness values are useful for an overall summary of the regression results, yet they may produce misleading conclusions, as regression is very sensitive to the presence of outlier points (see discussion of outlier points in Section 7.3.1). On the other hand, visual inspection will reveal trends that regression analysis "misses," due to anomalies in the data distribution. To gain the advantages of both methods while avoiding the pitfalls, the problem was approached from both ends, with the goal to coalesce the results into a single whole. From one side, regression results were tabulated, and compared with slopes to gain a range of parameters corresponding to well correlated runs. From the other side, plots were visually inspected and grouped into categories according the correlation present, as determined visually, and similar parameters assigned heuristically.

8.3.1 Fitness value trends, and their relation to slope

If a perfect CPU load measure did exist, one could characterize a given computation-bound program's behavior with a linear "load line." This load line would correspond to a plot of runtime versus CPU load for that program. Thus, given a specific load value, the load line allows prediction of program execution time, and vice versa. Being a simple line, the load relationship

for the program is completely specified by a slope and an intercept value. The slope represents the correlation between runtime and CPU load for a class of programs whose behavior is typified by the program in question; a program's membership in this class would be dependent on the instruction mix of the program. The intercept calibrates the line for any particular program in that class. Thus, given the class of a program in question, one could find the slope of its load line by examining results for other, known, programs in that class. A single test run of the program would correctly position the line and reveal its intercept.

Unfortunately, no such perfect load measure exists, or is likely ever to be found for a real system. However, existing load measures, such as load average, will approximate this perfect measure. The question to answer: how reliable is the approximation? An answer to this question may be found by examining individual runs and calculating a range of slopes which encompass the observed trends. The points lying significantly near those trends can then be counted, and probabilities that any random run lies upon the resulting load line calculated.

As a real load measure such as load average approaches the perfect load measure described above, a plot of a series of program runs will approach the ideal load line. In a non-ideal situation, then, a regression analysis will provide an approximated load line whose closeness to the ideal load line is dependent upon the quality of the data. If it is assumed that only one load line exists, then, a series of data which closely fits a trendline may be considered to closely fit the load line, and the load line is approximated by the trendline. Whether it is reasonable to assume that only a single load line exists for a given program depends upon the scheduling algorithm used within the OS. For example, if the scheduler changes the priority level of a program based on how long it has been running, a single program whose run time depends upon external parameters may produce multiple load lines, depending on the values of those parameters. If, however, all runs of a given program execute in a similar period of time, one may assume that it is typified by a single load line. For simplicity, and based on the nature of the test application used in these experiments, it is assumed that any particular set of input parameters to the test application will produce a run that always exhibits the same ideal load line. Modification of the parameters may change the load line, but it is furthermore assumed that slight modifications will produce only slight changes. Because the range of parameter values used within the experimental test runs was limited, and because the scatter of points in many cases

was fairly large, the following assumptions were made regarding the experimental results: (1) The range of trendlines is due as much to the spread of points as to the spread of ideal load lines, (2) the correlation between the test application run time and load average is far enough from ideal that the notion of a single, unique load line is meaningless, and (3) the spread of ideal load lines across the entire data set is small enough that a range of load line slopes may be derived and used for the purpose of scheduling. The validity of these assumptions will be examined as the data results are derived.

A plot of trendline slope versus fitness value for run time versus load average experiments is given in Figure 7.11. As discussed earlier, plotted points show a general trend toward larger slopes with larger fitness values. The trend is more visible in Figure 7.14, after the removal of two outlier points. However, the spread of slope values is still large enough that conclusions are difficult to make from these plots alone. In addition, one must decide at what fitness value a regression line will be considered a good fit, or, in other terms, how high must the fitness value be for the points to be considered as lying on a significant trend?

Examination of all plots reveals that visible trends are generally present in plots with trendline fitness values of 0.1 or better. In addition, trendlines with negative slope values--an impossible runtime-load relationship in most systems--all have fitness values less than 0.1. Overall, 78% of the plots generated in the runtime versus load average experiments have trendline fitness values of 0.1 or greater. This percentage rises if outlier points are removed. For Figure 7.11, it can be seen that, ignoring the two outliers, slope values tend to fall between 0.01 and 0.05 for fitness values of 0.1, while at 0.5 to 0.8, they range between 0.02 and 0.08. The two outlier points were ignored because the plots from which they were generated, shown in Figures 7.12 and 7.13, contain a few outliers themselves which significantly shift the trendlines. These two runs are thus considered unrepresentative of the average runtime versus load average relationship.

As implied earlier, fitness value does not alone define the quality of a particular plot, where quality refers to the degree to which the data points fall on the corresponding load line. Depending on the arrangement of the points, very widely scattered points may sometimes produce trendlines with high fitness values, even though visual inspection will reveal little or no obvious correlation in the data. In plots such as these, moving a single point slightly will cause a

large movement in the trendline, thus revealing the trendline to be "unstable" and of little use. Similarly, other plots in which the points obviously lie upon a line produce trendlines with low fitness values, again due to quirks in the data distribution. Thus, visual inspection of the plots is an important addition to the analysis by fitness value. Plots with anomalous fitness values, as defined above, may be given less consideration when determining the useful range of load line slope values.

As mentioned above, 78% of the runtime versus load average plots showed trendline fitness values of 0.1 or above. This amounts to 50 plots out of a total of 64. These 50 plots were visually inspected. Of the 50, 23 showed strong trends in the data, while 27 contained weaker trends or large numbers of outliers. A histogram of the slope value for the 23 plots with clear trends is shown in Figure 8.1. The total range of slopes is from 0.01 to 0.07, though the majority fall between 0.005 and 0.025 (the given bounds were obtained through examination of the data values--they are more accurate than could be obtained from the histogram alone).

A total of 14 plots fall within that slope range, out of the 23 plots with visually strong trends and fitness values above 0.1. Out of the original 64 plots, 27, or 42%, have trendline slopes within the range 0.005 and 0.03. 11 plots, or 17%, contained trendlines with slopes below 0.005, while 26, or 41%, contained trendlines with slopes above 0.025. With almost half of all plots, and more than half of the strong-trended plots, containing trends with slopes in the range 0.005 to 0.025, this range will be considered to represent the range of slopes of the load lines corresponding to the various program runs. The fact that 57% of the plots overall, and 39% of the plots with strong trends, fall outside this range is interpreted as due to a combination of four factors. (1) High network traffic on certain days caused many outlier points, or a general spreading of runtimes, masking the effect of the CPU load on runtime and thus obscuring the load line. (2) The UNIX scheduler may cause longer runs of the test application to exhibit load lines significantly different from the shorter runs. A second peak in the histogram, in the slope range 0.035 to 0.05, supports this interpretation; overall , only 8 plots, or 13%, showed trendlines with slopes above this range. (3) High loads occuring shortly before execution of a given test application run skewed the first two load average values, causing an artificially high estimate of load during the run. (4) Other factors not yet identified occasionally skewed runtimes, causing outlier points which distorted trends and masked load line relationships.

Interestingly, trends were largely absent in plots of slope or fitness value versus test application parameters, such as number of processors or number of iterations. These plots are shown in Figures 7.5 - 7.10. Slight trending is seen in the plot of slope versus number of iterations, but this is difficult to interpret, especially since changing the number of iterations will greatly change the runtime, and is thus probably the most significant factor contributing to the multiple load line ranges mentioned above. In other words, a long running program might fall closer to its load line, since transient anomalies in the system which effect run time would tend to cancel out over time; on the other hand, a long running program may have a load line significantly different from that of a short running program, due to its lowered scheduling priority.

Similarly, no trend is seen when fitness values are plotted against the average load value for each experiment, as shown in Figure 8.2. This is surprising, since low loading levels during an experiment could be considered typical system "noise," and the resulting plot expected to be trendless. However, the average load during an experiment seemed to have no bearing on the presence or absence of a strong trend. In fact, some plots with very low average loads showed well correlated runtimes, while plots with unusually high average loads showed poor runtime correlation.

8.3.2 Outlier points: significance and probability

In the above discussion, several references have been made to outlier points present in the plots. Outlier points were also examined in Chapter 7, Experimental Results. The importance of outlier points, in terms of the scheduling problem, is that they are deviations from the expected behavior. If a scheduling algorithm tries to predict program behavior based on a load measure, it must be aware of the probability that the runtime will turn out drastically different from the value predicted based on the current load, i.e. that the run will represent an outlier point.

In many plots, outlier points are obvious during a visual inspection. An example of such a plot with two outliers is shown in Figure 7.3. In other cases, such as that shown in Figure 8.3, the presence or absence of outliers was not so easily determined. For purposes of this work, an outlier point was defined as either, (a) any point two standard deviations or more away from the trendline, if the trendline slope is non-negative, or (b) any point two standard deviations or more

away from the best fitting line with non-negative slope, if the trendline slope is negative. The latter half of this definition--ignoring negative trendlines--was justified by two factors: (1) It is known that a negative relationship between load average and run time is not possible in the system; any negative correlation must be an artifact of other factors, and (2) In most cases, plots with negative trendlines contained a few errant points which, if removed and the regression repeated, would then produce a positive sloping trendline; these arrant points, causing a negative slope, were assumed themselves to be outliers from another, positive sloping load line.

An estimate of the probability of an outlier, over the entire data set, was calculated by inspecting all plots and counting the total number of points in the plot, and the number of outlier points. From a total of 64 plots, representing 5665 test application runs, 177 outlier points were identified. Thus, for a random run of the test application program, using parameters in the range of this experiment--40, 80, 120, or 160 iterations, 1-4 processors, and a communication to computation unit ratio of 0.0, 0.1, 0.2 or 0.3--one should expect an arrant result, representing an outlier point, 3.1% of the time. Conversely, 96.9% of the runs will lie near a load line with positive slope. As mentioned above, 42% of these load lines fall in the slope range of 0.005 to 0.025 and 28% fall in the slope range of 0.035 to 0.05. Thus, 74% of those positive sloping load lines had slope values in one of the two stated ranges.

One might object to the above statements with the argument that the 64 runs were not identical; that is, the parameters to the test application program varied between sets of runs, and thus the expected behavior of the program is not uniform. In one sense, this objection is valid, in that application run times varied greatly over the 64 runs, depending on the number of iterations specified; 40 iteration runs took on the order of 55 seconds, while 160 iteration runs took on the order of 220 seconds (see Figures 7.40 and 7.44). That this difference in average run times effected the expected load lines is born out by the dual peaks in the slope histogram in Figure 8.1. On the other hand, the lack of trends in the plots of slope and fitness versus the various test application parameters, (shown in Figures 7.5 - 7.10) is taken as evidence that, in general, these parameters do no effect the load line response of the program, at least over the limited range of values attempted during these experiments. And the value range was limited on purpose; it is intuitively obvious, for instance, that a program with a communication to computation unit ratio of 0.95 is going to exhibit a runtime much more closely tied to network traffic than CPU load. In

fact, the parameter values were chosen to deliberately fit two criteria: (1) application behavior was suitable for a workstation network multiprocessor, based on prior expectations of such suitability (i.e. low communication percentages), and (2) total application runtimes were short enough to allow an experiment to run in an 8 hour period, approximately, assuming 50 to 60 runs per experiment. Note that this latter requirement was instilled purely for convenience, to limit the number of experiments aborted due to machine and network outages.

So what do these results mean, in terms of scheduling? For now, consider only the test application used in the experiments; the results will be generalized to other programs later. If this application is submitted to a scheduler, how might the scheduler use the above information to choose one or more sites for execution? To begin with, the scheduler knows that, 3.1% of the time, the application's run time will be determined by factors other than load average, and thus the best placement of the task in these cases is unpredictable. The rest of the time, the runtime will fall near a load line, and thus its execution time, or range of possible execution times, may be predicted if the load line is known. In general, ranges of execution times were small for a given experiment; typical ranges are seen in the execution time histograms shown in Figures 7.53, 7.54, 7.57 and 7.58. On 160 iteration runs, for example, the majority of the runtimes differed from the average by 15 seconds, or about 7%, given all other parameters equal. These numbers are calculated over all loads occurring during the experiment. For a given load during a given experiment, similar deviations are seen in plots with clear trends. For example, in Figure 7.4, the deviation in runtimes for a load average of 1.5 is approximately 8 seconds, or 11%, while the deviation at a load average of 2.5 is approximately 6 seconds, or 6%.

Thus, knowing the number of iterations to be performed, the scheduler may estimate execution time for the test application based on load for each possible machine, and assume its estimate will be within about 10% of the actual value 96% of the time. A collection of test application tasks with differing iteration numbers could then be scheduled for maximum throughput or minimum response time, using scheduling algorithms previously described. Since number of iterations is an input parameter, it is not unreasonable to assume that the scheduler could obtain this value for a given submitted task; it might be read directly from the submitting command line, or provided by the submitting user.

8.4 Significance of the Network Load Statistic

It is logical to imagine that network load could be measured and used in the same way as CPU load. Such a statistic, the traffic level in packets per minute, is available using special network monitors. However, it is not normally possible to record such a statistic without a dedicated workstation running in promiscuous mode. As a substitute, the round trip time of a single message was measured, under the assumption that the travel time for that message should reflect relative network load during that time period. Results of experiments proved this assumption to be incorrect. In fact, the message generally reached its destination with no noticeable delay. As described earlier, this phenomena is due to the nature of the network protocols used. Only under heavy traffic did network delay appear to reflect load, though even then the response was prone to large fluctuations from sample to sample. These results higlight the fact that network traffic analysis through software is not trivial [32].

Although network delay does not represent the load on the network, it does respond to the load the same way an application would respond to the load; most packets get through undelayed, but as load rises, a larger and larger percentage see a delay. Thus, the behavior of the load monitor program is still useful to a scheduling algorithm.

It was shown that a network segment could be classified based on three measured parameters: the typical (background) delay value, the delay spike frequency, and the delay spike amplitude range. The first parameter is the delay seem by most packets produced by the load monitor; this value is nearly constant for the three segments examined, generally fluctuating only a few milliseconds around the mode value. The second parameter is the frequency of large delay spikes, defined as a delay at least one order of magnitude above the background; busy network segments have a much higher spike frequency than quiet segments. The last parameter is the average height of the spikes, and this value also reflects the traffic on the segment, with larger spikes on busy segments. Thus, the character of the network traffic on a given segment can be derived through delay measurements. However, the results are only valid over comparatively long periods of time, on the order of hours. Such a load measuring scheme cannot respond to minute by minute fluctuations in network traffic.

A scheduler could make use of such information when locating processes initially, if any information is known about the relative amount of communication that will occur between the processes. A group of processes with no communication might be located on a busy network segment, while a group of processes which must communicate would be placed on a quiet network segment. Obviously, the placement is more important to the communicating processes, since the non communicating processes will execute similarly on either segment. However, placing tasks on appropriate segments, based on inter-task communication, will prove important to the overall schedule, and the overall throughput of the system, by avoiding the placement of communicating tasks on busy segments.

The sampling interval for the network monitor is not critical. The network segments analyzed for this work and discussed in the previous chapter showed fairly constant behavior over a period of months. To characterize the network segment, the load monitor should examine the net periodically during each characteristic usage segment; once in the evening, and once during the day, for example. These examinations could be performed a few times a week. Each examination would consist of a monitor period of a few hours, in which a sample is taken on the order of once a minute. The collection of samples taken during a given examination could then be processed to provide the statistics described above. In most cases, these statistics should remain fairly constant. An exception, for example, could be a segment serving an academic computing lab, on which traffic reflects the progress of the current semester, and the number of assignments due the students.

8.5 The Failure of the Benchmark-Based Load Measure

The rational behind the use of the benchmark-based load average was the idea that the relative amount of time a small program spent executing, versus the time it spent in the ready queue waiting to execute, would reflect the same ratio of times for a larger program running at the same time. Plots of application run times versus the benchmark load measure results, however, failed to reveal significant correlation between the two, and the histogram of fitness values of the corresponding trendlines showed very few measurable trends (see Figure 7.26). On the other hand, the consistently negative slopes of the trendlines, as revealed in Figure 7.31,

demonstrate that some degree of correlation is present; otherwise, one would expect an even distribution of positive and negative slopes.

A hypothesis to explain the failure of this method involves the UNIX scheduler. As a process runs and accumulates CPU time, its scheduling priority is lowered. Thus, a longer running program, such as the PVM test application, will be given a gradually lower priority, and will thus obtain a smaller percentage of the CPU time than the short running benchmark load measure task. Therefor, the ratio of running to waiting time for the benchmark program would not equal the same ratio for the application program.

One way to test the above hypothesis is to examine a histogram of the resulting benchmark load measurements, taken over a period of time. If a given load measure is sensitive to fluctuating load, one would expect to see a spread of load measure values. The benchmark load measure gives a minimum value of 1 when wall clock run time equals CPU run time, indicating that the task executed the entire time it was in the system. Thus, it is straightforward to examine the spread of values and compare them to other load measurements taken at the same time, such as load average. Figure 8.4 shows a histogram of benchmark load measures, and Figure 8.5 gives a histogram of load averages taken during the same time period. Notice the much larger distribution of load average values; the benchmark load measure values all fall between 0.9 and 1.1. Since a value of 1.1 is theoretically impossible to attain, it is apparent that the resolution of the clock values used to time the benchmark produces uncertainties of at least +/- 0.1, thus invalidating the entire range of benchmark values. Similar histograms were produced from all run periods examined.

One final feature present in the load average histograms and absent in the benchmark load histograms that demonstrated the fundamental difference between the two is especially visible in Figure 8.6. This histogram contains two peaks, one corresponding to load measurements taken during quesient periods, the other corresponding to periods of test application execution (the load measure program and test application were running together, but load measurements were not synchronized with application runs). This bi-modal shape is also visible in Figure 8.5, but was absent from all benchmark load measure plots, indicating that the benchmark load measure did not respond to the execution of the test application.

8.6 Correlation Between Application Runtime and Communication/Computation Ratio

One of the most interesting results of the data analysis is the revealed correlation between application runtime and the measured communication to computation time ratio. The correlation is much stronger than that seen between runtime and load average. Of course, the communication to computation time ratio is a product of the application itself, not an external measure of the system. Thus, application as a load measure in the same sense as load average presents problems. However, it will be argued that the correlation between the time ratio and runtime may be exploited for the purposes of scheduling.

It is important to notice that, in a multi-process application, not all communication to computation time ratio values correlate with total application runtime. For a given process, it is likely that the returned time ratio is correlated with runtime. However, individual process completion times were not recorded in the experiments, and thus this hypothesis is impossible to verify. The recorded runtime value, for the entire application, is actually the runtime of the slowest process, the limiting process. Since all processes are identical for a given run, and each processor's hardware is identical, the limiting process is the process running on the most heavily loaded machine. Note that the previous statement is true by definition, as "load" is defined as the phenomena, independent of hardware, that affects program execution time on a given processor; load average and network delays are imperfect attempts to measure this ideal quantity, "load."

On a given plot, it is hypothesized that the highest communication to computation time ratio is returned by the limiting machine. This statement cannot be proved directly because of the above stated limits in the data; the individual process runtimes were not recorded. However, the statement is supported by the following circumstantial evidence: (1) In all cases, the highest time ratio is correlated with runtime, as evidenced by a non-zero data trend slope, while lower time ratios often follow trends with slopes of 0, indicating no correlation. (2) When the set of highest time ratios includes values from multiple processes, all values fall on the same trend line, indicating a consistent relationship between highest time ratio and runtime, regardless of the machine involved. (3) In runs involving three or four processors, the processor returning the highest time ratio values was usually one of two on the busier of the two network segments; in addition, these machines had higher typical load averages than the machines on the quiet

network segment. (4) In runs of three of more processes, time ratio values below the high value trend were sometimes spread over a wide vertical distance, indicating little or no relation to runtime. (5) In runs of one processor, the time ratio values were scattered widely across the plot with no visible trends, as shown in Figure 8.7. The importance of this last point deserves clarification. In a single processor run, the communication time is simply a measure of the computation overhead required to prepare for communication, without any actual communication. Thus, the time ratio is actually a ratio of two computation times, both of which would be expected to respond to load in the same manner. The fact that no trend is seen when these values are plotted against runtime indicates that the computation portion of the ratio does not itself determine runtime. On the other hand, a close relationship between computation time and communication time on a given machine is unlikely, since communication time is dependent on two independent processors, while computation time is local to a single machine. Therefore, it is inferred that the correlation between runtime and time ratio must be largely due to a correlation between runtime and communication time, and thus that high time ratios, representing high communication times, determine longest runtime.

Given the above hypothesis, it is apparent that communication-to-computation time ratio could be used by a scheduler to locate the limiting machine during an application run. The time ratio, produced by a low priority process running concurrently with the application, could be sampled periodically by the scheduler, and the limiting machine(s) avoided during scheduling of processes that communicate. In a sense, the time ratio represents a cumulative history of the network loading on the involved machines, providing a more stable measure of network loading than the network delay measure used during these experiments. In addition, apparent network loading caused by high local CPU loads will tend to be normalized out of the measure, due to the division by computation time; plots of time ratio versus load average indicate that the time ratio is insensitive to CPU load. Typical plots are shown in Figures 7.38 and 7.39.

The Cm/Cp ratio may also be used to predict runtime, using the same techniques described in the context of load average. Slopes of dominant (upper most) Cm/Cp-runtime trends were calculated for charts of 2, 3, and 4 processors. Resulting slopes ranged from 0.01 to 0.07, though most concentrated in the range 0.01 to 0.04. A plot of the slope values versus number of processors is shown in Figure 8.8. Note that the range of values and the values themselves tend

to rise with increasing number of processors. Both features are likely due to the increasing number of network segments and traffic which accompany the increasing number of processors. A plot of slope versus number of iterations is shown in Figure 8.9. In this plot, it is apparent that the range of slope values decreases as number of iterations increases. For the largest number of iterations, 160, the range of slope values is small, approximately 0.01 to 0.02. Because increaisng the number of iterations will reduce the proportion of time spend performing startup and other overhead, it is likely that the slopes occurring in the 160 iteration plots are most representative. The resulting slope range is much tighter than that calculated for load average. On the otherhand, the Cm/Cp slopes are closely tied to the communication characteristics of the application program under test. Thus, their applicability to a general class of programs cannot be estimated from such a small data set. More investigation is needed in this area.

8.7 Program Length and Available Parallelism

A few words should be said about the degree of parallelism of the PVM test application, and its relation to the number of program iterations. Speedup plots indicate a gain in moving from one to two processors for 40 and 80 iteration runs, but little gain with the addition of a third processor. Runs with 120 iterations showed improvement up to the maximum number of processors investigated, four, though gains moving from two to three and three to four were much smaller than the gain moving from one to two. Surprisingly, 160 iteration runs showed less improvement upon addition of the fourth processor than did 120 iteration runs, though improvement upon addition of a third processor was significant.

In general, as processors are added to a fixed total workload, the amount of computation performed by each processor drops while the amount of communication rises. Of course, computational overhead associated with communication will offset the computation drop somewhat. Except through experimentation, it is difficult to know the proper level of parallelism for a given application program. Since the test application was written to mimic general program behavior, however, some general conclusions may be drawn from examining its behavior.

With typical run times around 220 seconds for the 160 iteration case, it is apparent that significantly longer run times will likely be required to take advantage of more than two or three

processors. On the other hand, communication percentages in the range examined did not seem to play are large roll in fixing parallelism. This is surprising since the communication time appears to play a large roll in determining execution time on the slowest machine (see previous section). In general, runs with no communication showed improved performance over runs with communication, but runs with varying non-zero levels of communication were difficult to distinguish. This behavior is due to the nature of the program's response to network traffic, as discussed earlier; packets tend to reach their destination with either no delay or comparatively long delays, with few intermediate delays. The delay experienced by a given application would thus bear closer relation to network traffic level on the segment than on the number of messages sent by the application; a few delayed messages will produce noticeably longer run times. This conclusion is supported by the results of the network delay monitor period experiment results, which revealed the frequency of delay peaks to be largely insensitive to sampling interval, and thus to the number of messages sent (since each sample represents a message).

To use a large number of workstations efficiently will require a long running application. For programs whose execution time is on the order of minutes on a single processor, two or three processors will likely produce the best performance to be expected. Such information is currently more useful to application writers than schedulers, since few schedulers are able to dynamically parallelize an application program. On the other hand, a program consisting of many tasks might be clustered onto two or three processors by one of the clustering algorithms described earlier, with the level of clustering determined by the expected run time of the program.

8.8 Implementation of a Scheduler Information Policy

The experimental results described in this work apply largely to the development of the information policy of a given scheduling algorithm. This aspect of scheduling seems to have attracted the least amount of research. When reading the description of a scheduling algorithm, it is common to find only vague references to "system load," and "network congestion" when the information policy is discussed. In many studies it is taken for granted that one can call an OS kernel function and retrieve perfect load statistics. This work provides insight into the behavior of several load measures on a typical system. In addition, techniques are provided for

determining the relationship between load and application run time for computation-intensive programs. Such information is precisely that required to implement the information policy of a scheduler. In particular, many scheduling algorithms require a prediction of program run time, and some rely on expected load profile to make important decisions such as whether or not to migrate a task.

8.8.1 Information policies

As described in Section 3.2.1, the information policy is the part of a scheduling/load balancing algorithm which determines the type of system information that is collected, how often it is collected, and how it is used. The load statistics described in this work are simple to collect, and thus fit easily into the load policy in any system. In addition, their applicability is not limited to PVM or even parallel processes; they could as easily be used for batch scheduling ordinary programs on workstation networks. Because the amount of information collected is fairly small, either a centralized or distributed policy is possible. Within a centralized policy, the master program would reside on a single machine, with slaves collecting CPU statistics on each processor, and network delay statistics on each segment. The collected information would be routed back to the master program, either as it is collected, or in periodic packets containing multiple measurements. Of course, if the distinction between "master" and "slave" program is blurred, one can see that the above policy is also a distributed policy, in a sense. If the duties performed by the master are transferred partially to the slaves, the policy becomes fully distributed. Similarly, it is straight forward to envision a master assigned to each cluster of workstations, where a cluster is defined as machines on a single segment. The master for a given segment could serve as an information server for masters on other segments, answering requests for local loading information. One or more concurrent schedulers could use this information to inject processes into various locations within the system. Such an arrangement could easily be expanded to almost any size, limited only by the amount of network traffic generated through the exchange of information. It is claimed that, given the limited sensitivity of application run time to the load measures described here, a great many servers could exchange information periodically without generating noticeable network congestion. Updates every five or ten minutes would be sufficient.

8.8.2 The target application program type

Such a system is not suitable to all jobs. However, jobs targeted for workstation networks fall within the group finding most benefit from the above described information policy. These jobs tend to be long-running, computationally intensive programs, which, when multiple processes are involved, perform a minimum of interprocess communication. Such applications are common in the sciences, particularly physics and chemistry, and it was for the purpose of running such jobs that the notion of workstation network multiprocessors first evolved. Most workstation network support software is aimed at this target group of jobs.

Long-running, frequently executed computation-bound programs provide the perfect target application for a scheduler based on the load statistics examined in these experiments. With the general results from use of the PVM test application in mind, an investigator could apply the given techniques to determine the quantitative relationships between load average, network delay, and the specific application program in question. The results of such experiments would allow the scheduler to be tuned to that particular program on that particular set of workstations.

Suitable levels of interprocess communication within the target application are more difficult to quantify. Qualitatively, the communication to computation ratio should be kept small. The determination of that ratio for a given program, however, is mostly intuitive. For a real program, it is difficult to define what exactly is meant by communication to computation ratio; attempts are made in [23: p.22-27] and [51: p. 309-325]. The PVM test application provides a good example of this difficulty.

The computation-communication structure within the PVM test application is somewhat artificial, designed to allow levels of computation and communication to be independently adjusted. However, results of a few test runs demonstrated the complexity of defining what is meant by "communication-computation ratio." Two definition schemes are possible: (1) define communication and computation "operations," based on the structure of the code, and then count the number of each type of operation performed in a given run, or (2) measure the total time spent communicating, and the total time spent computing (which may or may not be defined as all time not communicating). Both methods were used in the test application. Subroutines which

performed a certain amount of communication or computation per call were written and then called from the main program loop. The amount of time required to execute each subroutine on a typical run was measured, and parameters within the subroutine adjusted so as to produce units of computation and communication that typically took roughly identical time periods to execute. Each execution of one of these subroutines resulted in a "unit" of computation or communication. The ratio of the two has been referred to as the "unit ratio" within this work. Calls to time functions were then added to measure the total time spend executing each of the two unit routines, and the ratio of the two, called the "time ratio" or Cm/Cp, returned. In theory, for a given run of the program, the unit ratio should be roughly the same as the returned time ratio.

In fact, the unit ratio and time ratios typically differed by an order of magnitude or more. This is indicative of the fluctuating load conditions within the CPU and across network segments encountered by the program. It also highlights the difficulty of determining the communication-computation ratio for a real program. First, one must define the ratio itself; is it time-based or work unit-based? Second, one must somehow instrument the program or runtime environment to make the required measurements, a non-trivial task even in an experimental system. Ultimately, for a given run of a program, the time ratio provides the clearest picture of the proportion of computation and communication performed by the program. However, the time ratio is only available after the run is finished, and it might be quite different for the next run. Similarly, one may calculate the unit ratio by counting instructions within the program, but if unit ratio bears little or no relation to the actual time spent performing each type of operation, it is of little use. In conclusion, communication-computation ratio is another term which, like load average, is frequently employed but seldom investigated in detail. More work remains to be done in this area.

8.8.3 A scheduling expert system

The most attractive possibility presented by the experimental data gathered in this work is the creation of a scheduler which monitors the results of its decisions and learns from its successes/mistakes. Such a scheduler would become familiar with the behavior of a specific program and, using current loading statistics, be able to predict run time and, possibly,

computation and communication times for that program. This ability to know behavior before execution allows implementation of the many scheduling schemes which require normally unavailable a priori knowledge of the queued jobs. Careful monitoring of frequent runs of a program would, in time, provide the scheduler with more detailed knowledge of the program's behavior than that of the writer of the program. If the scheduler was able to monitor the network traffic produced by the program--which is not a simple task by any means--it could also calculate a meaningful communication-computation ratio for the program. Programs discovered to be spending an inordinate amount of time communicating could be examined by their authors and possibly rewritten to improve performance.

Time of day and day of week, even month of year load predictions are another possible ability of an intelligent, system monitoring scheduler. That such predictions are possible is revealed by experiments described in [31], though intuitively, the possibility is not surprising; human behavior is keyed closely to daily and weekly schedules, and human behavior for the most part determines job submission patterns in a computer system. Information on time-based loads could be used by schedulers in both batch-oriented and preemptively migrating interactive environments. A batch scheduler, containing a queue of tasks, could increase or decrease job placement rates based on the time of day, reducing the number of processes injected during times with typically high usage. Similarly, a typical interactive scheduler which supports preemptive migration could move running tasks away from system regions at times when usage patterns predict an impending load increase.

8.8.4 Load measures and preemptive migration

Load measures and run time prediction are especially pertinent to scheduling policies employing preemptive task migration. Preemptive migration is entirely based on load measurements, with the goal of moving tasks away from loaded regions of the system. To be able to accurately measure load on a CPU, and then use the load to predict remaining execution time for the task, is a key element of a successful load migrator; to justify the expense of moving a task, it must be known that, (1) the task is not about to terminate, and (2) the task will perform better on the destination machine. Experience-gained knowledge of program behavior in the presence of load provides that capability.

To migrate a task, the scheduler must determine that the target machine has a significantly lower CPU load than the source machine, where a significant load difference is a load difference that will be reflected by application runtimes on the two machines. A detailed knowledge of a load statistic, such a load average, is necessary to predict how various load levels will affect program behavior. This conclusion was reached by the sometimes unexpected results of the described experiments and data analysis. In addition, knowledge of local network loading is important if a task is a communicating part of a parallel program. To move a process onto a lightly loaded machine on a heavily loaded network is a mistake that a scheduler could easily make, without knowledge of network segment statistics. As discussed in the previous chapter, network segment statistics, at least in many cases, are relatively stable over time. Thus, knowledge of network segment characteristics could be gained with very little work by the scheduler and very little interference with system performance. This is an important result, because classification of network traffic generally requires dedicated hardware to count packets and monitor collisions. Because machines on congested network segments are typically avoided, if possible, by human users, the direction of a communicating process to one of these machines is a significant risk for a scheduler without network statistics knowledge.

8.8.5 Applicability to specific scheduling algorithm types

Four classes of scheduling algorithms were discussed in Section 3.3. A few words will be said below regarding the way in which the experimental results might be incorporated into each type of scheduler.

List schedulers: The assignment of priorities to tasks waiting in the queue typically requires knowledge of program behavior. Behavior prediction through knowledge of program parameters and current loading conditions will provide this knowledge. In addition, knowledge of loading levels will allow the scheduling algorithm to determine when a server processor is "available." This ability is particularly important in a non-dedicated environment such as a general purpose network of workstations. In such a system, the scheduler cannot assume a server is available just because it has finished all tasks submitted by the scheduler.

Folding/clustering schedulers: The most important question to answer in a clustering or module clustering scheduler is how many tasks should be fused for the given application on the given system? Clustering too few tasks will result in excess network traffic. Clustering too many tasks will overload the target processors. A common assumption made by module fusing algorithms is that the optimal number of modules to produce is one per processor. Such a conclusion is invalid in a workstation network environment, in which external load, not under control of the scheduler, is constantly injected into the system. Ideally, tasks should be combined to match the level of parallelism available in the program, as described above. A scheduler with prior knowledge of that program could make such a decision, based on application run parameters, current load and network characteristics. Without knowledge of the program, the scheduler could choose a number of tasks based on the number of machines currently considered available, as defined in the discussion of list schedulers.

Queuing schedulers: Most of the queuing scheduling strategies are static in nature, meaning they place tasks based on predefined patterns or probabilities. Some schedulers, however, allow the probabilities directing tasks to individual servers to change over time, and such changes could be made based on the CPU and network load seen by each server. In addition, loading measures are useful for determining when a given server is available, especially if the server is being used for other, unrelated processing as well. Finally, estimated run times for queued programs could be used to calculate more accurate queue lengths, rather than simply counting the number of jobs. In fact, this last optimization applies to most scheduling algorithms.

Threshold schedulers: Application of load statistics to threshold schedulers is straightforward; the load, rather than the number of jobs, is used to calculate the current work level, which is then compared to the threshold. This is intuitive, as a machine with one six hour job should not be considered less loaded than a machine with six five minute jobs. Quite to the contrary, additional jobs should be routed to the latter machine. If preemptive migration at a reasonable cost is implemented, this distinction is not so critical, since jobs placed on the first machine could be moved off as the short jobs on the second machine terminate. However, even if

preemption costs are small, the number of migrations for a given task should be held to a minimum; many existing systems limit this number to one.

8.9 Scope and Limitations of Results

The techniques described were designed to apply in the majority to typical workstation network systems. However, certain limitations were necessarily imposed upon the data, and it is important to understand these limitations when applying the results to a given system.

The most important limitation is in the selection of target application types. As stated previously, this work targeted computation-intensive applications, whose execution time is bound by the CPU. Such applications spend little or no time performing I/O or communication. The techniques presented in this work are designed with the assumption that the application program is CPU-bound. The relationship between CPU load and run time cannot be expected to hold as communication levels are increased. This weakness is due to the fact that network loading directly effects communication but only indirectly effects CPU load. Therefore, in predicting run time for communicating tasks, it is necessary to consider both CPU and network load. Unfortunately, the network load measure considered in this work is not appropriate for run time prediction, for reasons stated earlier.

A related limitation arises when the target application is scaled to a large number of processors. In this situation, the overhead involved in task startup and initialization on each machine may become significant in the overall run time, particularly if the run time of each task is comparatively small. The source of the problem involves the use of the network to transfer startup data and final results between each task and the master processor. In this situation, overhead time will be effected by network traffic, and thus requires the use of a network load measure in addition to CPU load measure for execution time prediction. However, this limitation is considered minor for two reasons. (1) Typical application programs do not scale to large numbers of processors; five to eight processors are common limits stated for applications tested in the literature [17, 40, 43]. (2) Typical application tasks are designed to ensure that the amount of time spent performing startup chores is insignificant compared to the total computation time; task distributions spending a significant amount of time performing initialization lead to

inefficient use of processing facilities. Therefore, the effect of network load during startup on total run time is typically expected to be negligible.

Also, it is apparent that the particular behavior of the test application program, while designed to mimic typical applications, is unique to the test application. Other programs, even if computation-bound, will nevertheless display unique behavior. However, the techniques presented may be generally applied to any computation-bound program, and a load-run time relationship derived. Any computation-bound program will respond in a linear manner to CPU load.

Finally, it is important to note that the use of run time prediction as described here applies only to scheduling algorithms which do not perform preemptive migration. The reason for this limitation lies in the fact that run time prediction assumes that total execution time is limited by the busiest processor in the configuration. Preemptive migration, however, allows tasks to be moved off the busiest processor before completion. In addition, the overhead cost of task migration is attributable, at least in part, to network loading. Therefore, when migration overhead is included in run time, a possibly significant communication component is added to the programs execution profile. As stated earlier, programs that are not CPU-bound will not respond linearly to load average.

8.10 Future Research

A number of issues remain associated with this research remain to be investigated. In addition, the results suggest new questions, which require additional data collection and/or different analysis techniques. This section briefly lists a few of these future research directions.

The experiments performed here used a relatively small subset of the total parameter space available with the PVM test application. In particular, only one communication pattern (circular) and one work distribution (distributed) was used. Also, comparatively small communication ratios were employed. The question thus arises, would the correlation between runtime and load average continue to hold when these parameters are varied? What effect would pairwise or random communication have on total runtime? Would communication percentage

affect runtime differently if the communication were lumped together, rather than distributed with the computation?

Many questions also arise regarding the relationship between communication-computation time ratio and application runtime. Because this relationship was unexpected, data necessary for its full analysis was not recorded during the data gathering stage of the experiment. Initially, it is important to verify that the limiting processor in a given run produces the best correlated communication-computation ratios. Then, it is important to determine whether samples of this ratio, taken as the application executes, would also correlate with the runtime, thus allowing runtime prediction.

Although the PVM test application was designed as a fairly generic application, representative of many parallel programs, the fact remains that it is a single program. Further research could examine other application programs and compare their correlation with load average to that of the test application. Is the relationship universal, or unique to a certain class of applications? Another question to be answered, are outlier points ubiquitous, or do they come about due to features of the running application.

A large void remains in the area of network load measurement. Currently, dedicated hardware is generally required, calculating network load by counting packets on the wire. Further work remains in the development of a software monitorable loading statistic that is also portable. One possibility is the monitoring of CPU activity in the network stack, which could be correlated between machines on a shared segment to calculate an estimate of network traffic on that segment. Whether such a measure is implementable remains to be determined.

The most important "next step" in this research is the writing of an actual task scheduler which uses the above described results. Such a scheduler could be fairly easily attached to the PVM software, or implemented on top of another package. This scheduler would use load average correlation, network characteristics, and possibly Cm/Cp ratio to place tasks amongst a group of workstations. The relative importance of these three items could be compared, and any weaknesses examined. This scheduler could learn from past scheduling decisions, developing a

repertoire of known program profiles, which would include slope and y-intercept of runtime-load average regression lines. Any of a number of scheduling algorithms could be employed, or a number of different algorithms could be compared, in each case inserting an information policy based on the results of this research.

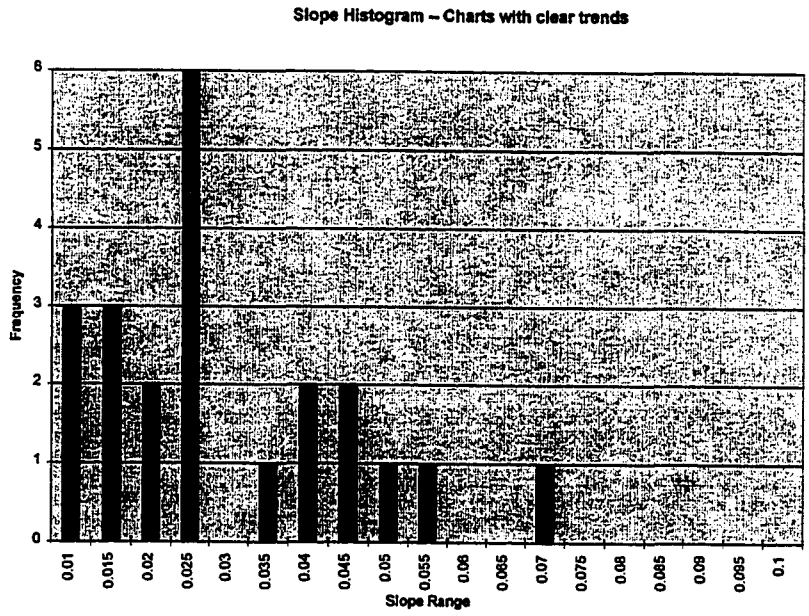

Figure 8.1 Histogram of trendline slopes in load average plots with strong trends.

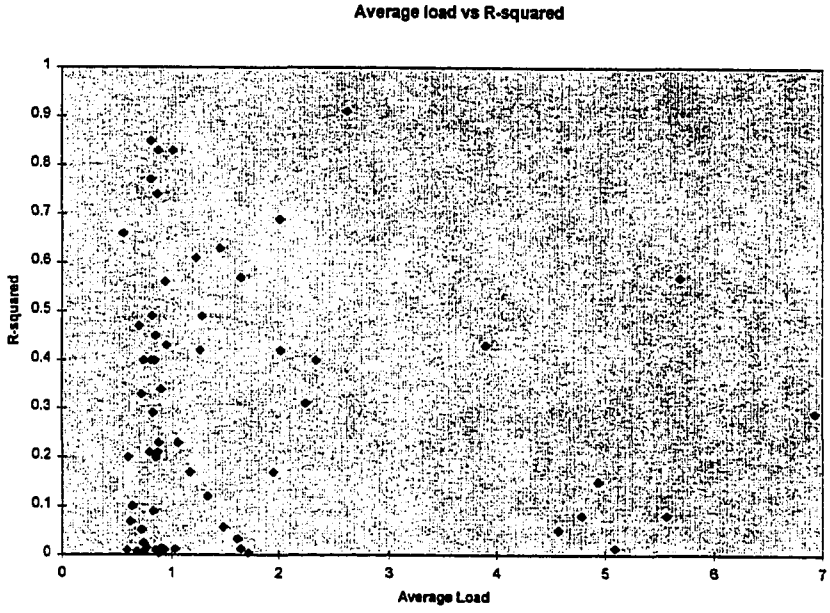

Figure 8.2 Trendline fit values plotted against average load average, for load average plots.

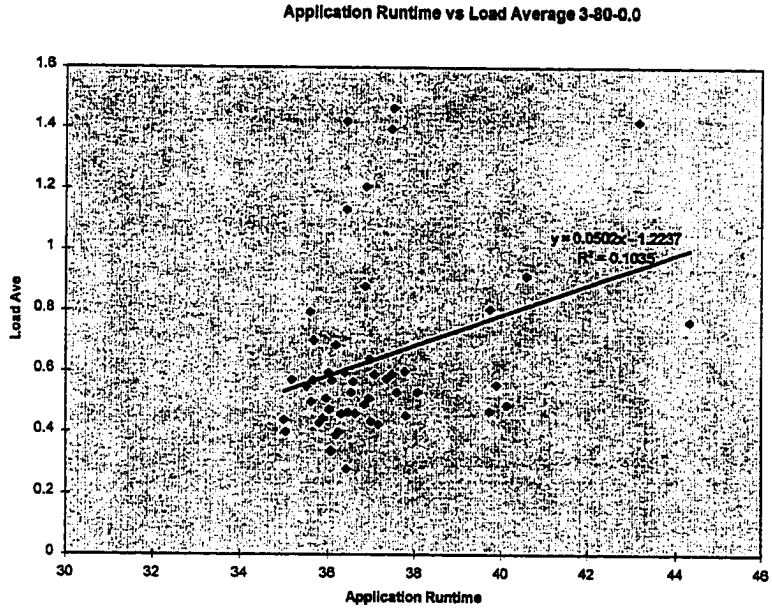

Figure 8.3 Plot of runtime versus load average with many scattered outliers.

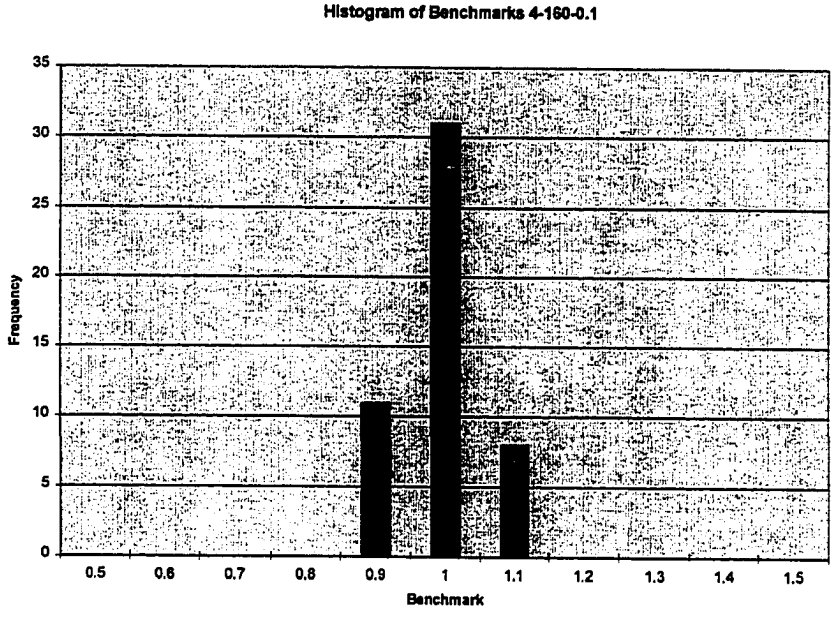

Figure 8.4 Histogram of benchmark loads during a typical set of application runs.

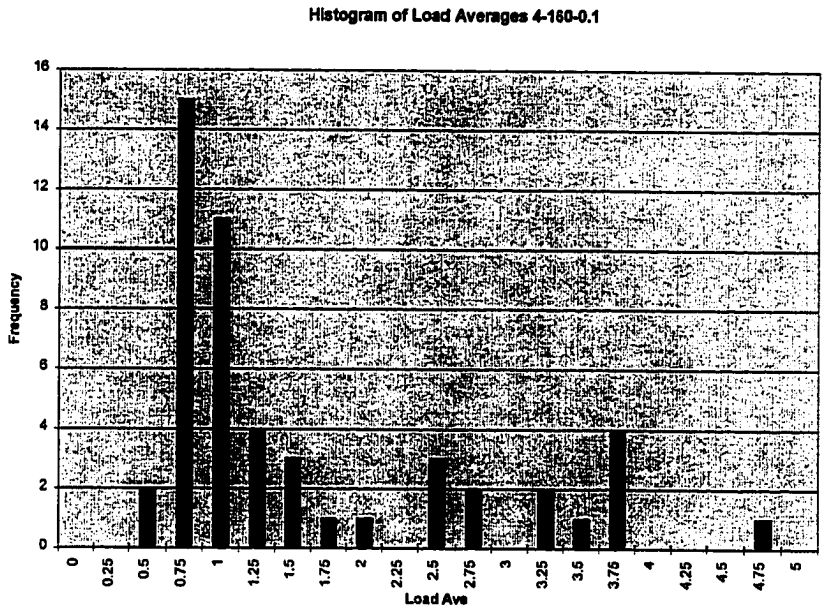

Figure 8.5 Histogram of load average values during the same set of application runs.

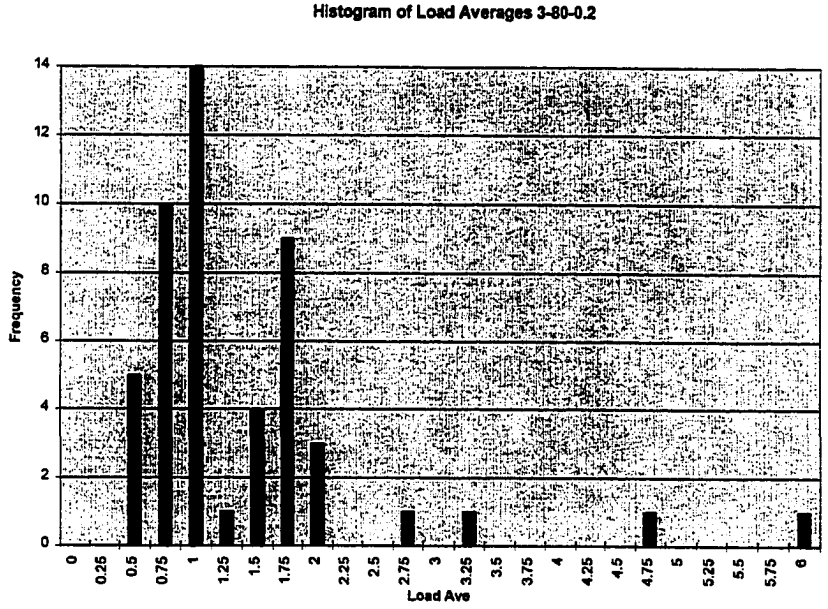

Figure 8.6 Histogram of load average values, showing two peaks.

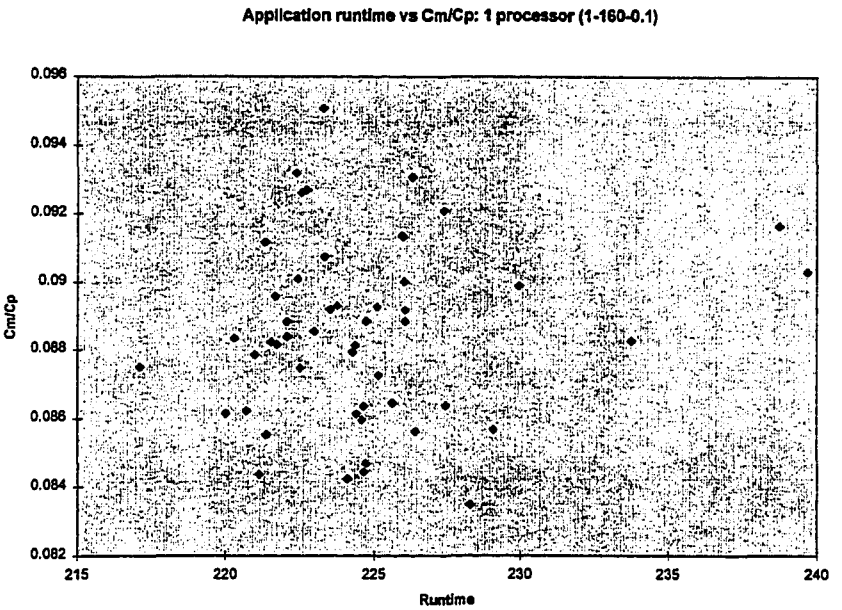

Figure 8.7 Cm/Cp plotted against application runtime for a single processor.

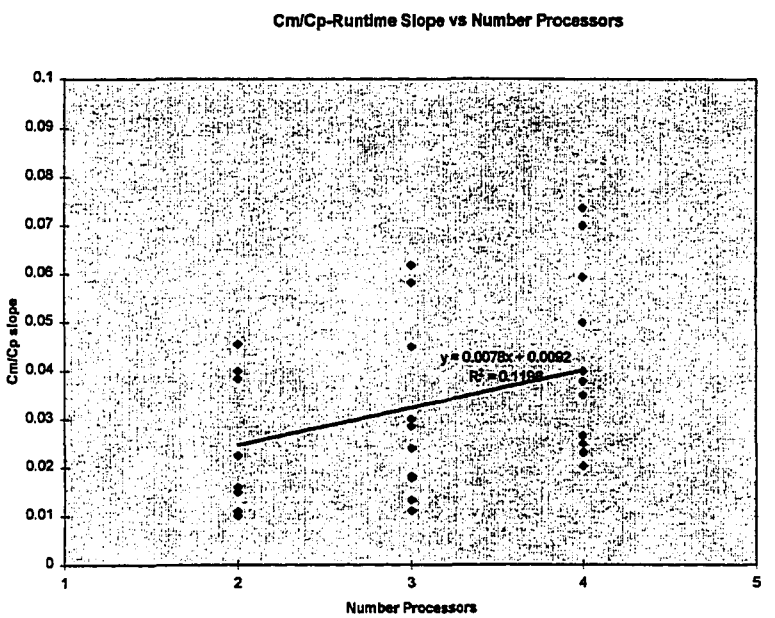

Figure 8.8 Cm/Cp-runtime slopes plotted against number of processors.

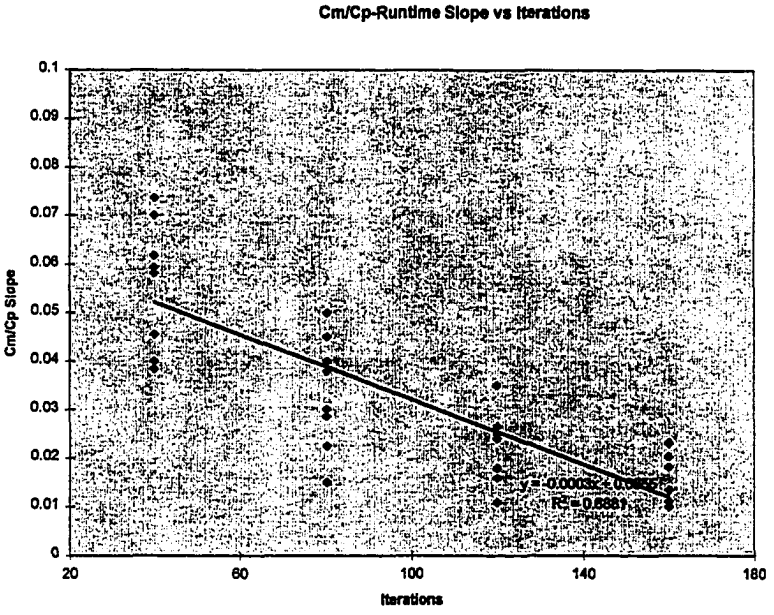

Figure 8.9 Cm/Cp-runtime slopes plotted against number of iterations.

REFERENCES

1. R. Agrawal and A. Ezzat. Location independent remote execution in NEST. IEEE Transactions on Software Engineering, vol. SE-13, no. 8, pages 905-912, August, 1987.
2. Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, pages 47-56, September 1989.
3. A. Beguelin. Xab: A tool for monitoring PVM programs. In *IEEE Workshop on Heterogeneous Processing*, pages 92-97, Los Alamitos, CA, April 1993.
4. A. Beguelin, J. Dongarra, G. Geist, R. Manchek, K. Moore, R. Wade, J. Plank, and V. Sunderam. HeNCE: A user's guide, version 1.2. Technical report, Oak Ridge National Laboratory, Oak Ridge, TN, December, 1992.
5. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing '91*, pages 435-444, Albuquerque, NM, 1991.
6. A. Beguelin, J. Dongarra, A. Geist, and V. Sunderam. Visualization and debugging in a heterogeneous environment. IEEE Computer, pages 88-95, June 1993.
7. A. Bricker, M. Litzkow, and M. Livny. Condor technical summary. Technical report, Computer Sciences Department, University of Wisconsin - Madison. 1991.
8. J. Casas, R. Konuru, S. Otto, R. Prouty, and J. Walpole. Adaptive load migration systems for PVM. Technical report, Oregon Graduate Institute of Science & Technology, Portland, OR, March, 1994.
9. T. Casavant and J. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pages 141-154, February 1988.
10. R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *4th ACM Sigplan Symposium on PPOPP*, pages 249-259, San Diego, CA, May 1993.
11. A. Cheung and A. Reeves. High performance computing on a cluster of workstations. In *1st International Symposium on High-Performance Distributed Computing*, pages 152-160. September 1992.

12. T. Chou and J. Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pages 401-412, July 1982.
13. Y-C Chow and W. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers*, vol. c-28, no. 5, pages 354-361, May 1979.
14. P. Crandall, M. Quinn, and N. Nedeljkovic. A performance monitoring system for networked parallel computing. Technical report 93-80-12, Oregon State University, Corvallis, OR, September, 1993.
15. P. Dasgupta, R. LeBlanc, M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, pages 34-44, November 1991.
16. J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. Technical report, Oak Ridge National Laboratory, Oak Ridge, TN, January 1993.
17. F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software-Practice and Experiance*, vol. 21, no. 8, pages 757-785, August 1991.
18. D. Eager, E. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, pages 662-675, May 1986.
19. D. Eager, E. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. *Performance Evaluation Review*, vol. 16, no. 1, pages 63-72, May 1988.
20. D. Eager, J. Zahorjan, and E. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, vol. 38, no. 3, pages 408-423, March 1989.
21. K. Efe. Heuristic models of task assignment scheduling in distributed systems. *IEEE Computer*, pages 50-56, June 1982.
22. H. El-Rewini and T. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, no. 9, pages 138-153, 1990.
23. T. Freeman and C. Phillips. *Parallel numerical algorithms*. Prentice Hall, New York, NY, 1992.

24. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine, A user's guide and tutorial for networked parallel computing. MIT Press, Cambridge, MA, 1994.
25. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, TN, May 1993.
26. G. Geist and V. Sunderam. Network based concurrent computing on the PVM system. *Concurrency: Practice and Experiance*, vol. 1, no. 4, pages 293-311, June 1992.
27. B. Grant and A. Skjellum. The PVM systems: An in-depth analysis and documenting study - concise edition. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, September, 1992.
28. T. Green and J. Snyder. DQS, a distributed queuing system. Technical report, Florida State University, March 13, 1993.
29. E. Haddad. Dynamic load distribution optimization in heterogeneous multiple processor systems. Technical report TR 93-02, Virginia Polytechnic Institute and State University, Blacksburg, VA, January, 1993.
30. W. Hsieh, P. Wang, and W. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *4th ACM Sigplan Symposium on PPOPP*, pages 239-248, San Diego, CA, May 1993.
31. R. Huber. Network latency and workstation load measurements in a heterogeneous distributed system. Masters thesis, Iowa State University, 1995.
32. B. Jamoussi, M. Bowman, J. Metzner, and W. Adams. A framework for traffic characterization of LAN internetworks. In *Proceedings of the ISMM International Conference on Intelligent Information Management Systems*, pages 16-19, Washington, DC, June 1994.
33. R. Keller and F. Lin. Simulated performance of a reduction-based multiprocessor. *IEEE Computer*, pages 70-82, July 1984.
34. R. Keller, F. Lin and J. Tanaka. Rediflow multiprocessing. In *Proceedings of CompCon '84*, pages 410-417, San Francisco, CA, February 1984.

35. T. Lewis and H. El-Rewini. Parallax: A tool for parallel program scheduling. *IEEE Parallel & Distributed Technology*, pages 62-72, May 1993.

36. F. Lin and R. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pages 32-38, January 1987.

37. M. Litzkow and M. Livny. Experience with the Condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL, October 1990.

38. M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In *Usenix Winter Conference*, San Francisco, CA, 1992.

39. M. Loukides. *System performance tuning*. O'Reilly & Associates, Sebastopol, CA, 1991.

40. E. Markatos and T. LeBlanc. Load balancing vs. locality management in shared-memory multiprocessors. In *1992 International Conference on Parallel Processing*, pages I-258-267, 1992.

41. K. Mehrotra, S. Ranka, and J-C Wang. A probabilistic analysis of a locality maintaining load balancing algorithm. In *The 7th International Parallel Processing Symposium*, pages 369-373, Newport Beach, CA, April 1993.

42. S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba, a distributed operating system for the 1990s. *IEEE Computer*, pages 44-53, May 1990.

43. R. Prouty, S. Otto, and J. Walpole. Adaptive execution of data parallel computations on networks of heterogeneous workstations. Technical report CSE-94-012, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 1994.

44. G. Ramanathan and J. Oren. Survey of commercial parallel machines. *Computer Architecture News*, vol. 21, no. 3, pages 13-33, June 1993.

45. L. Revor. DQS users guide. Technical report, Argonne National Laboratory, September 15, 1992.

46. D. Sept. The design, implementation and performance of a queue manager for PVM. Technical report CS-93-196, University of Tennessee, Knoxville, August 1993.

47. B. Shirazi and M. Wang. Analysis and evaluation of heuristic methods for static task scheduling. *Journal of Parallel and Distributed Computing*, no. 10, pages 222-232, 1990.

48. N. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, pages 33-44, December, 1992.

49. W. Stallings. *Data and computer communications*, third edition. Macmillan, New York, NY, 1991.

50. W. Stevens. *TCP/IP illustrated, volume 1: The protocols*. Addison-Wesley, Reading, MA, 1994.

51. H. Stone. *High-performance computer architecture*, second edition. Addison-Wesley, Reading, MA, 1990.

52. A. Tanenbaum. *Computer networks*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

53. D. Tavangarian, M. Klein, G. Hipper, and M. Koch. Parallel computing in workstation clusters using a concurrent network architecture. In *Proceedings of the ISMM International Conference on Intelligent Information Management Systems*, pages 101-104, Washington, DC, June 1994.

54. A. Thomas and M. Neilsen. A load sharing system for a network of independent workstations. In *Proceedings of the ISMM International Conference on Intelligent Information Management Systems*, pages 97-100, Washington, DC, June 1994.

55. L. Turcotte. A survey of software environments for exploiting networked computing resources. Technical report, Engineering Research Center for Computational Field Simulation, Mississippi State University, 1993.

56. Y-T. Wang and R. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, vol. c-34, no. 3, pages 204-217, March 1985.

57. R. Whiteside and J. Leichter. Using Linda for supercomputing on a local area network. Technical report SAND88-8818, Sandia National Laboratories, Livermore, CA, June 1988.

58. N. Wilson. Personal communication. Instructional and Research Computing, University of Toronto, 1993.

59. T. Yang and A. Gerasoulis. A fast static scheduling algorithm for DAGs on an unbounded number of processors. Technical report, Rutgers University, New Brunswick, NJ, 1991.

60. T. Yang and A. Gerasoulis. PYRROS: Static task scheduling and code generation for message passing multiprocessors. Technical report, Rutgers University, New Brunswick, NJ, 1991.

61. S. Zhou, J. Wang, X. Zheng, and P. Delisle. UTOPIA: A load sharing facility for large, heterogeneous distributed computer systems. Technical report CSRI-257, University of Toronto, Toronto, Canada, April 1992.

ACKNOWLEDGMENTS

I would like to thank my major professors, Dr. Jennifer Davidson and Dr. Jim Davis, for their support and guidance through this work. I also wish to thank my committee members, Dr. Suraj Kothari, Dr. Glenn Luecke, and Dr. Prasant Mohapatra, for their participation and useful comments and suggestions. Finally, thanks are due to Dr. Di Cook and Dr. Paul Hinz for their suggestions regarding the data analysis techniques.