PERFORMANCE MODELLING OF MESSAGE-PASSING PARALLEL PROGRAMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF THE UNIVERSITY OF ADELAIDE
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By

Duncan A. Grove, B.E.(Comp. Sys.)(Hons)

May 30, 2003

Contents

List of Algorithms

List of Tables

List of Figures

Abstract

Parallel computing is essential for solving very large scientific and engineering problems. An effective parallel computing solution requires an appropriate parallel machine and a well-optimised parallel program, both of which can be selected via performance modelling. This dissertation describes a new performance modelling system, called the Performance Evaluating Virtual Parallel Machine (PEVPM). Unlike previous techniques, the PEVPM system is relatively easy to use, inexpensive to apply and extremely accurate. It uses a novel bottom-up approach, where submodels of individual computation and communication events are dynamically constructed from data-dependencies, current contention levels and the performance distributions of low-level operations, which define performance variability in the face of contention. During model evaluation, the performance distribution attached to each submodel is sampled using Monte Carlo techniques, thus simulating the effects of contention. This allows the PEVPM to accurately simulate a program's execution structure, even if it is non-deterministic, and thus to predict its performance.

Obtaining these performance distributions required the development of a new benchmarking tool, called MPIBench. Unlike previous tools, which simply measure average message-passing time over a large number of repeated message transfers, MPIBench uses a highly accurate and globally synchronised clock to measure the performance of individual communication operations. MPIBench was used to benchmark three parallel computers, which encompassed a wide range of network performance capabilities, namely those provided by Fast Ethernet, Myrinet and QsNet. Network contention, a problem ignored by most research in this area, was found to cause extensive performance variation during message-passing operations. For point-to-point communication, this variation was best described by Pearson 5 distributions. Collective communication operations were able to be modelled using their constituent point-to-point operations. In cases of severe contention, extreme outliers were common in the observed performance distributions, which were shown to be the result of lost messages and their subsequent retransmit timeouts.

The highly accurate benchmark results provided by MPIBench were coupled with the PEVPM models of a range of parallel programs, and simulated by the PEVPM. These case studies proved that, unlike previous modelling approaches, the PEVPM technique successfully unites generality, flexibility, cost-effectiveness and accuracy in one performance modelling system for parallel programs. This makes it a valuable tool for the development of parallel computing solutions.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university. This thesis contains no material which has been previously published or written by another person, except where due reference has been made in the text. I consent to this copy of my thesis being available for loan and photocopying from the University Library.

Duncan A. Grove, B.E.(Comp. Sys.)(Hons)
May 30, 2003

Acknowledgements

Firstly, my sincerest thanks go to my supervisors, Dr. Paul Coddington and Prof. Ken Hawick. Ken, your exuberance for parallel and distributed computing whetted my desire to undertake advanced research in computer science, if not my taste for whiskey; slàinte! Paul, your earnest and steadfast mentorship helped reveal to me the true nature of scientific enquiry, for which you have earned my deepest respect and gratitude. I would also like to thank my friend and unofficial advisor – on matters of thesis, life, the universe and everything – Dr. Francis Vaughan: Francis, you were often wrong, but somehow managed to always help me find the right answer!

Many thanks are due to the University of Adelaide and in particular its Department of Computer Science, as well as the Advanced Computational Systems and Research Data Networks CRCs, for supporting my research and giving me the chance to present the fruits of that labour at conferences around the world. I am also indebted to the School of Informatics at the University of Wales, Bangor, for inviting me to spend six months there to pursue my work. Likewise, I am grateful to the Centre for High Performance Computing and Applications at the University of Adelaide and the Australian Partnership for Advanced Computing for making supercomputer time available to me.

Studies aside, I am deeply thankful to the many friends who filled my life with laughter and joy throughout my PhD candidature. While it is impossible to name them all, some demand special mention: Benji, our lunches and walks by the river have held immeasurable pleasure for me; Craig, your insight for happenings in the department provided no end of fun; Kate – yes, you won, but the impetus from our show-down did us both much good; Richard, Jezz and Mike, thank you for looking over bits of thesis draft – it was much appreciated; and finally, to the Hungers crowd – I can't imagine a better bunch of friends to have had along for the ride.

Most importantly, I owe an immense debt of gratitude to my family. Gray, Bron and Lach: I could never be as happy as I am without each of you, who are so precious to me, in my life. Mum and Dad, you are more deserving of my thanks than anyone for bringing this thesis to fruition. Dad, your continual interest, advice, enthusiasm and support were my life-raft; Mum, your love and encouragement were my ration-kit. Last, and most of all, I would like to thank my fiancée, Alex, for, quite simply, everything: Alee, your companionship means the world to me.

Another turning point, a fork stuck in the road.

Time grabs you by the wrist, directs you where to go.

So make the best of this test, and don't ask why.

It's not a question, but a lesson learned in time.

It's something unpredictable, but in the end is right.

I hope you had the time of your life.

So take the photographs, and still frames in your mind.

Hang it on a shelf of good health and good time.

Tattoos of memories and dead skin on trial.

For what it's worth, it was worth all the while.

It's something unpredictable, but in the end is right.

I hope you had the time of your life.

Good Riddance, Green Day

Chapter 1

Parallel Computing

1.1 Introduction

Ever since the invention of the computer, users have demanded more and more computational power to tackle increasingly complex problems. In particular, the numerical simulation of scientific and engineering problems creates an insatiable demand for computational resources, as researchers seek to solve larger problems with greater accuracy and in a shorter time. A typical example of such a problem is weather forecasting. This involves dividing the atmosphere into a three-dimensional mesh, where each cell represents the state of a different part of the atmosphere and varies with time. The temporal and spatial characteristics of the weather can then be simulated by iteratively carrying out a number of calculations (derived from theoretical models) based on the states of each cell and its neighbours. The quality of a forecast using a given theoretical model is determined by the size of the cells that the atmosphere is divided into and the duration that each model iteration signifies. Consequently, the amount of computation required for an accurate forecast is enormous, yet the calculations must be completed in a timely manner to be of use.

A common means of increasing the amount of computational power available for solving a problem is to use parallel computing. A parallel computer consists of two or more independent processors connected by some form of communication network. If a problem can be sub-divided into n smaller problems, a parallel program can be written to concurrently solve those sub-problems on n independent processors. Ideally this would take $\frac{1}{n}^{th}$ of the time that would be required to solve the same problem using one processor, but this is rarely the case in practice for two main reasons. Firstly, many problems contain significant amounts of computation that cannot be parallelised easily or at all. While these serial parts are executing on one processor, other processors remain idle. Secondly, many problems require significant amounts of communication and synchronisation between sub-problems, which can introduce long delays. For examples of both of these effects, consider the weather forecasting example from above. Prime examples of parts of this problem that are difficult to parallelise are reading the initial states into each cell, and coalescing the simulation results to one processor for output at the end of simulation. Communication between cells (and hence processors) is extensive and frequent, because calculations at each of the many cells depend on values at that cell as well as its neighbours, for each of many iterations. Synchronisation is also required at the end of each iteration to keep the entire simulation in step.

Creating efficient parallel programs is notoriously difficult and time-consuming for two main reasons. Firstly, it can be very difficult to determine a near-optimal parallel algorithm for solving a given problem. Secondly, the debugging process for parallel programs is far more difficult than for serial programs because of the exponential number of interactions that can occur between processors [235]. This thesis focuses on the first problem. In addition to all of the well-known problems that are associated with constructing a good serial algorithm, there are a number of problems specifically associated with constructing a good parallel algorithm. These mainly revolve around ensuring that all processors are kept busy and that they have timely access to the data that they require. Quite simply, as mentioned earlier, controlling a number of processors operating in parallel can be exponentially more complicated than controlling one processor. Furthermore, unlike data placement in serial programs, where sophisticated compilation techniques that optimise cache behaviour and memory interleaving are common, optimising data placement throughout the vastly more complex memory hierarchy present in parallel computers is often left to the parallel application programmer. All of these problems are compounded by the large number of parallel computing architectures that exist, because they often exhibit vastly different performance characteristics, which makes writing well-optimised, portable code especially difficult.

This thesis is not about how to write well-optimised parallel programs *per se*; rather, it is about how to construct models of parallel programs that can accurately predict program performance. The main performance metric of a parallel program is normally the wall-clock time, for example as measured by a wrist-watch, required to run a program from start to finish. Other metrics that are also often important are efficiency (i.e. the wall-clock time required by one processor to run a program divided by the total number of processor-hours required to run a parallelised version of the same program) and resource usage (i.e. the amount of memory and disk space required to run the program). Compared to actually building parallel computers and parallel programs and measuring their performance, performance models can be quickly and easily produced and evaluated for both real and hypothetical parallel machines and programs. It is important to realise that these models cannot actually solve the problems that the parallel programs are being written for, they can only estimate how well various implementations will perform.

However, this information is critical at the program design stage, and can help programmers to quickly make effective decisions about parallel algorithm design and the parallel architecture required to achieve good performance for a particular application.

This chapter introduces the *Performance Modelling* (in Section 1.4) *of Message-Passing Parallel Programs* (in Section 1.3) *on Parallel Computers* (in Section 1.2). Section 1.5 outlines the structure of the rest of this dissertation.

1.2 Parallel Computers

Parallel computers [91,321] are commonly classified as either tightly coupled shared memory multiprocessors, loosely coupled distributed memory multicomputers (often called clusters), or a mixture of the two. These different types of parallel computers are distinguished by how processors and memories are connected to each other. In shared memory systems, all processors can access memory through a global address space. In such systems, processors communicate by operating on shared data structures using shared variables for synchronisation. Shared memory systems can be subdivided into Uniform Memory Architectures (UMAs), which are also known as Symmetric Multi-Processors (SMPs), and Non-Uniform Memory Architectures (NUMAs). The distinction between these two classes is that processors in UMAs can access memory in constant time, whereas processors in NUMAs take different amounts of time to access different memory banks, depending on how far away they are from the processor. It is easier to efficiently connect a large number of processors using NUMA techniques – typical current UMA machines have between 2 and 64 processors, whereas some current NUMA machines scale to 1024 processors. Almost all implementations of both UMA and NUMA machines are also Cache Coherent (CC), which means that hardware mechanisms are employed to ensure that cached copies of data stored at individual processors are kept synchronised with global memory. In contrast, in distributed memory systems every processor has its own local memory, and processors communicate by explicitly copying data from one processor's memory to another using message-passing. Distributed memory systems usually have slightly slower communication speeds than shared memory machines, but allow even larger systems to be built – a number of existing systems are built from many thousands of processors – and are significantly cheaper for a given number of processors. A popular contemporary trend in constructing parallel computers is to connect large numbers of SMP nodes using multicomputer communication networks, in an attempt to get the best of both systems.

Many different types of communication networks are in common use, some of which are shown in Figure 1. The simplest is the communication bus (shown in Figure 1(i)), which is often used in low-end shared memory systems. A bus is conceptually a single

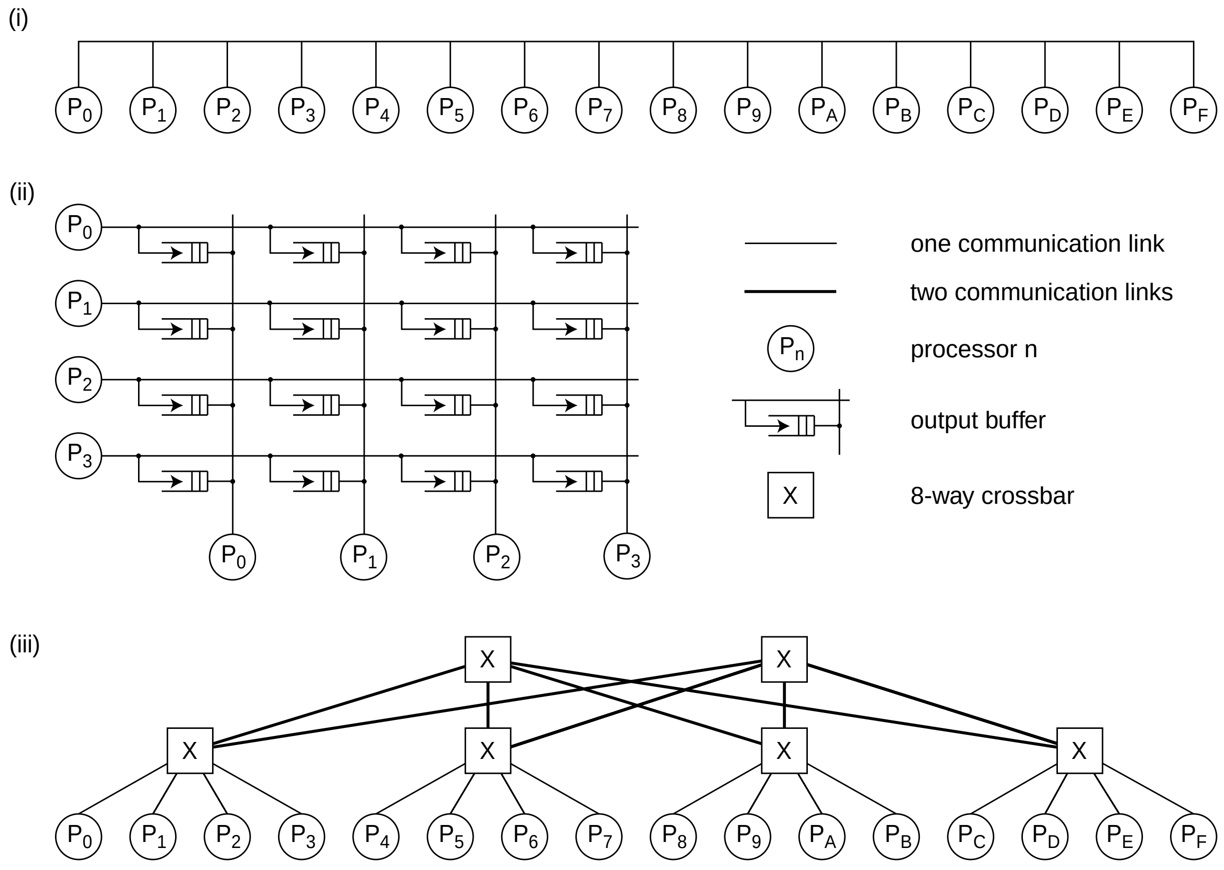


Figure 1: Common interconnection networks in parallel computers: (i) a 16-port communication bus; (ii) a 4-way output-queued crossbar switch; (iii) a 16-port fat-tree, constructed from 8-way crossbar switches.

wire that acts as the connection between processors. It is a shared medium, where the available bandwidth must be distributed amongst all the processors that are connected to it. This approach does not scale particularly well. A slightly more sophisticated network is the multiple-bus, where every processor is connected to several independent buses. Although this provides a relatively easy means of increasing the bandwidth available to processors using commodity bus parts, it suffers from the same fundamental limitation to scalability as the single bus. More sophisticated again is the crossbar network (shown in Figure 1(ii)), where all processors can communicate simultaneously without a reduction in bandwidth (provided that a number of processors are not trying to communicate with one processor, which will result in output queue contention). A crossbar network for n processors requires n^2 switches and either n input-queued buffers or n^2 output-queued buffers. Consequently, while the bandwidth of this network increases linearly with the number of processors, its scalability is also limited.

A crossbar network is perhaps the simplest form of multi-stage communication networks. The term multi-stage means that a message from one processor to another needs to be routed via some intermediary. In the case of the crossbar network, these intermediaries are buffers in the network. More complicated multi-stage networks are often constructed by connecting small crossbar switches in series, which trades off the number and size of crossbar switches required with the number of switches that a message must traverse. Examples of such networks are Delta, Omega, Clos and fat-tree networks. A fat-tree network (shown in Figure 1(iii)), for example, connects processors to each other through a hierarchy of switches, where the bandwidth at each level of the hierarchy grows to allow every processor to communicate simultaneously with full bandwidth (once again, provided that a number of processors are not trying to communicate with one processor, which will result in output queue contention). An alternative to buffering messages in the network is to use processors in nodes with multiple network interfaces as routing intermediaries. Communication networks of this kind are often connected using ring, star, mesh, torus, hypercube and completely connected topologies. The main distinguishing feature between these networks is their degree of connectedness, which determines the number of hops between switching elements that a message must make to travel from one processor to another. For example, in an n processor bidirectional ring network, a message will require up to $\frac{n}{2}$ hops to reach its destination; in an $n = 2^k$ processor k-dimensional hypercube network (where every node has k output links in orthogonal dimensions) at most k hops will ever be required; and in a fully connected network only one hop will ever be required. Naturally, the increased bandwidth and decreased latency available to the more connected networks comes at the price of requiring more network links.

For the purposes of performance modelling, the only information about the communication network that is really necessary is how fast processors can communicate with each other. Therefore, regardless of the network topology that actually connects processors in a parallel computer, all types of communication network can simply be viewed as collection of processors and memories connected by "some" interconnect, along with a performance function that describes how fast messages can travel between various processors under various circumstances. Likewise, the processors can be considered as black boxes capable of computing at a certain speed, thereby abstracting over the details of superscalar functional units, pipelining, vectorising, cache behaviour, and the like. This abstract view of processing and communication is used for the rest of this thesis.

1.3 Parallel Programs

There are a number of different programming methodologies that can be used to create parallel programs that can run on parallel computers [144, 181]. A good programming methodology for parallel computation needs to: 1) allow a programmer to decompose a problem into pieces that may be evaluated in parallel; 2) provide a means for mapping that parallelism to processing power; and 3) allow communication and synchronisation between processes. Note the introduction of this new term: processes are parts of a parallel program that run on processors. Usually in large-scale parallel computing (and exclusively assumed throughout this thesis) exactly one process runs on each processor, so the terms are often used interchangeably. Two of the most fundamental programming methodologies are closely associated with the main parallel computer architectures that are available: shared memory programming for shared memory multiprocessors and message-passing programming for distributed memory multicomputers. In Flynn's famous taxonomy of parallel computing [131], these are both categorised as Multiple Instruction stream, Multiple-Data stream (MIMD) programming methodologies, which signifies that every processor can independently operate on independent data. Flynn termed the other main class of parallel programming methodologies as Single-Instruction, Multiple-Data (SIMD) methodologies[1]. In SIMD computing, a single processor acts as a control unit, and all other processors work in synchrony with it to process data in parallel. For obvious reasons, this is often called data-parallel computing. While large scale SIMD parallel architectures were popular in the early years of parallel computing, they have all but disappeared – although SIMD hardware remains common on the small scale, in the form of high performance vector processors and multimedia instruction sets for commodity microprocessors. Despite this, the data-parallel approach to large scale parallel programming remains alive and well. This is because data-parallel programs can be run on MIMD hardware, which is sufficiently powerful to run all types of parallel programs. In this context, data-parallel programs are often called Single-Program, Multiple-Data (SPMD) programs. MIMD programming methodologies, such as shared memory or message-passing programming, can also be used to create SPMD programs. In addition, however, they can be used to express more general Multiple-Program, Multiple-Data (MPMD) programs.

Shared memory programming is the oldest extant approach to parallel programming, mainly because shared memory multiprocessors were among the first parallel computers and they are still commonplace today. As noted in the previous section, processes in shared memory programs, which are called threads, communicate by operating on shared data in a global address space. Synchronisation using semaphores is required to prevent other threads from reading from/writing to a shared data structure while another thread is writing to it, as described by Dijkstra's seminal papers on critical sections, deadlock, mutual exclusion and the dining philosophers [98, 100], as well as Courtois *et al's* related paper on the multiple readers and writers problem [80]. Shared memory programming is considered by many to be the easiest of the low-level ways in which to write parallel programs, because programmers need only worry about synchronisation; communication

[1]Flynn also specified two other classifications, but they are of little relevance. They were: SISD, for serial programming architectures; and MISD, for completeness rather than for any practical purpose.

is unnecessary because there is always only one copy of any data structure. Moreover, some compilers are available that can automatically extract parallelism from normal sequential source code, which makes writing a shared memory parallel program even easier. Unfortunately, as the memory hierarchy is hidden from the programmer in the shared memory programming paradigm, the performance implications of the memory hierarchy cannot be exploited for maximum scalability.

Message-passing parallel programming, on the other hand, requires the programmer to move copies of data throughout the memory hierarchy using explicit communication. In message-passing programs, each process has its own local memory. Processes must communicate cooperatively and pair-wise, with one process initiating a send operation that must match with a receive operation issued by another process. These send and receive operations may be either synchronous, where a sender or receiver blocks until the call completes, or asynchronous. An asynchronous send operation initiates message transmission and then immediately returns programmed control to the calling process, without waiting for the message to be received. An asynchronous receive operation checks to see if there are any messages in an arrival queue; it receives a message if one is available, otherwise it immediately returns programmed control to the calling process. In order to allow synchronisation, asynchronous message-passing operations must be used in conjunction with operations that test whether or not messages have arrived. Using asynchronous message-passing is more complicated than using synchronous message-passing, but it provides programmers with the ability to overlap communication with computation. This permits programmers to ameliorate the effects of latency by delaying synchronisation until it is absolutely necessary, so that useful computation can be carried out in the mean time. While these factors necessitate greater programmer effort, they provide greater control over data locality and synchronisation, which can be exploited to maximise performance. Because message-passing is a MIMD approach, it is ideally suited to distributed memory parallel computers, and can also efficiently cope with irregular problems. However, careful data distribution and load balancing is required to ensure that processors are kept busy, which further increases programmer effort.

While shared memory and message-passing programming techniques are very different in practice, they are both fundamentally able to describe parallelism in a general way. Because of this, any shared memory program can be translated into a semantically equivalent message-passing program, and vice versa [205,323,369]. However, this equivalence is usually leveraged at a lower level than source code translation. For example, Distributed Shared Memory (DSM) libraries are available to simulate the appearance of shared memory on a distributed memory multicomputer and message-passing libraries are available for shared memory multiprocessors. DSM libraries are usually supplied to bring the convenience of shared memory programming to distributed memory machines, but they do not allow the same levels of performance as pure message-passing programs on the same machines. The same convenience factor is true of bringing message-passing libraries to shared memory multiprocessors (often in nodes of SMP clusters), but these libraries are still able to provide high performance because they continue to expose the memory hierarchy to application programmers. Finally, so-called remote memory operations have recently become possible on some essentially distributed memory hardware platforms and thus one-sided communication routines have been introduced into many message-passing libraries. These routines allow processes to directly read and write data owned by remote processes without their cooperation or any copying involved. This provides the option of shared memory programming simplicity from within the message-passing paradigm.

Data parallel programming techniques evolved out of the use of computational pipelines in high performance vector processing. These pipelines were devised to reduce the control overhead required for highly repetitive but independent low level computation, mainly found in array processing. Obviously these pipelines could be evaluated even more quickly in parallel, and so SIMD data parallel computing was born. As explained earlier, while large scale SIMD parallel architectures have all but disappeared, SIMD-style data parallel programming techniques for MIMD parallel architectures remain popular today. This is because they reduce the amount of effort required by programmers to express parallelism. Data parallel programming techniques provide operations that work on arrays, rather than individual elements of arrays. This simplifies programming because computations that require loops in lower-level parallel languages can be written as single operations. Compilers for data parallel programming languages, such as High Performance Fortran (HPF) [174], OpenMP [258] and HPJava [62], typically allow programmers to use their insight to annotate source code with distribution directives that indicate how data partitioning should take place, so that performance may be maximised. This information is used to automatically schedule whatever computation and communication (usually in the form of message-passing) is required. In addition, procedural data parallel languages usually provide directives that allow programmers to specify that the iterations of a particular loop can be executed concurrently. These directives are not generally verifiable by the compiler, so it is the programmer's responsibility to guarantee their correctness. There are three other issues that also hamper the data parallel programming approach. Firstly, most data parallel languages do not provide a means to intricately control data distribution on machines with complex memory hierarchies. While this is intended to be managed by the compiler, it is extremely difficult to optimise automatically, so performance cannot always be perfectly tuned in these cases. Secondly, load balancing of irregular problems must be achieved by dynamically redistributing data, which can attract a large performance overhead. Finally, the data-parallel paradigm cannot be used to express task-parallel programs at all.

From a performance modelling perspective, data parallel programs are simply message-passing programs in disguise, and shared memory programs can be expressed as message-passing programs. Furthermore, neither shared memory nor data parallel programming paradigms are as powerful as the message-passing approach; for example, they cannot express programs that achieve optimal performance on machines with complex memory hierarchies. With these facts in mind, it becomes apparent why message-passing was once described as "the assembly language of the 1990's" [370]. In this regard, a performance modelling system for message-passing parallel programs can serve as a performance modelling system for all parallel programs.

1.4 Performance Modelling

The word *model* is a very over-loaded term in the field of parallel computing. It is commonly used in three ways, all of which have been described in several papers [172,227,320]. Firstly, a model can be an abstract representation of a computing system where details are removed in order to reveal basic characteristics. A common example of this is an architectural model, which focuses on the structure of the physical and technological properties of underlying hardware components, such as processing units, memory units and communication network. Secondly, the word model is often used as a substitute for the word *methodology*. This is intended to denote a broad way of programming, for example, a code could be programmed using a data parallel programming model. These first two uses for the word model are often used in the context of parallel computers and parallel programs, described in the previous two sections respectively. Finally, however, in the context of performance modelling, a model is a formalised description of a program running on a machine that can be used to reason about the program's performance characteristics [163, 190].

As elucidated in Section 1.1, the main purpose of running message-passing parallel programs on high performance parallel computers is to solve problems that would take too long to run on conventional sequential machines. In order to make enlightened decisions about which parallel computer to buy, or how to effectively code a solution to a given problem, it is very important to understand the performance characteristics of such systems. To reiterate a point made in the introduction to this chapter, the performance characteristic usually of principal concern is overall execution time. Unfortunately, communication performance in message-passing programs is non-deterministic in many realistic situations [211], mainly due to contention. Not only does this make predicting the performance of communication events difficult, it can also result in an enormous number of possible execution sequences that completely alter program structure, resulting in further effects on performance. Although in pathological cases this will create an overwhelmingly chaotic system that makes performance prediction nigh impossible, in almost all real-world programs, the temporal extent to which small timing variations can cause uncertainty is far more bounded, because of synchronisation points during program execution [309]. This is why many message-passing parallel programs exhibit remarkably stable overall execution time, despite internal processes replete with non-determinism. Because of this performance stability, it should be possible to predict the overall execution time of message-passing parallel programs with some degree of certainty. The difficulty lies in accurately modelling both the direct and flow-on performance effects of variability in computation and message-passing time.

Some research groups have developed enormously detailed models of specific parallel hardware/parallel software systems, and used these to accurately predict performance. These sorts of performance models suffer from four main problems. Firstly, they are very complex and time-consuming to create. Secondly, they are large and expensive to solve. Thirdly, they are not usually very flexible so new models need to be constructed for every new situation. Finally, they can be extremely difficult to understand, mainly because they are usually completely numerical rather than symbolic. These four factors make such models fairly useless in the design stage, when it is necessary to quickly decide upon an effective parallel architecture and parallel algorithm for a given problem from a very large number of possible choices.

At the other end of the spectrum, some research has focussed on simple, abstract models that allow the performance of parallel programs under different conditions to be quickly and easily estimated. These approaches are generally designed to be so simple that programmers can carry out estimates using back-of-the-envelope calculations; Amdahl's Law (see Section 2.2) is a famous example of this. While these techniques provide reasonable ball-park estimates of performance in some cases, they fail to provide much useful information for most real parallel applications because they do not take into account any of the complex, non-linear effects such as contention and non-determinism which play such an important part in the performance of large parallel systems.

An intermediate approach allows a trade-off between accuracy and flexibility. This dissertation presents such an intermediate approach. It describes a new performance model that, unlike other intermediate approaches, achieves extremely good accuracy and can be applied to arbitrary parallel programs. In this model a set of parallel program primitives, or fundamental building blocks, are identified that can be used to compose performance descriptions of message-passing parallel programs. Next, a means for precisely quantifying the performance of these building blocks with a new tool called MPIBench is presented. Finally, a Performance Evaluating Virtual Parallel Machine (PEVPM) is used to simulate program performance, with particular attention paid to the effects of contention and non-determinism, which play a crucial role in the performance of large parallel programs. This makes it is possible to accurately predict how real or hypothetical parallel programs will behave at a macroscopic level on real or hypothetical parallel machines. This new model can be used to support better parallel algorithm design and purchasing decisions for high performance computers destined to run specific codes.

1.5 Thesis Outline

The introduction to this chapter in Section 1.1 explained why high performance parallel computing is an essential part of solving very large and complex scientific and engineering problems in a reasonable amount of time. The two main tasks that must be carried out to deliver a good parallel computing solution to a given problem are choosing an appropriate parallel machine (of which various types were discussed in Section 1.2) and writing a well-optimised parallel program to solve the problem (using the paradigms described in Section 1.3). These tasks are often carried out in concert, usually cycling through manifold candidate solutions, all the while conducting time-consuming empirical benchmarking until a satisfactory solution is found. An alternative approach is to use performance modelling techniques to more quickly and effectively choose from a number of possible implementations. This approach has its own complications, of course, and Section 1.4 described the challenges associated with accurate performance modelling of parallel programs, and of message-passing programs in particular.

The remainder of this thesis is devoted to developing a useful, general and accurate performance modelling system for message-passing parallel programs. Many of the inspirational ideas for this work are derived from previous performance modelling techniques, which are discussed in Chapter 2. Using these ideas as a starting point, Chapter 3 develops a new performance modelling system for message-passing parallel programs, called the Performance Evaluating Virtual Parallel Machine, with novel techniques to accurately yet inexpensively account for and detail the many sources of non-determinism and hence performance variability that are observed in parallel programs. In order to validate this new performance modelling system, case studies of three parallel applications on three parallel computers are presented in Chapter 6. Vital elements of these case studies (and of interest in their own right) are detailed studies of the message-passing performance on those three parallel machines, which can be found in Chapters 4 (on point-to-point communication) and 5 (on collective communication). Importantly, these studies required the design and implementation of a new tool for accurately benchmarking and analysing the performance of message-passing operations, called MPIBench, which is also described in Chapter 4. Finally, Chapter 7 summarises the major findings of this dissertation, and indicates some areas that are worthy of further research.

Chapter 2

Performance Modelling Techniques

2.1 Introduction

Throughout the genesis of parallel computing in the 1960s and 1970s, research developments in parallel algorithms were strongly grounded in theory. The main early formalisms for parallel processing were summarised in a book by Peterson [275]. In particular, he identified the work of Dijkstra on P/V constructs in 1965 [98], Karp and Miller on computation graphs in 1966 [200], Bredt on finite state machines in 1970 [50], and extended petri net models in 1974 by Peterson and Bredt [274], Agerwala [3] and Lipton *et al.* [217]. Although these works have provided robust formalisms that continue to serve as a useful basis for modelling parallel processing, the computational requirements of solving these models grow exponentially with the size of a system. This makes them practically applicable to only relatively small systems or where very high costs in terms of solution time or resource requirements are acceptable (see Sections 2.4 and 2.27).

In the 1980s significant research attention was focused on practical ways of devising efficient parallel algorithms using the Parallel Random Access Machine (PRAM) model and closely related variants (see Section 2.3). These models require an algorithm to be completely described in terms of instructions that control individual memory accesses. This allowed researchers to understand the very fine-grained characteristics of the parallel algorithms that they were studying. Unfortunately, the underlying machine models upon which these studies were based were unable to accurately reflect the complexity of the physical processes that occur in a parallel computer system. Although many extensions to the PRAM model were suggested to try and overcome its various limitations, these were largely unsuccessful because the numerous modifications could not be effectively unified.

During the late 1980s and early 1990s researchers began contemplating new ways to tackle the problem of modelling the performance of parallel programs. New methodologies were devised that allowed a direct evaluation of performance from the structure of source code (such as those discussed in Sections 2.5, 2.8 and 2.15). In many cases the programming structures that these techniques encouraged allowed performance to be modelled using several parameters and simple equations. Although these methodologies allowed the creation of efficient programs with predictable performance, the techniques were not amenable to the increasing number of irregular problems that high performance computing was aiming to solve and which required more complicated control structures.

To deal with these more complicated programs, researchers turned to statistical and Markov modelling techniques to try to estimate the performance characteristics of a program. While these approaches were relatively successful in modelling specific situations, they suffered from two problems. The first of these was the high computational requirement of these approaches because solution techniques were generally of exponential order, although solution requirements were potentially less than those generated by the very early techniques mentioned previously. Still, these techniques remain in use today for certain well defined problems. The second difficulty was more significant: these techniques did not really help a programmer to understand their parallel program better or provide insight into ways that would allow them to improve it.

More recently, modelling research has favoured techniques that involve a detailed understanding of the performance of small chunks of code (see Sections 2.7, 2.12, 2.13 and 2.14) rather than a general notion of the average performance of macroscopic blocks of code. In a way, this is very similar to the early PRAM modelling techniques, although significant advances have been made in understanding the general causes of performance degradation in parallel programs (see Sections 2.10, 2.11, 2.15, 2.16, 2.17, 2.18, 2.19, 2.20, 2.24 and 2.25). This philosophy has encouraged the development of performance modelling tools able to tackle arbitrary parallel programs (see Sections 2.21, 2.22 and 2.26).

Research in the field of performance modelling for parallel programs has still to settle on an appropriate level of abstraction. At one extreme, a complete model of every intricacy of a problem is intractable and, what is more, it will not usually increase a programmer's insight into a problem. At the other extreme, reducing the performance of a program to a number of simple equations (such as the techniques discussed in Sections 2.2, 2.6, 2.9, 2.12 and 2.23) is too simplistic and likewise does not significantly help a programmer see ways to improve a solution to a problem. It seems that the most promising modelling techniques lie somewhere in between.

A convenient way of providing an overview of the performance modelling of parallel programs is to present a historical summary of research that has been carried out. This is possible because most of the advances that have been made occurred sequentially, or addressed independent issues. The following sections are organised so as to highlight the major contributions that have formed the basis for the work that will be presented in the remainder of this thesis, although attention will also be drawn to related material within each section.

2.2 Amdahl

The earliest model commonly used to determine the performance bounds of parallel programs was devised by Amdahl in 1967 [14]. When applied to parallel processing, this model can be used to compute the maximum possible speedup S of the parallel version of a program compared with a corresponding serial version of the same program using the formula:

$$S(P) \leq \left(f + \frac{1-f}{P}\right)^{-1}$$

where f is the serial fraction of the program and P is the number of processors available to the parallel version. The serial fraction of a code represents the run-time that is associated with any parts of the code that cannot be parallelised and must therefore be run on only one processor.

2.3 Fortune and Wylie

In 1978, Fortune and Wylie [132] described an abstract model of parallel computation based on the Parallel Random Access Machine (PRAM). Early models related to the PRAM were described by Schwartz [314] and Goldschlager [147]. The PRAM aimed to provide a general model of parallel computation, in contrast to a special purpose model of parallel computation that could fully exploit the available hardware. Unfortunately, special purpose models were rarely portable to other situations. The general model provided by the PRAM aimed to abstract over the details of specific machines and programming styles and instead focus on the inherent parallelism available in a given problem, but possibly at the cost of optimisations based on full knowledge of those details.

The basic PRAM is an idealised parallel processing machine, consisting of P synchronous processors communicating via shared memory. Each processor is able to execute one instruction or perform one communication operation per clock cycle. There are several families of PRAMs which are classified by the semantics used for accessing shared memory. These are: the Exclusive Read, Exclusive Write (EREW) PRAM; the Concurrent Read, Exclusive Write (CREW) PRAM; and the Concurrent Read, Concurrent Write (CRCW) PRAM. In the case of the CRCW PRAM, a further sub-classification applies, based on the conflict resolution used to arbitrate over concurrent writes.

The simplicity and generality of the PRAM model led to its wide acceptance as a research tool, especially for research into: the *concurrent access problem* of how to service concurrent requests without underlying hardware support; the *memory management problem* of how to layout data in order to minimise contention [43, 44]; and the *routing/interconnection problem* of minimising slow-down caused by the routing of data. Unfortunately, the cost model associated with the PRAM did not prove to be very useful in practice. It was unable to express the disparate costs associated with real machines, such as the difference between accessing local or remote memory. Several additions to the basic PRAM model were made over the years in attempts to fix the mismatch between the model and reality. These extensions included attempts to account for processor asynchrony [74], network latency and limited bandwidth [4] and topological locality [171,185,347]. A good survey of the basic PRAM model and its variants can be found in [159]. Despite these improvements, the basic PRAM model is still unable to generate accurate cost estimates for code running on real hardware platforms and it usefulness for performance prediction on actual parallel machines is of very limited scope.

2.4 Hoare; Milner; Alur and Dill

There are several well known methods for the formal analysis of concurrency. Principal among these are Hoare's famous Communicating Sequential Processes (CSP) [177, 178, 301], proposed in 1978, and Milner's Calculus of Communicating Systems (CCS) [243] from 1980; both are closely related [53] and can be traced to Dijkstra's pioneering work on Cooperating Sequential Processes in 1968 [99]. CSP and CCS are often described as *process algebras*, because they provide a powerful mathematical framework for specifying the behaviour of parallel processes. However, brief perusal of any work on, for example, CSP will attest that "all of these dialects have been 'blackboard' languages: they have been used for describing parallel systems when the intended audience is human" [308] (although that cited work was aimed at making CSP definitions more amenable to mechanical analysis). The reason for this is connected with the way in which these formal methods are constructed.

Formal methods define a rigorous syntax and semantics for a small number of basic operators that describe sequential processing, parallel composition, synchronisation, communication, interruption and deterministic or non-deterministic choice. These operators can be used to create models of specific applications. These models are then evaluated for all possible sets of event sequences that could occur, typically with respect to some sort of acceptance condition for the purpose of model validation. Common acceptance conditions are the absence of deadlock or livelock, or constraint checking for model parameters. Essentially, the main purpose of most formal methods is to rigorously prove or disprove the correct operation of concurrent systems. In order to achieve this, every operational detail of a parallel system must be specified in great detail. Because of the complexity involved with this, both in terms of model construction and model evaluation, formal methods are mainly reserved for modelling concurrent systems where failure is not an option. For example, communication protocols are frequently verified for correctness using formal modelling techniques.

One of the main extensions to the original CSP definition was the inclusion of timing information [92,93,291,310,311], resulting in Timed CSP (TCSP). Obviously performance modelling would be impossible without this facility. The extensions are relatively straightforward, and merely consist of annotating CSP operators with a deterministic quantity that represents the time required for the operation to complete. A further experimental addition to CSP resulted in Probabilistic Biased Timed CSP (PBTCSP) [222, 223], which allowed probabilistic models to be attached to the choice operator. However, these probabilistic choice operators only allowed binary decisions to be made based on two probabilities, which seems overly restrictive. Despite this simplification, proofs involving PBTCSP were described as rather complex; perhaps tellingly, PBTCSP models do not appear to have been described or applied elsewhere. Both the binary choice limitation of PBTCSP and the difficulty just described highlight the primary aim of all TCSP-based approaches. Despite the notion of time, TCSP-based approaches are obviously designed to prove the correct ordering of modelled events, rather than the time at which they occur; time is merely added to facilitate modelling of systems where time is inextricably linked with the ordering of events, such as timeouts in network protocols. Arguably, therefore, TCSP-based approaches are not truly intended for performance modelling of parallel systems, although in theory they could be used in such a way.

Conversely, similar work by Alur and Dill on a formal theory of timed automata [13] was fundamentally designed around the notion of time. Therefore, although their approach was also mainly intended for model-checking, it is arguably more applicable to the performance modelling of concurrent systems. Furthermore, and unusual for formal modelling systems, their approach can incorporate probabilistic timing delays with state transitions [12], which makes it particularly appropriate for realistic modelling of physical processes. Recently, some research has applied these general principles to the performance modelling of parallel processes [166].

Unfortunately, however, despite the accuracy and provable correctness that can be achieved with formal modelling tools, they are yet to be generally useful for the performance modelling of large-scale parallel programs. Currently, it is essentially impossible to translate many real-world problems – for example non-trivial message-passing parallel programs – into realistic formal models. Even if this could be achieved, these models generally take an exponential (in the size of the model) amount of time to solve, and any performance implications they uncover may be difficult to understand due to model complexity. In summary, formal methods are not currently able to effectively model the performance of large-scale parallel programs.

2.5 Valiant

In 1990, Valiant [354] described a promising technique for writing efficient, portable parallel programs with predictable performance called Bulk Synchronous Parallelism (BSP). Valiant saw that one of the biggest problems with the message-passing approach was that deriving analytic cost models for performance prediction was very difficult because of the number and complexity of data transfers associated with it. Instead, the BSP methodology requires parallel programs to be structured so that their computation and communication are separated so that each can be considered as a bulk quantity. Because BSP considers communications *en masse* it is simpler to estimate bounds on communication time compared with unstructured message-passing. A good overview and comparison of BSP to other techniques can be found in [326].

Computation in BSP programs flows through a series of parallel *supersteps*, each of which is divided into three phases. In the first phase, each processor/memory pair P is involved in computation using only local data. This can be modelled by McColl's parameter s [234] which represents the number of basic operations (such as addition or multiplication) that can be carried out by a processor in one second. In the second phase, processes share data in a communication phase. During the communication phase, any number of messages can be sent and received. The communication pattern is defined by what is called an h-relation, which involves each process sending and receiving at most h messages. Usually, h is used as a compound parameter that also accounts for the total size in bytes of messages encountered by a process, m; i.e. hm is usually abbreviated to h. The communication time is modelled by the parameter g which represents the time required for an h-relation to complete under continuous message traffic between random processes. The parameter g is normally determined empirically for a particular machine, and is related to the machine's bisection bandwidth, the performance of the network stack, the buffer management used, the routing strategy and the BSP run-time system. In the final phase a barrier synchronisation is performed, where the duration of this synchronisation is modelled by the parameter l which is also determined empirically.

The execution time of a BSP superstep can be computed from the text of the program and the parameters of the target architecture which were described above. The *standard cost model* used to do this is:

$$cost\ of\ a\ superstep\ =\ max\{w_0, ..., w_i, ..., w_P\}\ +\ max\{h_0g, ..., h_ig, ..., h_Pg\}\ +\ l$$

where i ranges over the processes and w_i is the time for local computation at process i. Hence, subject to the constraints of predicting the run-time of serial programs (for example using techniques such as those of Knuth [203] or Dunning [109]) performance can be predicted. Inspecting the standard cost model, it is clear that efficient BSP programs must: balance the computation between processes to minimise w_i; balance the communication between processes to minimise $h_i g$; and minimise the number of supersteps to reduce the number of barrier synchronisations of duration l that are required.

Achieving these requirements for a specific program can be aided by tools from the BSPlib Toolset [176], which can create a performance description from trace data obtained by a once-only run of the code on any parallel machine. The performance model that is generated can be used in the design process for writing the BSP program, when porting the BSP program to new parallel computers, or when making purchasing decisions for a parallel computer (based on its s, g and l parameters). The BSPlib Toolset also provides some insight into the extent of performance improvements that could be made by using an asynchronous message-passing implementation such as the Message-Passing Interface (MPI) [239, 240, 152] or E-BSP [194], which extended the definition of the h-relation by adding notions of locality and unbalanced communication to the BSP model.

2.6 Hockney

In 1991, Hockney introduced a model for describing asymptotic performance [180]. It was based on the maximum rate at which some activity can be performed (r_∞) and an associated value, based on a parameterisation of size, at which half the peak performance can be sustained ($n_{1/2}$). The original purpose of this model was to characterise performance on vector processors, where r_∞ represented the maximum rate of floating point operations that could be achieved, and $n_{1/2}$ represented the vector length for which half of that rate was actually achieved. More recently, this approach has been applied to characterise the performance of many systems. Notably, it is used when specifying a machine's performance on the Linpack benchmark [105] to rank the Top 500 fastest computers in the world [102]. More relevant to performance modelling, in a paper by Getov, Hockney and Hey [143], it was applied to distributed memory multicomputers. They showed that a machine could be empirically characterised by peak and half-performance parameters for a variety of small parcels of computation and communication. The performance of a program could then be predicted by counting its number and size of operations and summing their contributions to overall execution time. While this model is certainly useful for modelling simple, regular parallel programs, it suffers from two main problems in general. Firstly, it becomes intractable for complex parallel programs (at least without some form of automation), because it requires a large number of separate empirical models. Secondly, it does not provide particularly accurate models. Indeed, it simply provides a first order linear approximation to performance characteristics.

2.7 Saavedra and Smith

A simple yet attractive approach to performance modelling was proposed by Saavedra and Smith in 1992 [303,304]. Although this method was not designed to account for parallelism at all, it is possible that the general principles on which it operates could be applied to parallel programs. Saavedra and Smith's work noted that traditional benchmarking techniques alone fail to characterise programs and machines, hence results generated in such a way were tied to a specific program on a particular machine. They showed that a more general performance modelling system could be achieved with: a narrow-spectrum (micro) benchmarking tool [33] as a *machine characteriser* to determine the performance of abstract operations on a particular machine; a *program analyser* to statically count the number of those abstract operations in each basic block and dynamically count the order of those basic blocks during a trial run of an instrumented version of the program; and an "execution predictor", which could combine the results of the previous two stages and predict overall program performance. The one notable limitation of this technique is that it does require each program to actually be run at least once (and again where any different compiler optimisations are in play) prior to modelling, which limits its usefulness as a prototyping tool. To verify their approach, Saavedra and Smith used it to predict the Standard Performance Evaluation Corporation (SPEC) '89 [332] and Perfect Club [87] benchmark performance of several Reduced Instruction Set Computer (RISC) machines. In summary of those results, their simulations could quickly predict overall run-time to within 30% accuracy for 95% of cases. The difficulty of conducting accurate micro-benchmarking was listed as the major factor limiting overall modelling accuracy.

Clearly this general technique could be applied to the performance modelling of message-passing (in particular) parallel programs if accurate micro-benchmarking of message-passing operations could be conducted, and the problems of performance variability and non-determinism (described in 1.4) could be addressed.

2.8 Culler *et al.*

In 1993 Culler *et al.* [85] noted that vast amounts of previous research had focused on overly detailed but flawed models of parallel computation, for example the PRAM which they considered unrealistic because it was synchronous and it assumed instantaneous interprocessor communication [330]. Furthermore, although the BSP model had attempted to bridge these limitations by allowing processors to communicate asynchronously and by accounting for memory latency and finite bandwidth availability, it did so at the cost of prescribing a restricted programming methodology.

Accordingly, Culler *et al.* developed a new model called LogP that is based on BSP but requires less enforced program structure and allows a programmer more control over their program. In contrast to the BSP model, the LogP model does not require a global barrier to separate communication and computation phases and it adds the notion of a finite network capacity that can only support a certain number of messages in transit at once. This makes LogP slightly more general than BSP, although the two models are able to efficiently simulate each other in most circumstances [34]. In other circumstances, LogP empowers programmers to take into account technology trends in order to improve the performance of their solution to a problem. In particular, Culler *et al.* realised that "technological forces [were] leading to massively parallel machines constructed from at most a few thousand nodes, each containing a powerful processor and substantial memory, interconnected by networks with limited bandwidth and significant latency." [85]. The LogP model uses four parameters that were designed to capture the effects of these factors. These are:

- *Computing bandwidth* supplied by the number of processors/memory units, P.
- *Communication bandwidth* between the processors $1/g$, where g represents the minimum gap between consecutive messages.
- *Communication latency* between processors, modelled as a constant L which represents the upper bound of the actual latency that may be observed by a short message when measured under unloaded conditions.
- *Coupling efficiency* between communication and computation, which is modelled by a parameter o that represents the overhead involved in message transmission.

Culler *et al.* [85] also suggested some general programming recommendations to guide efficient parallel algorithm design. Although these are not specifically related to the LogP model, they do highlight common parallel programming techniques that a performance model should be capable of accounting for:

- The *coordination of work assignment* [15, 358], which is concerned with how the processing that must be done should be divided up between available processors.
- The *coordination of data placement* [45, 86], which is concerned with how the data that is required by individual processors should be distributed.
- The *provision of balanced communication* to make the best and most timely use of bandwidth availability.
- The *overlapping of communication with computation* because it is important to keep processors busy with useful work while waiting for data to arrive from remote processes.

2.9 Grama *et al.*

Finding a good parallel algorithm to solve a given problem is usually very difficult because the efficiency of parallel algorithms is often very dependent on critical system parameters, such as the number of processors used, and the latency and bandwidth of the network that connects them. Therefore, scalability analyses are often done to determine how well particular algorithms perform, or scale, as the number of processors, interconnection speed or problem size are varied. One of the most common techniques for assessing scalability is to use an isoefficiency function, described by Grama *et al.* in 1993 [148]. The isoefficiency function of a parallel algorithm is an analytic expression that expresses the increase in problem size that is required to maintain efficiency as the number of processors assigned to a problem is increased. In order to find the isoefficiency function for a given algorithm, the efficiency E of the algorithm is first found by either direct measurement and subsequent curve fitting or complexity analysis (such as that found in [204]) and expressed as:

$$E \;=\; \frac{Speedup}{Number\ of\ processors} \;=\; \frac{1}{1+\frac{T_o}{T_1}}$$

where T_o represents the parallel overhead as a function of problem size and number of processors used, and T_1 is the execution time of the algorithm on one processor. This expression is then transformed via algebraic manipulation to determine an isoefficiency expression for T_o that maintains constant efficiency for increasing numbers of processors.

Practically speaking, because of their simple nature, isoefficiency functions are not able to cope with non-linear sources of performance loss, such as load imbalance or contention. However, the isoefficiency function of an application with regular computation and communication can be used to predict the performance of that application for fixed-size problems on various numbers of processors, or on machines with different network characteristics (provided that these characteristics can be explicitly stated in the expression for T_o). In another form, the isoefficiency function of an algorithm can also be used to determine the size of a problem that can be solved by a given machine in a fixed time. Finally, the scalability of different algorithms can be contrasted by comparing their respective isoefficiency functions with respect to the number of processors used. Algorithms are said to be highly-scalable if the data size only needs to increase linearly with the number of processors used. Poorly scalable algorithms require the data size to be increased more rapidly in order to maintain constant efficiency.

2.10 Adve

In 1993, Adve [1] submitted a dissertation that analysed the behaviour and performance of parallel programs. The model he presented was a significant step forward in the performance modelling of parallel programs because it provided far more qualitative and quantitative information about the performance of a parallel program than earlier methods had, but for comparable computational effort. While most of the models developed prior to Adve's model could only be applied to programs with simple synchronisation structures or required complex and heuristic solution techniques, his model could enable a programmer to relatively accurately predict the impact of underlying system changes as well as guide program design decisions for finding an effective, efficient parallel solution to a problem.

Adve's thesis developed and validated a deterministic model of parallel program performance prediction and testing its accuracy, efficiency, and practicality for real programs on realistic input data sets. The model used deterministic values for mean task times and communication times, while shared resource contention was computed from a separate, stochastic model. Combined with abstract representations for the separate behaviour of programs and systems, the model made it possible to analyse hypothetical programs and systems as well as combinations of these.

Although a fundamental limitation of Adve's model is that it cannot account for variance due to communication delays, his research showed that in reality, for many codes on many machines, the principal effect of random delays is to increase the mean execution time between synchronisation points and to leave the variance unaffected. This result contradicted a common assumption at that time that there was a large variance in parallel execution times. The key implication of Adve's thesis is that "it could be reasonable to ignore the variance of task and process execution times when computing synchronisation costs in a parallel program" [1].

In Adve's model, the total execution time was the sum of four components:

$$t_{total} = t_{computation} + t_{communication} + t_{resource\ contention} + t_{synchronisation}$$

where $t_{computation}$ excluded any computation performed when overlapped communication was occurring. While this model was conceptually simple, in practice it was non-trivial because of the non-deterministic nature of resource contention and because it can be extremely difficult to estimate average synchronisation delays.

Deriving model inputs is an important part of the modelling process. Adve's approach required two main inputs. The first was the task graph of the program. The second were the sets of resource usage parameters for individual tasks which were either deduced or measured by experiment. Constructing the task graph for a program is equivalent to reproducing the parallel control structure of the program. This can be achieved from a basic understanding of the program and carrying out little or none of its actual computation. Adve used a task graph to represent program behaviour because he believed that it is an appropriate level of abstraction for an analytical model. He defined:

- A *task* as the basic unit of work;
- A *task graph* as a directed acyclic graph that describes the inherent parallelism in a program, where nodes represent tasks and edges represent the relationships between tasks.
- A *process* as an entity that could be scheduled on a processor to execute tasks.
- A *condensed task graph* as a task graph, reduced so that each node denotes a collection of tasks that could be executed by a single process.
- A *fork-join task graph*, consisting of parallel phases of computation separated by full barrier synchronisations.

Construction of a condensed task graph reduces a graph so that each vertex represents the work performed between synchronisation points. Condensed task graphs can be orders of magnitude smaller than task graphs and are a useful construct for reducing complexity. A commonly occurring subclass of condensed task graphs are fork-join task graphs which are able to describe programs written in procedural languages. These fork-join task graphs have boundaries between synchronisation points and this helps to avoid a state-space explosion. A number of other models have been developed for reducing task graphs but these are likewise restricted, in particular, to the relatively simple but extremely common fork-join task graphs [16, 106, 348, 349, 360].

One troublesome problem of these approaches is that some programs have non-deterministic processing requirements, which can vary significantly between different executions of a program. This occurs in part because of the presence of data-dependent effects such as conditional branch probabilities or dynamic loop bounds. Although other techniques had been successfully used to model this using stochastic task execution times [16,106,196,230, 231, 306], they were not compatible for incorporation with the deterministic task model used by Adve. Even so, Adve's technique remains applicable for a significant proportion of message-passing programs, with the added advantage that:

> "the deterministic assumption ... implies a unique execution sequence for the program, and furthermore [that] the delay at each synchronisation point in this sequence can be calculated as simply the numerical maximum of the execution times of the synchronising process." [1]

2.11 Singh *et al.*

In 1994 Singh *et al.* [317] published a paper that examined the advantages of emerging methodologies over PRAMs. Singh *et al.* suggested that a useful parallel programming methodology must be abstract enough to be usable, detailed enough to capture fundamental properties, and general enough to run efficiently on different platforms without algorithmic changes. They argued that in order to achieve this, the modelling community needs to obtain a better understanding of the communication properties of parallel algorithms. Although several communication patterns such as dense linear algebra computations, computations on regular grids and fast Fourier transforms are determinable analytically, it is difficult to model the dynamic computations that are crucial in many real world applications such as computations on irregular grids (for example [31,65,90]), in Monte Carlo simulations, or in codes exhibiting adaptive parallelism (such as [63]). Still, they believed that even for these more complex algorithms, some form of characterisation of their communication properties should be possible.

A necessary input to performance models is a description of the communication properties of the program. Determining these properties is one of the most difficult parts of the modelling process. Singh *et al.* identified three sources of communication in a program:

- Inherent communication in the algorithm, i.e. the communication which would occur even if every processor had the entire dataset of the program in local memory.
- Communication resulting from finite local memory capacity.
- Communication from memory organisation effects.

This is instructive because it makes a distinction between local and remote data that a process must access. Furthermore, it highlights that there will be complex performance considerations even for local data depending on where the data reside in the storage hierarchy. In addition to these concerns, modelling of communication is made difficult by several other factors:

- Realistic data sets are often non-uniform over the input domain creating data-dependencies which make analysis difficult.
- Algorithm complexity can render even uniform domains difficult to model because of processing dependencies and interacting data structures.
- Some algorithms have a dynamically changing structure or utilise dynamic load balancing techniques [67, 90, 95, 219, 269, 316, 328].

Most models prior to the work of Singh *et al.* focused on the second and third factors identified above. The main contribution made by Singh *et al.* was the demonstration that characterising data sets is often just as important as characterising algorithms.

2.12 Mehra *et al.*

In a 1994 paper, Mehra *et al.* [237] described how simulation could be a convenient tool for answering "what-if" questions during program design (see also [38, 293]) such as:

- What if the communication links were twice as fast?
- What if the CPU on each node could be sped up twofold?
- What would happen if one could have a machine with 8192 processors running a scaled-up problem?
- What if algorithm B was used instead of algorithm A?
- What if data were distributed across the processors differently?

They suggested several characteristics that a modelling technique should possess in order to answer such questions:

- *Generality.* A model should not be tied to a particular problem size, process mapping or system configuration and performance estimates should be easily adaptable to other platforms by substituting appropriate constants for the relative costs of computation and communication.
- *Accuracy.* A model should provide some quantitative measure as to its accuracy.
- *Rapid modelling.* It should be possible to create a model quickly without having to model minutiae.

At the lowest level, instruction level models have very accurate predictions but simulated runs take even longer than actual runs and they will not generalise to other platforms easily. Higher level models are more amenable to rapid modelling and can provide generality through abstractions that can model data distributions, interprocessor data movements and the like, but it is more difficult to quantify the accuracy of such models.

Mehra *et al.* [237] demonstrated that an effective way to represent a performance model at arbitrary levels of detail was to use a performance language. They found that the performance characteristics of Single Program, Multiple Data (SPMD) programs, which they noted form the majority of all message-passing programs, could be described with a performance language that supported syntactic constructs such as subroutines and loops so that repetitive behaviours could be expressed compactly. Furthermore, they were able to preserve procedure and block boundaries throughout the modelling process. Their modelling technique considered the order complexity of sequential blocks of code as well as dependency information for communication operations. This required a knowledge of the run-times of sequential blocks of code, the lengths and destinations of messages, and the extent of loop bounds. Obtaining this information from a program was a two-step process that involved the extraction of relevant information for parameter estimation from the program followed by formula discovery to fit parameterised equations to measured or hypothetical run-times.

An object-oriented "behaviour description language" was used to represent parallel programs as a collection of autonomous computing objects called players. These players had methods called key application subroutines that could be invoked by messages. Their model used the following constructs to model parallel programs:

- Sequential blocks of code were modelled using the statement (**Run** *duration*); which represented *duration* seconds of computation on a processor.
- Non-blocking send statements were modelled by the statement (**Post** *recipient message* : **length** *bytes*); where *recipient* represented the destination processor of a message containing the data *message* of length *bytes*.
- Blocking receives were modelled by the statement (**Receive** *message* [: **from** *sender*]); where *message* was the label of the data contained in the message and the optional *sender* parameter denoted the source of the message.
- Overheads such as buffer copying were simulated using the (**Hold** *duration*) statement which caused *duration* seconds of idling on the processor. In particular, message-passing delays were determined empirically and modelled using a (**Hold** *msg-xmitdelay*) statement.
- Program control flow was modelled with the following C-like expressions:
  - (**BdlRoutine** *name* (*args*)(*variables*)*statement+*)
  - (**Repeat** *times statement+*)
  - (**If** *condition statement+*)
  - (**Branch** (*probability statement+*)...)

They developed a simulator called Axe which modelled multicomputers of homogeneous processing elements connected by a point-to-point network where each node had its own local memory, CPU and operating system kernel for message forwarding, task scheduling and memory management. Axe was sufficiently expressive to parameterise message-passing programs on distributed memory machines and was used to present a comparison of the profiled performance of two example programs with models that were generated for the same programs. The models were built by hand and required many man-months of effort but were very accurate. However, it was believed that the modelling procedure could be automated and investigations were subsequently begun into an automatic model generator [236].

2.13 Parashar and Hariri

Parashar's 1994 PhD thesis [266] and a related paper authored with Hariri [267] described a novel, interpretive approach for making accurate and cost-effective predictions about the performance of parallel programs. In contrast to existing tools, which required either hand-crafted models or post processed run-time traces to enable performance visualisation and analysis, their techniques provided the means for purely compile-time performance estimation. This allowed the ramifications of decisions about problem decomposition, communication and synchronisation strategies to be far more easily explored. The three step process they described involved: 1) the creation of an abstract model of system hardware capabilities; 2) the creation of an abstract model of application structure; and 3) subsequent evaluation of the execution of the abstract application on the abstract system hardware.

System abstraction involved the manual hierarchical decomposition of a parallel machine's hardware into a system abstraction graph (SAG), where each node in the graph, or system abstraction unit (SAU), represented the performance characteristics of either a processing component, memory component, communication/synchronisation component or I/O component. In contrast, application abstraction was performed by an automatic compiler. Parashar and Hariri implemented a compiler for translating HPF or Fortran 90D programs into Fortran 77 plus message-passing programs, augmented with parametric information describing the performance behaviour of individual programming constructs called application abstraction units (AAU). In particular, they used AAUs to signify sequential computation, process forking, iterative and conditional execution, communication and synchronisation. Iterative AAUs were subclassified into either deterministic, synchronised or non-deterministic varieties. If the compiler could statically decide that a loop had a fixed number of iterations and did not contain any communication or synchronisation calls, this would result in a deterministic AAU that linearly extrapolated the execution time of the sequential computation. The case where a loop contained communication or synchronisation operations would result in a series of interleaved deterministic AAUs and communication/synchronisation AAUs, which would be used to construct a recursively defined analytical performance expression. In the most general case of non-deterministic loop conditions, loop unrolling would evaluate each iteration separately. Conditional AAUs were similarly subclassified, using functional interpretation to resolve execution flow. Where variables affecting control flow could not be automatically determined, they would be tagged and the user would be asked to specify their value during the model evaluation phase. Finally, all AAUs would be combined into an application abstraction graph, defined by the execution structure of the program, with nodes representing computation events and edges representing communication and synchronisation events.

Given the resources defined by a parameterised SAU, an interpretation engine was designed to recurse over AAGs, evaluating the time required for each of its AAUs to execute (and timestamping those events), thereby predicting overall program performance. Estimation of the time required for each AAU to read or write various parts of the memory hierarchy used an approximation of access patterns based on global access and miss counts for each program variable as well as local block access counts and last used timestamps for each program variable, and the cache block size, associativity and replacement algorithm defined by the SAG. Communication and synchronisation performance was modelled using fixed latency and bandwidth parameters, plus the waiting time required to access communication links and buffers. The waiting time was modelled by a global communication structure, which maintained information such as the source, destination, and transmission time of each communication and synchronisation event. This information could theoretically be used by the interpretation engine to simulate the effects of access contention to shared network resources, given a sufficiently detailed SAG. However, it seems that this information was only intended to be used for roughly synchronising the simulated start/finish times of group communication. Indeed, this is confirmed by the existence of a global, user-defined factor $f_{overlap}$ in Parshar and Hariri's model, which must be empirically derived from performance measurements of a specific code, that weights the time required for communication AAUs to account for the overlapping of communication and computation. These facts suggest that the underlying network models used by Parashar and Hariri were not accurate enough to account for network contention nor the non-determinism to which they contribute.

Because of the parametric nature of SAGs, Parshar and Hariri's modelling system could easily predict the effects of different hardware platforms on program performance simply by re-evaluating AAGs for various parameter settings. The output module of the interpretation presents performance statistics, including a breakdown of computation time, communication time and wait time, as well as execution traces that can be viewed in Paragraph [162]. In [267], Parashar and Hariri provided an experimental validation of their modelling system using the NPAC HPF benchmark suite [245]. On tests using up to 8 processors, they managed to achieve prediction accuracy within 5% most of the time, and 20% in the worst case – the latter for higher numbers of processors. In addition, they used their system to make (unvalidated) predictions for larger configurations, and showed predicted results for 16 and 32 processor jobs, with varying processor speed and network load. Importantly, they showed how a hypothetical increase in network load would become extremely important for jobs utilising a large number of processors, clearly showing that very good network models will be required to obtain good performance predictions of programs run on large parallel machines.

2.14 Skillicorn

In 1995, Skillicorn [322] identified three criteria that a parallel programming methodology must meet to be generally useful:

- *Architectural independence.* Since code is likely to be run on a wide variety of platforms, efficiency should not be tied to a particular architecture.
- *Congruency.* It is essential that the true cost of a program in terms of time and resources on the machine is reflected in the programming model.
- *Descriptive simplicity.* If software that is developed for a particular methodology is to outlast a specific architecture, it is crucial that that it be built around a model that is sufficiently abstract.

Widely available libraries for message-passing such as MPI or the Parallel Virtual Machine (PVM) [137] have dealt successfully with the issues of architectural and platform independence (for instance, refer to Baker and Fox [29]) and descriptive simplicity, but congruency has not been adequately dealt with. Although message-passing inherently has the properties that make it a congruent methodology, this had not been specifically studied before Skillicorn's work and the property is still rarely used in practice. One of the contributions of Skillicorn's work was his systematic development of a theory and practice of congruency in message-passing programs.

Since the design of many programs is driven more or less by performance concerns, it is important for a programming methodology to have predictable costs. Run-time is usually the primary concern but others include development costs or resource costs. Cost measures are more difficult to analyse for parallel programs than sequential programs. The construction of cost measures for sequential programs is often divided into two phases: algorithm choice and fine tuning, since only the constants in front of asymptotic costs are affected. This means that the cost functions of serial software are usually composable and total cost is easily computable by summing the cost of its parts since it is convex; importantly, it is not possible to reduce the overall cost by increasing the cost of one part. In 1994, Skillicorn wrote more about the desirable properties of cost systems:

> "It is highly desirable to have the cost function defined in the same compositional way as the program semantics: A cost is associated with each basic operation, and rules are given to compute the cost of a composed program from the cost of its components". [319]

Given this goal, he went on to develop the idea of a *cost calculus* for the performance of parallel systems [325]. This is not simple for parallel systems because there are cost implications associated with rearranging operations. Because of this, such a calculus would require a calculational transformation system where the cost of a transformation could be calculated by a set of rules. Such a system could be made to work for any programming language for which deterministic costs could be obtained for the basic operations, even if they were in a parametric form.

Skillicorn argued that without this kind of composability it would be impractical to derive or optimise program performance in a modular way because implementation choices at some particular part of the program would necessitate a consideration of the cost implications for every other part of the program. Also, for this idea of composability to remain valid, Skillicorn noted that the decisions made by the compiler and run-time system must be reflected in the cost system, even though they may not be explicitly described by the programmer at implementation time. One way to build such a cost system would be to make the programming model low-level enough so that all of the decisions such as data decomposition, process placement and communication would need to be explicitly made by the programmer. Determining the cost of a program would then become an exercise in analysing program structure. Skillicorn acknowledged that MPI programs are amenable to such analysis in another paper also published in 1994, although in that paper he also alluded to shortcomings with the basic MPI program model:

> "The big problem with parallel computation today is finding the right level of abstraction. Machines and architectures change frequently. There is a need to develop software that can run on new machines with relatively little change. A good level of abstraction should be mathematically based. MPI makes it easy to build efficient implementations but does not help much with properties that programmers want." [323]

Although MPI does not always provide the features that programmers want, such as those provided by higher level languages such as High Performance Fortran (HPF), OpenMP, or other massively parallel systems [89], these properties can easily be (and often are) implemented on top of message-passing primitives [205]. Since message-passing can serve as the technological base for such features and many foreseeable developments in parallel programming methodologies (for example those that emerge with new technologies such as the Virtual Interface Architecture (VIA) [78]), this issue is not so important. Developing performance models of MPI primitives will automatically provide a performance model for these new methodologies.

Skillicorn also extended the algorithmic skeleton idea [73, 55] of using pre-structured building blocks for computation to the realm of communication skeletons [324]. Algorithmic skeletons encapsulate control structures such as frameworks for divide and conquer algorithms or task queueing systems. Each skeleton corresponds to a standard algorithm fragment which can be used as part of a larger program. The compiler or library writer chooses how each algorithm is implemented and how intra-skeleton and inter-skeleton parallelism can be exploited on the target architecture. This raises the level of abstraction considerably. It also means that the implementation of each building block needs to only be done once per architecture. A communication skeleton is an interleaving of computation steps and fixed patterns of communication on an abstract topology. Communication skeletons are efficiently implementable and can have defined cost measures. It allows building blocks to be internally parallel but composable sequentially so programmers do not need to be aware of parallel programming pitfalls. A real world example of this philosophy is ScaLAPACK [66], which combines a set of basic communication subroutines [368] and linear algebra subroutines [35] to create a package of parallelised linear algebra subroutines.

2.15 Crovella and LeBlanc

In 1994 Crovella and LeBlanc [82, 83] presented a novel approach to the performance estimation problem called *lost cycles analysis* which distinguished between productive computation and parallel overhead. They reasoned that if they could predict the total performance lost to overhead in a parallel program and then subtract this from the peak performance that it is possible for the machine to reach, they would be able to predict the overall performance of a parallel program.

Although this may seem a convoluted way of predicting performance, it was not so strange after all, since if one knows the characteristics of a machine, its peak performance is trivial to calculate. Furthermore, predicting the performance degradation induced by the individual sources of overhead in a parallel program simplified the overall performance prediction problem by explicitly separating those effects that had a bearing on performance.

The model they designed was carefully crafted to be *complete* so that it captured all possible sources of overhead, as well as *orthogonal* so that the sources of overhead were mutually exclusive. Completeness is critical but it is often ignored in performance modelling tools, usually due to a focus on particular metrics such as cache-hit ratios or message traffic. Although such a focus would be useful if it corresponded to a dominant source of overhead, in reality, performance is often dominated by unexpected effects. Together, the properties of completeness and orthogonality ensured that their system could correctly calculate lost cycles and hence, indirectly determine pure computation.

In practice, the lost cycles approach used a tool to measure the sources of overhead in a program and another tool to fit the measurements that were made to analytic forms. A tool called `pp` measured parallel overhead by processing simple event logs that had been collected at run-time by using a logging library. A tool called `lca` was used to guide the user through the selection of a model to fit the output data from `pp` to analytic forms.

All categories of overhead were measured using the unifying metric *lost cycles*, an aggregate in seconds of parallel overhead. An advantage of the lost cycles approach was that it allowed quantitative study of the trade-offs between effects often modelled in incompatible ways. The categories of performance loss that were measured were:

- *Load imbalance* where any idle processor cycles occurred while unfinished parallel work existed.
- *Insufficient parallelism* for any idle processor cycles that occurred while no unfinished parallel work existed.
- *Synchronisation loss* for cycles that were spent acquiring a lock or waiting for a barrier synchronisation to complete.
- *Communication loss* for cycles that were spent waiting for messages to arrive from remote processes.
- *Resource contention* for cycles that were spent waiting for access to a shared hardware resource.

A small number of measurements for each effect was sufficient to parameterise an example model and lead to an aggregate model spanning the entire parameter space. The model was useful under varying conditions and crossover boundaries where one programming technique outperformed another were able to be obtained by solving simultaneous equations.

2.16 Mraz; Tabe *et al.*

Variation in communication time, often called jitter, is a well-known phenomenon in telecommunication networks. However, studies by Mraz [248] in 1994 and Tabe *et al.* [340] in 1995 seem to be the only substantive investigations into the variance of communication time on parallel computers. The dearth of studies on this topic is probably because highly accurate clocks have not been generally available for most parallel computers in the past, which makes it impossible to time individual message-passing operations, and hence obtain distributions of message-passing performance. Therefore, most benchmarking efforts have focussed on average communication time over a large number of communications. Mraz and Tabe's studies of message-passing variance were possible, however, because their investigations focussed on the high-end IBM SP2 running AIX, which does provide a high resolution globally synchronised clock.

It is important to minimise jitter in real-time systems to maintain a steady flow of information. For example, too much jitter creates pop sounds in audio signals or jerkiness in video signals. Some parallel programs provide real-time output, and in these cases it is obviously important to minimise jitter. More problematic for parallel programs in general, however, is the detrimental effect of jitter on performance. If message delivery to one process of a parallel system is slow, that delay, to some extent, will eventually propagate through to every other process. In the worst case, every other process will sit idle, waiting for the delayed process to catch up. The chances of one process suffering late message delivery increases at least proportionately with the number of processes involved, but usually even more so due to increased contention. Further the performance degradation of a delayed message also increases in proportion with the number of processes. Unfortunately, therefore, while this effect may be negligible for parallel programs running on a small number of processors, it has the potential to severely limit performance when a larger number of processors are used. In fact, at some point, overall performance will begin to reduce as more processors are assigned to a problem; and beyond even that, another point will be reached where overall performance will become slower than if only one processor were used.

Mraz developed a "hot-potato" benchmark that measured the time to pass a virtual token around a ring of processors. Although, strictly speaking, this did not measure individual communication times, the number of processors in the ring was relatively small (2 to 64 in different cases) compared to the large number of repetitions used in other benchmarks to obtain average times (typically many thousands), so fine-grained timing characteristics did not become completely washed out. One noteworthy limitation of Mraz's approach is that he assumed that the time a message spent actually traversing the interconnection network was constant. Although in the case of his benchmark this would have been essentially true, because only one message could ever traverse the network at any given time (ignoring operating system traffic), it does not hold true in general: contention will introduce even more variance. Despite this he obtained some very interesting results. Mraz conducted 100,000 point-to-point communication tests using various message and ring sizes, and recorded the best, average and worst completion times observed, as well as a histogram of results for one test. His results showed that the average and minimum times were of the same magnitude, while the maximum times were up to two orders of magnitude larger. Closer examination of the histogram revealed that the bulk of the measurements formed an exponential-type distribution that tailed off within several multiples of the message latency, while a small but appreciable number of results accounted for the outlying events. Despite the significant variance of the delays, which in themselves would severely affect parallel program performance on a large numbers of processors, Mraz largely disregarded the main (i.e. exponential) part of the distribution in favour of analysing the outliers. By timing the iterations of a busy loop and correlating the results with the message-passing times that were observed, he deduced that the long delays corresponding to his outlying observations were due to operating system interruptions. The most significant of these was the operating system's process scheduler, which ran for 30-40μs every 10ms. Other interruptions he was able to identify were due to the parallel program environment and page faulting. This insight was used to improve the AIX operating system by ganging common interrupts across all processors simultaneously, thereby removing the effect of unsynchronised stalls.

Tabe *et al*'s work extended Mraz's study by quantitatively investigating the effect of the very slow, outlying message-passing times on an all-to-all communication pattern. In particular, Tabe *et al* used a simulator to show how the performance variance of point-to-point message-passing introduced load imbalance on a microscopic scale, which, when summed over all communication operations, caused macroscopic performance degradation. This was validated against actual measurements of all-to-all performance, for the first time providing reasonable proof of why collective communication performance does not live up to expectations based on simple (constant) point-to-point microbenchmark performance.

Two other studies have also briefly mentioned the variance in message-passing communication times. Georgitsis [140, 141] observed that the distributions measured in Mraz's work could well be Poisson distributions, although no quantitative verification of this statement was provided. More recently, in a study on the architectural requirements of NASA's NAS Parallel Benchmarks (NPB) [28], Wong *et al.* [373] noted that the actual distribution of times observed in low-level message-passing on heavily loaded commodity networks did not correlate at all well with typical microbenchmark performance. For their machine, they found that while the return trip time measured by common microbenchmarking tools was only 50μs, the mean time actually observed was 5ms with a similar variance.

2.17 Clement, Quinn and Steed

An interesting approach aimed at dealing with the performance variability of parallel programs was presented by Clement and Quinn [70, 71]. They believed that because the low-level operations in a parallel program could take different lengths of time to complete, for example due to contention, the performance of those low-level operations should be modelled by stochastic values. Rather than trying to measure those values directly, their approach centred around inferring them from an analysis of program structure as well as measurements of overall program performance. The first stage of this process required a static program analysis tool to count the number of low-level operations in each basic block and a (dynamic) instrumentation run to determine the execution frequency of each basic block. In the second stage, this information would be mechanically converted into a (long) analytic expression parameterised by the performance of each low-level operation and input into a standard symbolic manipulation package. Finally, given a number of assumptions, they showed how multivariate data analyses could be used to discern the mean, variance and confidence interval for the performance of each low-level operation. The important assumptions were that: the performance parameters could be well-approximated by normal distributions with constant variances under all circumstances; the program comprised of a deterministic task structure, and especially that the number of loop iterations and the shape of data structures scaled linearly with problem size; communication performance could be well-approximated by a linear model based on message size; and that all models remained invariant across machine/compiler pairs. If all of these assumptions could be met, a program's performance sensitivity to critical system parameters such as message latency or bandwidth could easily be estimated by perturbing the inferred parameter values and re-evaluating the analytical performance expression. As an obvious extension to this, different (even hypothetical) values for the performance of low-level operations could be used to estimate the performance that could be achieved for the application on various parallel machines.

Although this approach was originally designed to cope only with regular communication in data parallel languages, it was later extended by Clement and Steed to deal with arbitrary PVM programs, and incarnated in a tool called APACHE [72, 333]. In this revised approach, they also introduced a simple means to roughly account for contention in shared networks (such as non-switched Ethernet) which involved augmenting the standard linear model of communication time T with a contention factor γ:

$$T = l + \frac{b\gamma}{W}$$

where l is link latency in seconds, b is the size of the message in bytes and W is the bandwidth of the link in bytes per second. Although, in general, γ could vary for every message, the APACHE model does not retain enough information to account for this, so γ must be assumed to be constant and equal to the number of processes. Consequently, a restriction of this model is that it assumes that all processes communicate simultaneously, which is only even roughly true for problems exhibiting regular computation and communication patterns. In several example cases where this assumption was shown to hold, however, Clement and Steed found the simple contention model greatly enhanced the accuracy of their performance predictions for essentially zero extra effort. For up to 8 processors on each of three different parallel machines, they found that APACHE was able to model the performance of Jacobi iteration to within 10% accuracy and matrix multiplication to within 30% accuracy. Prediction accuracy using more processors or on codes with less regular computation and communication patterns would presumably suffer because of APACHE's inability to predict the effects of non-linear performance factors.

2.18 Islam

In a book published in 1995 [188] there is a useful chapter by Islam on *Characterising Parallel and Distributed Applications*. In it he explained that a parallel code has static attributes which are explicitly defined by the programmer, as well as dynamic attributes which can vary from one run of the program to another, even on the same machine.

Even though many dynamic events occur during a parallel program's execution, the processes of a parallel computation generally synchronise the execution of various parts of their own computations with subcomputations of other processes. The result of this is that common process interaction patterns between communication and computation phases can be identified. This is often referred to as a communication pattern. Of course, for any particular program there are dynamic attributes such as message size or iteration time that determine a unique version of the pattern. Typically, an application consists of a series of basic patterns, which Islam identified as:

- *Asynchronous* process interaction patterns with no explicit dependencies between processes.
- *Synchronous* process interaction patterns with explicit synchronisation points at the end of each phase of computation. This commonly occurring pattern is often found in iterative phases of SPMD programs, for which synchronisation occurs at the end of a loop.
- *Pipelined* patterns, either synchronous or asynchronous, where data flow through a process in a predefined order.
- *Client-server* and the related *bag-of-tasks* patterns where client processes compute results and communicate these with several servers or a single server, respectively.

These very standard process interaction patterns are often built on top of a message-passing paradigm using the following primitives:

- *Synchronous sends and receives* where both the sending and receiving processes block until they have completed successfully.
- *Asynchronous sends* where the sending process returns control immediately to the current thread of control and a separate thread is started to deal with the outgoing message. Similarly *asynchronous receives* do not block and wait for incoming data but must be used in conjunction with a polling method that determines when a message has arrived.
- *Exchange* calls where two processes send and receive data simultaneously.
- *Request-response* calls where a message is sent by a process that initiates processing on a remote process and a response is returned.

- *Multicast* and *broadcast* calls where a message is sent to some or all of the processes of an application respectively.
- *Reduction* where one process recombines messages from many sources at one destination.

Although it is possible to view all communication patterns as a collection of sends and receives, and they are usually implemented this way at low levels, aggregation into higher level patterns can provide a better characterisation for modelling purposes.

2.19 Jonkers

In October 1995 Jonkers published his PhD thesis [192] which presented a new modelling formalism and associated software tools for the efficient performance prediction of parallel programs called GLAMIS (GeneraLised Architecture Modelling wIth Stochastic techniques).

To put his approach into perspective, he categorised the ways in which parallel program performance could be evaluated into three broad groups, namely *measurement*, *simulation* and *analytical techniques*. Measurement is only of very limited use during the initial stages of program design and implementation because it forces a measure-then-modify programming cycle which is very labour intensive. In some cases it is not even possible to measure program performance in the early stages of program design because the hardware is not available. Simulation is a more flexible option and can be used when the hardware is not available. It is also more suited to parameter studies although the results of a simulation are numerical and a separate evaluation is required for each set of input parameters. This can be computationally very expensive. Analytical techniques provide the most useful approach in the early stages of program development. Either numeric or symbolic models can be created, usually involving a trade-off between accuracy and flexibility. Numeric approaches are similar to simulation and usually more accurate, but symbolic models have far more scope for providing performance estimates under a wide range of conditions. This is useful in the early stages of program development because it can enable a programmer to find the best solution to a problem. Jonkers classified analytical performance modelling techniques according to their:

- *Expressive power* which is concerned with the ability of a model to describe the performance characteristics of a program and the machine it is running on.
- *Prediction accuracy*, where there are two sorts of errors that can be incurred. Firstly, there are inherent modelling errors because models are a simplification of reality and secondly, analytical errors which occur where small modelling inaccuracies are tolerated in order to make the models analytically tractable.

- *Robustness* which refers to the reliability of the predictions that are given by a performance model. This is often in the form of a sensitivity analysis to identify the situations that may lead to high prediction errors.
- *Analytical complexity*, in particular, the computing resources that will be required for computing a prediction and how long it will take to get that prediction.
- *Scalability* through the utilisation of replication constructs to allow the specification of machines with high degrees of symmetry. Large machines are often highly symmetric and model constructs for replication can improve comprehensibility and decrease analytical complexity.
- *Ease and comprehensibility* of modelling, especially in large and complex systems.

Jonkers chose to use a formal modelling language that allowed for the explicit specification of parallelism because it provided the most expressive power for an analytical modelling technique. It allowed a unified description of both parallel architecture models and program models to be constructed. Within this modelling framework, Jonkers described two general sources of performance loss. The first of these is called *condition synchronisation.* In the static form, which is familiar from normal serial computing, this is also known as a *precedence relationship* and it is implicit in the structure of the program. In parallel programs a dynamic form exists and it is associated with communication between cooperating processes. The second source of performance loss is *mutual exclusion* which is inherently dynamic. This can occur both at the machine level where it is called *resource contention* and at the program level where it is referred to as *critical sections.* Mutual exclusion is one of the most noticeable sources of non-determinism in parallel programs.

Because precedence relations are static, both deterministic and stochastic task times can be naturally modelled using task graphs, although complications arise due to conditional execution. In deterministic models, all quantities such as timing parameters and loop bounds need to be constant or at least representable in a symbolic fashion. In probabilistic models, some degree of uncertainty exists in timing parameters and stochastic quantities are used in the model. Many of the parameters that need to be modelled in a real system are nearly deterministic. For example, there is usually very little variation in time needed for a floating point operation. At a program level, numerical applications often use fixed loop bounds, for example to iterate over the number of columns in a matrix. Message-passing, however, is more problematic. It is dynamic and it cannot be modelled with task graphs unless communication times are assumed to be deterministic. Worse, mutual exclusion is inherently non-deterministic and cannot be modelled using task graphs at all.

Jonkers adopted a hybrid approach that used task graphs to express condition synchronisation and queueing networks to express mutual exclusion. This allowed both the machine and program to be described in a natural, comprehensible way. He devised an algorithm for the analysis of program models represented by general task graphs under the assumption of deterministic task completion times. This considerably simplified his analysis compared with purely stochastic approaches but still yielded very accurate predictions for a wide range of parallel applications.

In GLAMIS, Jonkers represented tasks dependencies with *Simple* (or *series parallel*) graphs. He did this because they could describe most commonly occurring message-passing programs and they were relatively simple to solve. For deterministic series parallel graphs, the total execution time could be deduced simply by using a critical path algorithm. A number of tasks in series could be replaced by one task with a delay equal to the sum of the delays, and a number of tasks in parallel could be replaced by a task with a delay equal to the maximum delay of the parallel tasks. Although Jonkers was focused on deterministic models, the previous work of Gelenbe and Liu [138] had shown that stochastic models were still feasible. In those models Gelenbe and Liu had obtained the overall distribution for parallel constructs by multiplying the constituent distributions and the overall distribution for series constructs by convolution.

Although Jonkers assumed task times to be deterministic in GLAMIS, service times for the queueing model representing the machine were probabilistic. Parallel architectures were modelled as virtual machines using a logical instruction set. Each of the instructions was parameterised and modelled by a queueing centre. The virtual machine instructions he used were floating point instructions, memory loads and stores, as well as message-passing primitives such as sends and receives. Programs were modelled using a sequence of series-parallel sections using a directed acyclic graph. The annotation of nodes in a task graph included an instruction count of the average number of times each instruction type is called in that task. Mapping from instructions to visit counts, which defined the queue structure, was machine-dependent. This machine model was formally defined using the tuple $< Q, S, Y, M, \delta, I, F >$. Q represented the queueing model elements with mean service time S, queue type Y, relative speedup δ and number of equivalent queueing centres M where replication was involved. I represented the logical instruction set of the machine and the function F was used to map instructions to visit counts.

One of the drawbacks of Jonkers' technique is that model construction involves significant manual effort, especially while building machine models. Regarding program models, Jonkers believed that construction could be partly automated using the by-products of parallelising compilers.

2.20 van Gemund

In 1996, van Gemund described the performance modelling of parallel systems [355]. He attributed the difficulty in modelling parallel processes to the role that synchronisation plays in parallel systems. Like Jonkers, he divided this synchronisation into a static component and a dynamic component. The static component, known as condition synchronisation, occurs because of the precedence relationships between tasks. The dynamic component is caused by mutual exclusion which is due to resource contention at the machine level and critical sections at the program level. Van Gemund extended the scope of Jonkers' work to include the effects of conditional control flow, which also alters the dynamic nature of a program. The dynamic forms of synchronisation and conditional control flow lead to non-determinism which is very computationally expensive to model accurately. Accurate modelling involves solving a problem where the solution complexity grows with exponential order compared to the size of the system. This is because it requires evaluation of all the combinations of event orderings that can occur in the problem. As background for his work, van Gemund classified existing parallel program performance modelling techniques according to their ability to model synchronisation and conditional computation. Brief summaries of these approaches follow.

Deterministic graphs are a popular choice for modelling condition synchronisation in parallel programs because they have a very low solution cost. The crucial abstraction is performed during the modelling step which extracts a graph representing the inherent parallelism in a code from the program text. After this, the technique yields to an exact analysis and execution time can be determined using a critical path algorithm. Unfortunately, these graphs cannot model mutual exclusion or conditional control flow which limits their predictive capability.

Stochastic graphs are an extension of the deterministic graph concept. Stochastic rather than deterministic values are used to represent task times by annotating graph edges with distributions that approximate the effect of conditional control flow and mutual exclusion (see Tayyab [344] and Yazici [376] for examples). Unfortunately, while it is relatively easy to determine average execution time from such graphs, determining the distribution of run-times that will be observed is very difficult unless restrictive assumptions are made [139,215]. Instead, some approaches have investigated the simpler problem of determining performance bounds [110, 215].

Queueing networks can model mutual exclusion between contesting processes by using queueing centres to simulate the delays associated with access to a shared resource. For example, message-passing delays can be approximated by setting the mean service demand and service times at queueing centres based on the number of messages in transit and minimum message latencies. Such networks are usually solved for an average case using steady-state analysis. Often, exponentially distributed service times are assumed (citing a justification offered by Salza [305]) since this allows the queueing networks to be mapped to Markov chains. This is advantageous because such models can be solved using Mean Value Analysis (MVA) [289,294] which allows an exact solution in polynomial time. Unfortunately, this assumption places unrealistic restrictions on the structure of code that it can model effectively [192]. Furthermore, although queueing networks can be good at estimating the delays in single phases of a parallel program, they are not able to model condition synchronisation. Because of this, they have been combined with task graphs to form hybrid models [192], resulting in a model that is essentially equivalent to a stochastic graph.

Petri nets (see Breanl [49] or Murata [250] for an overview of their properties) have more modelling power than task graphs or queueing networks and they can accurately model parallel systems [19, 199]. In fact, the closely related *extended petri nets* are as powerful as a Turing machine and theoretically they can describe any program written in any programming language. In the case of performance models, timed petri nets represent the time of basic operations with a transition time at each node of the network, condition synchronisation in the structure of the network, and mutual exclusion by using a non-determinism operator. In the case of a message-passing parallel program, a petri net model is normally constructed by separately modelling the blocks of computation between two consecutive communication statements. These submodels are then merged using rules that model communication among the processes. Although petri nets are very powerful, they have exponential solution complexity and are prohibitively expensive to solve for models of parallel programs of practical size. Furthermore, models of large programs become difficult to comprehend and aid programmers very little in understanding performance implications of algorithm choice during program design.

Simulation languages allow for arbitrarily high levels of modelling detail which gives them the potential to be most similar to the actual system. Simulation languages naturally account for condition synchronisation and mutual exclusion. Data-dependent control flow can be supported although this is often in a probabilistic context using weighted model parameters. Deriving performance estimates from simulation has the advantage over directly measuring actual systems because it is non-invasive and can even be done for hypothetical machines. Unfortunately, simulation languages are typically numerical rather than analytical in nature, which restricts the insight they can provide for performance optimisation.

Analytical modelling techniques (such as CSP and CCS from Section 2.4, APACHE from Section 2.17, PEL [215], TCAS [288] and others [25, 30, 88]) have an underlying calculus that can be used to describe the performance of a code. Successive approximations are used that produce less accurate but simpler models and ultimately, a performance model is expressed as a system of equations that retains parameters of interest. The advantage of this is that a model could then be used to easily investigate the performance implications of different parameter choices [70]. For example, Mendes *et al.* [238] produced a symbolic model for a code in terms of its problem size, the number of processors used, and a number of system parameters that were evaluated via benchmarking.

None of the analytical approaches before that of van Gemund's accounted for mutual exclusion. Van Gemund developed an analytical technique with a mathematical framework and calculus for approximating the performance of parallel systems aimed primarily at the initial phases of program design where extremely low solution cost rather than high accuracy is important. Techniques with low solution cost are of vital importance in initial design phases in order to study the performance implications of design choices through exploration of large parameter spaces. Van Gemund described a static approximation to the effects of mutual exclusion using his structured PerformAnce ModEling LAnguage (PAMELA) which defined a symbolic calculus for describing synchronisation. This calculus provided a set of approximation rules that, when repeatedly applied, reduced a model to an analytical expression in the time domain where machine and program parameters of interest were symbolically retained in the final time domain performance model. This allowed the computation of a deterministic time domain result that could be easily evaluated for different input parameters. This relied on several restrictions (in particular it assumed individual task times to be deterministic, based on previous work by Adve [1,2]) to support low-cost, statically determinable models that had robust accuracy across entire parameter spaces. Examples of similar techniques can be found in a number of studies [112, 113, 114, 115, 116, 155, 356].

PAMELA evaluated performance models using critical path analysis for condition synchronisation and bounds analysis to approximate mutual exclusion. It used the same syntax to describe programs and machines which allowed a unified description of mutual exclusion at both the program and machine level. Van Gemund called the combination of critical path analysis and bounds analysis *serialisation analysis* since it was based on identifying the potential serialisation of contending model parameters. In addition, the symbolic nature of his analysis process allowed the effects of conditional control flow to be retained in the final performance model. The constructs that PAMELA used to model condition synchronisation, mutual exclusion and conditional control flow are summarised below.

Condition synchronisation was modelled in two ways. Condition synchronisation that was implicit in the structure of a program code was modelled using the sequential operator ";" and the parallel operator "||". Explicit condition synchronisation could be used to expressed using $wait\{c_1, c_2, c_3...\}$ and $signal(c_1)$ directives, which acted on conditions C_i. *Mutual exclusion* was supported at two levels of accuracy. At an exact level, achieved using numerical simulation, PAMELA supported the P/V construct of Dijkstra [98]. Van Gemund also defined a PAMELA construct to approximate mutual exclusion where access to resources (such as memory banks or processor scheduling slots) was controlled by *use* constructs. He defined $use(U, \tau) = P(U); delay(\tau); V(U)$ where U represented a set of resources. From this definition he developed a series of equations for calculating the delays caused by *use* statements in the time domain.

PAMELA provided an interesting solution for a fundamental problem of analytic prediction: static un-decidability due to data-dependent program parameters. *Conditional control flow* was partially supported by symbolic parameter retention. For example, consider the conditional operator $if(p)\ L$ where p is the probability of running the code L. In the final model the parameter p would be retained, which could then be specified by human intervention, or the model could be subjected to analysis given various p's. Another example of this is the $while(c)\ L$ loop where c is the loop condition and L is the loop code. Here, the unbounded while loop would be modelled as a loop bounded by a user specified parameter c. In other cases, van Gemund showed that it was possible to derive valid performance models of a program despite data-dependent execution since it is often known how much work needs to be done, even if the exact order in which it is computed is not known.

Calibration was not supported by PAMELA although van Gemund recognised its importance. Some analytical approaches feature calibration where certain parameters are determined by direct measurement of calibration routines rather than by calculation [135,216]. A similar example of this can be found in a paper by Gupta and Banerjee [155] where communication cost was estimated in terms of an MPI-like abstract kernel of high level instructions.

In general, the execution time of code modelled using PAMELA should be stochastic with a finite distribution between a lower bound T^l and upper bound T^u. However, calculation of these distributions, even given the assumption of deterministic task times is a very complicated task and was not tackled by van Gemund in his dissertation. Instead, he chose to approximate these distributions by their lower and upper bounds. He developed exact solutions for the lower bound of run-time, T^l. Developing an exact bound for T^u proved to very complicated. Instead he derived equations for the upper bound of a distribution that was correct to within an unknown factor less than 2, which depended on specific circumstances. This was not a major concern, however, since he also showed that the vast majority of systems had a T^{mean} much closer to T^l than to T^u anyway.

2.21 Labarta and Girona *et al.*

One of the most mature tools for predicting and visualising the performance of message-passing programs is Dimemas, which was conceived by Labarta and Girona *et al.* and described in their 1996 paper [210]. Dimemas simulates the time behaviour of a message-passing program using a trace file of its computation and communication structure and some simple parameters describing a target machine. For MPI programs, Dimemas uses trace files generated by linking a program with the VampirTrace [260] instrumented MPI library and running the resultant code on an existing parallel machine. Any parallel machine can be used for this instrumentation run - even a uniprocessor workstation running a number parallel processes in a time-shared fashion. Dimemas analyses the trace file based on elapsed CPU time measurements rather than wall-clock measurements for each process, and automatically adjusts the trace file to remove the time spent on unrelated processes. This introduces non-linear errors, for example by ignoring the effect of cache flushing as processes are swapped in and out, but a paper by Girona and Labarta [145] argued that the overall accuracy of the technique remains reasonable in many cases. A more fundamental limitation of Dimemas, however, is its inability to model non-determinism. Because the trace file is immutable, determined by the order of events during the instrumentation run, applications where the number and sequence of computations and communications depends on timing cannot be accurately modelled using Dimemas.

The target architecture supported by Dimemas is a network of SMP nodes, each with their own local memory and one or more processors. Processors are connected by multiple levels of bus-based communication links. Point-to-point communication performance is computed using a simple linear performance model, which can be roughly determined from the trace file or explicitly specified, augmented by a simple principle to roughly account for dynamic (non-linear) network contention: only one message may traverse a given bus at a time. More specifically, if there are m messages ready for transit but only b available buses then the messages are serialised into $\lceil m/b \rceil$ waves. This constitutes a first order approximation to contention, which is inherently non-linear.

For collective operations, a fan-in/fan-out model based on point-to-point communication is used [146]. The time required for the fan-in stage of a collective communication is modelled by:

$$T_{fan-in} = \left(l + \frac{s}{w}\right) * model\_in\_factor$$

where l is the link latency, s is the size of the data to operate on, w is the link bandwidth and $model\_in\_factor$ is used to describe the number of steps in the collective algorithm. The expression for the time required during the fan-out stage of a collective algorithm is identical, except that “in” is replaced with “out”. The $model_in/out_factor$s for any particular collective routine can be specified by the user, although a number of sensible defaults exist. Typical *model_in/out_factor*s are *constant-*, *linear-* or log_2-based functions, parameterised by number of processors. It is worth noting that this model assumes an implicit barrier so that all collective operations start at the same time. While this greatly simplifies the model, it comes at the cost of ignoring the effect of staggered starts on collective performance, which is common, especially in irregular programs.

Given a trace file of a program and the architectural parameters of a target platform, Dimemas performs a critical path analysis. It reports summary statistics such as total CPU usage for different code blocks and their importance for overall program execution time. It also produces output traces that can be viewed as space-time diagrams using Vampir [260]. Because the architectural parameters can be changed and the performance model re-run, Dimemas also allows a user to easily analyse a program's performance sensitivity to factors such as network latency or bandwidth. All of this information can be used to study load imbalances and potential parallelism, and ultimately predict the benefits of particular code optimisations.

2.22 Dunlop and Hey *et al.*

In 1997, Dunlop and Hey showed that using instruction counting to model serial performance and simple ping-pong results to model communication performance fails to account for the memory hierarchy and can lead to a very significant mis-estimation of performance [170]. While it is well appreciated that cache behaviour plays a significant role in most serial codes [165], its effects are often neglected when predicting the behaviour and performance of parallel programs [170]. A possible reason for this is that cache behaviour is critically dependent on memory access patterns, hence a good model of cache behaviour can only be obtained by exactly simulating every single memory access. This is tremendously expensive; simulating the cache behaviour of a program will be far slower than actually running that program. Despite this, it can be very worthwhile to model and tune the cache behaviour of serial code because of the enormous performance improvements to which it may lead. Unfortunately, modelling the cache behaviour of parallel code is far more problematic. Obviously, there will be many processors to model, which makes cache simulation even slower. Far worse, though, are the effects that process scheduling and non-determinism – which are exponentially greater for parallel programs – can have on cache behaviour. Unlike the execution of serial code, where interruptions almost always lead only to transient anomalies in cache behaviour, the slightest change in the execution structure of some parallel codes can lead to radically different cache behaviour. Solving each of these behaviours is completely intractable. Despite this, however, it should still be possible to individually model the cache behaviour of the serial/computational parts of a parallel program in many cases. Strangely, however, very few modelling systems for parallel programs seem to take advantage of this reprieve.

In order to fill the void of performance estimation tools for parallel programs that take into account the memory hierarchy, Dunlop and Hey developed a Performance EstimatoR FOr RISC Microprocessors (PERFORM) [107, 108]. PERFORM uses execution-driven simulation to run the control framework of serial sections of code, taking into account any variables that may affect control flow. In addition, PERFORM relies on several simple but effective heuristics to avoid having to execute the entire control structure of a code, yet takes care to maintain reasonable accuracy. These mainly involve avoiding the need to simulate every loop iteration. For example, one heuristic is to only simulate a small number of loop iterations, check whether the completion time for each repetition has reached a steady state, and if so then use extrapolation to approximate the time required for all remaining iterations. Putting these optimisations aside, the time required to execute a serial section is computed by summing the time required to execute any data movement, computation or library calls that it encompasses. The time required for data movement - either explicit stores or implicit loads – through register sets, cache memories and main memory, is simulated using a fairly detailed cache model; computation time is determined using instruction timing formulae for basic arithmetic and logic operations (but note that multiple functional units are not taken into account, so timing overestimates will result for multiple-issue CPUs); and the time required for library calls needs to be supplied from empirical benchmark data. The PERFORM simulator was shown to achieve good accuracy for a Jacobi iteration example, but took about the same amount of time to run as the actual program [170]. In the same paper, Hey and Dunlop also presented results that showed that cache behaviour could also have a direct effect on message-passing performance. Using a tool based on Lebep [122], they showed that the message-passing time for strided data (for example, communication of a row of a matrix stored by column) could be significantly different than for contiguous data. Using PVM on a Meiko CS2 they showed that the extent of memory stride could alter communication performance by up to 1000%. On a workstation cluster using MPI, however, an overhead of only 20% was observed.

By considering message-passing calls as library calls, PERFORM could also be made to simulate message-passing parallel programs. This idea was investigated by Reeve [292], although he chose to abstract over the details of serial computation. Instead, he focussed on the communication operations in a message-passing program, statically generating instruction streams for each processor based only on the total number of processors and each individual processor's identity tag. In particular, he conceived a model with six basic operations: send, receive, asynchronous send, asynchronous receive (all parameterised by message size), wait (for asynchronous operations to complete) and work (parameterised by a number of floating point operations). Although very thin on details, Reeve explained that the instruction streams could be deterministically evaluated using two queues for each process, to represent local execution time and potential communication events [292]. Whenever two processes participating in a communication become ready, their local execution time could simply be incremented according to a model of communication time; Reeve chose to use a linear latency/bandwidth model with bandwidth sharing, ignoring non-linear network contention. The results and discussion presented by Reeve suggest that this is reasonable for a small number of processors, but that detailed contention effects for the interconnect network and memory hierarchy will play a significant role when run on a larger machine. While the work described above has made good progress at accounting for the memory hierarchy, a related study by Hernandez and Hey [167] claims that substantial effort is still needed to marry the performance results of low-level communication benchmarks with the performance achieved in real parallel codes.

2.23 Becker *et al.*

In 1997, Becker and his colleagues from the Whitney project published a paper [32] that developed several simple models of NASA's NAS Parallel Benchmarks (NPB) version 2 [28]. The models were used to explore the performance trade-offs involved in building a balanced parallel cluster computer capable of supporting scientific workloads. The models that were developed were simple analytical expressions using the number and size of messages sent by each benchmark. Coupled with measurements of single processor performance, network latency and network bandwidth, these models were used to predict performance and hence find a well balanced machine for running NPB-type applications.

It was recognised that the parameter space for cluster design was quite large. Designers must consider factors such as CPU type, the amount and type of memory per node, the network technology and topology, and operating operating system. The Whitney team's design goal was to choose the most suitable configuration of a machine for a certain code based on metrics of speed, efficiency and financial cost. Clearly, not every possible cluster architecture could be physically tried. Thus, by developing simple abstract models of the NPB benchmarks as representatives of the target workload, the Whitney project evaluated the effectiveness of various architectures by calculation and simulation.

The architectural model they used for the system was characterised by network latency l in seconds, bandwidth b in bytes per second, single node performance f in operations per second, the number of nodes p and monetary hardware cost. This allowed the comparison of different technologies, for example Fast Ethernet versus Myrinet, at any given price. The communication model they used for calculating the message-passing time was $msgtime = l + message\ size/b$. The program models for each of the NPB benchmarks were created by hand and were parameterised by the number of iterations i, the total number of operations required to complete the benchmark m, and the grid size used by the benchmark. The overall time for a benchmark could then be calculated by:

$$time \;=\; m/f + i * comm(n, p, b, l)$$

where $comm(n, p, b, l)$ represented the total communication time of the code. For example, the completion time for the "BT class A" benchmark was modelled by an equation representing the three communication phases of the algorithm:

$$\begin{gathered}6 * msgtime * (\sqrt{p} - 1) * \tfrac{n^2}{p} * 2 * 5 * 8 \\ +3 * msgtime * (\sqrt{p} - 1) * \tfrac{n^2}{p} * 2 * (25 + 5) * 8 \\ +3 * msgtime * (\sqrt{p} - 1) * \tfrac{n^2}{p} * 2 * 5 * 8\end{gathered}$$

A verification of this model using four nodes and Fast Ethernet was presented. The model predicted overall run-time to within 30%. However, the analysis of the situation was questionable because the modelled situation was dominated by computation, which is relatively easy to model accurately, while the communication cost, which is more difficult to model, was severely underestimated. Importantly, the analysis process would tend to underestimate the communication cost even more if the number of nodes were increased.

2.24 Gautama

The low-cost performance analysis techniques developed for PAMELA by van Gemund (see Section 2.20) had assumed that model parameters were deterministic. This assumption ignored the fact that data-dependencies could significantly alter program execution. Schopf and Berman [313] had shown that the use of stochastic values for model parameters could give far more insightful predictions of parallel program execution time than single point values. In 1998, Gautama [136] added statistical distributions to the PAMELA performance modelling approach, although it was primarily focused on PAMELA's serial constructs rather than its parallel constructs, which were too difficult to analyse. Gautama characterised the probability distribution function (PDF) of model parameters using statistical moments including mean, variance, skewness and kurtosis. His treatment was mainly mathematical and did not focus on the sources of variability for the distributions. Instead, his raw data were obtained from trace-driven simulation over many runs of manually instrumented source code. While Gautama's work was not focused on parallel performance modelling, the notion of generating performance distributions is particularly relevant to the performance prediction of parallel programs, because parallel programs tend to exhibit variable performance even more than regular serial programs.

2.25 Tam and Wang

An important part of modelling parallel programs is understanding the communication network that they use [233, 295, 327, 334, 336]. Tam's PhD thesis published in 2001 [341] and a related publication with Wang in 1999 [342] recognised that modern parallel architectures were converging towards machines built around collections of general purpose nodes with local memory, interfaced to each other through a reliable and scalable network. Taking this into account, they developed a performance model for communication in such networks. In their model, costs were induced by data movement through an extended memory hierarchy, consisting of a node's local cache and memory, and remote memory based at other nodes. A detailed investigation was made of the common physical and operating system processes that supported this data movement. The data movement from a sender's memory to a receiver's memory was abstracted by three phases. In the first of these, send time O_s was characterised by the expression $O_s(m) = \kappa_s + \tau_s m$ where κ_s accounted for any initial operating system queueing overhead and $\tau_s m$ accounted for the time associated with buffer copying a message of size m. In the second phase, transfer time L through the network was modelled using the costs associated with connection setup and finite bandwidth. Contention in the network was modelled by introducing a send gap g_s between output packets from a node, as well as a minimum service time at network routers g_r. In the final phase, asynchronous receives were characterised using O_r, which was analogous to O_s for the send time, plus U_r which represented the time for the receiving process at user level to access the new data using a polling scheme, or to un-block and wake-up. To test their modelling system, models of a gather operation [342] and complete exchange operation [341] were constructed. In contrast to existing models of point-to-point message-passing performance, Tam and Wang's detailed studies showed that the overall execution times for these collective operations are dominated by network contention and congestive packet loss, the effects of which are very difficult to quantify.

2.26 Kranzlmüller and Schaubschläger

The only substantial attempts to tackle head-on the problem of simulating non-determinism in message-passing programs seem to have been performed by Kranzlmüller in 1992 [207] and (extended by) Schaubschläger in 2000 [309]. In general, the dynamic computation and communication structure of message-passing programs can evolve non-deterministically, depending on the exact times when messages are sent; these times vary because of processor speed, process scheduling, processor load, cache effects, memory contention, network contention, interactions with inherently non-deterministic physical processes, randomly generated input data (which is common in scientific simulation), or even dynamic program structure itself. This last situation introduces a feedback loop, where non-determinism begets yet more non-determinism. In such cases, non-determinism will obviously play a critical role in program execution. A very common parallelisation technique that exhibits this behaviour is the simple master-slave task farm, where the master distributes small work parcels to slave processes as they become idle.

The most difficult aspect of modelling non-determinism is that it introduces a potentially exponential number of different possible execution paths for any given input data. This greatly amplifies the risk of many problems, such as deadlock, livelock and race conditions. Kranzlmüller and Schaubschläger's work was aimed at automatically testing message-passing parallel programs for these problems to aid in the debugging purposes. In particular, they created a NOn-deterministic Program Evaluator (NOPE) that could record and replay program execution up to points where non-deterministic choice could occur, systematically make every possible choice, and thereby recurse through all possible program execution sequences. These executions could then be automatically validated for the absence of livelock, deadlocks and race conditions. While this work was not directly related to performance modelling, many of their ideas are extremely relevant here.

In order to determine a program's computation and communication structure, Kranzlmüller created an instrumented MPI library that could log the entry and exit points of all communication routines. Special consideration was given to the crucial points at which non-determinism could occur. These fundamental points are encountered during wildcard receives. In contrast to an explicit receive operation where the destination process is listening for a message from a specific process, in a wildcard receive the destination process does not care about the source of incoming messages – it will simply accept the first message to arrive from any source. Therefore, all wildcard receive operations imply a point of non-deterministic choice. In order to evaluate every different choice that could possibly occur in these situations, NOPE checkpoints program execution and, using Lamport's "happened-before" concept of causal dependence [211], systematically selects any possible inbound message to pair with the wildcard receive. This choice will be recorded and the program will be allowed to continue; depth-first traversal will eventually yield all possible execution paths.

Unfortunately, it is completely intractable to actually evaluate all possible program execution sequences, even for relatively short running parallel programs. Therefore, it is necessary to try and identify event orderings that are likely to happen and thus prune the tree of possible execution sequences to test. Schaubschläger recorded the event orderings of several highly non-deterministic benchmark programs on two parallel machines. He discovered that despite the huge number of event sequences that could theoretically occur, only a small number of event orderings were ever observed in practice. For example, for one application run on an nCUBE-2 with 7 points of non-determinism, hence $7! = 5040$

possible execution paths, only 116 of those paths occurred in 10 million runs. In fact, 96% of the executions resulted in one of only 3 different execution paths. On a heavily loaded Origin2000 running the same application, 100 paths accounted for 80% of 10 million runs; for the same machine in an unloaded state, 100 paths accounted for 92% of 10 million runs. Schaubschläger's basic conclusion was that network topology and load means that some messages are more likely to be preferentially received, for example because of the number of hops required to get from source to destination. Thus, if the communication time for particular messages could be accurately estimated, that information could be used to steer NOPE to only evaluate the most probable executions.

To achieve this, Schaubschläger used a modified version of a tool called SKaMPI [296, 297] to try and accurately benchmark the communication performance of the nCUBE-2 and Origin2000. By making measurements using both lightly and heavily loaded networks he discovered that contention can cause extreme variation in communication times. At this point he became discouraged and proclaimed that:

> "making exact predictions about message transfer times on a heavily loaded [parallel machine] seems impossible ... this makes predictions hardly possible and our desired estimations too inaccurate." [309]

In acquiescence to this setback, fixed latency/bandwidth parameters were chosen to model message-passing time in NOPE, although Schaubschläger lamented the inadequacies of that approach.

Finally, it is important to make two points about the direct applicability of this work to performance modelling. Firstly, it requires message-passing programs to be actually executed for each possible event ordering that is chosen for simulation; this is immensely slow. Secondly, the presented algorithms do not handle timestamps accurately during artificial replay; only information about the ordering of events is preserved. Hence, as it is, the system cannot be used for performance modelling.

2.27 Magnusson *et al.*; Hughes *et al.*

Several tools are able to directly simulate program execution at the instruction set level of an architecture, and hence determine performance. Simics [228, 359], for example, is able to precisely simulate complete computer systems including the operating system and its device drivers as well as user programs on a huge variety of hardware platforms. Simics is an instruction level simulator that is able to accurately model complex cache and memory behaviour and out-of-order processing, although it does have some limitations. Other advantages of the simulation approach are that debugging sessions and performance analyses are completely non-intrusive, can access arbitrarily detailed information and can be started, stopped or replayed at will. In addition, because Simics exactly simulates entire computer systems, it is able to simulate complete communication stacks, and hence multiprocessor parallel computers. It is, however, up to the user to specify timing models for network transfers, although a default model exists. The default model for network communication uses fixed latency and bandwidth parameters. While this allows deterministic simulation, it is unrealistic because it does not account for contention. Since parallel programs are typically heavily dependent on communication performance, more accurate network models are obviously a necessity for accurate simulation results.

Rsim is another widely used instruction-level simulation tool [183,302]. Unlike Simics, Rsim does not support full system simulation. It only simulates statically linked Sparc v9 binaries and ignores the operating system entirely. In its favour, however, Rsim is able to accurately simulate Instruction Level Parallelism (ILP), multithreading and CC-NUMA architectures. The default network model used by Rsim is a wormhole routed 2D mesh network, complete with contention and buffer space modelling. Like Simics, it is also possible to replace the network model with more accurate network simulators. For example, the SICOSYS communication simulator, which is capable of determining parallel application performance using arbitrary communication topologies and common traffic patterns, has been integrated in Rsim (see [286, 287]).

While tools like Simics and Rsim are extremely accurate, they have one major drawback: they are extremely slow. For example, Simics is only able to simulate applications at about 1/80 to 1/350 of their native speed, depending on architecture. Because Rsim also simulates ILP it is even slower and can only simulate applications at about 1/2400 to 1/7100 of their native speed. Obviously these tools run counter to one of the requirements of a performance prediction system for designing parallel algorithms: it needs to be fast. Performance predictions using these tools would take many, many times longer than actually implementing different algorithms and measuring their performance. These methods are still useful, however, for accurately simulating application performance on hypothetical or difficult to obtain parallel machines, albeit slowly. The main reason for the existence of these tools is to help hardware, operating system and compiler designers optimise the performance of very low level operations – it is not for simulating full scientific workloads.

2.28 An Overview of the Approaches

The specific performance modelling techniques for parallel programs that have been presented in this chapter are by no means exhaustive. An enormous number of other general performance modelling techniques as well as models of specific systems have been proposed, such as those by Blasko [36, 37, 38, 39, 40, 41], Fleischmann [127, 128], Foster [64], Halderen [158], Juurlink *et al.* [193, 195], Kerbyson and Nudd and Papaefstathiou *et al.* [202, 256, 257, 262, 263, 264], Mohan [246], Moritz and Frank and Al-Tawil [8, 247], Nelson [253], Nicol [254], Norman and Thanisch [255], Prakash [282, 283], Riley [299], Shaw [315], Sötz [331], Wabnig and Haring [361], Wen and Fox [367], Worlton [374] and more [27, 51, 202, 209, 216, 280, 339, 373]. Many further techniques have been proposed [23, 24, 118, 123, 124, 313, 312, 375] to take into account specific problems for the increasingly abundant class of cluster computers [17,96,175,298,335,362]. Recently, performance models have also begun appearing for programs running on the Grid [133], such as that of Bu and Xu [154]. The techniques that were presented in this chapter, however, are sufficient to introduce the difficulties that are involved with modelling the performance of parallel programs on a wide range of parallel architectures. It is these difficulties that the majority of the research community have reached consensus on; the plethora of different solutions that have been proposed merely serve to indicate that the answers to these difficulties remain to be comprehensively solved. There are four main themes from this chapter that set the scene for the remainder of this dissertation. These are:

- The properties that a parallel programming methodology must possess to support efficient parallel programs, namely: the coordination of work assignment and data placement; the provision of balanced communication; and the ability to overlap communication with computation. In order to be generally useful, this needs to be done in an architecturally independent, congruent and descriptively simple manner.
- The sources of performance loss that occur in parallel programs, namely: condition synchronisation due to static precedence relationships; mutual exclusion, which is dynamic and includes resource contention at the machine level and critical sections at the program level; and conditional control flow caused by data-dependencies.
- The characteristics that a performance modelling technique possesses, such as its generality, expressive power, comprehensibility, accuracy, robustness and cost.
- The fact that no existing performance modelling methodologies are optimal (or even sufficient) with respect to all of the characteristics listed in the previous point.

While many previous performance modelling techniques for parallel programs were designed to address various subsets of the important issues listed above, those techniques are impossible to reconcile into one, generally useful approach, because of their piecemeal nature. The rest of this dissertation presents a new, unified performance modelling technique for parallel processing, in particular, for programs that follow the message-passing methodology. In contrast to previous approaches, this simple-to-use modelling technique is able to accurately and cost-effectively model all the sources of performance loss in a parallel program and thus provide programmers with insightful details that will help them improve the performance of their programs.

Chapter 3

The PEVPM Performance Model

3.1 Introduction

Optimising the performance of parallel programs is significantly more difficult than optimising the performance of serial programs. The main reasons for this are the wide variety of architectures and associated performance characteristics that parallel computers exhibit compared with serial computers. To a large extent, the asymptotic performance of serial programs can be optimised for all serial computers during the design phase, and performance tuning is required only to streamline the program's execution. In contrast, creating high performance parallel codes depends intimately on exploiting the specific characteristics of the machine on which a particular code will run. Unfortunately, there is already great complexity involved with creating parallel programs, and a parallel solution cannot be hand-crafted for every architecture. Instead, programmers struggle with the difficult task of designing parallel programs with the potential to run efficiently on a wide variety of target architectures. In order to meet this potential, a large burden is left on the performance tuning phase of implementing an effective parallel solution to a problem. Despite the best efforts made at the program design stage, most parallel performance tuning still relies on a "measure-modify approach" [82] because of the difficulties in foreseeing the effect on performance of factors such as input data, the number of available processors and the characteristics of the communication network that connect them. The measure-modify approach is heavily dependent on making detailed performance measurements of programs during execution and can be an extremely time consuming process, particularly if it is necessary to repeat this process for many target machines.

Performance modelling provides an alternative to the measure-modify development process by bringing the focus of performance optimisation for parallel programs back from the tuning phase to the design phase. This is possible because of the predictive capabilities of performance models, which empower programmers to make better decisions at the design stage. Of course, the benefits to development of using performance models need to be weighed against the costs incurred in their development. Indeed, one of the reasons that performance models are rarely used in practice is the high cost of their creation and/or solution. However, while performance tuning is a relatively mature art, performance modelling is not. There are certainly many advances that can be made to the theory and practice of performance modelling that would both lower the cost of model development and increase model effectiveness, thereby enhancing the usefulness of performance modelling. For example, the cost of model development could be reduced by improving the accessibility of performance modelling techniques to the average programmer, not just experts in performance modelling. Also, model effectiveness desperately requires researchers to focus on techniques with sufficient accuracy but without prohibitive development cost, so that programmers can choose appropriate solutions from among many alternative implementation possibilities.

This chapter describes a new system for modelling the performance of parallel programs, called the Performance Evaluating Virtual Parallel Machine (PEVPM), that aims to address the need for a sufficiently powerful yet generally accessible performance modelling technique. Based on the models discussed in Chapter 2, Section 3.2 explores the requirements of a performance modelling system, as well as the properties of parallel programs that make this possible. Section 3.3 clarifies the range of programming methodologies and parallel architectures that are of particular (although not exclusive) interest, namely message-passing codes on distributed memory computers. Sections 3.4, 3.5 and 3.6 form the central discussion of this chapter. They explain the fundamentals of modelling message-passing code, a way of describing the features of parallel code that are salient to performance, and how to evaluate the performance implications of those features. Finally, Sections 3.7 and 3.8 summarise the relevance of this technique to other programming methodologies and to the question of performance modelling for parallel programs in general.

3.2 Key Features of Performance Models

While the mechanics of the PEVPM modelling technique presented in this chapter do not simply involve the extension of any existing methodology, the problems and issues that the PEVPM modelling technique faces have been extensively researched. Because the PEVPM does not explicitly build upon any previous performance modelling techniques, this chapter contains few direct references to the prior research discussed in Chapter 2. Instead, this section summarises the key research ideas that inspired the design of the PEVPM, in terms of:

- The requirements for a good performance modelling system.

- Some postulates concerning the performance properties of parallel programs that are generally regarded to be true.
- Some provable theorems and corollaries that can be applied to the problem of performance modelling for parallel programs.

These requirements, postulates, theorems and corollaries will be called upon throughout this chapter to aid in the development of the PEVPM modelling system.

Many early performance modelling techniques for parallel programs were restricted to a particular architecture, such as the PRAM model of Fortune and Wylie (see Section 2.3), or to a particular programming model, such as the BSP model of Valiant (see Section 2.5). Because of these restrictions, none of these models enjoy wide-spread relevance today. Thus:

Requirement 1 *A performance modelling technique for parallel programs should be applicable to any parallel programming methodology and to any parallel machine.*

The work of Culler *et al.* presented in Section 2.8 highlighted that technical and economic forces are driving parallel computers to cluster architectures. Fulfilling Requirement 1 implies enough flexibility to cope with these architectures, however:

Requirement 2 *A performance modelling technique for parallel programs should be particularly relevant for cluster architectures.*

The work of Crovella and LeBlanc, presented in Section 2.15, introduced the benefits of using completeness and orthogonality properties for performance modelling. Unlike the non-descriptive performance modelling techniques proposed by Amdahl (see Section 2.2), Hockney (see Section 2.6) and Grama *et al.* (see Section 2.9), ensuring these properties in a modelling formalism allows the sources of performance loss to be modelled in a holistic way that allows programmers to understand the performance implications of the design and implementation of their parallel programs. Thus:

Requirement 3 *A modelling technique should provide ordinary programmers with insight into the effects on performance of load imbalance, insufficient parallelism, synchronisation loss, communication loss and resource contention on their code.*

The predictive capability of performance models was shown to be of great value in the introduction to this chapter. The work of Mehra *et al.* on Axe, which was reviewed in Section 2.12, provided an example of the usefulness of tools for answering "what-if" type questions. Although the models they constructed were very expensive to create, the accuracy with which they could predict performance on hypothetical architectures and hypothetical problem sizes enhanced the ability of programmers to design their code right "the first time". Hence:

Requirement 4 *Performance modelling techniques should help programmers to accurately answer "what-if" questions about their code at the design stage.*

Because of the effort involved with creating models using Mehra *et al.*'s technique, or indeed many similar techniques, such as that of Becker *et al.* (see Section 2.23), performance modelling during the design process has not found wide-spread use. Instead, like the techniques developed by Saavedra and Smith (see Section 2.7), Parashar and Hariri (see Section 2.13), Labarta and Girona *et al.* (see Section 2.21) or Dunlop and Hey *et al.* (see Section 2.22):

Requirement 5 *A performance modelling technique should allow the construction of performance models without prohibitive creation cost.*

Related to this, unlike the extremely computationally expensive modelling techniques of Hoare, Milner, Alur and Dill (see Section 2.4) or Magnusson *et al.* and Hughes *et al.* (see Section 2.27):

Requirement 6 *A performance modelling technique should allow the construction of performance models without prohibitive solution cost.*

As discussed in Section 2.20, van Gemund investigated the idea of a performance compiler. This required a program/machine description to be produced in a performance modelling language. Like the modelling language defined by Mehra *et al.*, van Gemund's performance modelling language had syntax for describing loops, limited conditional control flow and message-passing. Van Gemund went further, however, and developed a calculus with a set of rules to reduce performance descriptions to simpler models by statically approximating the effects of dynamic sources of performance loss. This allowed approximate models to be used for wide-ranging parameter studies where low-cost was important, or higher accuracy models where high-quality solutions were required:

Requirement 7 *A performance modelling technique should support a trade-off between low evaluation cost and high solution accuracy.*

If truly accurate models are required, Gautama showed that the execution time of code should be modelled probabilistically (see Section 2.24). Although Gautama tackled this problem for serial problems, the applicability of his research to parallel codes was very limited. In fact, although techniques to generate performance bounds have been investigated, no previous techniques have been able to produce probabilistic performance evaluations of parallel codes. Despite the fact that the variance in the performance of most parallel programs is bounded (see Postulate 1, shortly), this variance can be quite extensive. Therefore:

Requirement 8 *A performance modelling technique should be able to produce models that describe the variability in performance between runs of a parallel program.*

Related to this and briefly investigated by Mraz as well as Tabe *et al.* (see Section 2.16) and Clement, Quinn and Steed (see Section 2.17):

Requirement 9 *For accurate modelling, the performance of communication primitives must be treated as probabilistic quantities.*

Furthermore, Tam and Wang showed that it is important to look closely at how these primitives are implemented in order to model them accurately (see Section 2.25). Doing so requires a detailed evaluation of all the processes that occur in end-to-end communication. Thus:

Requirement 10 *A model of a communication event must include all of the processes involved in the end-to-end transmission of information, in particular operating system overhead and buffer copying at either end, as well as transmission time and routing latency in the network.*

Interestingly, Jonkers showed that in many cases, variance in the individual task times constituting a parallel program usually increases the mean execution time of a program but leaves the total variance unaffected (see Section 2.19). He found that this was due to synchronisation in parallel algorithms, and this suggests that:

Postulate 1 *The effect of non-deterministic task times during the execution of a parallel program does not usually affect the control structure of the program but merely increases the execution time between synchronisation points.*

A form of sensitivity analysis described by Singh *et al.* was presented in Section 2.11. They noted that characterising the input data set to a parallel code is just as important as characterising the algorithmic structure of the code itself. They found that the performance of many code structures is stable for different input data sets (of the same size), so:

Postulate 2 *There is scope for estimating the effect of data-dependencies on performance in many codes.*

As discussed in Section 2.14, Skillicorn showed that message-passing systems are congruent. This means that the cost of a message-passing program can be determined in a composable way from the cost of its pieces, if they are modelled at an appropriate level of abstraction:

Theorem 1 *The performance of message-passing parallel codes can be modelled by modelling the performance of their constituent parts.*

He went on to show that message-passing primitives are sufficiently powerful to describe all parallel programming methodologies. For example, shared memory operations can be modelled using send and receive operations; and data parallel operations can be implemented on top of message-passing systems. So:

Theorem 2 *A performance model for message-passing codes can be used as the basis for a performance model for codes written using other parallel programming techniques, such as shared memory or data parallel programming.*

It is crucial to understand the underlying communication primitives of message-passing parallel programs in order to model such programs effectively. These primitives were identified by Islam, and enumerated in Section 2.18. Only a handful of primitives are needed to completely describe even the most complex communication patterns:

Theorem 3 *There are only a small number of message-passing primitives that are required to describe any message-passing parallel program.*

Hence, by Theorem 1:

Corollary 4 *Only a small number of performance models need to be created to model the performance implications of communication operations in message-passing programs;*

and by Theorem 2:

Corollary 5 *Only a small number of performance models need to be created to model the performance implications of communication operations in all parallel programs.*

Adve's work on performance modelling covered in Section 2.10 showed that a unique execution sequence is implied if task times are modelled by deterministic task times. In most cases this allows the basic control structure of a parallel code to be determined statically. An extension of this concept by Kranzlmüller and Schaubschläger (see Section 2.26) showed that:

Theorem 6 *The control structure of a parallel code can be determined statically and automatically in most cases, by carrying out small amounts of computation to determine data-dependencies in most of the remaining cases, and cannot be done in only a very small proportion of parallel programs (which may still be modelled in abstract ways through human intervention).*

The remainder of this chapter is dedicated to designing a performance modelling system for parallel programs that fulfils all of the requirements that have just been listed.

3.3 Scope

The introduction to this thesis in Chapter 1 showed that there are many ways of writing parallel programs, as well as a large range of parallel architectures that they may run on. This makes devising a performance modelling technique that is capable of satisfying Requirement 1, i.e. that a performance modelling technique should be applicable to any parallel programming methodology and to any parallel machine, a potentially difficult task. Fortunately, this problem is made more amenable to solution by the fact that message-passing is sufficiently powerful to describe any parallel program. Hence Theorem 2, i.e. a performance model for message-passing codes can be used as the basis of a performance model for codes written using other parallel programming methodologies. Therefore, the performance modelling technique that will be presented in the remainder of this chapter will focus on modelling message-passing codes. Codes written using other programming methodologies can be modelled by first translating them to an equivalent message-passing code, as discussed in Section 3.7.

Interestingly, focusing on the message-passing programming methodology simplifies the exercise of choosing a parallel architecture that can sufficiently describe all other parallel architectures. Hoare's CSP and Milner's CCS (see Section 2.4) separately showed that regardless of the physical complexities that underly any particular parallel machine, conceptually, a generalised machine that is capable of supporting message-passing programs needs to support only two tasks: local processing and communication between processing elements. Therefore, for a performance model, it is possible to abstract over the architectural details of a parallel machine, instead focusing only on the speed of local processing and communication events. Of course, it is necessary to evaluate the performance of these tasks taking into consideration their relationship with each other, which must be determined from the structure of the message-passing program. Admittedly, modelling these tasks and determining program structure are complicated endeavours (that will be covered in the remainder of this thesis), but the important point for now is this: as long as a performance model is based on the performance of local processing and the communication between processes, it is applicable to any parallel machine.

There is one important caveat that has not been addressed yet: contention from *unrelated* processes, caused by either non-essential operating system services or other user programs[1]. The effect on performance of interference from such processes is highly variable and depends intimately on each program's specific structure and scheduling. Hence, incorporating the effect of such processes in a performance model is immensely complex and can only be even loosely approximated in certain, specific cases [19,375]. The

[1]Of course, this does not include contention from processes that are related to the program that is being modelled, for example operating system involvement during communication; understanding the contention between those processes is critical in understanding the performance of the entire program.

performance modelling technique that will be presented in the rest of this thesis ignores this issue entirely, so it is therefore only valid for systems where a parallel program will have dedicated access to the parallel machine that it is to be run on, and in particular where only one user process is run on any processor. However, this restriction is becoming less of a concern because of trends in the way that high performance computing is being used.

In the past, because of the expense of high performance computing hardware, resources dedicated to running a single program have been relatively rare. However, with the development of affordable high performance computing using clusters [17, 96, 97, 175, 290, 298,335,334,362], whose computing resources and, to an extent, communication resources can be easily partitioned, single purpose partitions of parallel machines have become far more widespread (for example, see [161]). This is particularly advantageous because such parallel machines provide the perfect candidates for performance modelling using only the performance of local processing and communication events, because architecturally they are completely independent processing nodes that are connected by a distinct communication network.

So, to summarise, the primary focus of the performance modelling technique that will follow is for message-passing codes that run in a dedicated fashion on distributed memory (for example, cluster) computers. The applicability of this performance modelling technique to other parallel programming techniques and parallel machines will be discussed in Section 3.7.

3.4 Modelling Message-Passing Codes

In the previous section it was stated that the performance of a message-passing code on a parallel computer can be determined from the structure of the code, the performance of local processing and the speed of communication events. To understand the process by which this is achieved, imagine an arbitrary execution sequence of an arbitrary message-passing program on an arbitrary parallel machine. When a processor is performing local computation, it is acting entirely independently from the rest of the system. This serial segment of code is amenable to separate performance modelling using traditional techniques. A key point is that the times taken to run segments of local computation are relatively stable in most parallel applications. The reasons for this and the techniques for modelling serial segments of code are presented in Section 3.4.1.

The end of a local computation is encountered whenever a processor becomes involved in a communication operation. For most parallel programs (i.e. those with the statically defined communication structures identified in Theorem 6) it is at this point that there is the most potential for non-deterministic program execution to occur. This is because the times taken for communication can be very variable due to the effects of network contention. Accurate modelling of the effect of this non-determinism in the time domain is the key to accurately modelling the performance of an entire parallel program. This can be achieved by modelling each individual communication event separately, based on properties such as from whence and to where it was sent, when it was sent, its length, and the number of other messages simultaneously in transit. Section 3.4.2 introduces a taxonomy of the communication events that commonly occur in message-passing programs, and describes how they can be effectively modelled.

The time at which messages are sent, and therefore the number in transit simultaneously, are dynamic properties. They are not apparent initially so must instead be derived from the execution of the program that will have occurred prior to their invocation. Section 3.4.3 describes how they are derived using a semi-static approach that combines the static models of serial computation and static program structure with the dynamic effects of network contention on the actual run-time structure of a program.

Using this conceptual model of the execution of a parallel program, the unimportant details of serial computation remain hidden, while the important and variable structure of the communication between the processes is manifest. Following discussion of the target computational model, Section 3.4.4 defines a mapping from programming constructs to a performance evaluation language and Section 3.4.5 discusses how this can be automated.

3.4.1 Modelling Local Processing

There is a large body of literature that addresses the problem of estimating the performance of serial code. Many techniques [5, 6, 160, 203, 204] are concerned with determining the asymptotic performance of code fragments and are expressed using complexity analysis with "big-oh" notation. A big-oh expression is usually used to choose from among several alternative algorithm choices for solving a certain problem. Although determining the exact performance of a code fragment requires detailed knowledge of the compiler and target architecture, it is possible to use big-oh expressions to roughly determine which algorithms perform best under which situations. To do this a big-oh expression represents the approximate run-time of a code fragment as a function f of "input size" n, for example $O(f(n))$. The function f represents the rate of growth of run-time for the fragment given increasing input size. The input size for a big-oh expression of a code fragment is a reflection of the amount of work that must be carried out.

Usually, the input size for a big-oh expression depends on the size of the data structure that is being operated on by the associated code fragment, for example array size or the number of elements in a list. Common big-oh run-times for code fragments are $O(1)$ (constant), $O(\log n)$ (logarithmic), $O(n)$ (linear), $O(n^2)$ (quadratic), $O(n^3)$ (cubic) and

$O(2^n)$ (exponential). The function f is usually unit-less, so any constants c are ignored, i.e. $O(cf(n))$ is equivalent to $O(f(n))$. Big-oh expressions are transitive, so the big-oh expression for a succession of code fragments is the summation of the individual big-oh expressions for each fragment. Furthermore, because big-oh expressions are aimed at the asymptotic cost of a code fragment, i.e. for large n, low order terms of f are ignored. For example $O(n^2 + n)$ reduces to $O(n^2)$.

Of course, accurately modelling the performance of a serial block of code requires far more effort than merely determining its asymptotic big-oh expression. Instead, the performance implications of every source instruction need to be accounted for. This can be achieved by *not* applying the two rules that are usually applied to big-oh formulae, i.e. that neither constants nor low order terms can be ignored. Determining the constants of a big-oh expression analytically is a very complicated task. It can only be achieved for specific instances of code generated using a specific compiler for a specific architecture. Even then assumptions must be made about the scheduling and memory management of the code by the operating system. Assuming that the code has dedicated access to the CPU and memory (which is a reasonably accurate assumption provided that no other user programs are running on the CPU), the code's performance can theoretically be determined by evaluating the time taken for the assembly language instructions in the compiled object code [279]. This can be done using instruction timing formulae for the instructions set architecture, a knowledge of the pipelining employed by the CPU and an understanding of the memory hierarchy of the architecture. Unfortunately, however, dynamic program behaviour caused by data-dependencies and conditional statements on both program structure and caching behaviour can greatly complicate matters. While some approaches [107] attempt to model these factors, others [366] opt to use statically approximated cache hit/miss ratios and compiler generated branch prediction coefficients together with an instruction counting approach.

The last common cause of complication is disk I/O, which is heavily dependent on a huge range of factors ranging from operating system caching to disk drive geometry. While disk I/O will not affect the validity of the model discussed in this chapter, for the sake of simplicity, it is going to be assumed from here on that no local computation involves significant disk I/O. This also precludes access to virtual memory, i.e. more than enough core physical memory is assumed to be available.

In the end, the information that is of interest for each serial segment of code is simply the time that the serial segment will take to complete. Although this can certainly be determined using any of the methods described above (or even the simulation tools described in Section 2.27), it is worthwhile to realise that for many parallel programs, it may be far more practical to simply measure the completion time of some representative serial segments.

3.4.2 Modelling Communication Events

The modelling technique proposed in this thesis calls for modelling each communication event that a program will run. In the same way that serial segments of code were considered independently in the previous section, each communication event will be considered independently. Before delving too deeply into the generalised model of a communication event, it is important to recognise the implicit assumption that communication performance is memory-less, i.e. that the amount of data sent in the past does not affect future performance. While this section provides a rationale for that assumption, a more rigorous justification can be found in Section 4.7, which demonstrates the statistical independence of a series of identical performance measurements.

In order to build a general performance model of a communication event, it is necessary to consider all the possible messages that could occur. The first step towards developing this model is to broadly identify the different types of messages that may be used. Since the scope of this model is the message-passing paradigm, this is a relatively straightforward task. Because all message-passing implementations, for example MPI, PVM and the Ada rendezvous mechanism [352], support essentially the same set of operations, any one implementation can be chosen as a reasonably representative programming system. Since MPI is the most widely used message-passing system, it will be considered here. Now, determining the types of messages that may be used simply involves studying the functions that are defined in the MPI specification.

There are 125 functions defined by the MPI specification [240]. These functions provide programmers with a vast array of tools for tasks such as instantiating a set of processes, creating data types, specifying communication topologies and performing communication and synchronisation. Fortunately, not all of these functions have significant consequences for modelling the performance of a communication event. In fact, it is only the functions that perform communication or synchronisation that need to be considered.

The communication functions in MPI are commonly divided into simple sends and receives, and collective operations. A simple send and receive involves a point-to-point communication from one process to another. Several different send/receive protocols are defined in MPI, which offer various levels of performance, flexibility and robustness. There are standard blocking sends and receives, ordered sends and receives, combined sends and receives, buffered sends and receives, and asynchronous sends and receives. Based on performance characteristics alone, most of these can be treated the same. Both ordered sends and receives as well as buffered sends and receives are mainly used to ensure correctness in a program. For example, buffered sends and receives give a programmer more control over the transmission process than standard sends and receives by allowing them to use explicitly allocated buffer space. However, these functions perform exactly the same as the standard send and receives, provided that there is enough buffer space available. So, ignoring the issues associated with program correctness[2], it turns out that for performance modelling there are only four point-to-point operations that need to be modelled. These are blocking sends, non-blocking sends, blocking receives and non-blocking receives. Both varieties of send can be modelled in a very similar way, as can both varieties of receive. In fact, either type of send can be matched by either type of receive. Although the inherent performance difference between a synchronous or asynchronous operation is negligible, their semantic differences completely change the way they need to be interpreted by a program, with vast potential for affecting program performance. Separate models for each of these four operations will be discussed shortly.

Collective operations in MPI are a convenient way of instructing a large group of processes to participate in primitive but large-scale operations. These can be either data movement, synchronisation or collective computation (reduction) operations. The data movement operations are designed to provide a descriptively simple means of distributing data (usually called scattering data) among processes and collating data (usually called gathering data) from many processes. This allows programmers to use one function call to gather or scatter data, rather than using a large number of point-to-point operations to achieve the same effect. For example, one `MPI_Broadcast` call can be used to send a common piece of information from a source process to all other processes, rather than sending it to each process individually. The synchronisation operations, for example `MPI_Wait` which synchronises on the delivery of a point-to-point message or `MPI_Barrier` which synchronises a group of processes, provide a means of ensuring precedence relationships within a parallel program. Lastly, there are the collective computation operations, which are similar to gather operations, but include data reduction based on the results of a small amount of computation. Commonly used collective computation operations are minimum, maximum, sum, etc, which are used to find the minimum, maximum or sum, etc, of data contained at each process. Once again, operations like this allow a programmer to specify a large-scale operation in a simple way that avoids the need to use a large number of individual send and receive operations.

Now, although a (very) few parallel machines have specialised hardware that can enable them to execute some of these collective operations in an atomic manner, most parallel machines do not. Even in the event that a parallel machine may have special hardware to accomplish this, it must be supported by the machine's MPI implementation. Since almost all MPI implementations are based on reference designs that do not support this, very few systems have this capability in practice. Instead, the vast majority of implementations deconstruct collective operations into individual point-to-point

[2]Evaluating correctness in parallel programs presents very difficult problems, and is a related but distinct field of research. Although issues of program correctness are largely beyond the scope of this thesis, some properties of this modelling process may lend themselves to use in that field. These are briefly discussed in Section 3.6.

operations. (In the rare cases where specialised hardware is available and utilised, a special purpose model should be built to accommodate it). Therefore, from a modelling perspective, the performance of a collective operation can usually be modelled using the performance models of point-to-point operations. This is, of course, provided that it is possible to model how an implementation will describe a collective operation in terms of point-to-point operations. Fortunately, because collective operations are conceptually quite primitive, yet designed to operate over a large number of processes in an efficient manner, they exhibit very structured communication patterns. Given little more than a knowledge of the total number of processes involved, and the source process of a message for scatters or destination process of a message for gathers, it is usually possible to determine how a collective operation will be constructed from point-to-point operations. How this can be done is discussed in Chapter 5.

To summarise the preceding paragraphs, all of the communication in a message-passing parallel program is ultimately produced by a point-to-point operation. Furthermore, the only distinct varieties of point-to-point operation that need to be considered for performance modelling purposes are synchronous sends, asynchronous sends, synchronous receives and asynchronous receives. Therefore, performance models of only these operations are required to create a basis for modelling the performance of any communication that can occur. Having established the types of calls that need to be modelled, the next question becomes how should this be done? To answer this, it is necessary to understand the processes that occur during message transmission and reception.

An example of a common communication pathway for a message sent from one process to another is shown in Figure 2, although some of these steps may be absent in systems that have special support for message-passing. The figure shows a message moving from the user-space program, through the MPI library to the operating system for transmission via the host's network subsystem. At any of these points there is the potential for the message to be buffered, and possibly for control to be returned to the calling process, while the message is dealt with asynchronously. Once the message has been delivered to the operating system, it will undergo further processing by the network protocol stack (for example an IP stack [281, 337]). At this point the network interface card will be instructed to fetch the data from wherever it is in memory and to place it on to the external communication network. Up until now the data may have traversed the local bus, such as a PCI bus, several times. Within the scope of using a dedicated parallel machine with uniprocessor nodes, a process will only contend with operating system functions conducting message-passing for access to local resources such as memory. This may cause some small delays due to processor or memory contention if communication is overlapped with computation. However, when the message enters the external communication network (or similarly, the internal communication network of an SMP node), there

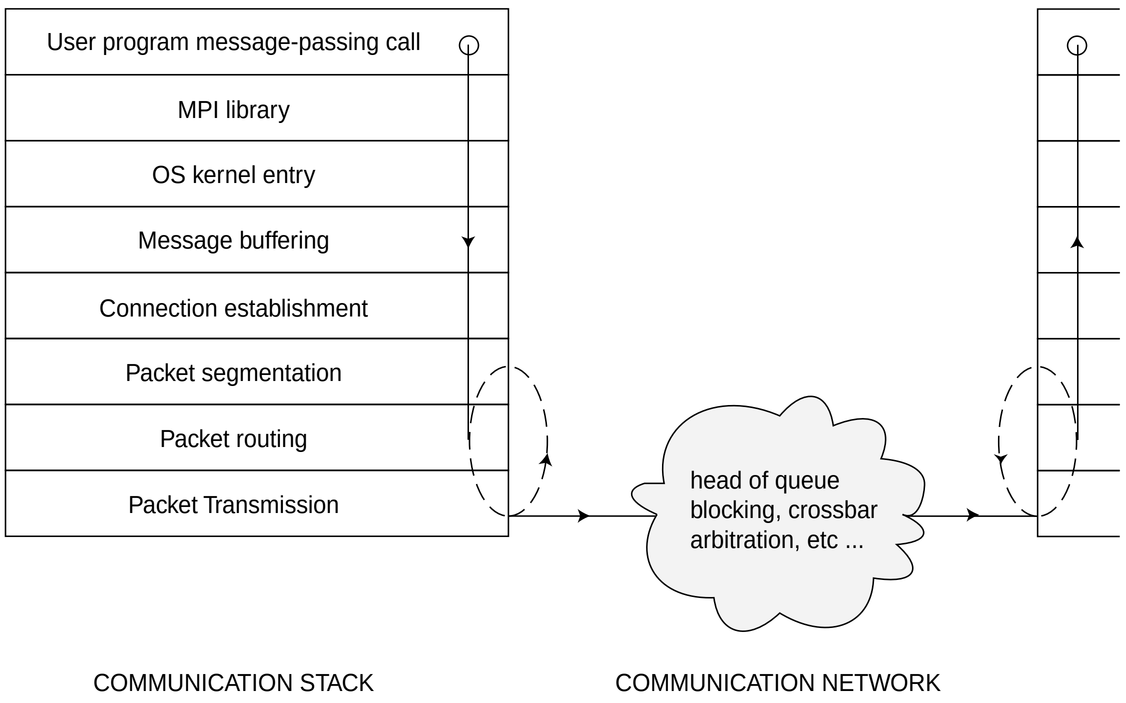


Figure 2: The pathway for a point-to-point communication between two MPI processes on different nodes. The sending process is on the left and the receiving process (which mirrors the communication stack of the sender) is on the right.

is much more potential for contention. This is because it becomes increasingly difficult to provide full-speed point-to-point links between processors as the number of connected processors grows, and at some point the communication network will begin degrading towards a shared medium. When this happens messages will begin to experience delays through the network because of contention for the shared communication resource. Once the message arrives at the destination host, it follows the reverse chain of events that occurred at the source, and it finally arrives for the user program to process. This entire process is obviously quite complex and modelling every step of the process individually would be quite difficult and extremely time-consuming (although it can be done; refer to Section 2.27). Fortunately, this does not need to be done because the communication pathway is connected end-to-end. This means that the entire multi-stage process can be quite accurately described by an equivalent single stage model.

For synchronous messages, all that is necessary for performance modelling processes is to determine the overall time that a message will take to move through the entire communication pathway, since this is how the entire process appears to the user program. Because contention may occur during the transmission process, this cannot be modelled by a single value, but must be modelled in a probabilistic manner. A slight variation on this model is needed for asynchronous messages, because programmed control may be returned to the calling user program before a message has successfully been delivered. For an asynchronous message, there are three distinct phases that are visible from a calling MPI program. These are send phase at the source process, transmission through the communication network, and the receive phase at the destination process. An asynchronous send will queue a message for delivery and then return control to the user program. The local completion time required for this phase is mainly dependent on message size, cache/memory speed and message-passing implementation. On processors with dedicated memory bus access, the contention and hence performance timing variation encountered in this stage will be small. On other systems, notably commodity SMP systems with a shared memory bus and where outgoing messages are buffered by the MPI implementation or operating system's network stack, more contention and hence more performance variation will be observed. In order to take this into account, the queueing time to initiate the asynchronous send must be modelled as a probabilistic value. Likewise the most contention-susceptible phase, transmission through the external communication network, it must also be modelled by a probabilistic value. Its value may be inferred from the time it takes for a complete synchronous point-to-point transmission minus the time required for the send and receive phases attached to each of its ends. The time required for an asynchronous receive must be modelled slightly differently from an asynchronous send. When used correctly, an asynchronous receive must be applied in conjunction with operations that either wait for the transmission phase to end, or test whether it has finished. In the latter case, if the transmission phase has not finished, more computation may be carried before testing for completion again, etc. Therefore, while the local completion time of an asynchronous receive operation must also be modelled as a probabilistic value, account must be taken of whether the data transfer has completed or not. The three phases just described are shown in Figure 3, labelled as $t_{q-isend}$, $t_{transmission}$ and t_{irecv} respectively; the figure also shows the complete end-to-end completion time for a synchronous send, labelled t_{send}, which is what is commonly referred to as the latency of a message-passing operation.

As well as showing the relationship between the models for synchronous and asynchronous messages, Figure 3 gives hints about how contention affects the completion time of a point-to-point message. The figure shows the component and total probability distributions of the time a message will take to make its way through different parts of the communication pathway. For each distribution, there is a distinct lower bound to completion time, which depicts the performance of a message that (by chance) avoids any contention at all. Each distribution then rises smoothly because of the effects of contention (the particular distributions that may occur are discussed in detail in Section 4.7 of the following chapter) and tails off towards infinitely slow completion. Because the

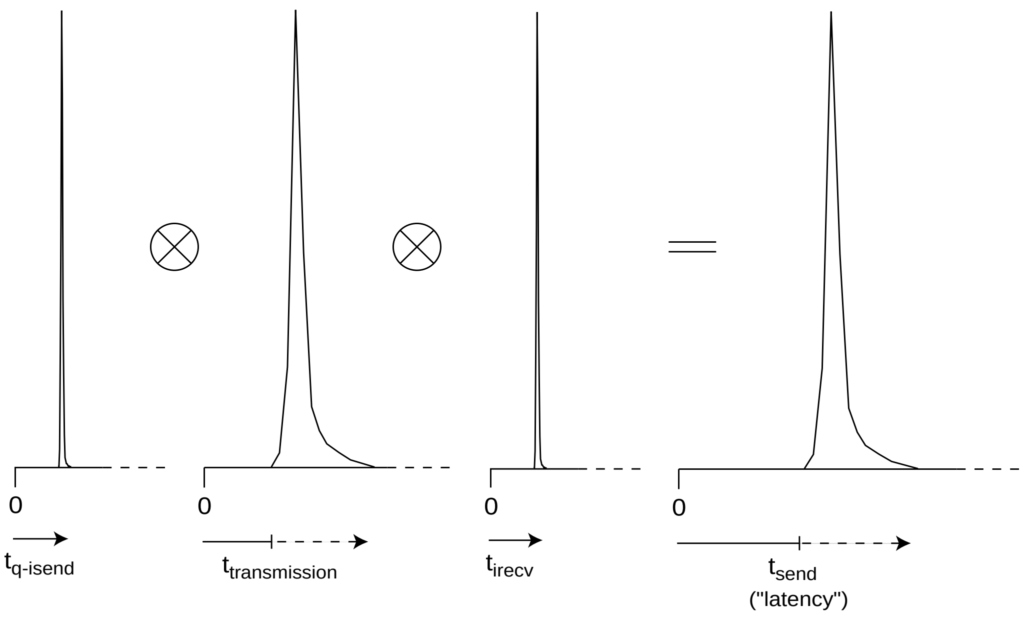


Figure 3: Three phases of point-to-point communication are visible to message-passing programs: the time to asynchronously process an outgoing message ($t_{q-isend}$), the transmission time through the communication network ($t_{transmission}$), and the time to process the incoming transmission (t_{irecv}). The completion times of each of these phases individually, or all together (t_{send}), are inherently variable due to contention, and can be described by probability distributions. For each distribution shown in the figure, the horizontal axis represents the times that a communication phase may take, with associated probabilities shown on the vertical axis.

three phases are connected in series, the overall probability distribution that describes the end-to-end completion time of the entire communication pathway is described by the convolution of the three probability distributions.

To understand the probabilistic interpretation of contention it is important to re-examine the cause and effect of that contention at both microscopic and macroscopic levels. At a microscopic level contention is caused by two or more competing processes trying to gain exclusive access to a shared resource. In the case of the communication network, this is affected by the exact number and exact timing of all the messages in the network. Although any particular situation will ultimately result in an exact sequence of events, it is a finely balanced chaotic system so this sequence of events cannot be predicted reliably. However, the possible outcomes of contention delays are well-described by probability distributions at a macroscopic level. This proves to be the saving grace for bottom-up modelling of the macroscopic performance of high performance computing applications because they typically involve a very large number of messages. Essentially, using a probability distribution to describe message-passing time macroscopically simulates the dynamic and microscopically unpredictable performance implications of contention.

3.4.3 Combining Processing and Communication Models

This section describes a framework that can be used to construct an overall model of a complete message-passing code, i.e. one that consists of many segments of local processing connected by many communication events. To construct an overall model of any given message-passing code involves two broad problems. Firstly, there is the issue of which submodels of local computation and message-passing need to be generated. Secondly, there is the problem of how these many submodels should be coalesced into an overall model describing the performance of the entire code.

As was explained in Section 3.4, the boundaries between submodels occur wherever a communication operation is encountered. Since the platform of interest here is MPI, submodels must be generated every time an MPI operation that induces communication is called. There is a subtle but crucial point that must be taken from the previous sentence: the submodels are separated by MPI operations at run-time, not in the source code. For example, consider an `MPI_Send` operation within the body of a loop that is called many times. At the source code level, the `MPI_Send` only divides the code into two sections, but at run-time there are many unique `MPI_Send`s. It is these many sends that form the boundaries for submodels.

It was explained previously that the most potential for non-deterministic behaviour in message-passing programs is caused by the variable performance of communication due to contention. Since any non-deterministic execution can have an enormous impact on program performance and execution structure, the modelling system must account for this. Furthermore, because non-determinism is a dynamic property, it must be modelled in a dynamic way. Coupling this realisation with the necessity of submodel division at dynamically determined points in program execution, it becomes clear that a wholly static modelling technique is insufficient. Still, static modelling is achievable at the level of individual submodels, so the submodels are treated statically because of the lower solution requirements that this affords. However, while the submodels themselves are constructed in a static fashion, the initial conditions that they are developed from are based on a dynamic notion of program state. This is why this modelling technique was described earlier as a semi-static approach.

The dynamic program state that is used as a basis for creating static submodels of local processing and communication is provided by a virtual machine. The virtual machine, which is described in Section 3.5, does this by maintaining a representation of the state of the program and the parallel machine that it is running on. Then, when either a segment of local computation or a communication operation is encountered, the stored state is used to create a submodel of the computation or communication. This entire process can be considered as a form of partial execution that only performs work to evaluate the dynamic execution structure of the program, and hence program performance. This exemplifies a goal of the modelling technique, namely Requirement 6, that a performance model should require as little effort as practicable to evaluate.

So, combination of the submodels of serial computation and communication events is done in a dynamic manner. It depends only on knowing what local computation will occur until the next communication event is processed and the details of that particular communication event. Therefore, before describing the algorithms that are used to combine the submodels (in Section 3.5), it is necessary to examine how these instruction streams that describe local computation and the immediately following communication event can be determined.

3.4.4 The Modelling Formalisms

The main simplifying abstraction used by the PEVPM so far is that the execution of a message-passing code follows a statically determinable sequence of events between message-passing calls. At the boundaries of these statically determinable sequences of events, message-passing calls are susceptible to contention. Although this contention is dynamic, its effect on performance can be simulated by using probability distributions of message-passing performance, and by considering each message-passing call as a potential source of non-determinism that can affect program execution structure. In order to apply this performance abstraction to real (or hypothetical) programs in practice, a means of converting common programming structures into a description of serial sections of code and message-passing calls needs to be developed.

Because message-passing is built upon the notion of separating data from the instructions that process those data (unlike data-parallel or object-oriented programming paradigms), it is ideally suited to procedural programming languages. For this reason, message-passing libraries are almost universally developed for such languages, and in particular for C, C++ and Fortran. Although the correlations between programming constructs and model descriptions that are about to be discussed hold equally well for any procedural language, the C programming language will be used here to facilitate the discussion.

There are only a handful of programming constructs that provide the basic utility of any procedural programming language. These are simple and compound statements, loops, conditional statements and subroutines. For the purposes of the performance model that has been developed in this thesis, message-passing calls must be considered separately. This does more than simply create the need for a means to describe these message-passing calls individually. It also affects the way that the other programming constructs (other than simple statements) must be considered, because message-passing calls can occur in the body of those constructs. While these constructs can traditionally be modelled as single entities, if they contain message-passing calls then the modelling abstraction that has been developed in this thesis requires them to be divided into multiple entities that are separated by message-passing calls. The rest of this section presents models for each of these programming constructs and any associated message-passing.

Simple and Compound Statements

In C, a simple statement is an expression followed by a semicolon. Expressions are a sequence of operators and operands that are used to perform computation. For example, consider the code:

```
z = a + b;
```

This entire fragment of code is a simple statement, where the expression `z = a + b` has operands `a` and `b` operated on by the `+` operator. Simple statements are (essentially) directly translated into object code during the code generation phase of program compilation. The assembly language instructions that are used by compilers to implement simple statements have very explicit execution timing which can be determined from the Instruction Set Architecture (ISA) of the target machine. For the purposes of the performance model being developed, the time of a simple instruction can be trivially modelled:

$$T(\textit{simple statement}) \;=\; t_{\textit{simple statement}}$$

In this equation, T takes a *simple statement* and computes the time $t_{\textit{simple statement}}$ it requires from the instruction timing formulae specified in the ISA of the target machine (or alternatively, this may be achieved empirically by measurement). Since the performance of multiple simple statements is transitive, compound sequences of simple statements can be used to reduce the computational requirements of the model:

$$T(\textit{compound statement}) \;=\; t_{\textit{simple statement}_1} \;+\; \ldots \;+\; t_{\textit{simple statement}_n}$$

So far, this is really only a more explicit description of the well-known performance modelling techniques for simple statements described in Section 3.4.1. However, a slight modification is needed to account for the variable performance of message-passing. Even though message-passing calls could be considered as a form of simple statement in that they merely represent a data access (albeit a remote data access), they cannot be rolled into compound statements for modelling because they are the essential boundaries between submodels that are imposed by the PEVPM system. Instead, if a message-passing call falls in the middle of a block of serial computation, this must be modelled by three separate submodels. For example, the code:

```
// serial segment 1
z = a + b;
z = z + c;
...
z = z + n;
MPI_Send(&z, ..., from, to, ...);
// serial segment 2
y = y - m;
y = y - l;
...
y = y - d;
```

must be modelled as:

$$T(overall\ segment) \;=\; T_{serial\ segment_1} \;+\; T_{MPI\_Send} \;+\; T_{serial\ segment_2}$$

Modelling the code listed above in this way is necessary because the completion time of the `MPI_Send` call (T_{MPI_Send}) is dynamic, depending on the level of network contention that is present when it is called. Using this technique, the start time of the `MPI_Send` call is retained in the model. This information is required when determining the number of other messages simultaneously in transit, which will be used to dynamically select an appropriate probability distribution for the performance of the `MPI_Send` call. The performance models for various MPI operations are examined in detail in the following chapter, but for now they can be generalised as:

$$T(MPI\_operation) \;=\; t_{MPI\_operation}(type, when, size, from, to)$$

Here the time $t_{MPI\_Operation}$ for a specific MPI call is determined dynamically from the state of the virtual machine, introduced in the previous section, based on the type of the call, when it was initiated, the size of the communication, where it was sent from, its destination and the level of contention in the communication network.

A minor side track is necessary here to explain why the model described above was chosen rather than considering message-passing calls as subroutines. Although, strictly speaking, message-passing calls are subroutines, the subroutine models used in this technique (which will be covered shortly) are intended for the user program level. They are not necessary for the majority of library functions because such functions provide the illusion of simple statement performance. These library functions, such as MPI communication calls, can be better modelled as quasi-simple statements, i.e. their completion time can be "looked up" in an analogous way to the instruction timing formulae of simple statements.

However, unlike simple statements they may depend on a number of parameters where no simple mapping between source code and completion time exists. Instead, rather than simply looking up the completion time of library calls in a table, the completion time of these quasi-simple statements must be computed by some special purpose model such as that described above. Some examples of commonly used HPC library functions other than MPI operations that could be modelled in the same way are math library calls or optimised sorting routines.

Loop Constructs

The next programming construct to be examined is the loop. Loop constructs provide programmers with a concise way of repeating a series of statements, either across a range of values using a `for` loop, or as long as some some condition remains true using a `while` loop. A pseudo-code fragment containing a basic example of each type of loop is shown below:

```
for (i = 0; i < n; i ++) {
  // loop body
  ...
}
...
while (c) {
  // loop body
  ...
}
```

Generally speaking, `for` loops are used in situations that demand an explicit amount of repetition, for example processing all the elements of a data structure in order. Conversely, `while` loops are used where a statically indeterminate amount of repetition must be carried out, for example when applying an iterative calculation that must continue until some specified measure of accuracy is reached. More pedantically, however, either of these loops can be used to imitate each other (with equal performance) by manipulating the loop condition and/or by using `exit` statements. Therefore, the different types of loop can be considered equivalently provided that they can be transformed to some common description. From a performance modelling perspective, this common description is simply the number of loop iterations that will occur and, necessarily in some situations, the value of the loop condition at the beginning of each iteration. Once these facets of the loop have been established, the loop can be unrolled [9] and the resultant code can be modelled accordingly, i.e. it can be modelled as a separate segment of code with its own interspersed communication, further loops, conditional statements or subroutine calls.

Also, in situations with fixed per iteration processing requirements, the computational expense of simulating a highly repetitive loop can be reduced by only simulating the performance of its first so-many iterations; if the time for each iteration converges to a steady-state value, the time required to complete the loop can be simulated by that value multiplied by the number of remaining iterations [107].

As a manifestation of the well known *halting problem* [350], statically establishing these parameters for modelling loop behaviour is impossible in the general case. However, in many situations, and especially in the way that high performance computing codes are written, it is possible to achieve this. Because of the types of problems that high performance computing codes aim to solve, i.e. those with very large computational requirements, they are quite often purpose built to solve very specific problems. More precisely, they are usually designed to solve problems of a very specific size and to a very specific accuracy. As a result of these processing requirements, `for` loop bounds in such codes are usually quite explicitly defined and do not depend on dynamically derived values. When this is the case, it is usually a relatively simple matter to statically extract the loop conditions from the source code, although it can be complicated (but not impossibly so) if a chain of calculation on static variables is used by the programmer to specify loop bounds. If this is not the case, one of two simple fall-back strategies must be applied. Firstly, since the human in charge of the model building process may possess more insight into the problem than an automatic compiler, he or she could be prompted to manually specify the loop condition. This could either be a deterministic condition, or some form of symbolic condition, in which case the condition will remain a parameter to the final performance model. For example, if neither the automatic compiler nor human intervention can determine the number of loop repetitions that will occur when the program is run, it could be modelled by a parameter, say `i`, that signifies the number of iterations that will occur. The second strategy is related to this. Rather than requiring human intervention to parameterise the loop during the model building process, a loop bounds parameter could be automatically assigned by the compiler. Either of these approaches are particularly appropriate for `while` type loops, because the number of iterations that will be required to satisfy such loops is usually data-dependent and therefore statically indeterminate. What this means, essentially, is that in any case where a symbolic performance model results, the problem of loop repetition is being deferred until model evaluation.

Rather than being problematic, this is advantageous for several reasons. Primarily, it uncovers important data-dependent model parameters, thus providing a solution that satisfies Postulate 2, i.e. that it allows the effect of data-dependencies on performance to be expressed. For example, the input data size will be retained as a parameter to the final performance model. As another example, consider an iterative code where termination depends on reaching some given value of accuracy. Since this will depend on the input data, and cannot be determined statically, a useful performance model would identify the time per iteration. This is exactly what this technique provides, retaining the number of iterations that will occur as a parameter. Secondly, and related to this, a parametric model can be subjected to parameter studies to provide insight into program performance over a range of situations (see Requirement 7).

So, the general modelling formalism for loops is:

$$T(loop) \;=\; t_{loop\ body_1} \;+\; ... \;+\; t_{loop\ body_n}$$

Here the completion time of the loop is calculated by unrolling the loop, applying the appropriate modelling techniques to each iteration i of the loop body in turn to obtain $t_{loop\ body_i}$, and summing the results. For the common case where the loop body contains no message-passing calls and performs the same amount of computation per iteration, this can be optimised to:

$$T(loop) \;=\; t_{loop\ body} \;*\; n$$

and the resultant performance model may be incorporated with the models for surrounding code using the modelling technique for serial segments described above.

Conditional Constructs

Many of the concepts that were just discussed for modelling loops are also relevant to modelling conditional statements. Conditional statements are used to run different segments of code depending on the value of some condition. In C, this can be achieved using:

```
if (c1) {
  // segment 1
  ...
}
else if (c2) {
  // segment 2
  ...
}
...
else {
  // segment n
  ...
}
```

This syntax represents the most general form of conditional statement that can exist. It can express the simplest `if`-only condition, all the way to (via trivial transformations) arbitrarily complex `switch` statements. Semantically, this conditional statement executes the first segment of code where c_i is found to be true. Similarly to modelling loops, modelling this conditional statement requires the conditions c_i to be determined. Once again, the halting problem makes it impossible to statically establish the value of these conditions in general. Thankfully, as was the way with loops, the manner in which conditional statements are often used in high performance computing codes can simplify this problem in many circumstances. The following paragraphs identify the different ways that conditional statements can be used within the context of the PEVPM.

The simplest case to deal with occurs where no message-passing functions are contained in any of the conditionally executed code. This is directly analogous to the case where a loop body contains no message-passing calls, which was discussed above. For any code where this is the case, modelling the performance of that conditional construct becomes the responsibility of the local processing model, so traditional performance models can be used. There are two main traditional methods for modelling the performance of conditional execution – simulation (for examples, see Section 2.27) and probabilistic methods (for examples, refer to Section 2.10). Simulation is very accurate but can be very computationally expensive to evaluate and it requires a specific input data set to generate a model. Since the performance model being sought here requires both low-cost evaluation and must generalise to different input data sets, simulation is inappropriate. Probabilistic methods, however, exhibit exactly these desired features. Probabilistic methods assign a fractional value to each c_i which corresponds to the probability that it will evaluate true. A performance model is then constructed by summing the run-time of each segment multiplied by its probability of being executed. Of course, the tricky part of this problem is determining the probability estimates for each c_i.

Putting the problem of determining the probability estimates for c_i aside for now, consider the other case, i.e. where message-passing functions are contained in the body of a conditional construct. Fortunately, with a few modifications, the probabilistic method described above is also applicable in these situations. Recall that a basic necessity of the overall modelling technique is to determine the local instructions that will be executed at each process until the next communication event occurs. That communication event serves as a demarcation line between the current state of program execution and all possible future executions. Only the execution time of the current stream of instructions is evaluated before a decision is made about the next stream of instructions that will be executed. That time is then used as a property of the message-passing call, so that an approximation to the dynamic contention in the system can be made. Therefore, simply summing the probabilities of each outcome multiplied by the execution time of each outcome to obtain an average execution time is inappropriate. Instead, the probabilities must be used to randomly (but according to the specified weightings c_i) select a specific execution every time a conditional statement is encountered. This guarantees a unique instruction execution sequence until the next message-passing event. This is essentially a form of Monte Carlo sampling (see Fishman [126], Liu [218] or Robert and Casella [300]), where the outcome sampling is repeated (per conditional construct) during the life-time of the modelled program (for example when statements are repeated in loops).

The caveat of this technique is that the instruction execution sequence that is being modelled may not correspond exactly with what will occur in the real system. However, since the variation in execution sequence in the real program is caused by data-dependencies, it cannot be modelled statically anyway and the existence of this limitation is a moot point. Pragmatically, however, what this limitation means is that less confidence can be placed in the performance estimate of a brief execution sequence than a long execution sequence. In a short execution sequence, the probabilities chosen at random will have had less chance to converge to their desired values, c_i. In a long sequence, however, it is more likely that the probabilities will have converged to the desired c_i values; if necessary, a statistical error for completion time could be estimated using standard Monte Carlo techniques, based on uncertainty in the c_i values and the length of the execution sequence. Lastly, this probabilistic approximation to execution sequence closely parallels the probabilistic approximation to contention that was discussed in Section 3.4.2, and hence it inherits the same reprieve for high performance applications: by their nature, such applications are extremely long running, so for any realistic cases, long execution sequences are intrinsically guaranteed.

Returning to the problem of determining the probability estimates for c_i, as with modelling loop conditions described above, the conditions c_i may either be statically determinate, or statically indeterminate. One extremely common case of a statically determinable condition is shown below:

```
MPI_Comm_rank(MPI_COMM_WORLD, &procnum);
if (procnum == 0) {
  // code to execute on processor 0
}
else if (procnum == 1) {
  // code to execute on processor 1
}
...
else {
  // code to execute on all remaining processors
}
```

In fact, this is another example of the semi-static nature of this modelling approach. Here, *procnum* is not, strictly speaking, a static quantity because its value is not discernible from the code alone. However, it is not truly dynamic because its value does not change at run-time[3]. Rather, *procnum* is defined semi-statically by the MPI run-time system when the program is launched. In contrast with the loop conditions, however, in the majority of cases conditional execution will be statically indeterminate. In these situations, the conditions will need to be dealt with in the same way that loop conditions are dealt with, i.e. by prompting the user (who may have more insight into the problem) to enter either a determinate value or parametric value, or by automatically assigning parameters to each condition. Once again, this delays the problem of condition determination until model evaluation, with the same advantages as before, i.e. it retains data-dependent model parameters in the model and therefore allows parameter studies of performance in differing circumstances.

In summary of this section, the modelling formalism for conditional statements is written:

$$T(conditional) \;=\; c_1 : t_{segment_1} \mid ... \mid c_n : t_{segment_n}$$

In this expression the time to run a particular outcome is chosen at random from the weighted probabilities c_i, and then $t_{segment_i}$ is evaluated as a compound segment.

Functions and Subroutines

The final programming construct that completes the set of structures that are necessary for writing procedural programs is the subroutine. Subroutines are used as an abstraction that allows programmers to create simple interfaces to complex or recurring bodies of code. A subroutine consists of a declarative part that defines how it should be called, a list of parameters that can be passed to the subroutine, a body of code that is executed when the subroutine is called, and optionally a return value (in C, a subroutine that returns a value is called a function). Many examples of subroutines have already been described. For example, all of the MPI function calls that have been mentioned are functions. At the time, however, it was noted that those calls (and more generally most library calls) were going to be considered as quasi-simple statements, i.e. their performance would be looked up or computed by a special purpose model. That was because such library calls are mainly used to provide some form of complex but well-defined operation, and are therefore attractive to modelling as an indivisible entity. More importantly, the code for these library calls is not intended to be visible to a programmer. Instead, the declarative part of the library call and the parameter list is intended to give the programmer access to the

[3]Although in reality *procnum* is represented by a variable and can therefore be changed, it is intended to represent a constant quantity at run-time. Changing its value is nonsensical and serves no practical purpose.

functionality of the subroutine without having to worry about its implementation details. It is impossible, however, to model a user code subroutine in anything but a piecemeal fashion. At this level, subroutines are used to provide application-specific abstraction rather than the appearance of a complex instruction. They are likely to contain a large range of both local computation and message-passing operations in extensive loop and conditional structures. Therefore, modelling user level subroutines must follow on from the same modelling techniques that have already been discussed.

Similarly to modelling loops and conditional execution, modelling subroutines in the general case is fraught with difficulty. Once more, however, the way that subroutines are mainly used in high performance computing programs simplifies the problem. The main difficulty in modelling subroutines occurs when recursion is present. Recursion occurs when one invocation of a subroutine leads to the invocation of the same subroutine before the original subroutine has finished executing. While understanding about such situations can be determined using induction, this is a complicated endeavour. Fortunately, although recursion can produce elegant solutions to some problems, it is rarely used in high performance programs. This is mainly because recursion tends to add significant overhead to program execution time and alternative solutions can always be achieved without using recursion. Therefore, rather than provide a complicated procedure for dealing with such an uncommon and unimportant problem, the possibility of recursion will be excluded from this modelling technique.

Without recursion, modelling subroutines becomes a trivial problem. Semantically, a subroutine merely serves to transfer from the source of the subroutine call to the beginning of the body of the subroutine, where execution continues. When the subroutine completes, execution returns to where it left off in the calling body of code. Therefore, from the perspective of the PEVPM, all that is required is to insert the instruction stream from a subroutine body (up until a message-passing operation occurs) where the subroutine is called. Thankfully, the ability to do this is possessed by all C compilers by inlining functions. Therefore, the problem of modelling subroutines can be offloaded to an existing technique. This is summarised by the formalism:

$$T(subroutine) \;=\; t_{subroutine\ body}$$

Where $t_{subroutine\ body}$ is evaluated by applying the rules for a serial segments, loops and conditional statements to the body of the subroutine.

3.4.5 Building a PEVPM Model

The previous section formalised a mapping between the basic constructs used in procedural message-passing programming and a modelling language that can be used to describe their performance. This section builds on that foundation to show how complete models can be created to describe the performance of entire programs containing many basic constructs. The C language coupled with MPI will continue to be used as the vehicle for describing this, although the applicability to other procedural message-passing languages (including pseudo-code) should be clear.

Two methods for supporting the model building process are possible. These are the use of an automated compiler [173, 363] to produce performance models from existing code or a simple pre-processing compiler to interpret programmer supplied performance directives (for example like in HPF [174] or OpenMP [258]). The idea of augmenting a code with compiler directives provides an attractive way of developing a prototype compiler. In contrast to a completely general compiler which requires an extensive set of rules and associated data structures, the well-structured use of appropriate compiler directives can support a prototype compiler that requires only simple representations of general programming structures to be processed. More importantly, there are many details about program execution that a compiler cannot possibly determine at compile time, but for which a programmer may have some insight. In addition to this, programmer insight may allow manually specified directives to focus only on sections of code that have the potential to greatly affect program performance, thus creating smaller and simpler models. Although an automated compiler would provide a very useful tool for practical application to real problems, creating it would require a substantial effort in software engineering that would not provide any insight into the research questions associated with the performance modelling of parallel programs. Therefore, an easy-to-parse set of programmer-supplied performance directives were conceived to facilitate the model building process. This meta-language that simplifies the translation of program code into a performance model is described in the following sub-sections.

Since a performance directive is required to simplify the translation process for each of the programming constructs identified in the previous section, the following discussion will closely resemble the structure of that section: directives are presented for application to serial segments of code, message-passing calls, loops, conditional statements and functions and subroutines. In addition to this, some supplemental directives are needed to represent the characteristics of the machine that will support the parallel program. These machine-dependent directives will be examined first, followed by the directives that apply to each programming construct. All of these directives can also be found in Appendix A.

Machine Dependencies

The machine dependent directives are necessary to characterise the processing and message-passing capabilities of the parallel machine a code will run on. These directives must be incorporated at the beginning of the source code file containing the `main` procedure of the program being modelled. In order to prevent the directives from being interpreted by the normal C compiler, they must be contained in comment structures. The syntax of the directives used to define processing and message-passing capabilities are shown below:

```
// PEVPM Processor description = <processor identifier>
// PEVPM &         timing basis = <processor speed>
// PEVPM Network   description = <network identifier>
// PEVPM &            link <id> = <from> <to> <performance profile>
// PEVPM &                  ... = ...
```

The structure of these directives typifies all of the other directives that will follow. A performance directive is identified by the tag `PEVPM` at the beginning of a comment line. This alerts the Performance Evaluating Virtual Parallel Machine (which will be described in the following section) to the presence of a performance model construct when the source code is parsed by the performance modelling compiler. Following the `PEVPM` tag is the type of performance directive which may be continued on subsequent lines using an `&` symbol. In this case, there are `Processor` and `Network` directives.

The `Processor` directive has two attributes. The `identifier` attribute must be set to a string which describes the serial processing architecture of the machine, for example a CPU/compiler/OS triplet. The `timing basis` defines the speed of the CPU in MHz for which any serial processing times in the following model are valid (for the given processing architecture). This flexible approach to describing serial processing capability allows many performance models using different processor architectures and processor speeds to coexist in the relevant source file. This can be achieved using several `Processor` directives, each with their own unique `identifier` and `timing basis` attributes. The `Network` directive describes the communication capabilities of a parallel machine as a directed acyclic graph, where each edge represents an individual link in the communication network, and each edge is annotated by a `performance profile`. Each `performance profile` attribute is a path that points to a file containing the set of probability distributions that describe message-passing performance on that link based on message size for a given level of contention.

When it comes to evaluation, a particular processor architecture, processor speed and network characteristics can be manually selected in the PEVPM. Alternatively, a `Default` directive may be used to specify the default models that should be used for each facet of the machine model:

```
// PEVPM Default processor = <processor identifier>
// PEVPM &        numprocs = <number of processors>
// PEVPM &           speed = <processor speed>
// PEVPM &         speed_i = <processor speed>
// PEVPM &         network = <network descriptors>
```

Notice that there is a default `numprocs` attribute here that does not correspond to any attribute of the `Processor` or `Network` directives. The `numprocs` attribute represents the semi-static number of processors that the code will run on. In the same way that the number of processors is specified on the command line when an MPI program is launched, the number of processors must be specified when a performance model is evaluated. The speed of each of these processors is specified by the `speed` attribute, although, for heterogeneous architectures, this value can be overridden for any processor number using a subsequent `speed_i` attribute, where i is a number between 0 and `numprocs-1`.

In addition, the PEVPM assigns a semi-static value `procnum` to each virtual processor numbered `0..numprocs-1` that it instantiates to represent each of the `numprocs` processors. Since these pieces of information are critical to the structure (and hence performance) of almost all message-passing programs, the values `numprocs` and `procnum` are exalted above other variables in a program whose performance is being modelled. They are treated as constants so that a special form of conditional directive can be automatically evaluated. (The general form of the conditional directive is covered below). This allows the per processor instruction streams that the PEVPM relies on to be constructed without unnecessary user intervention. An example of this special form of conditional directive and the program code that it models is shown below:

```
// PEVPM Runon c1 = procnum < (numprocs / 2)
// PEVPM &     c2 = procnum > (numprocs / 2)
if (procnum < (numprocs / 2)){
  // run this on the ``bottom half'' of the machine
  ...
} else {
  // run this on the ``top half'' of the machine
  ...
}
```

The `Runon` directive takes condition attributes `ci` that represent the code to run on each processor depending on the evaluation of the conditional part. The value of the `ci` attributes can be either a list of processors, for example `1,2,3,8..10` which would run on processors 1,2,3,8,9 and 10, or an inequality containing only constants and/or the semi-static values `procnum` and `numprocs`.

Serial Processing

Once a processor capability has been defined, the time to execute serial segments of code can be specified. The syntax of the directive used to model the performance of serial segments of code is shown below:

```
// PEVPM Serial [on <processor identifier>] time = <basis time>
```

This directive contains the `Serial` keyword followed by an optional `on <processor identifier>` attribute and a `time` attribute. The optional `on` attribute can be used to specify an existing `processor architecture` for which this `Serial` directive applies.

Since every serial segment can validly be subdivided into other serial segments, a system for bracketing sections of serial code is needed to ensure that all serial code is associated with one and only one `Serial` directive. This requirement is easily met by a combination of the underlying bracket-delimited block structure of C and a convention: every bracket-delimited code block within a serial segment (i.e. up to conditional, loop or MPI-call boundaries) must have exactly one `Serial` directive per processor identifier; which can be easily verified automatically. Many `Serial` directives, however, can be attached to any serial segment of code provided that they are differentiated by their `<processor identifier>` attribute. This is important because different compilers/architectures may produce vastly different assembly instruction streams for a serial segment due to differing Instruction Set Architectures, compiler optimisations, available libraries, etc. For any particular processor architecture, the `time` attribute specifies the amount of time that the serial segment would take to run on a CPU running at `timing basis` speed (refer to the related `Processor` directive above). By the crude assumption that there is a linear relationship between serial processing speed and clock speed for any given architecture[4] this modelling method allows all speeds of CPU for each modelled architecture to be roughly catered for. For example, if a `Processor` directive has a `timing basis` of 500 MHz but a performance model for a 1GHz processor is required, the value of all `Serial` directives must simply be divided by two. Apart from using a `<basis time>` determined by using any of the techniques that were covered in Sections 3.4.1 and 3.4.4, the syntax `parameter <parameter name>` can be used to introduce a symbolic parameter into the performance model. This is useful for deferring modelling of the performance of serial segments of code until model evaluation in cases where the run-time of these segments is not statically determinable.

[4]In reality the relationship is non-linear due to bus and memory speeds. Note, however, that it is particularly accurate for small cache-bound problems because cache access times usually decrease linearly with clock speed for a given architecture.

Message-Passing Calls

The next performance directive to tackle is the one that describes a message-passing event. Any message-passing directive that is encountered implies the end of a segment of serial computation. The time that the specific message-passing operation described by such a directive will take to complete is dependent on the type of the call, when it was initiated, the size of the communication, where it was sent from, its destination and the level of contention it will encounter in the communication network. With the exception of contention, which is a dynamic property that must be determined from the state of the PEVPM (which will be discussed in Section 3.5), this information can be determined from the program source code. The syntax for this performance directive is shown below:

```
// PEVPM Message type = <MPI operation>
// PEVPM &       size = <message size>
// PEVPM &       from = <processor number(s)>
// PEVPM &         to = <processor number(s)>
// PEVPM &   req/stat = <request/status identifier>
```

A `Message` directive consists of a series of attributes that identify the `type` of MPI operation whose performance is being modelled – for example an `MPI_Isend`, `MPI_Recv`, or `MPI_Bcast`, etc – the `size` of that message in bytes, `from` where the message will originate, `to` where the message is being sent, as well as a `request/status` identifier that will be used by the PEVPM when it is testing for the completion of asynchronous communication operations using `MPI_Test` or `MPI_Wait`, etc. The `size`, `from` and `to` attributes can all be modelled similarly to the `time` attribute for serial segments, i.e. by using a constant number, or using `parameter <parameter name>` syntax. In addition, the source and destination attributes may be constituted from a simple equation using the semi-static `procnum` and `numprocs` values. Furthermore, in the case of group operations a range of processor numbers may be specified. For example, an `MPI_Bcast` operation would consist of a singular `from` processor and `to` processors `0..procnum-1`. The following two chapters expand on the details of how the performance of specific MPI operations are modelled.

Loop Constructs

In the previous section, loops were the easiest programming construct to produce a performance formalism for. This maps to a correspondingly simple PEVPM directive, which is shown below:

```
// PEVPM Loop iterations = <number of iterations>
```

Typically, the `iterations` attribute will be a constant for `for`-type loops and a `parameter <parameter name>` for `while`-type loops, although some exceptions exist. The exceptions are encountered when conditional statements in the body of the loop are used to manipulate the loop variable. In these cases, a `parameter <parameter name>` attribute must be used, and determining the number of iterations must be deferred until the model is evaluated.

Conditional Constructs

The final PEVPM directive models conditional constructs. For practical reasons, it does not correspond exactly to the modelling formalism for conditional constructs that was described in the previous section. Rather than directly specifying the probability of certain outcomes, a weight is assigned to each outcome. Then the probability of each outcome is calculated by divided the probability of each outcome by the sum of weights of all the outcomes. This is more useful than explicitly specifying probabilities because if conditional outcomes are later added to account for new situations then their weights may possibly be added such that the dependent probabilities will change appropriately. The `Condition` directive is given below:

```
// PEVPM Condition c1 = <weighting of condition 1>
// PEVPM &         c2 = <weighting of condition 2>
// ...
// PEVPM &         cn = <weighting of condition n>
```

Checks and Balances

A useful bonus from using directives to describe the performance of parallel programs is that any desired elements of an automatic compiler can be added piecemeal. While PEVPM directives can be included manually, an automated solution could be created by implementing a compiler to parse the source code and target the constructs of the PEVPM directive language. As part of this solution, the application of some simple semantic rules would greatly enhance the robustness of such a compiler. For example, checks could be made to ensure that every serial segment is preceded by a `Serial` directive, every message-passing call is preceded by a `Message` directive, attributes of `Condition` directives correspond to every possible outcome of a conditional statement and so on. Although no attempts have been made to implement an automated compiler or semantic checking (because that would be too time consuming and it involves no significant research questions) it is useful to realise that there are no fundamental hurdles to creating such valuable tools.

Related to this discussion on the syntactic and semantic checks that could be introduced into the PEVPM system is the question of model validity in the presence of compiler optimisations. Most optimisations in modern compilers are based on techniques such as common subexpression elimination and basic block reordering [7]. Because these procedures are practically impossible to apply across function call boundaries or conditional statements, etc, the effects of optimisation on this performance model will be confined to the `time` attribute of `Serial` directives. Therefore, using compiler optimisations will not invalidate a PEVPM model but serial segment submodels will only be applicable for a particular processor architecture using a particular compiler and any associated optimisations.

3.5 Automatic Performance Evaluation

Once a performance model of a message-passing code has been created using PEVPM directives, the implications of that model can be evaluated on the Performance Evaluating Virtual Parallel Machine. The PEVPM consists of an abstract parallel machine and an associated set of algorithms that govern the execution of PEVPM directives on that machine. This combination provides a platform that is capable of reproducing the time-behaviour (and hence performance) of programs abstracted by the generalised model of message-passing codes detailed in Section 3.4, i.e. where processors independently run segments of local computation until they cooperate by exchanging messages. This section begins by describing the computational basis of the PEVPM, continues on to formalise the rules that are used to simulate program performance, presents an illustrative example of how the technique works, discusses the power of the methodology, and concludes by presenting some situations where it could be usefully applied.

Commensurate with the simplicity of the generalised model of message-passing that has been described is the conceptual simplicity of the abstract parallel machine that supports it. The computational basis of the abstract parallel machine is illustrated in Figure 4, which shows `numprocs` virtual processors that are capable of serial computation, each with its own send and receive message queues for communication. However, instead of executing real code for serial processing or message-passing, the instruction set of this abstract machine is the collection of PEVPM directives that are detailed in the previous section. At the beginning of a model evaluation, each virtual processor is provided with a copy of the entire PEVPM model of the message-passing code, which it then begins executing according to the placement of `Runon` directives. Execution progress at each processor is recorded by a local program counter, which is incremented every time a PEVPM directive is processed. For example, when a `Condition` directive is processed the program counter will move forwards in the directive stream to point at the directive associated with the outcome of the condition. When a `Loop` directive is encountered, the PEVPM directives in the body of the loop are unrolled to meet the loop bounds and the program counter jumps to the first directive in the first iteration of the loop. For simplicity,

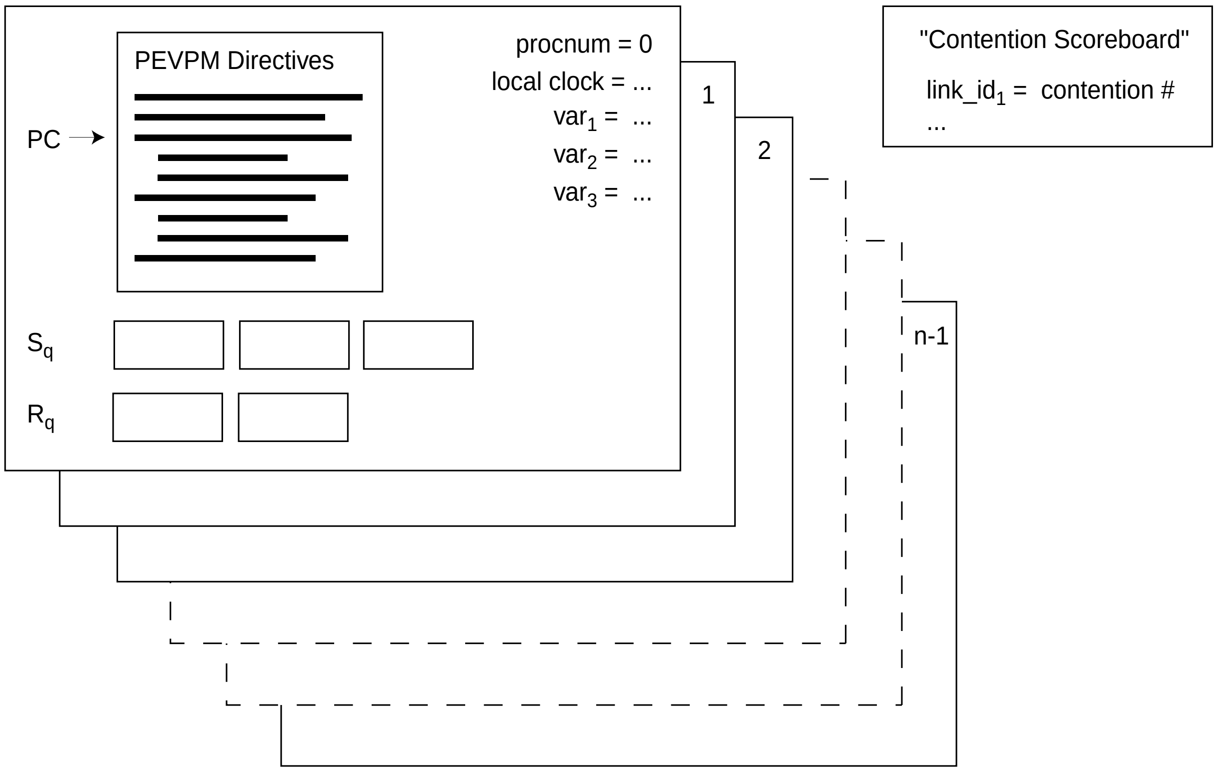


Figure 4: The computational basis of the Performance Evaluating Virtual Parallel Machine is an abstract machine with n virtual processors capable of executing PEVPM directives, a program counter (PC), send queue (S_q), receive queue (R_q), local clock and any specially exalted local variables.

it is assumed that neither of these control flow actions incur any explicit time penalty. While this is usually a reasonable approximation for most message-passing codes, nominal processing times could easily be associated with these constructs for situations where this is not the case. Since the time-behaviour of the program is the desired product of this analysis, each processor also maintains a local clock that is incremented when a PEVPM directive calls for local processing or a message-passing operation. For local processing the clock is simply incremented according to the `time` attribute of the associated `Serial` directive. For message-passing operations, the clock is incremented by an amount that is computed using the performance model for message-passing operations that is described in the next chapter. Although it is not necessary to understand precisely how those times are computed for now, the structures on which those calculations rely are pertinent here. These structures are the send and receive queues that are maintained by each virtual processor, as well as a global *contention scoreboard* that maintains an account of the number of messages in transit throughout the communication network.

Using the computational basis detailed above, the PEVPM employs an iterative two-phase process to simulate the performance behaviour of a program based on its PEVPM directives. The first phase, called a *process sweep* and described by Algorithm 1, simulates

Algorithm 1 PEVPM Process Sweep

```
for $p$ in 0 .. $numprocs - 1$ do
  while $p.runnable$ is true do
    get $p.directive$
    if $p.directive$ is Condition then
      evaluate condition
      set $p.program\_counter$ to branch location
    else if $p.directive$ is Loop then
      unroll loop
      set $p.program\_counter$ to first loop location
    else if $p.directive$ is Subroutine then
      inline subroutine
      set $p.program\_counter$ to start of subroutine
    else if $p.directive$ is Serial then
      get $time$
      increment $p.local\_clock$ by $time$
      increment $p.program\_counter$
    else if $p.directive$ is Message then
      get $type$
      if $type$ is MPI_Isend or MPI_Irecv then
        get $size, from, to$
        increment $p.local\_clock$ by $local\_completion\_time$
        push $(type, size, from, to, p.local\_clock)$ onto $p.send/recv\_queue$
        increment $p.program\_counter$
      else if $type$ is MPI_Send or MPI_Recv then
        get $size, from, to$
        push $(type, size, from, to, p.local\_clock)$ onto $p.send/recv\_queue$
        increment $p.program\_counter$
        set $p.runnable$ to false
      else if $type$ is MPI_Test or MPI_Wait then
        get $from, request/status$
        if $request/status.completed$ is true then
          if $type$ is MPI_Wait then
            set $p.local\_clock$ to $request/status.match\_time$
          end if
          set "$flag$" to true
          increment $p.program\_counter$
        else {$currently\ unknown$}
          push $(type, from, request/status, p.local\_clock)$ onto $p.recv\_queue$
          increment $p.program\_counter$
          set $p.runnable$ to false
        end if
      else {$end\_of\_input$}
        set $p.runnable$ to terminated
      end if
    end if
  end while
end for
```

Algorithm 2 PEVPM Match Sweep

```
set $match\_time$ to $\infty$
for $p$ in 0 .. $numprocs - 1$ do
   sort all unmatched $p.send\_queue$s into $g.send\_queue$ by $local\_clock$
   sort all unmatched $p.recv\_queue$s into $g.send\_queue$ by $local\_clock$
end for
set $s$ to head of $g.send\_queue$
set $r$ to head of $g.recv\_queue$
while $r$ is not NULL and then $r.from.runnable$ is false do
   get $r.type$
   if $r.type$ is MPI_Test then
      get $r.from$
      set $r.completed$ to false
      set $r.from.runnable$ to true
   else if $r.type$ is MPI_Wait then
      set $r$ to successor in $g.recv\_queue$
   else {$r.type$ is MPI_Recv}
      get $r.from, r.to, r.size, r.local\_clock$
      while $s$ is not NULL and then $s.local\_clock < match\_time$ do
         get $s.from, s.to, s.size, s.local\_clock$
         set $s.arrival\_time$ using $s.*$ and $contention\_scoreboard$
         if $s.to$ is $r.from$ and $s.arrival\_time < match\_time$ then
            set $match\_from$ to $s$
            set $match\_to$ to $r$
            set $match\_time$ to $s.arrival\_time$
         end if
         set $s$ to successor in $g.send\_queue$
      end while
      if $match\_time$ is not $\infty$ then
         if $match\_from/to.type$ is blocking then
            set $local\_clock$ of $match\_from/to$ to $match\_time$
         end if
         set $match\_from/to.completed$ to true
         set $match\_from/to.runnable$ to true
      end if
      if $r.from.runnable$ is false then
         set $r$ to successor in $g.recv\_queue$
      end if
   end if
end while
if $r.from.runnable$ is false then
   set all $r.from.runnable$ to deadlocked
end if
```

the processing that will be performed by all processes until they reach their next *decision point*, i.e. a message-passing call that signifies the end of a segment of local computation by that process. The second phase, called a *match sweep* and described by Algorithm 2, searches for the path that further processing will take by identifying the earliest decision point from all processes, known as the *pivotal decision point*, and deciding what will occur at that point. Chaining the decisions made by successive process/match sweeps simulates the entire execution structure of the program.

An evaluation begins by following the thread of control on the first processor: the program counter steps through PEVPM directives until a decision point in execution structure is encountered. Until this occurs: 1) the execution structure of the process is evolved according to `Condition`, `Serial`, `Subroutine` and `MPI_Wait` directives; 2) the local clock is incremented in concert with `Serial` directives or by submodels for the local completion time of asynchronous message-passing calls; and 3) any communication events are placed on the local send or receive queue as appropriate. At any point where a `parameter <parameter name>` attribute is encountered, execution is suspended while a higher authority is consulted for the value of the parameter. In most cases this symbolises user intervention, although in some cases it may be possible to infer the value of the parameter from the special variables stored in a process description. When a decision point in the control structure is encountered, e.g. a directive symbolising an `MPI_Send` or `MPI_Recv` operation is reached, further evaluation of that process is postponed by setting *p.runnable* to false and evaluation skips on to the next process. Directives symbolising `MPI_Test` or `MPI_Wait` operations also imply a decision point in the control structure: in correct programs they are required to test and "flag" the arrival (or non-arrival) of non-blocking message-passing operations. Inevitably, this boolean information will eventually be used to determine the progress of the process involved. Although this must occur in a conditional statement and could hence be dealt with by its associated `Condition` directive, it is sensible to suspend the process sweep at this point. This is because the arrival of that message cannot be determined until a match sweep has been completed. If complete process/match sweeps are allowed to occur, the arrival of this message will eventually be automatically determined and the result can be fed back into the modelling system, absolving the user in charge of the modelling process from `parameter`-like intervention in these situations.

A minor addendum is necessary to explain why non-blocking operations need to be treated differently to blocking operations. Consider the example show in Figure 5. This example shows a simple interaction between three processes in which deadlock would occur if blocking send operations were used. However, this program would complete if non-blocking send operations were used. Since these are both legitimate (and in fact, common) ways in which message-passing is used, the PEVPM must behave in a semantically

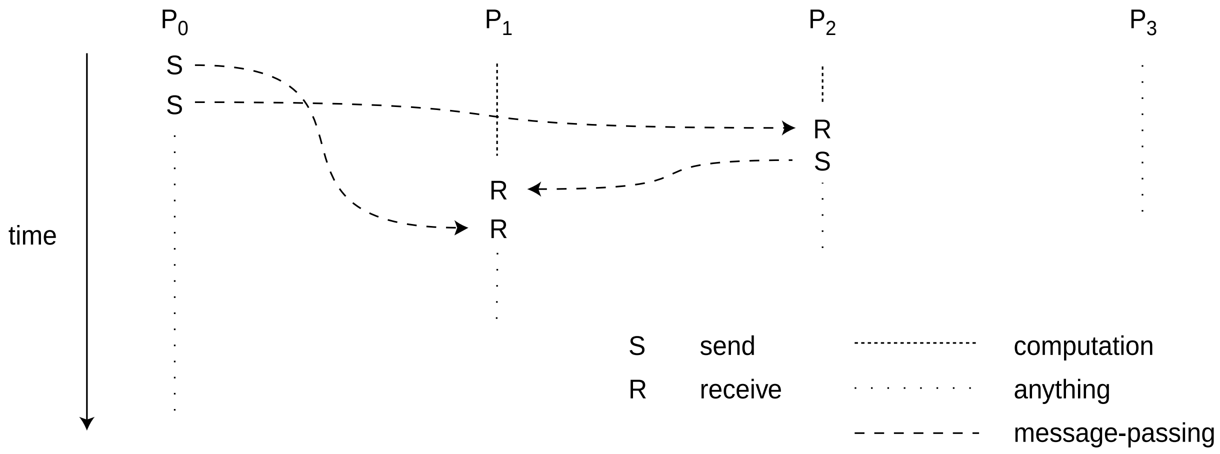


Figure 5: An example interaction between three processes in which deadlock would occur if blocking send operations were used; the disposition of the PEVPM to deadlock-or-not must be the same as the actual programming model.

identical way. Essentially this requires that all possible send and receive operations are identified during a process sweep so that the match stage can be guaranteed to complete correctly. This is achieved by allowing a process sweep to continue when non-blocking operations are encountered.

Related to this, and as will be seen in Chapter 4, many MPI implementations split long messages into a series of smaller messages to minimise buffering requirements. This can significantly affect the overall message-passing time that is visible to user applications. Although these performance effects are implicitly accounted for in PEVPM models through the benchmark results that are used as inputs, it is also possible to expose such effects explicitly. For example, if a particular MPI implementation's transmission strategy is known (via source code, documentation or the detailed examination of benchmark results) then that strategy can be simulated accordingly by special purpose models.

Once all processes have completed a process sweep, the second phase match sweep is initiated to determine the result of the earliest decision point that occurs for any process. This is the only decision point that can be solved during this match sweep because its outcome can affect any of the future decision points, as shown in Figure 6. The match-sweep line divides the execution progress into known events and unknown events. Clearly the earliest unmatched receive, i.e. the one posted by process P_3, can only be matched by the message labelled "1". However, the match at process P_2 cannot be made until all possible sends that could match it have been discovered during a prior process sweep, i.e. the messages labelled "?". Therefore, once a match has been made the match phase is terminated and a new process sweep is begun. How a match is made depends upon the type of operation that is at the head of the global receive queue. If it is an `MPI_Test` operation then the fact that processing did not continue during the process sweep means

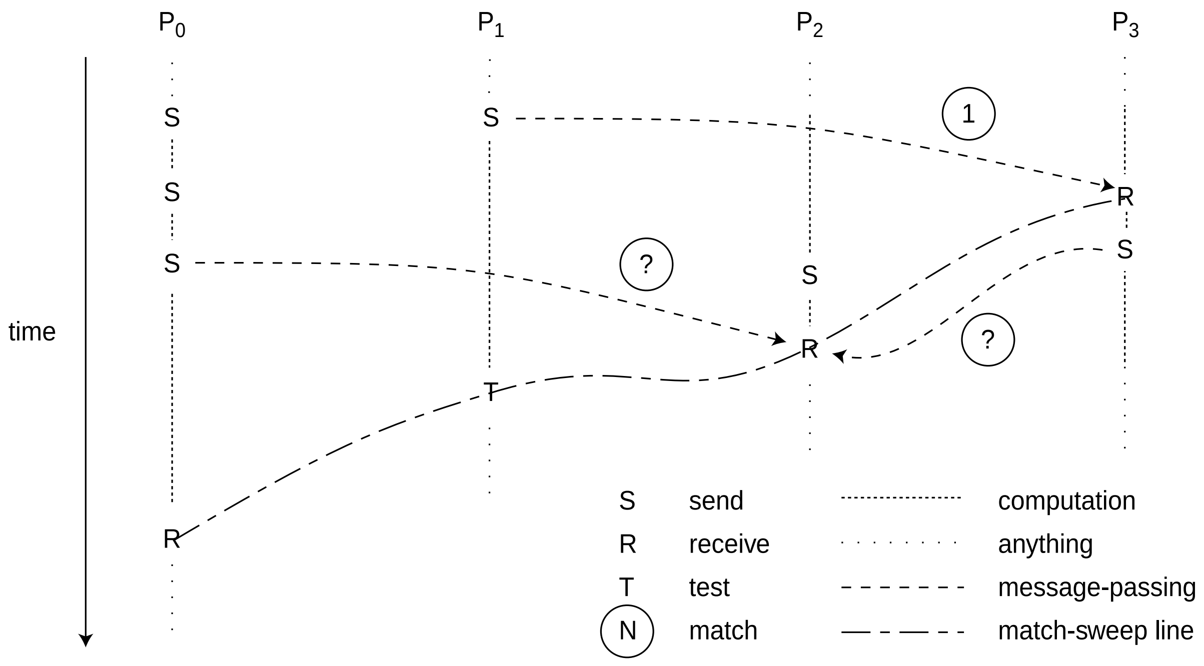


Figure 6: The structure of program execution is recorded by a match-sweep line that separates events that have happened from the set of future events that may happen.

that the result of the test was unknown at that point, i.e. the previous send or receive to which it was referring had not been matched yet. However, since this is now the earliest pivotal decision point, there can no longer be any pending matches between send and receive pairs that could occur before the test operation. Therefore, the *completion* parameter for the test operation can be set to false and a new process sweep can be initiated, thus providing a forward execution path. Dealing with an `MPI_Wait` operation at the head of the receive queue is similar, except that its blocking semantics means that the process must remain blocked (rather than being made runnable) until a match has been made. Therefore, having yet to determine a forward execution path, the match sweep must continue by trying the next earliest operation on the receive queue. Finally, on to the most interesting case: where a receive event must be matched with a complementary send event. A send provides a potential match partner if its destination matches with the process associated with the receive call that is currently being evaluated (and provided that other meta-data such as the MPI tags and communicators match). However, timing information must also be considered in order to find the correctly ordered match: a receive will match with the first appropriate message to arrive. Determining the unique message that fits this criterion can be achieved using the knowledge of when all the messages were sent (which they were tagged with during the process phase) and their transit times through the communication network.

Obtaining the transit times of messages through the network is a difficult task because they are affected by contention. As discussed in Section 3.4.2, the technique that is used to model the performance of messages subject to contention relies on knowing the number of messages travelling throughout the communication system with which the message being modelled must compete. This information is maintained in the PEVPM by a contention scoreboard. The structure of the contention scoreboard must be developed to match the network topology of the parallel machine that it describes: in the general case, every possible link in the network is described by a *contention number* which indicates the number of messages that are currently in transit on that particular link. For commodity crossbar switches serviced by a high-bandwidth backplane, the entire contention scoreboard is well-described by just one link which represents that backplane. While the events on the send queue are being scanned during the match sweep, the contention scoreboard is continually being updated. Every time a new message is encountered, the contention numbers of any links that it must traverse (which can be determined using source, destination, and topology information) are increased by one; and the contention numbers of any links where a previous message has completed are reduced by one.

This is not an exact means of determining the duration of contention on individual network links because it estimates the contention a message faces on each link only once, when the message is scanned during the match process. It does not take into account any increase or decrease in contention that the message will face during its lifetime. However, this merely serves to more accurately simulate the true effect of contention on a microscopic scale: any messages that get a head-start are more likely to hold up later messages than be held up themselves. Furthermore, since this has equal likelihood of producing both over-estimates or under-estimates of contention, it tends to even out over simulations with a large number of messages. Fortuitously once again, this is the very nature of the high performance codes which are of concern for the PEVPM modelling technique. Finally, once the contention numbers that a message must transit under are known, an appropriate distribution can be chosen to simulate message-passing time. From this, a value will be chosen at random, thereby simulating the effect of contention on the completion time of the message.

Because a probabilistic model is used to calculate communication performance, an important caveat of the PEVPM methodology is that it evaluates the performance of a run that *could* happen, not necessarily what *will* happen. When the PEVPM requests the transit time of a particular communication operation a probabilistic model will be consulted and a sample time chosen at random, thus allowing the non-determinism caused by message-passing delays to be simulated. Although this produces a valid, microscopically detailed performance trace, the event ordering that occurs in any one run is by no means representative of all runs. What is more reliable is the macroscopic performance prediction for the entire code because the large number of samples that it is constructed from will help the model converge towards a good prediction of average behaviour (for exactly the same reasons discussed in regard to the execution of conditional constructs in Section 3.4.4). This caveat represents more of an opportunity than a problem. By evaluating submodels many times over a whole simulation, each time calling upon the in-built Monte Carlo methods that direct execution according to conditional constructs and variable message-passing performance, a sensitivity analysis can be performed to produce a probability distribution that describes the performance stability of the system. Furthermore, for all but pathological non-deterministic programs, good estimates of the lower and upper bounds on execution time can be obtained by only two evaluations: one where the minimum time permitted by each submodel is always selected, the other where the maximum time permitted is always selected.

Related to this last point, it is important to note that even if a probability distribution of overall performance is not required, a good prediction of average behaviour cannot be obtained by merely modelling the delivery time of each message by its average value, since this does not take into account the non-linear nature of non-determinism in program execution. For example, consider an iterative parallel program where in each iteration every process communicates with exactly one other process and then takes part in a global barrier synchronisation. Ignoring the overhead of the barrier synchronisation (which would merely complicate the example), the completion time of any one iteration will be governed by the slowest message-passing time between any two processes in that iteration, not the average. This simple example highlights how contention delays in message-passing time together with precedence relationships in a parallel program result in performance deterioration, and therefore why an accurate performance modelling system must take into account the complete performance distribution (rather than, for example, an average) when modelling every individual message-passing operation.

3.6 Advantages of the PEVPM Approach

Much of the power of the PEVPM methodology lies in its quasi-execution of program code. Because computation and communication evolves in virtual time, the model automatically accounts for the effects of overlapping communication with computation, load imbalance and insufficient parallelism. Coupled with the facility to explicitly model communication losses, synchronisation losses and the associated resource contention issues of each of these, the PEVPM methodology is capable of accounting for all the sources of both performance and performance loss in message-passing parallel programs. The PEVPM methodology has the potential to produce highly accurate performance estimations for only a low-moderate evaluation cost. High quality results will be obtained when highly accurate times are used to model the run-time of serial segments of code and the completion time of message-passing events. A low-moderate evaluation cost is ensured because only the *structure* of the code is simulated. Therefore all the time-consuming segments of serial computation that occur in a real code are eliminated and reduced to simple numbers. Likewise, rather than waiting for time-consuming message-passing operations to finish, the completion times of those operations are determined using simple models within a fraction of the time that they abstract.

There are many situations where this performance evaluation methodology could be very valuable. Obviously, it could be useful to application programmers who are trying to optimise their code by automatically drawing attention to performance losses caused by load imbalance or insufficient parallelism, as well as to detect some program correctness issues such as deadlock. More ambitiously, it could find application in automated compilers for the same purpose. Also, apart from merely providing a probabilistic performance estimate for an entire code, with a little additional programming effort, individual execution traces could be made amenable to examination using existing performance visualisation tools such as Vampir [260] (or others [54]) in the same way as in Dimemas (see Section 2.21). These tools graphically display computation and the communication relationships between processes with respect to a global time-line. This can help programmers to spot the cost of message-passing and performance bottlenecks such as load imbalance and insufficient parallelism. Further to this, they can also assist in the process of debugging complex message-passing programs by helping programmers to search for unmatched send/receive pairs or race conditions. Typically these tools use trace data gathered at run-time from real message-passing code executing on real parallel machines. Gathering trace data at run-time is an intrusive process that can alter the system that it is measuring. Unfortunately this is a very real problem in the case of message-passing programs because of the volatility in program execution structure caused by non-determinism [309]; quite often the behaviour of message-passing codes will vary drastically between production runs and runs that are subject to the intrusion of a performance debugging tool. Using the PEVPM technique alleviates this because performance traces are produced by simulation, which is not susceptible to this problem. Moreover, producing execution traces using the PEVPM technique is especially useful for cases where access to target parallel machines for test purposes may be difficult or even impossible (for example for hypothetical or yet-to-be released machines) and hence traditional techniques cannot be used. Of course, this would require the provision of theoretical models for communication and computation performance, but these can usually be easily, if roughly, estimated.

Finally, although this has not been investigated, the PEVPM technique may possibly be extended to deal entirely in symbolic quantities. A symbolic calculation of execution time is foreseeable by using (deterministic) symbolic quantities rather than numerical data. This is exciting because it would allow for the automatic generation of algebraic expressions with very low solution costs for describing the approximate performance of program fragments. These expressions could be used for rapid performance evaluation over large parameter spaces. The caveat to such a technique would be that model re-evaluation would need to occur whenever parameter values would cause the ordering of events to change; the solution to this problem would probably involve using inequalities to keep track of break-points where execution structure would change. Despite this, the idea is not untenable because of the automated nature of the evaluation process.

3.7 Implications for Other Parallel Methodologies

Before concluding this chapter, a quick detour will be made to put the PEVPM system into perspective. Early in this chapter, the primary scope of the modelling technique was identified to be message-passing codes that run in a dedicated fashion on distributed memory parallel computers. It was also explained that codes written using other programming methodologies and running on different classes of hardware could be modelled by first translating them to an equivalent message-passing code. The other major parallel programming methodologies that need to be supported are shared memory programming and data-parallel programming. This thesis does not examine the details of how this can be done, but instead refers to previous work that has already tackled this problem.

The relationship between shared memory programming and message-passing is conceptually quite simple. Indicative of this are packages that allow shared memory programs to run on distributed memory machines [81]. At a very basic level, there are only two types of operation that each methodology supports: data access and synchronisation. The data access is very similar in both methodologies. Access to remote data is still achieved using some form of communication network. Other than the specifics of the function calls that achieve this, the only real difference between the two methodologies is where the read or modified data is stored. In shared memory systems, the data remains in its original remote location. In message-passing systems, a new copy of the data is stored locally, thus improving performance if it is accessed multiple times. Of course, this comes at the expense of a more complicated program, since the programmer must explicitly control the synchronisation between processes to ensure that they are acting on up-to-date data. These synchronisation issues are made easier for shared memory programming through the use of operators that exclusively lock shared data while it is being modified. These simple semantics can be easily emulated by the more powerful synchronisation operations available to message-passing programs. Taken as a whole, it is clear than shared memory programs can be easily adapted to message-passing and the distributed memory approach.

The relationship between data-parallel and message-passing paradigms is even simpler.

Data-parallel languages provide a higher-level way of describing parallelism than shared memory or message-passing programs. Essentially, they allow the programmer to specify *which* data can be processed in parallel, rather than the communication and synchronisation details of *how* parallelisation should be accomplished. A data-parallel compiler then uses this information to lay out the data on processes, and automatically determines the communication that needs to happen between these processes. At a lower level, however, the processing and communication must still be carried out – and the crux of the similarity is this - using the same operations that are available to message-passing (or shared memory) programs [56, 205, 377]. Therefore, a message-passing version of a data-parallel program can trivially be obtained by compiling to an underlying message-passing system.

3.8 Summary

The introduction to this chapter explained that the main reason for using parallel processing is to reduce the computation time required for what would otherwise be very long-running programs. Because poorly parallelised code tends to offer very little performance benefit, there is great incentive to ensure that parallel programs are highly optimised. Unfortunately, a lack of sufficiently accurate performance prediction methods for parallel programs has traditionally necessitated resort to a very time-consuming measure-modify design cycle to achieve this. This scarcity of useful performance modelling methods is quite simply due to the notoriously complex behaviour of parallel programs, which makes it very difficult to devise adequate modelling methods. The main contributor to this complexity is contention, which causes non-deterministic delays and therefore non-deterministic program execution. This chapter described a new performance modelling technique for parallel programs with sufficient power to accurately deal with these issues, thereby providing an opportunity to move much of the performance optimisation part of writing a parallel program to the initial design phase of the software development process, with the attendant advantages that that affords.

Much of the inspiration for this new performance modelling technique was based on previous work done by many researchers, and these influences were chronicled in Section 3.2. Section 3.4 built on this foundation and described how the performance of message-passing codes can be modelled in a very general way. Rigorous instructions were then provided on how to completely describe the salient performance features of a message-passing program written in a structural programming language (of which the combination of C and MPI was chosen as a representative example). Briefly, this involves annotating existing source code or writing pseudo-code using a performance directive language to define the computation and communication structure of a parallel program. Section 3.5 described an abstract Performance Evaluating Virtual Parallel Machine (PEVPM) that can execute these performance directives to simulate the time-structure of the program, and thereby predict its performance. Two properties make this essentially execution-driven simulation novel: 1) its ability to abstractly simulate the direct performance effects of contention; and 2) its ability to simulate the indirect performance effects caused by non-deterministic program execution due to that contention. This is achieved by dynamically creating submodels of individual computation and communication events on-the-fly using Monte Carlo sampling techniques based on data-dependencies, current contention levels in the system, and detailed probability distributions of the performance of all low-level operations for a given parallel machine. These probability distributions can either be hypothetical, or empirically determined by benchmarking a parallel machine using the techniques described in the next two chapters. This allows the PEVPM methodology to produce highly accurate performance estimations for only a low-moderate evaluation cost.

Section 3.6 explained that because a PEVPM simulation evolves in virtual time, it automatically accounts for the effects of overlapping communication with computation, load imbalance and insufficient parallelism. Coupled with its ability to explicitly model communication losses, synchronisation losses and the associated resource contention issues of each of these, the PEVPM methodology accounts for all the sources of both performance and performance loss in message-passing parallel programs. Furthermore, because all of these events can be annotated, the PEVPM is capable of automatically determining and highlighting the location and extent of performance loss due to any source; it can also automatically discover program deadlock, and help programmers trace down race-conditions. This information is of crucial importance in the design of well-optimised parallel programs; while it is easy to see how an application programmer could use this information, the PEVPM process and the information it can provide could potentially be integrated into tools for automatic or semi-automatic program parallelisation. Also, although the concept was only discussed but not investigated, the PEVPM could possibly be enhanced to produce entirely symbolic performance models rather than empirical ones, which would allow for even lower evaluation cost that would make the PEVPM approach even more attractive for very wide-ranging parametric-based performance optimisation.

Finally, Sections 3.7 and 3.8 summarised the relevance of the PEVPM technique to other parallel architectures, parallel programming methodologies and to the question of performance modelling for parallel programs in general.

Chapter 4

Benchmarking Point-to-Point Communication

4.1 Introduction

As was discussed in the last chapter, one of the keys to making accurate predictions about the performance of a parallel program is to have a very clear picture of the communication performance of the machine it is to be run on. This chapter introduces techniques that can provide an accurate characterisation of the MPI-based point-to-point communication performance of a parallel machine and the following chapter extends this to MPI-based collective communication. Together, these address the major outstanding requirements of the PEVPM performance prediction technique detailed in the previous chapter - namely the need for accurate submodels of communication operations. In particular, this chapter describes techniques that can produce very accurate probabilistic performance models for all types of MPI communication. This is made easier by the portability and widespread use of MPI. Much of the work involved with developing benchmark tests for characterising the performance of a parallel machine and associated message-passing system only needs to be done once, and the techniques may be readily re-applied. Aside from this simplifying property, however, the development of benchmarks that provide a good characterisation of MPI communication performance is quite difficult.

In order to obtain useful results from a synthetic benchmark, the performance of communication calls must be measured in the same context in which they will be used in real systems [142, 168, 373]. This requires knowledge of how a message will perform depending on its length, whether it is in cache or memory, whether it has synchronous or asynchronous completion semantics, whether it is a point-to-point or collective communication, and the the level of contention that it will encounter due to the global communication pattern that is in use. Fortunately, it is possible to enumerate all combinations of these properties and characterise the performance of each individually, although there will obviously be similarities and common trends.

More problematically, however, obtaining highly accurate performance results has been almost impossible until recently. In the past it has been necessary to obtain average results over a large number of iterations because of the inaccuracy of available clocks. This has the unfortunate effect of washing out any finely grained timing characteristics. Worse still, a lack of accurate global clock synchronisation has made it impossible to meaningfully compare communication patterns that start and finish at separate processes, i.e. almost all primitive communication calls. When combined, these time-keeping problems have made it impossible to accurately establish the effect that contention and other sources of non-determinism have on communication performance.

Although the motivation for the work described in this and the following chapter was primarily to support the PEVPM performance modelling technique, it can also be viewed as a stand-alone work with application in many other areas. Performance information that is detailed enough to provide insight into contention effects and other sources of non-determinism is potentially very useful. It can be used to analyse the performance of MPI implementations, compare the performance of different parallel computers, provide deeper insight into the inner workings of parallel programs, and of course, enable improved performance prediction for MPI programs.

Section 4.2 briefly examines the ability of existing MPI benchmarking techniques to provide accurate and detailed performance information. In short, they are inadequate, so new techniques are developed and described in Section 4.3, which details the design and implementation of a package called MPIBench [153]. The MPIBench package is used to obtain the performance profiles of several parallel machines in Sections 4.4 – 4.6. Some analytic approximations to these profiles are discussed in Section 4.7. Both the empirical results and analytic models will be used to evaluate the effectiveness of the PEVPM approach in the following two chapters. Finally, a short study on the stability of the measured results, detailed in Section 4.8, uncovers some important factors that relate to the scalability of large parallel programs.

4.2 Existing Message-Passing Benchmarks

A number of MPI benchmarks are currently available. The IEEE Task Force on Cluster Computing has identified those that are in widespread use [20]. The following subsections categorise the most flexible and wide-spread of these benchmarks according to the communication primitives and patterns that they test and the timing mechanisms that they employ.

4.2.1 Genesis/PARKBENCH

The PARKBENCH [169] suite of benchmark programs contains a series of low-level benchmarks called Genesis [212] that can be used to characterise some of the basic communication performance properties of an MPI implementation. In particular, its `Comms-PingPong1` and `Comms-PingPong3` benchmarks respectively measure the times for round-trip messages and simultaneous data exchanges over a range of message sizes where the messages are stored in contiguous memory segments. In addition, two variations on the `Comms-PingPong1` benchmark, namely `Comms-Strided1` and `Comms-Strided2`, allow strided memory buffers to be used instead of contiguous buffers, so that the effect of the memory hierarchy on message-transmission performance can be gauged. All of these results are then fitted using linear regression to obtain values for latency and bandwidth in each circumstance, which provides a simple description of the message-passing performance of the system under consideration. On top of this, the `Comms-Allgather` benchmark measures the total saturation bandwidth of the communication backplane as a function of the number of processes used. This provides valuable information about the scalability of the communication network. Finally, the `Comms-Synch` benchmark measures the time required for barrier synchronisation as a function of the number of processes taking part in the barrier. While the simple parameters that can be measured by Genesis are very well understood and of undoubted value for making rough comparisons between machines, they do not provide enough accuracy for detailed performance models.

It is worth noting that the instructions for using the Genesis benchmarks encourage any tests to be run several times to check whether the results obtained are repeatable, along with the basic advice that the lowest benchmark times are those that are least likely to have been affected by contention, and should therefore be considered the most reliable. Firstly, this is true only to an extent. Because the measurements made by Genesis are based on averaging the results of a large number of repetitions, even the minimum total time observed is likely to include time for individual messages that were influenced by contention effects. Secondly, while the results observed with minimum execution time are arguably the most repeatable results than can be obtained by the Genesis benchmarks, ignoring slow results can potentially misrepresent true message-passing performance.

4.2.2 NetPIPE

NetPIPE [329] is a protocol independent network performance measurement tool, although a module has been developed for MPI performance measurement. NetPIPE consists of a protocol independent driver mechanism, and a protocol specific communication section. The driver mechanism is only designed to measure the performance of simple ping-pong communication between two processes. It does this by timing individual ping-pong operations using the standard Unix clock at the initiating process, although it only reports the minimum time observed. While this simplistic performance reporting mechanism could be easily extended to report performance distributions, this would be of little value because: 1) those measurements are only made using a coarse-grained local clock; and 2) there is no support for complex communications patterns, where contention and hence performance variability is likely to be observed.

4.2.3 Pallas MPI Benchmarks

The Pallas MPI Benchmarks (PMB) [259] provide a set of low-level benchmarks that measure the performance of a large range of common MPI operations. The PMB are straight-forward, well-documented and have been widely used. Importantly, version 2.2 of the PMB introduced `Multi` versions of its range of low-level benchmarks, which test concurrent message-passing performance under global load. However, the accuracy PMB results is hampered by inadequate timing methods.

The PMB rely on using MPI's builtin `MPI_Wtime` routine to time the execution of MPI message-passing operations, which has two disadvantages. The low-level clocks that `MPI_Wtime` relies upon are often of relatively coarse granularity compared to the duration of message-passing events. This makes it impossible to accurately time the execution of individual message-passing events. To compensate for this, the PMB measures the time to execute a large number number of message-passing operations contained within a loop and thence determines the average completion time for an individual message-passing operation. While this would provide reasonable results if there were very little variance in the completion time of individual message-passing operations, this is not the case because of network contention. Hence, results obtained using the PMB do not truly reflect the performance of the message-passing process. Secondly, even if `MPI_Wtime` for a particular implementation is based on a high-accuracy local clock, it is still usual that `MPI_Wtime` is not accurately synchronised between processes (the reasons for which will be examined in following sections). This makes it almost impossible to accurately examine the performance of individual message-passing events or how message-passing events interact, because timings can only ever be accurately compared with other timings made on the same processor. Hence, the PMB are not capable of measuring the performance variability that may arise in message-passing.

4.2.4 MPBench

MPBench [249] measures the performance of the most common MPI communication primitives over a range of message sizes. The default message sizes range from 4 bytes to

64 Kbytes, increasing in size by powers of two. For point-to-point routines only two processes are used. This is an unrealistic test because it does not account for network contention caused by many communicating processes on a reasonably sized parallel machine. Round-trip bandwidth using synchronous sends or bidirectional bandwidth using asynchronous sends can be tested. The asynchronous tests also allow for the application latency of an asynchronous send to be measured, although the results are highly volatile due to the effects of buffering schemes employed by any MPI implementation under consideration. MPBench can also test a variety of collective routines, namely broadcast, reduce, all-reduce and all-all communication patterns, although the completion time of these routines is only measured at the root process.

MPBench averages the time for a message-passing call to complete over many iterations of the call plus a small acknowledgement message that is required to verify that all of the messages have been received. The verification message only has a diminishingly small effect on the calculated average completion time because the already small amount of extra time that it takes to complete is divided by the number of iterations in the averaging process. In their paper describing MPBench [249] the authors point out that some machines have hardware timer registers which could be used to measure the completion time of individual message-passing calls, but the idea was not pursued.

4.2.5 Mpptest

Mpptest [150] avoids many common pitfalls that are made when measuring message-passing communication performance. Mpptest can measure synchronous and asynchronous point-to-point communication, as well as a selection of collective operations. The message sizes it tests are chosen automatically by an adaptive scheme that uses linear interpolation to focus testing near discontinuities in performance.

The fundamental design principle of the techniques that under-pin Mpptest is that the results of performance tests should be reproducible. To achieve this, Mpptest measures the average completion time of many iterations of a message-passing call in the same way as MPBench, but then repeats that process many times and records the minimum of the measured averages. The purpose of this meta-measurement technique is to smooth out (and essentially ignore) the effect of slow messages on the measurement process. Although this does lead to reproducible results, it ignores the distribution of results that are actually taking place. Furthermore, because Mpptest relies on round-trip times which are measured at only one process, it cannot be used to obtain a global perspective on the network contention effects that messages between other processes can cause.

4.2.6 SKaMPI

SKaMPI [297, 296] is the most flexible and thorough of the existing benchmarking tools. It provides an extensible framework for testing various communication patterns, as well as built-in support for testing a huge range of point-to-point and collective communication patterns. Automatic message size selection using a linear interpolation scheme similar to that of Mpptest is used. Although SKaMPI measures the time of individual message-passing calls, this is only to allow outliers to be discarded. At the end of a test the collected data are processed to provide the average of the data observed, and no attempt is made to characterise the distribution of results or any outliers that have occurred.

SKaMPI makes timing measurements with either the Unix system clock or `MPI_Wtime`. No attempt is made to synchronise clocks between processes. Because of this, only round-trip times are measured, and like MPBench and Mpptest, the completion time of group operations is only measured at a single process. Hence SKaMPI is also unable to provide detailed insight into the effect of network contention arising from messages between other processes.

4.2.7 Profiling Tools

There is another class of tools, apart from synthetic benchmarks, that are often useful for MPI performance analysis. Profiling tools such as Vampirtrace [260] and others [54] are designed to analyse the performance of "real" codes rather than the performance of artificial tests. In general, these tools log time stamps for a selection of message-passing calls and allow a post-mortem analysis of the time at which communication events occurred. Many of these tools can provide very insightful performance measures. For example, Paradyn [265] can present histograms which show the variation in message-passing time of the recorded data. Unfortunately, however, the accuracy, scalability and nature of these tools limit their usefulness for extremely detailed performance modelling. Firstly, the global clock synchronisation techniques that they employ are usually only accurate enough to guarantee the ordering of time-stamps – so that, for example, a message is never recorded as having arrived before it is sent. Secondly, when instrumenting a benchmark with extremely high repetition counts (which is required to generate smooth timing distributions) these tools can cause significant interference with the measurement process because of memory requirements and as results are flushed to disk. Thirdly and finally, remember the purpose of these tools: they are primarily designed to analyse the performance of real codes. Although they could be used to instrument the performance of (synthetic) micro-benchmarks, which are of interest here, these benchmarks still need to be written, and it is far more convenient to roll them into a portable stand-alone application.

4.2.8 Limitations of Existing Techniques

The previous sections have shown that the current techniques used in performance benchmarks for MPI systems suffer from one or more of three main inadequacies. The first inadequacy is the use of relatively coarse grained clocks for timing measurements. This forces a benchmark to average results over a high number of test repetitions, thereby losing detailed performance insight. In particular, none of the current synthetic benchmarks can generate distributions that show the variability in completion times that occur, or outliers that occur, both of which may have a large effect on program performance. Moreover, this important factor has been almost completely ignored in the literature; in fact, Mraz and Tabe *et al.*'s statistical examinations of communication time on the IBM SP2 (see Section 2.16) seem to be the only significant works that have made an attempt to quantify its importance. Tabe *et al.*'s main conclusions were that "it must be emphasised that using only means to model communication performance of parallel computers is inadequate" and that "understanding the tail behaviour of relevant distributions is critical to the development of good simulators". Unfortunately, this praise-worthy advice appears to have been, until now, either unheard or unheeded.

Secondly, the benchmarks either rely on `MPI_Wtime` to provide a globally synchronised clock or simply use ping-pong type tests to measure the total round-trip time, often using only one pair of processors. While relying on `MPI_Wtime` may be acceptable on high-end systems that have a hardware-synchronised global clock, it is not acceptable on low-end cluster systems where the accuracy of `MPI_Wtime` is usually bounded – if at all – by half of the round-trip time of a zero byte message. Alternatively, measuring only round-trip message-passing times negates the need for a globally synchronised clock, but it does not allow the direct measurement of the performance distribution of individual message-passing calls that begin at one process and complete at another. In the case of round-trip point-to-point messages, only the convolution of the completion time of two messages can be measured, which tends to broaden the distribution observed. Furthermore, in the case of collective communication, it is almost impossible to measure anything significant other than completion time at the root process – which provides no insight into the actual completion times of collective calls seen by individual processes.

Thirdly, and finally, none of the communication patterns used in current benchmarks were designed with clusters of SMP nodes in mind. If care is not taken with process placement, this can lead to measurement of intra-node communication performance when the intention is to measure inter-node communication performance.

4.3 Design and Implementation of MPIBench

Despite the unsuitability of previous techniques to providing highly detailed characterisations of message-passing performance, much can be learned from them. The techniques for performance characterisation presented in this section build on earlier work, in particular that of Gropp and Lusk [150] which provides a list of many common pitfalls that are made when measuring message-passing performance. As was noted in Section 4.2.5, the fundamental requirement of the techniques presented in Gropp and Lusk's paper was that the results of such performance tests should be reproducible. With this aim, they concluded that only the minimum time and "sort of" average time of a number of tests are repeatable. However, as the work presented in Section 4.4 will show, it is possible to achieve reproducible results without needing to resort to such simple, single valued views of communication performance. This is achieved by using an accurate globally synchronised clock, retaining the timing data of many individual calls to communication routines and applying post-processing to reveal the timing characteristics of MPI operations – including statistical analyses of variations in timing and the generation of probability distributions. These techniques have been packaged into a benchmarking program called MPIBench, which is capable of gathering comprehensive performance profiles on a large range of MPI communication primitives under varying conditions.

The following sections describe the timing framework MPIBench is built around, a global clock synchronisation mechanism that produces accurate results, the communication patterns that MPIBench employs to conduct tests and how the vast amounts of data that are captured during the benchmarking process are distilled into a more usable form.

4.3.1 Constructing a Timing Harness

MPIBench provides an easily extensible framework that can test the performance of message-passing calls in various patterns. The exact range of tests is controlled by a wrapper script which can also vary the number of test iterations, allowing a trade-off between measurement time and the accuracy of results. At the heart of MPIBench, an outer loop controls the test parameters, such as the type of message-passing call, message size and communication pattern. Each of these tests is run in a tight loop over the number of repetitions requested. The only operations in the loop are the message-passing calls themselves (which can be easily modified or extended as described in Appendix B.2), time stamps before and after each message-passing call using the local clock register and, for group operations a small amount of code to synchronise the beginning of tests.

As will be discussed in the following sections, the overhead of the time-stamping is extremely low and has insignificant effect on the code that is being measured. Likewise, the synchronisation code for group operations was carefully designed to be as non-intrusive as possible. Rough synchronisation may be achieved using the `MPI_Barrier` operation, which is sufficiently accurate on many systems. However, on other systems, this imparts a systematic skew on the time at which each process begins a test. Therefore, a far more accurate technique was also developed, where each process polls a global clock (discussed in the next section) until a prearranged global time is reached before it begins the test proper. In this way, all processes are guaranteed to start simultaneously, within the accuracy of the global clock. This improved synchronisation technique should only be used where absolutely necessary, because the prearranged synchronisation points must be sufficiently distant in time to ensure that all processes are ready to start when required. In particular, its should only be required when the performance of the `MPI_Barrier` operation on a machine is prone to experiencing a large number of delays caused by packet loss. In that case, the separation time should be some small multiple (e.g. 1-3) of the communication network's retransmit timeout (see Section 4.8).

MPIBench was designed to allow for high repetitions of each test in order to generate good quality profile data. Because of the amount of data that is logged in such circumstances, processing is performed after each test has been repeated the desired number of times. Firstly, all the time stamps are normalised to an accurate global clock. Then log files from all the processes are collated, processed and synchronised to disk before proceeding, in order to avert network file access interfering with further results.

4.3.2 An Accurate Global Clock

The accuracy of the standard MPI timing routine `MPI_Wtime` varies widely between different MPI implementations on different machines. MPI implementations are required to define a boolean variable named `MPI_WTIME_IS_GLOBAL` which specifies whether or not `MPI_Wtime` uses a globally synchronised clock, but even if this is this is true, the MPI specification only requires that the clock is synchronised to within half of the round-trip time of a zero byte message. Although this ensures that `MPI_Wtime` is accurate enough to measure message ordering, it does not allow message-passing time to be accurately determined between processes. Therefore an accurate and globally synchronised clock capable of precise timings of individual calls to MPI routines was designed and implemented.

Most modern processors are now equipped with 64-bit cycle count registers that are incremented on every clock cycle. These can be used for far greater local timing precision than has previously been possible. Local timing measurements in MPIBench are implemented as a preprocessor macro that stores in an array the current contents of the CPU's cycle counter (if it is available), or failing that, the value of `MPI_Wtime`. MPIBench supports cycle counting in modern x86-based processors (using the `RDTSC` assembly instruction [186]) and Sparc v9 processors (using the `%tick` register [187]). Alpha 21264

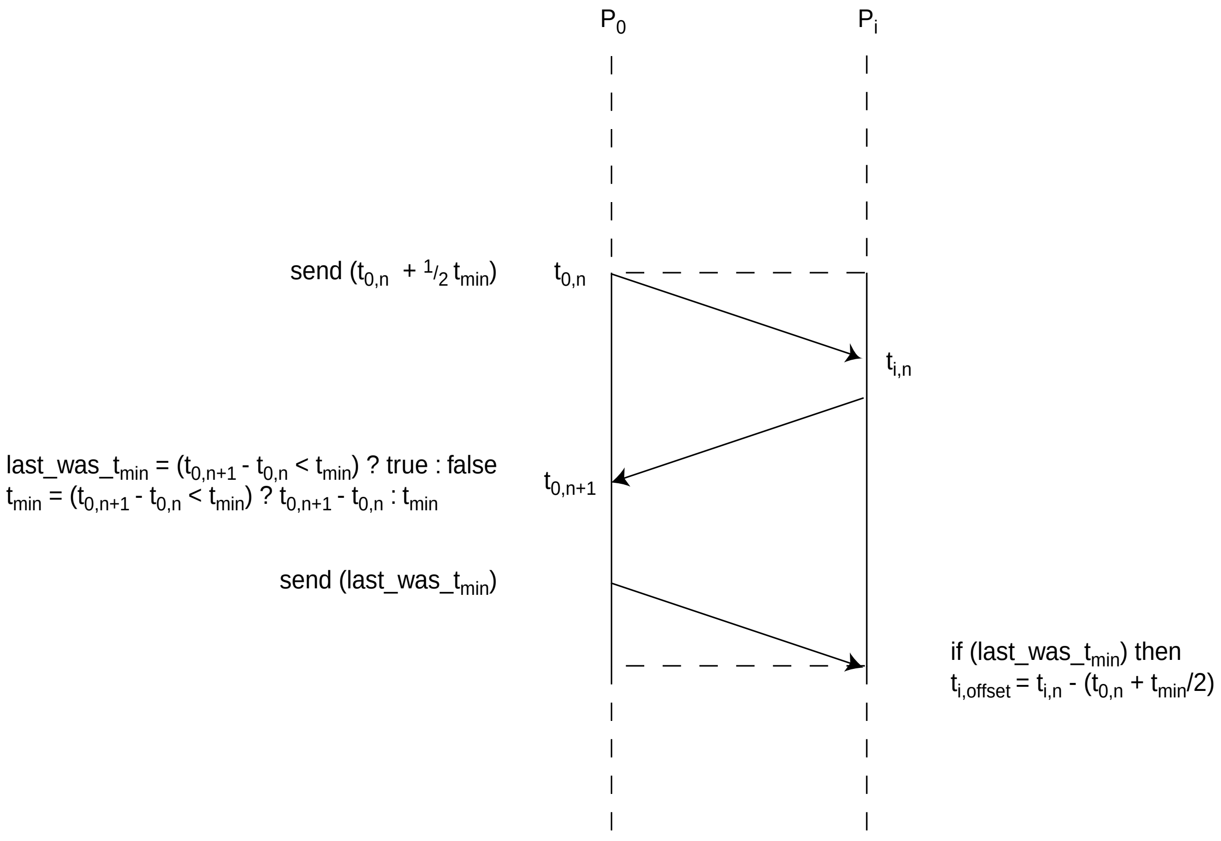

P0
Pi
send (t0,n + 1/2 tmin)
t0,n
ti,n
last_was_tmin = (t0,n+1 - t0,n < tmin) ? true : false
tmin = (t0,n+1 - t0,n < tmin) ? t0,n+1 - t0,n : tmin
t0,n+1
send (last_was_tmin)
if (last_was_tmin) then
ti,offset = ti,n - (t0,n + tmin/2)

Figure 7: One iteration of the clock synchronisation mechanism between processors P_0 and P_i.

processor support was attempted, but the cycle count register (accessed by the `RPCC` counter [76]) only provides 32-bit time-stamps, which overflow too quickly to be of practical use in MPIBench. As mentioned, `MPI_Wtime` can alternatively be used as the local time source, thereby providing complete portability of MPIBench. However, this should only be used on high-end machines where the granularity of `MPI_Wtime` is known to be acceptably fine, otherwise the accuracy of MPIBench measurements will suffer. Regardless of which method is used for local time measurements, the time-stamps are normalised to a synchronised global reference clock.

The global clock synchronisation techniques that were developed for MPIBench are similar to those of Christian [68] and Maillet and Tron [229]. Figure 7 shows one iteration of how processor P_i is synchronised to the reference clock on processor P_0. P_0 sends a message to P_i which contains the current time at P_0 plus an estimation of the minimum amount of time that the message will take to arrive, $1/2t_{min}$, where t_{min} is the current minimum round-trip time that has been observed by P_0. P_i receives the message and returns it to P_0 which calculates the total time that the round-trip message took to complete. If the round-trip time was the fastest observed so far, then the estimated time of arrival of the initial message was (probably) the most accurate yet, and P_i should use

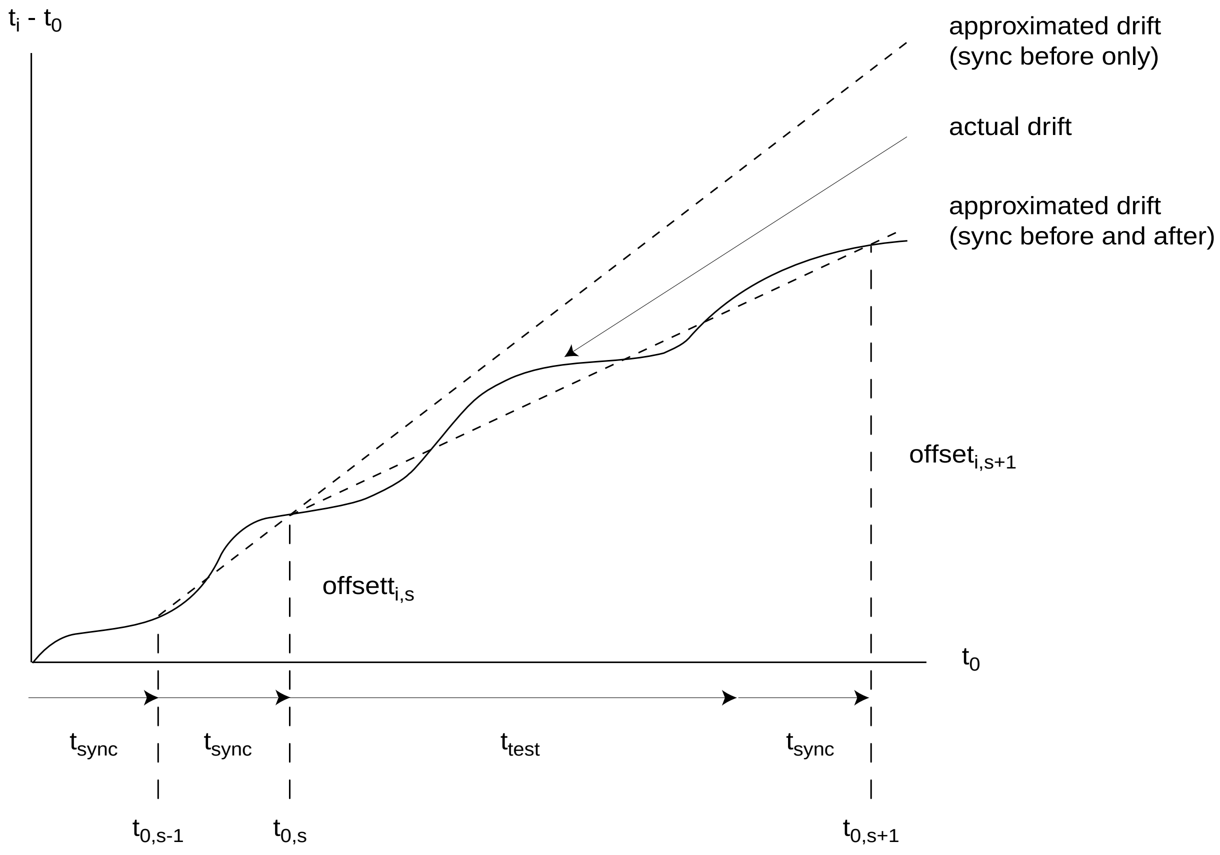


Figure 8: Comparing drift approximation using synchronisation only before a series of tests (extrapolation) with synchronisation both before and after a series of tests (interpolation).

the message's contents to calculate its offset from the reference clock. A second message is sent from P_0 to P_i as to whether or not this was the case, and if so processor P_i calculates the current approximation of the offset. Both processors repeat this process until a new t_{min} has not been observed for a prearranged number of repetitions. This user-defined parameter allows a trade-off between synchronisation time and synchronisation quality, which should be manually tuned for any given parallel machine until t_{min} can be repeatably obtained to within some desired accuracy. For the various types of network hardware examined in the remainder of this thesis, repetition values of between 100 and 1000 were found to achieve sub-microsecond accuracy. This pair-wise synchronisation is repeated for all the processors in the system, so the total time required for synchronisation is proportional to the number of processors in the system. Unfortunately, synchronising several processors at once is problematic using this technique because round-trip times close to the minimum are only observed if the reference processor is free to deal with only one message pair at a time, i.e. contention is absent.

Because the clocks in each processor run freely, merely calculating their offset at one point in time is insufficient for synchronisation to remain valid. The clocks on different processors will tend to drift apart from each other as shown in Figure 8. MPIBench uses a first order linear interpolation (rather than extrapolation) to account for this by calculating the drift between two synchronisation points: one before the tests themselves and one after the tests. (Note that clock drift has insignificant effect on the synchronisation process itself because any iteration of the synchronisation process occurs over a very short time). Using this approach bounds the error compared with the reference clock to the greater of the resolution of the local clock on each processor, errors in t_{min} estimates, and the maximum deviation from the actual linear drift between the clocks.

Firstly, the resolution of local clock measurements is linearly dependent on the CPU clock speed because a cycle counter is used; as a guideline example, resolutions of under 100ns were achieved on a 450MHz UltraSparc II and a 500MHz Pentium III. Secondly, as explained earlier, the magnitude of the errors in t_{min} estimates are inversely proportional to the number of synchronisation repetitions made by each process and can easily be reduced to sub-microsecond values. Finally, noise on the drift of clocks commonly used in computers is caused by thermal fluctuations in their operating environment. This noise has been shown to be of the order of about one microsecond in controlled machine-room environments [229]. Hence, since the overall error between any two clocks in the system is the superposition of the differences between each of the clocks and the reference processor, the global error can be reduced to about 2 microseconds.

4.3.3 Communication Patterns

MPIBench has built-in support for measuring the performance (as well as local completion time for any calls that return asynchronously) of many message-passing primitives, including `MPI_Send`, `MPI_Isend`, `MPI_Recv`, `MPI_Irecv`, `MPI_Sendrecv`, `MPI_Bcast`, `MPI_Barrier`, `MPI_Scatter`, `MPI_Gather`, `MPI_Allgather`, `MPI_Alltoall` and `MPI_Reduce`. Other MPI routines or even compound communication patterns can be easily inserted within the timing framework as described in the last section. In addition, MPIBench defines `each` and `total` variations of each of the group operation tests. These keywords control how the total amount of message data changes as more processors are added. If the `each` keyword is used, then each processor sends a fixed amount of message data, and the total amount of data grows as more processes are added. Conversely, if the `total` keyword is used then the total amount of message of data for the test is divided equally between all the available processes. This feature allows MPIBench to evaluate the effect of allocating more resources to a bigger problem, or conversely how performance for a problem with fixed size scales as a machine gets bigger. While this useful feature is a part of many parallel application benchmarks (for example in the NAS Parallel Benchmarks [28]), no other purely message-passing benchmarks offer built in support to achieve this.

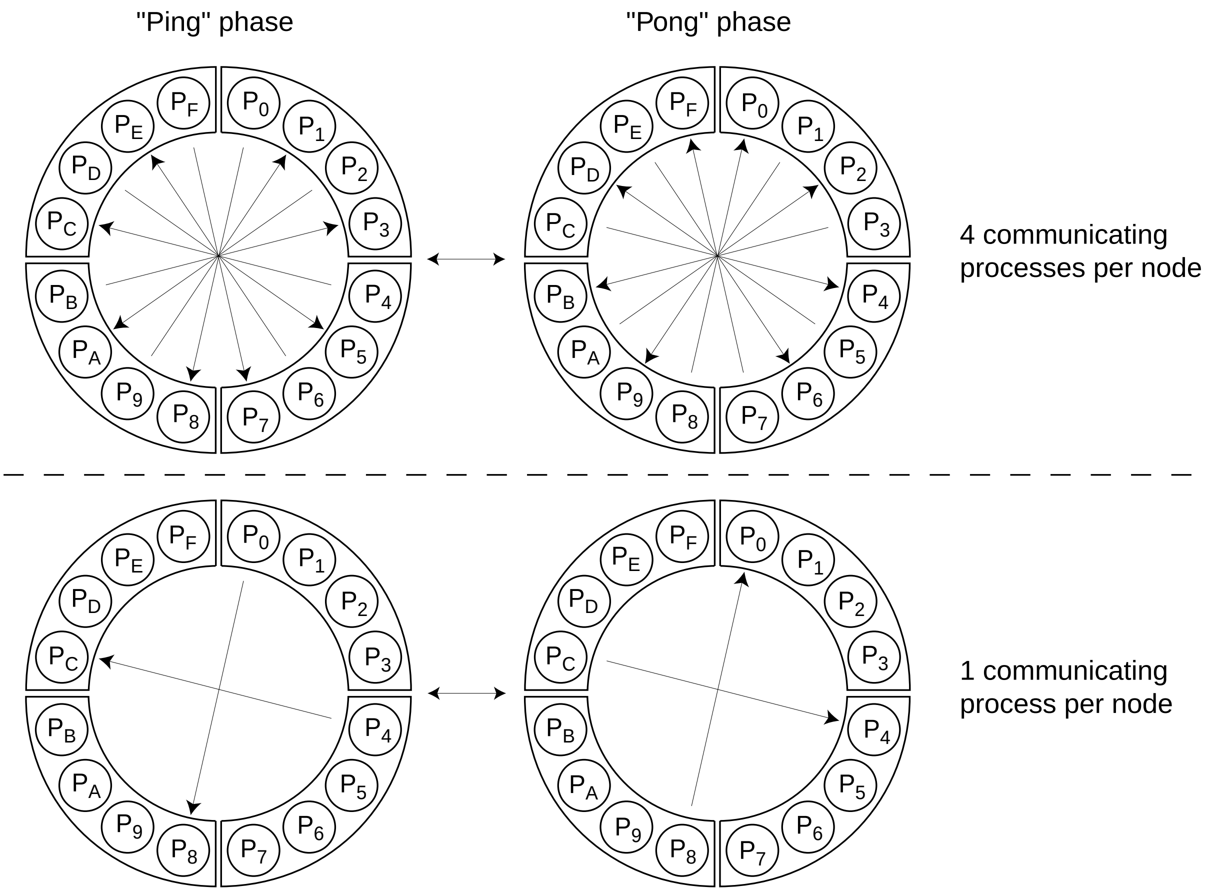

"Ping" phase
"Pong" phase
4 communicating processes per node
1 communicating process per node
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_A P_B P_C P_D P_E P_F

Figure 9: An example of how MPIBench controls process placement and balances communication for point-to-point tests that involve SMP nodes.

All of the MPIBench tests that measure the performance of peer-peer communication patterns were designed with clusters of SMP nodes in mind. This is an important consideration because such architectures have become very common over the last few years. Other studies on the effect of SMP nodes on communication performance, such as that of Capello *et al.* [61], have also started to emerge. Figure 9 shows how MPIBench allows the number of inbound/outbound messages from a node to be set, so that pairs of processes using point-to-point communication are prevented from residing on the same node. This ensures that all traffic during tests will transit the inter-node communication network. Note that intra-node communication performance can be determined by running a smaller test where test processes are only run on processors that are local to a node.

The top half of Figure 9 shows how a collection of 4-way SMP nodes communicate four concurrent messages, whereas the bottom half of the figure shows how the same nodes communicate just one message. In particular, it shows a case where the communication is divided into "ping" and "pong" phases so that all the nodes connected to one half of the communication network are configured to send while the other half receives. For many network topologies, and in particular for crossbar-switched (flat) networks and fat-tree

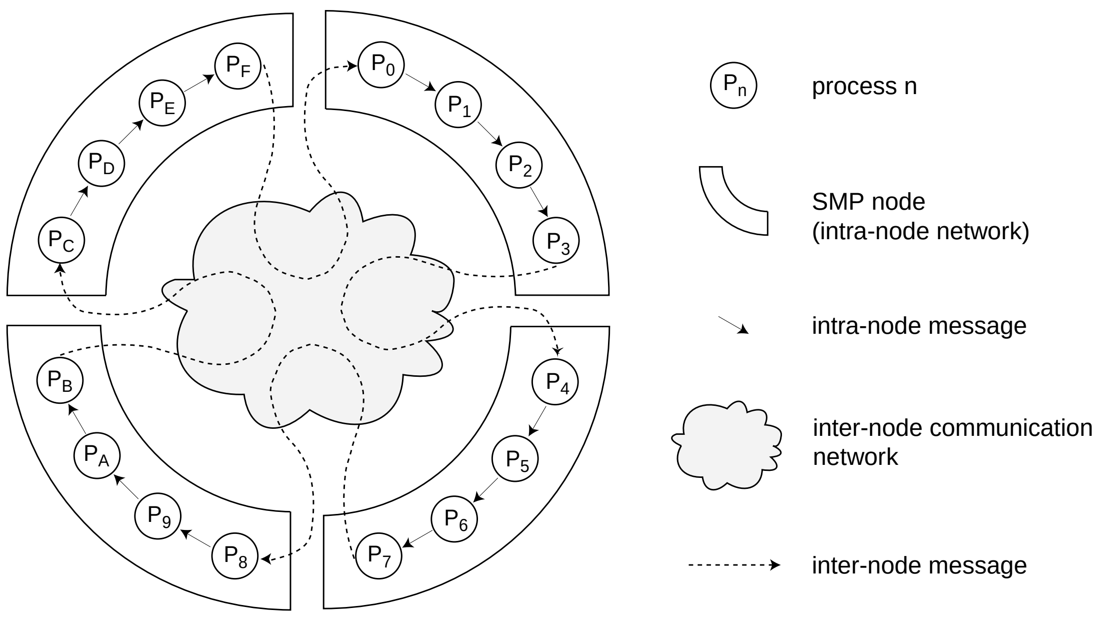


Figure 10: Intra-node and inter-node communication for a message-passing program with a unidirectional ring-shaped communication pattern on a cluster of SMPs; each type of communication must be modelled separately.

networks, which are commonly used to connect cluster computers (of either uniprocessor or SMP nodes), this communication pattern produces complement traffic [22]. Complement traffic exerts a uniform strain on the network where every message traverses the maximum number of communication links. For the topologies listed above, this provides a realistic simulation of point-to-point communication performance that can be expected under any properly balanced communication pattern, for example uniform load, butterfly permutation, matrix transposition, or nearest-neighbour communication [22]. For more exotic network topologies where complement traffic does not sufficiently characterise the performance of the network under different communication patterns, MPIBench provides hooks that allow custom communication patterns to be used as necessary; see Appendix B.2 for more details.

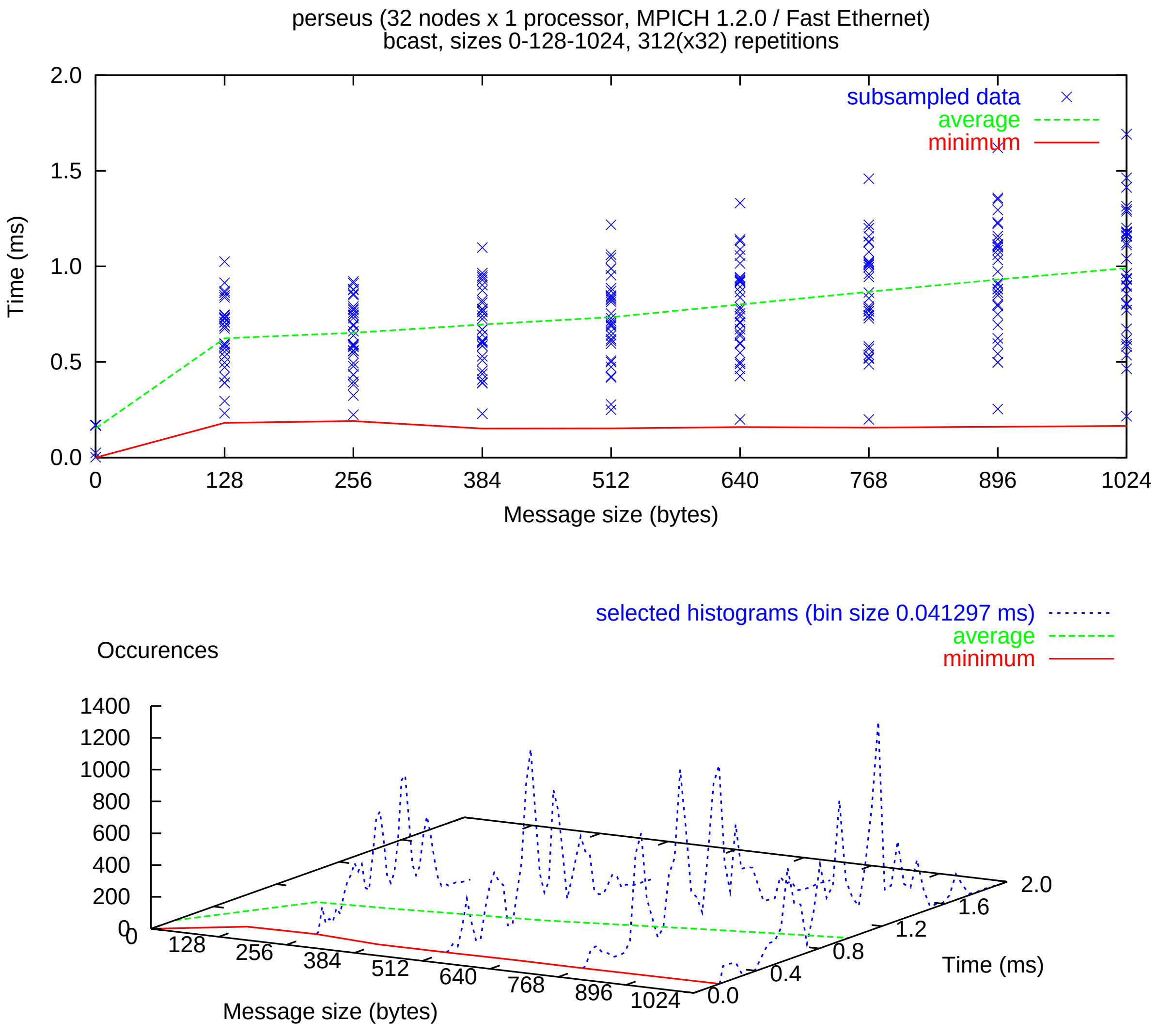
As an example of the flexibility of this approach, any application-specific communication pattern can be approximated by combining a collection of performance measurements from various network routes. For example, consider a cluster of 4-way nodes running an application with a unidirectional ring-shaped communication pattern, as shown in Figure 10. This involves four communication pairs per node, two of which are internal and two of which pair with a process on a remote node. The message-passing performance of this communication pattern could be modelled on a per process basis using measurements of point-to-point communication performance between two local processes and separate performance measurements of communication two processes running on adjacent nodes.

Any concurrent messages in either an intra-node network or the inter-node network will result in contention, which will need to be taken into account.

4.3.4 Generation of Results

By default MPIBench automatically generates results across a user-specified range of parameters, including communication type, message size and number of processes. The output of this process is a set of data files. These files contain raw timing information, as well as post-processed data. Although it is possible to log every single time-stamp to disk, the storage requirements can be significantly reduced by processing the raw data first to generate histograms of the timing data as well as lists of outlying events. MPIBench uses the following technique to separate outliers from the vast majority of the timing data. Firstly, the recorded times for individual messages are sorted into ascending order, and the time that encompasses the n^{th} percentile data is found. It then uses m times this value to separate outliers from systematic data. This heuristic, similar to a near mean filter, is effective because the tail of the most common results dies away very quickly, although the exact point at which it does this is very difficult to determine analytically. MPIBench uses default values for n and m of 99 and 10 respectively, which worked well for all of the machines that were tested in this thesis. A histogram is created for any times falling below the outlier criteria and any outliers are individually logged for further analysis.

To aid the data examination process, `gnuplot` files are also created that automatically generate Postscript plots of the data. An example of the output from a test (borrowed in part from Sections 4.4.2 and 5.2) is shown in Figure 11. The first line of the figure title shows that the test was conducted on a machine named Perseus. It also shows that the test used 32 Fast Ethernet connected nodes, with one process running on each node. For the remainder of this thesis, the number of processes that a test is run on will be denoted nxp where n is the number nodes connected to the network fabric, and p is the number of processes running on an SMP node (which will never be more than the number of physical processors in the node). The next line of the title shows that an `MPI_Bcast` test was performed, on message sizes from 0 bytes to 1024 bytes in steps of 128 bytes, and that measurements for each of these sizes were repeated 312 times at each of the 32 processes; hence producing 312x32=9984 individual measurements at each message size. All of this information is also used to provide a canonical naming scheme for any results that are produced. For example, in this test, the data collected was stored in filenames beginning with `perseus.32x1.bcast.0-128-1024` and ending with `.summary`, `.subsamples`, `.histograms`, `.outliers` and `.gnu`, which respectively contained the minimum and average values at each message size, a per-process subsampling of completion times at each message size, a histogram of completion times at each message size, outliers



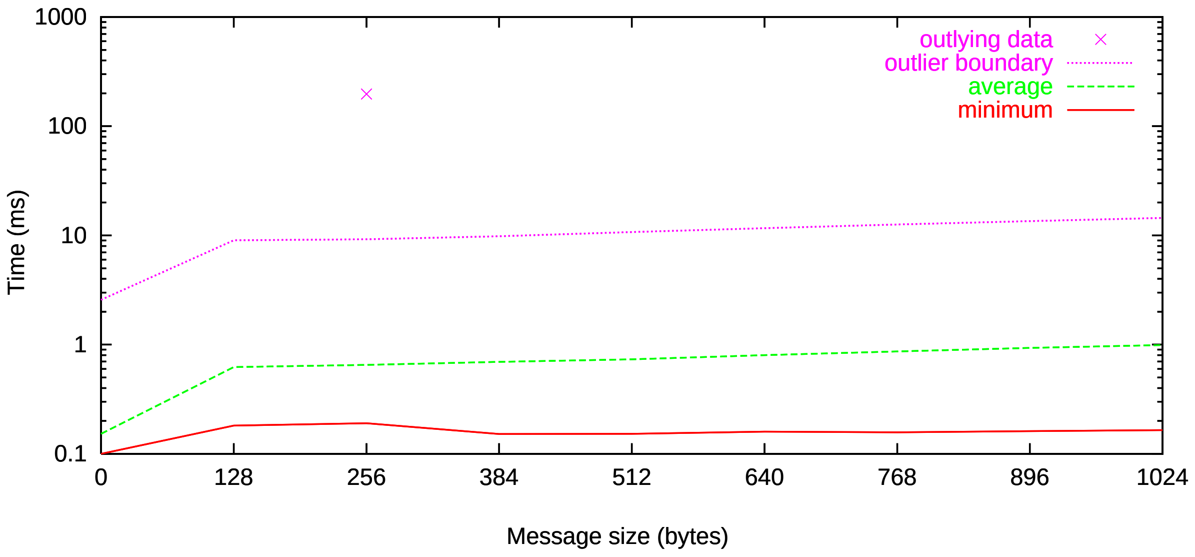


Figure 11: An example which shows the raw output of MPIBench for an `MPI_Bcast` test on a machine named Perseus over 32x1 processes using small message sizes. (Refer to the text for an explanation of each of the graphs).

recorded at each message size and gnuplot instructions.

From these data, three graphs are automatically generated and plotted on one page. Firstly, a simple overview of the data is produced. This includes the minimum time observed at each message size, the average observed for each message size and a subsampling of the raw data, which can be used to get an idea of the spread of the data. A collection of these plots for various tests is useful for observing general trends across large parameter spaces. Secondly, a three-dimensional plot with slices showing the distribution of recorded times for a selection of message sizes is produced. This provides an overview of how performance distributions change across message size, which can be used to manually select message sizes for which to plot histograms that show the performance distribution observed for a specific calls. Finally, there is a plot with a log time scale that shows any outliers observed in relation to the minimum, average and outlier boundary for each message size. This is useful for determining the performance robustness of a communication routine under exceptional circumstances, for example if messages are lost and need to be retransmitted.

4.4 Benchmarking Experiments

This section presents some results that were generated using MPIBench. These results were generated with two purposes in mind. Firstly, they are simply intended to show how MPIBench can provide detailed insight into the performance aspects of various MPI routines. More importantly in relation to this thesis they provide a set of benchmark results that can be used with the PEVPM prediction methodology that was developed in Chapter 3, and which will be tested in Chapter 6.

4.4.1 Machines Used

MPIBench was used to measure MPI performance on three separate parallel computing platforms. These were a Beowulf-type cluster of PCs connected by Fast Ethernet, a Sun Technical Compute Farm connected with Myrinet and a Compaq AlphaServer SC connected with Quadrics' QsNet. These three machines cover a wide range of currently available test platforms, from the low-end Beowulf-type machine to the high-end AlphaServer SC. The configuration of the three machines is described briefly in the following subsections.

Perseus: A Beowulf-type cluster

Perseus is a Beowulf-type cluster at the University of Adelaide. The cluster was purpose built by the Distributed and High Performance Computing Group to service the computational chemistry requirements of the three universities in South Australia [161]. The cluster is comprised of 116 dual processor (mainly 500MHz) Pentium III nodes, each with 256MB of RAM and a large local disk. It has a peak processing speed of 113 Gflop/s. When commissioned in March 2000 it was the largest and fastest cluster in Australia and one of the largest PC clusters in the world. Despite this, the machine is a good example of a low-end parallel computer. This is because the the nodes only have a small amount of memory and are connected by commodity switched 100 Mbit/s Ethernet. Because of these limitations, Perseus only achieved a Linpack benchmark result of just under 20 Gflop/s, which was about half the speed of the slowest machine in the June 2000 Top 500 supercomputer list [104]. The network is built around five 24 port Intel 510T switches with stackable matrix cards that provide 2.1 Gbit/s of backplane bandwidth per switch.

The MPIBench tests on Perseus were conducted under a software environment that comprised of a RedHat Linux 6.2 base using glibc-2.1.3-15 and a custom-built kernel. The custom-built kernel was required in order to provide the processor binding functionality that MPIBench needs to obtain accurate timing measurements. It was based on a vanilla 2.2.12 SMP kernel, with the Pset [179] kernel patch installed. The MPI implementation that was used was MPICH 1.2.0 [21, 151], which was the most recently available stable version at the time.

For the purposes of these experiments, MPIBench was run in a dedicated fashion on Perseus, i.e. no other user programs were allowed to run. Although the cluster normally operates using NFS and NIS as well as other services[1] these were all disabled to reduce the impact of sporadic system interference. When a minimal OS installation had been achieved, the only processes left running were the kernel and `inetd` services for logging into the machine to begin the tests and collect the results upon completion.

Orion: A commercial HPC cluster

Orion is a Sun Technical Compute Farm [242] at the University of Adelaide. It consists of a cluster of 40 Sun E420R SMP servers, each with four 450MHz UltraSparc II processors and 4GB of RAM. The nodes are connected by both 100 Mbit/s Ethernet and a Myrinet [42] network, which provides 1.28 Gbit/s of full duplex bandwidth and a significantly lower latency than Fast Ethernet. The Myrinet hardware provides an inherently connection-less data transfer mechanism, on top of which is built a GM protocol layer which provides

[1]This cluster routinely runs `xntpd`, `NFS`, `portmapper`, `sendmail`, `cron`, `syslog`, `NIS`, and `pump` on the internal nodes. This is not an uncommon practice on commodity clusters despite the overhead they can cause for parallel program execution.

reliable and ordered end-to-end packet delivery to user-space processes. Also in contrast to the flat topology of the Fast Ethernet network in Perseus, the Myrinet network in Orion is connected in a fat tree configuration. Orion has a peak speed of 144 Gflop/s. It was the fastest supercomputer in Australia when it was installed in June 2000 [103]. Despite only having a slight peak speed advantage over Perseus, its faster Myrinet interconnection network, and larger cache and main memory distinguish it from the lower-end Beowulf-type machine. Orion ranked number 188 in the November 2000 list of the Top 500 supercomputers in the world [104], with a Linpack benchmark result of 110 Gflop/s.

At the time of the tests, Orion was running SunOS 5.8 and the MPI implementation that was used was Sun HPC ClusterTools 4.0 [241]. In particular, ClusterTools was configured to use the low-overhead Myrinet GM transport layer [252] (version 1.4) for MPI programs, with all default settings enabled. As with Perseus, the experiments were conducted with dedicated access to the machine, thus avoiding interference from any other user programs.

APAC NF: A national supercomputing facility

At the high-end of machines that were benchmarked was the Australian Partnership for Advanced Computing (APAC) [26] AlphaServer SC [75]. At the time of the tests (March 2002), it was the fastest supercomputer in Australia. The APAC NF has 120 Compaq ES45's, each with four 1 GHz Alpha EV68 processors, 4 Gbytes of memory and 72 Gbytes of disk, all connected by QsNet [278]. In the same way that Myrinet is superior to Fast Ethernet, the QsNet is more powerful than Myrinet. Like Myrinet, the QsNet network was connected in a fat tree configuration. After link protocol requirements of 58 bytes per packet, QsNet provides 2.7 Gbit/s (peak) of full duplex bandwidth, and a latency of about 35ns. QsNet is based on a synchronous delivery protocol, where the sender of a packet always waits for a subsequent end of packet token from the receiver before it completes. The link protocol detects any lost messages or network faults and organises retransmission and re-routing as necessary. In addition to this, the communications processor on each QsNet network interface can do a substantial amount of the work required by higher level protocols such as MPI without the intervention of a node's main CPU. Overall, QsNet coupled with the Compaq MPI implementation on the AlphaServer SC provides zero copy, user level message delivery that bypasses the operating system. Processes initiating either MPI point-to-point or MPI collective communication simply block and wait for the end of packet token to be delivered.

Because the APAC NF is a shared resource, it was impossible to get completely dedicated access to the machine to perform MPIBench tests. However, the Portable Batch System (PBS) [164] that the APAC NF uses to control how jobs are scheduled made it possible to gain dedicated access to a partition of consecutive nodes on the machine. Although this was used to provide MPIBench with dedicated access to compute nodes within such a partition, communication between those nodes remained subject to interference from communication among nodes in other partitions. Because of the fat-tree topology of the QsNet communication network, the bandwidth available to point-to-point routines would never diminish, but the average latency and variance in completion time of point-to-point messages could, potentially, be slightly increased (because of small serialisation delays in the QsNet switches, which will be explained in Section 5.2). Fortunately the already small performance effects that could be caused by this interference were abated further in practice because, although other user programs were running on the rest of the machine during MPIBench tests, relatively few of those were parallel programs that spanned multiple nodes, and hence the load on the communication network was minimal.

Another reason for insisting on a partition of consecutive nodes was because this provided a collection of nodes connected by physically contiguous QsNet. Under such a circumstance, the MPI implementation is able to call upon hardware supported broadcast and barrier operations for drastically increased performance. With the exception of one noted software-based `MPI_Broadcast` test (for comparison purposes), all other tests on the APAC NF were run across physically contiguous nodes. Primarily, this was so that the performance of the more interesting hardware-assisted MPI calls could be evaluated, but also for the added benefit of reducing the run-time of MPIBench tests, many of which make significant use of `MPI_Barrier`. (A more in depth study of the quantitative performance differences between hardware and software-based collective communication using QsNet was done by Petrini *et al.* [276]).

4.4.2 Tests Performed

MPIBench was used to measure the performance of a range of MPI communication primitives across a variety of message sizes and number of processes. (Instructions on how to run MPIBench can be found in Appendix B.1). Because of the portability of MPI, as well as access to (almost) the same number of processors on each machine, it was possible to run an (almost) identical set of tests (enumerated in Table 1) across the three supercomputers. This was advantageous because it allows direct comparisons between the performance of MPI operations on each of the machines. In particular, because MPI performance is usually affected far more by network performance than the processing capabilities of individual nodes, these results are loosely applicable to other parallel machines utilising the same network infrastructure.

Because of the sheer amount of data that was generated by these tests, only a careful selection of results are presented. The results that have been included were chosen in order to facilitate a discussion of the performance characteristics of various MPI operations on

Table 1: A list of the tests that were run; the data generated from these tests are discussed in the remainder of this chapter and the following chapter. (Note: `MPI_Scatter`, `MPI_Gather` and `MPI_Alltoall` tests used the `total` option described in Section 4.3.3).

Machine	Number of procs	Test types	Sizes	Repetitions

Perseus	1x2
	2x1 2x2
	4x1 4x2
	8x1 8x2
	16x1 16x2
	32x1 32x2
	64x1 64x2
		`MPI_Isend`
		`MPI_Sendrecv`
		`MPI_Barrier`
		`MPI_Bcast`
		`MPI_Scatter`
		`MPI_Gather`
		`MPI_Alltoall`
			0-8-128	10,000

			0-128-1024	10,000
			0-4096-65536	10,000

Orion	1x2 1x4
	2x1 2x2 2x4
	4x1 4x2 4x4
	8x1 8x2 8x4
	16x1 16x2 16x4
	32x1 32x2 32x4
		`MPI_Isend`
		`MPI_Sendrecv`
		`MPI_Barrier`
		`MPI_Bcast`
		`MPI_Scatter`
		`MPI_Gather`
		`MPI_Alltoall`
			0-8-128	10,000

			0-128-1024	10,000
			0-4096-65536	1,000

APAC NF	1x2 1x4
	2x1 2x2 2x4
	4x1 4x2 4x4
	8x1 8x2 8x4
	16x1 16x2 16x4
	32x1 32x2 32x4
		`MPI_Isend`
		`MPI_Sendrecv`
		`MPI_Barrier`
		`MPI_Bcast`
		`MPI_Scatter`
		`MPI_Gather`
		`MPI_Alltoall`
			0-4-128	100,000

			0-32-1024	100,000
			0-4096-262144	100,000

each of the three machines, with particular attention paid to:

- The scalability of MPI operations on each machine across a range of messages sizes.
- The scalability of MPI operations on each machine across a range of processors, including the effect of utilising clusters of SMP nodes.
- The performance stability of MPI operations, by examining performance distributions and outliers observed for each test.
- A comparison of the performance of MPI operations between machines, i.e. the differences between Fast Ethernet, Myrinet and QsNet (when combined with a particular MPI implementation on each machine).
- Performance models of group operations built from point-to-point operations.

As explained in Section 3.4.2, point-to-point operations usually constitute the most elemental message-passing operation. From just a handful of these operations, the complete range of message-passing operations can (and usually are) constructed. Therefore a thorough understanding of the performance of these point-to-point operations lends itself to better understanding of the performance of more complex group operations. Section 3.4.2 also explained the differences between the range of point-to-point operations that the MPI specification defines. In summary of that discussion, the main differences relate to buffering and program correctness; as long as the different varieties of point-to-point operation are used appropriately, their performance is almost identical. Since it is only the performance of these operations that is the concern of this dissertation, only the (asynchronous) `MPI_Isend` operation is examined (as a representative example of the other point-to-point operations), with one exception. The exception is the `MPI_Sendrecv` operation, because it is conceptually different – it represents the combination of two simultaneous point-to-point operations, traversing the network between a pair of processes in opposite directions. Since the performance of this operation may potentially differ vastly between networks that use (physical) simplex links and networks that use (physical) duplex links, this operation is considered separately. Furthermore, in addition to the standard (simplex) `MPI_Isend` routine, a duplex `MPI_Isend` routine was benchmarked for comparison with the functionally equivalent `MPI_Sendrecv` routine.

4.5 Results for `MPI_Isend`

4.5.1 Inter-node, end-to-end completion time

Figure 12 shows the average times that were recorded for running MPIBench on Perseus over a range of small message sizes, for various numbers of communicating processes (indicated in the key by the number of nodes x the number of processors per node). It also shows another line, labelled *min*, which indicates the minimum time that was observed between one pair of communicating processes. Figure 13 shows the same information for larger message sizes. To validate the legitimacy of these results they were compared with results obtained from an equivalent test using SKaMPI. There was a very good correlation: although for small messages SKaMPI recorded times 15% higher than MPIBench, then as the message size increased the measured times quickly converged to within 2% of each other. The most likely reason for the discrepancy at small message sizes is that the clock used by SKaMPI did not have a sufficiently fine granularity to accurately measure the short transmission time of small messages.

First, consider the line marked 2x1 in comparison with the line labelled *min*, especially for larger message sizes. This represents a simple-ping pong test, which is commonly

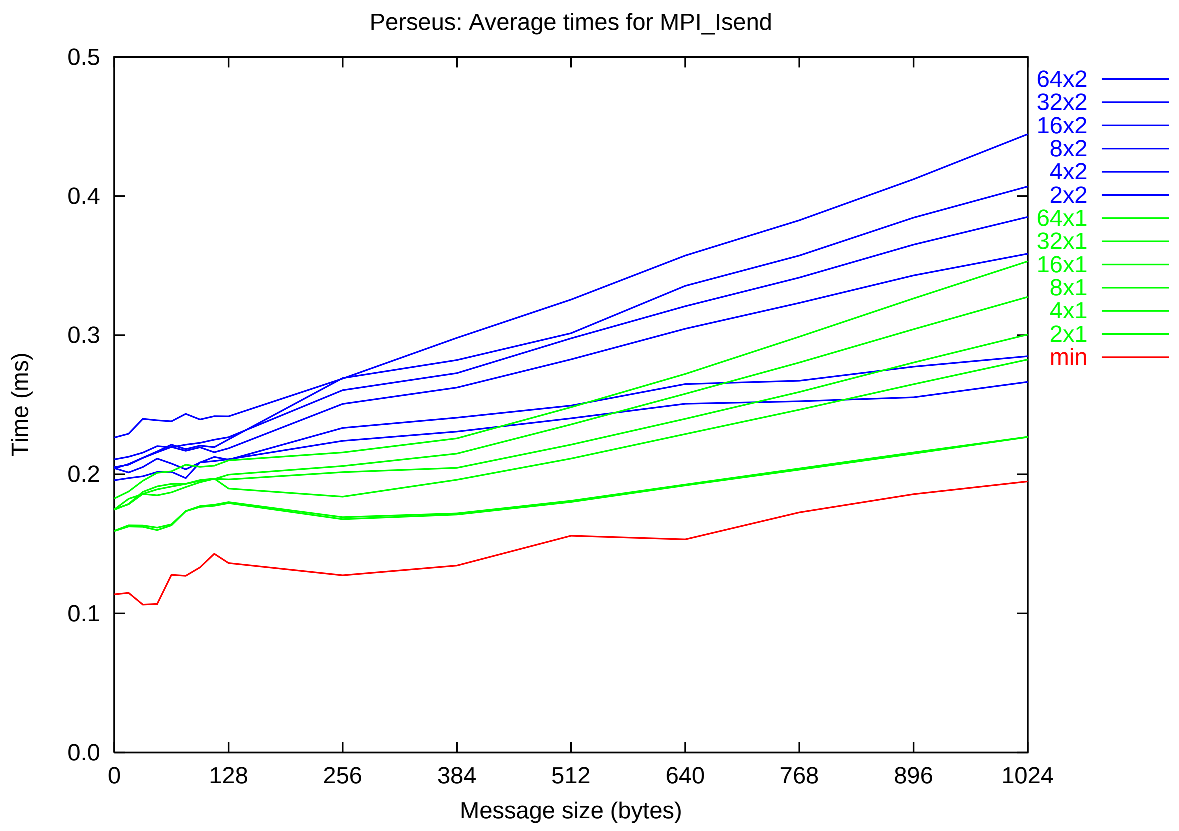


Figure 12: Average times for `MPI_Isend` using small message sizes with various numbers of communicating processes on Perseus.

used as a model of message-passing time. The similarity between minimum times and average times for this case highlights the extremely small timing variations that occur when network congestion is eliminated. When this is the case, message-passing time T can indeed be closely modelled by the common approximation $T = l + b/W$ where l is the link latency in seconds, b is the size of the message in bytes and W is the effective bandwidth of the link in bytes per second. The results in Figure 13 indicate an effective bandwidth of about 91 Mbit/s for 64 Kbyte message sizes, although the decreasing slope indicates that bandwidths closer to the theoretical limit of 95.7 Mbit/s will be obtained using larger messages[2]. Closer inspection of Figure 13 also reveals two distinct segments of the curve, with a knee occurring at 12 Kbytes. Although in this case there is only a slight difference between the scalability of performance in these segments, in some situations discontinuities can be far more pronounced (such as in the 64x2 case). What these breakpoints indicate are message sizes which delimit an MPI implementation's use of different protocols for sending short versus long messages, as well as the effect of packet size on the network infrastructure. For example, the results marked 64x1 and 64x2 in

[2]100 Mbit/s cannot be achieved for user data, because Ethernet has 64 bytes of overhead for every 1500 byte frame. In addition, each message will also have a negligible amount of MPICH overhead, which will reduce the bandwidth available to user processes to slightly below 95.7 Mbit/s.

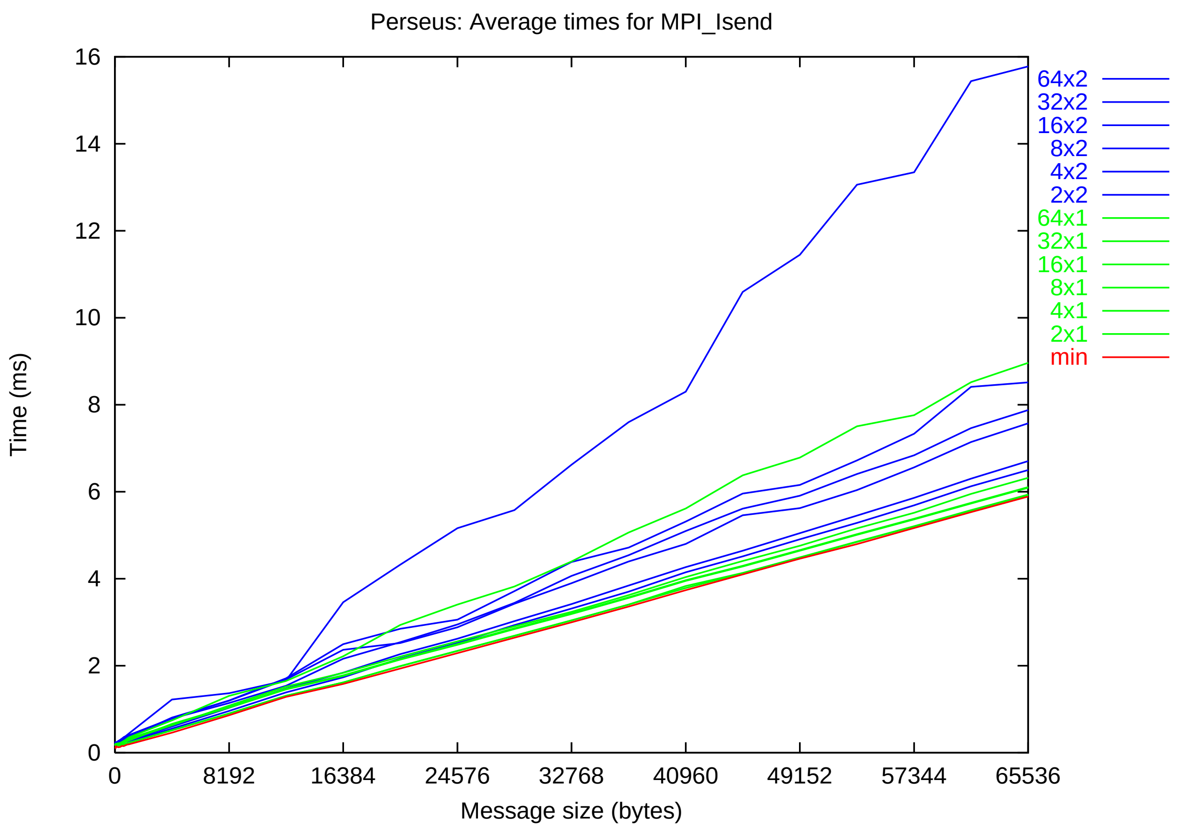


Figure 13: Average times for `MPI_Isend` using large message sizes with various numbers of communicating processes on Perseus.

Figure 13 indicate that some new factor clearly comes into effect for messages larger than 12 Kbytes. In this case, the cause of the divergence turned out to be saturation of the inter-switch network links, which resulted in greatly increased contention.

The results for Perseus shown in Figures 12 and 13 (and all future figures similar in nature) are colour-coded to indicate the effect of running multiple processes on a multiprocessor node. Each coloured set of data shows the results obtained for an increasing number of total nodes. Lines of the same colour can be disambiguated using the legend, which is sorted to match the the values of each line at the right-hand edge of the plot. The general effect of increasing the number of processes per node or the total number of nodes is to increase the level of contention. More precisely, increasing the number of processes per node increases the contention for the one network interface in a node as well as in the backplane network, while increasing the number of nodes only increases contention in the backplane network.

Consider the effect of contention in the backplane network, i.e. compare any data set of the same colour in either Figure 12 or 13. For small messages, the results become increasingly dispersed with increased numbers of communicating processes, which shows the susceptibility of the Fast Ethernet network in Perseus to contention. For example,

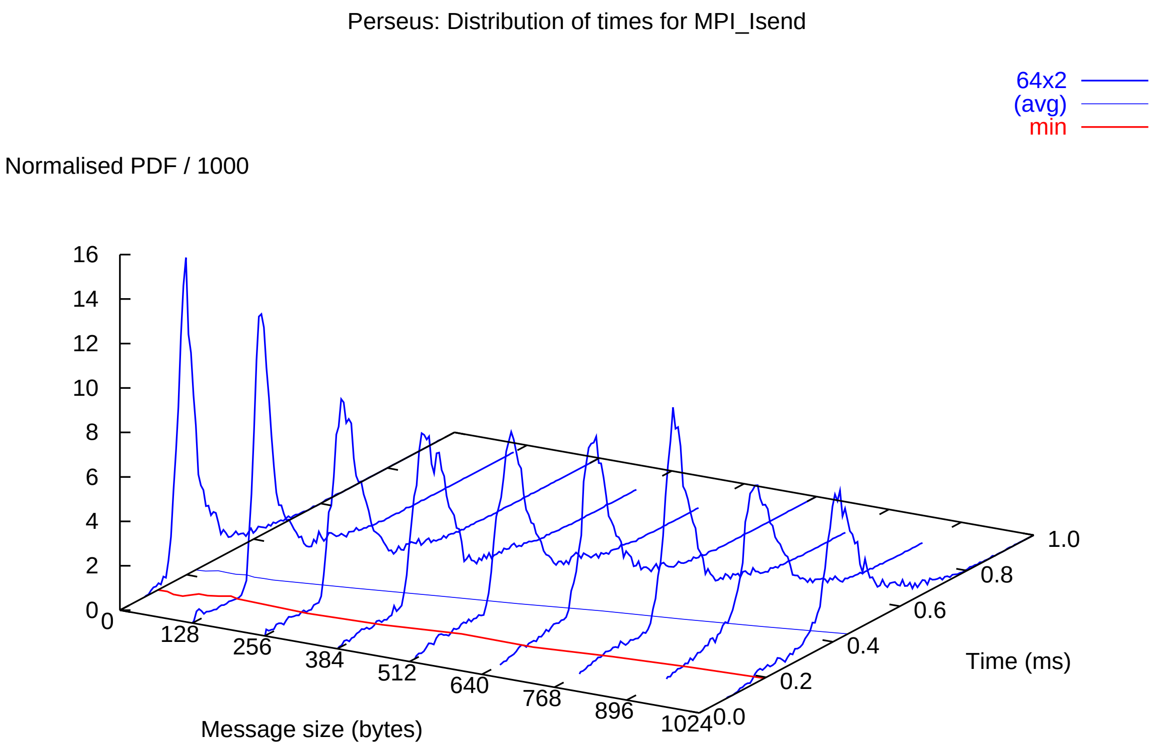


Figure 14: Sampled performance profiles for `MPI_Isend` using small message sizes with 64x2 processes (high contention for the local network interface and network backplane) on Perseus.

Figure 12 shows that, on average, transmission of a 1 Kbyte message takes 70% longer when 64x1 processes are communicating than when 2x1 processes are communicating. Clearly, modelling communication time in situations such as this may be very inaccurate if only a single point value is used. However, as the message size increases the effect of the number of communicating processes on delays in the backplane network becomes less noticeable (at least, until saturation of the network occurs leading to severe performance degradation, as will be discussed shortly). Although the magnitude of the delays for different numbers of processes remains about the same, the percentage difference between them rapidly shrinks because of the base time to transmit a message at a certain bandwidth. This is an important result which governs some approximations that may be made when modelling the performance of parallel programs. For accurate modelling, it shows that the effects of contention cannot be ignored for small messages; and that the effects of contention may only be ignored for large messages provided that saturation has not occurred in the backplane network.

Although a rough idea of the effect of contention on performance can be gauged from Figures 12 and 13, the effect is far more clearly demonstrated in Figure 14, which shows selected distributions of times that were recorded for 64x2 communicating processes on messages between 0 and 1024 bytes in size. Although similar effects can be observed when any number of communicating processors are present, this large number was chosen because the associated high level of contention accentuates these effects, and thus they are more readily apparent. The distributions have been normalised so that the area under each distribution is equal to one, resulting in a normalised probability density function (PDF). This allows the scale of the various distributions to be compared. Also plotted in Figure 14 are the minimum times from Figure 12 (i.e. the minimum times observed between two communicating processes) and the average times of the distribution at each size. Now it can be clearly seen how the minimum and average times relate to the performance distributions that they approximate: the distributions have a relatively smooth rise from a definite minimum time, through a peak which occurs very close to the average time and drops off quickly to some maximum time.

This is a slight idealisation that requires qualification. Firstly, Figure 14 shows that a very small number of messages appear to take less time than the values that denote the minimum time for message-passing between two processes at any particular message size (which are plotted in the series labelled *min*). These are erroneous measurements that occur because the quality of the synchronisation method, despite its usually high accuracy, cannot be strictly guaranteed. To briefly summarise the earlier explanation in Section 4.3.2: synchronisation quality between any process and the reference process is not bounded by an absolute time, but by a heuristic number of iterations where quality improves monotonically. More instructively, as the number of processes involved increases, the chance of any particular process being poorly synchronised and hence producing erroneous measurements increases proportionately. With the exception of this caveat, however, the results are reliable: poor synchronisation merely results in an observable noise floor that is insignificant in comparison with the majority of the recorded data. This is because, unlike the likelihood of erroneous measurement, the level of the noise floor grows inversely in proportion to the number of communicating processes. Although the chance of a few processes being poorly synchronised increases proportionally to the total number of processes in a test, the remaining processes – which are well synchronised – increase proportionally much faster. Therefore the ratio of good quality to poor quality measurements increases (i.e. the noise floor decreases) in proportion to the number of processes involved in the test.

The second qualification is actually more of an amplification. Unlike the minimum time, which is bounded by the performance of a contention-free message, the maximum time is theoretically unbounded. In practice, however, the tail of the PDF drops off so quickly that it soon becomes impossible to distinguish real data from the noise floor. There is also one caveat to the smooth decline of the PDF tail. Protocol timeouts for lost

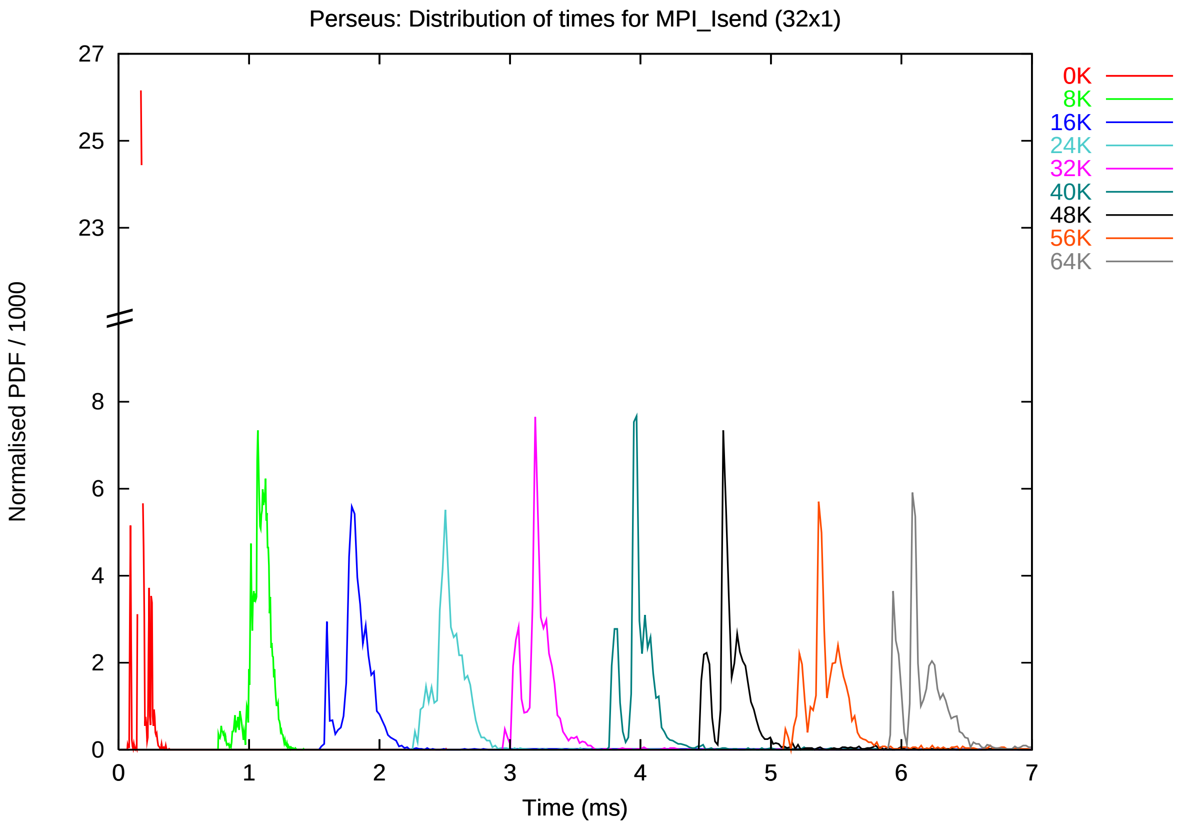


Figure 15: Sampled performance profiles for `MPI_Isend` using large message sizes with 32x1 processes (moderate network contention) on Perseus.

messages cause message retransmission. This produces similarly shaped outlier distributions (but of very low probability) at much longer time values. A closer investigation of the types of curves that fit typical PDFs that are observed can be found in Section 4.7 and a more detailed discussion of outlier distributions can be found in Section 4.8.

Figures 15 and 16 show how the distributions observed for `MPI_Isend` on Perseus vary across larger message sizes, in the case of 32x1 and 64x1 communicating processes respectively. An interesting feature of the performance distributions in these figures, and especially obvious in the 32x1 case, is the small peak before the main peak in each profile. Remember that the switched Fast Ethernet network topology in Perseus is not completely flat, but comprises of five Intel 510T 24 port switches connected by a stackable matrix card. This means that where more than 24 nodes are involved in communication (as in the 32x1 and 64x1 cases), messages between some nodes must necessarily traverse the inter-switch link as part of their route. This incurs a small amount of extra message-passing time due to the latency of the inter-switch connections. Therefore, the first peak is largely representative of the performance of message-passing between nodes residing on the same switch, while the main peak is largely representative of the performance of message-passing between nodes on different switches.

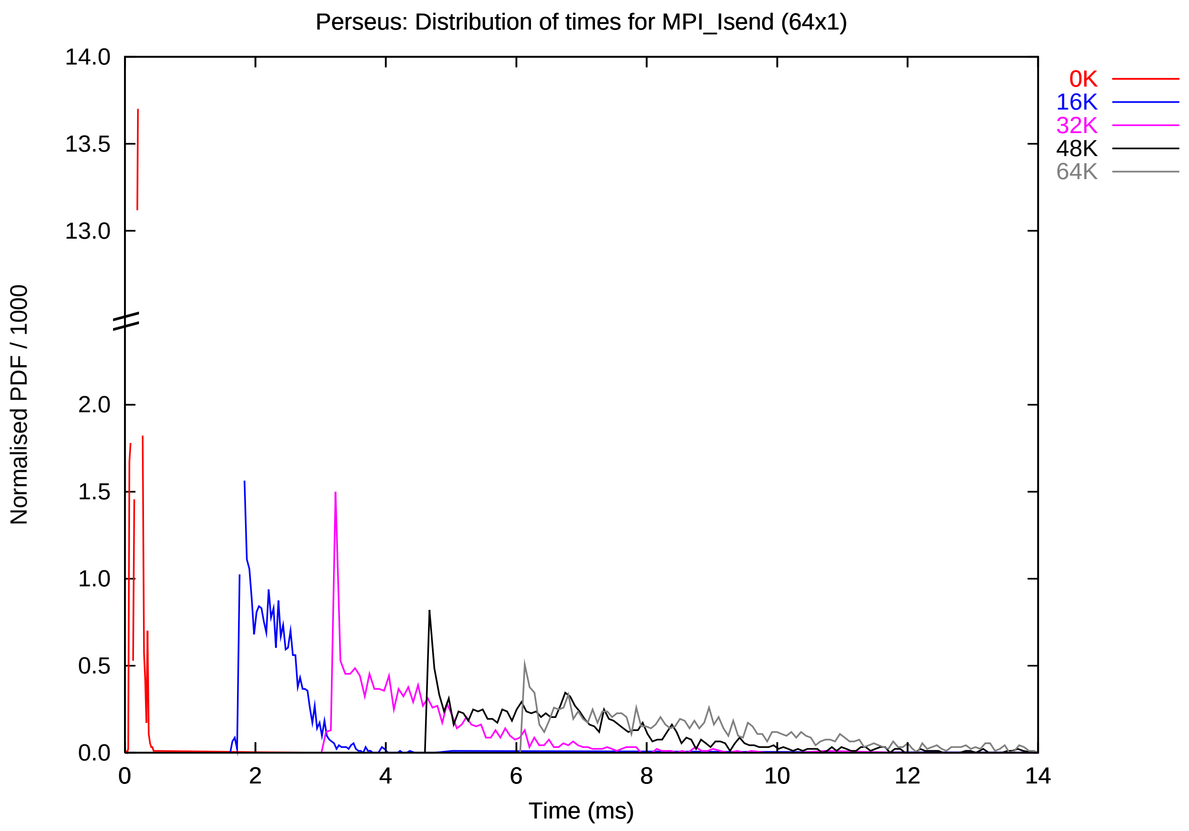


Figure 16: Sampled performance profiles for `MPI_Isend` using large message sizes with 64x1 communicating processes (high network contention) on Perseus.

The most significant information contained in Figures 15 and 16, however, is in their disparity. The obvious increase in average message-passing times on more than 32x1 processes of Perseus, shown in Figure 13, indicates that at most 32x1 processes can be used before performance begins to degrade markedly for larger messages. This is more clearly seen in the distributions of Figure 15, where larger messages perform as well as smaller messages (excluding the additional time required to send more information), in contrast to the distributions of Figure 16, where performance degrades significantly for larger message sizes. This degradation is readily apparent in the extreme elongation of the PDF tails for larger messages, and is due to massive contention. Closer inspection reveals that the degradation begins to appear when the messages reach about 16 Kbytes in size. For the 64x1 process case, three Intel 510T 24 port switches were spanned: two using 24 ports and one using 16. The onset of performance degradation began when a total of approximately 24x84.25 Mbit/s (since 81 Mbit/s is achieved between two processes for 16 Kbyte messages, plus 3.25 Mbit/s of Ethernet framing overhead) i.e. 2.02 Gbit/s was being delivered between the two fully utilised switches. Remembering that the stackable matrix cards connecting these switches provide 2.1 Gbit/s of backplane bandwidth each, it seems that the backplane limit of 2.1 Gbit/s per switch had been reached and the

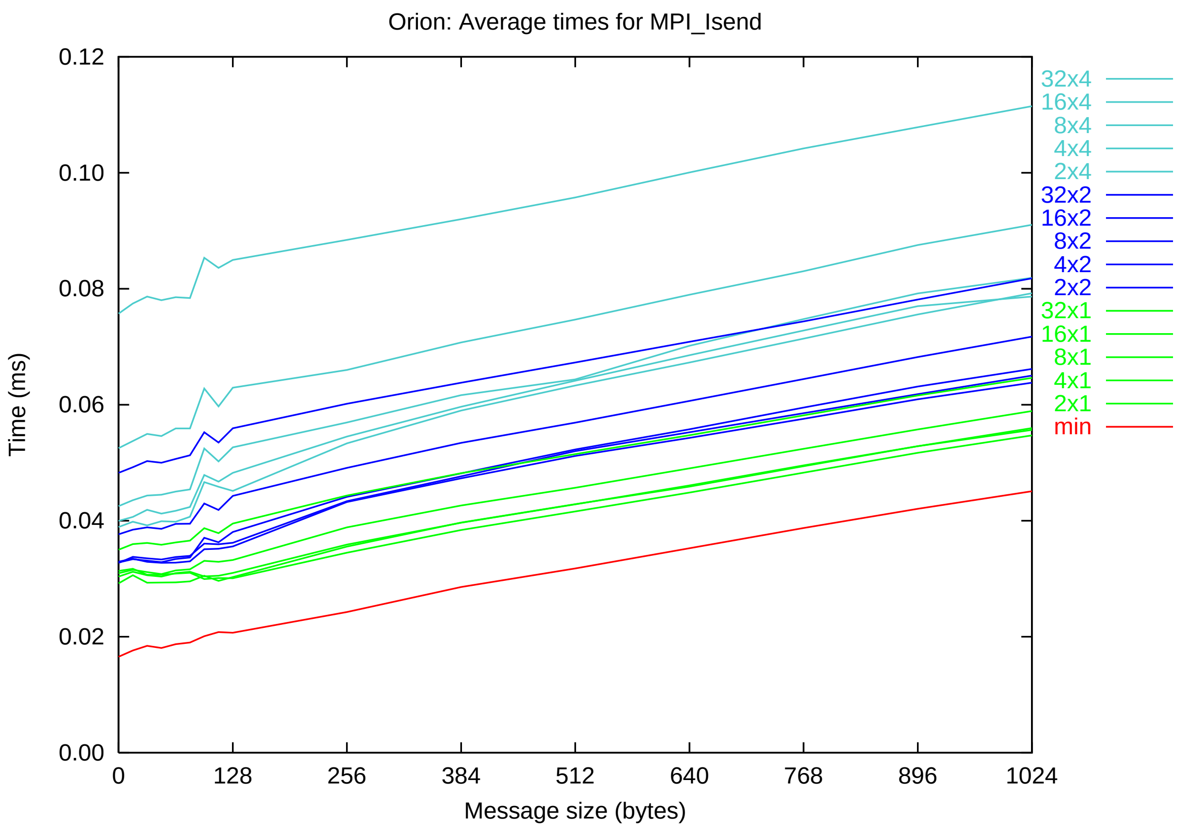


Figure 17: Average times for `MPI_Isend` using small message sizes with various numbers of communicating processes on Orion.

ensuing inter-switch saturation resulted in greatly reduced performance.

The same series of `MPI_Isend` tests that were run on Perseus using its Fast Ethernet network were also conducted on Orion using its Myrinet network. This second set of results is summarised in Figures 17-20. The most obvious differences between the performance of the two networks are that the Myrinet network in Orion has only about one-sixth of the latency of Fast Ethernet in Perseus for small messages (i.e. 20-80 μs depending on the number of nodes and processes per node) and slightly more than 4 times the bandwidth of Fast Ethernet for the largest message sizes used (i.e. 419 Mbit/s) in the case of one communicating process per node. However, in the cases where there were four communicating processes per node, some overhead was amortised and a (peak) effective bandwidth of 840 Mbit/s was observed. Sadly, this is still appreciably lower than the 1.28 Gbit/s that Orion's Myrinet network is listed to be capable of. Correspondence with Sun and Myrinet confirmed that this poor performance is characteristic of Myrinet using the GM 1.4 driver libraries on Sun hardware because of an under-performing DMA implementation. Unfortunately, an updated version of the GM driver library (v1.6) only became available after the majority of the work in this and the following two chapters had been completed and analysed.

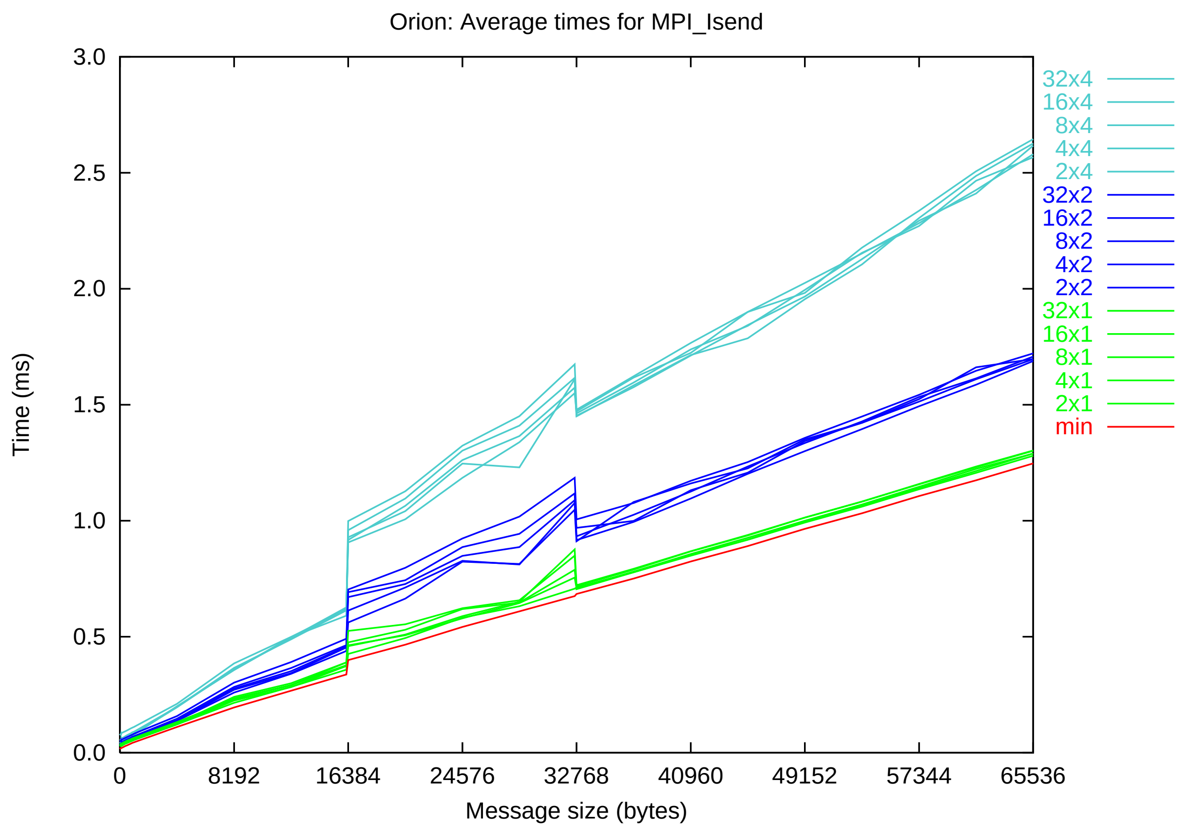


Figure 18: Average times for `MPI_Isend` using large message sizes with various numbers of communicating processes on Orion.

Returning to the subject of contention, consider the effect of contention for access to a unique network interface in a multiprocessor node. Coupled with message transmission overhead, this leads to some interesting non-linearities in performance for various numbers of processes (some if which have been recently noted but not explained by Capello *et al.* [61]). Figure 17 shows that 32 4-way nodes can each concurrently send four 128 byte messages in just over twice the time that it would take to send one 128 byte message. Although each Myrinet interface provides support for up to eight concurrent processes to multiplex their communication into the one physical connection at each Myrinet interface, this is based on a fair queueing system that divides up messages from individual processes into 4 Kbyte packets. This transmission serialisation means that the cost of startup and latency overheads have the potential to be partially amortised when more than one process is communicating per node, and therefore the network hardware is being utilised more efficiently. In bandwidth-limited programs this effect will lead to improved performance. However, access contention to the sole network interface also causes increased variance in message delivery times, as shown in Figure 19. This is likely to lead to degraded performance in latency-limited programs. To understand the reason for the increased variance, notice that the minimum message delivery times in these distributions remain

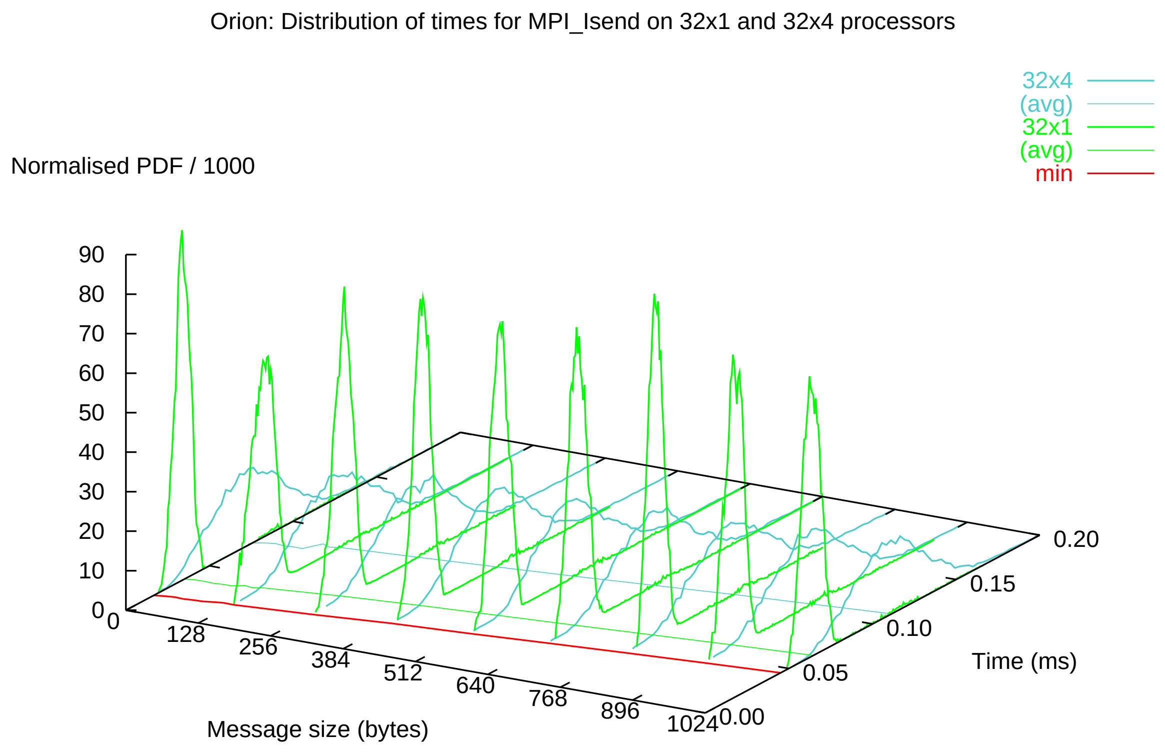


Figure 19: Sampled performance profiles for `MPI_Isend` using small message sizes with 32x1 processes (low contention for the local network interface) and 32x4 processes (high contention for the local network interface) on Orion.

the same regardless of the number of processes contending for the network interface. These minimum times represent the messages that were always granted immediate access to shared resources. In contrast, the great majority of times represent messages that were delayed access to shared resources due to contention; those that were delayed the most are represented at the far right of the distributions. As contention for the local network interface is increased, these delays become more common. Hence the normalised PDFs for message transmission time broaden (and reduce in amplitude appropriately) as the number of processes per node are increased. For small messages, where this effect is most significant, the normalised PDFs for message transmission broaden by the number of processes per node n and reduce in amplitude to $1/n$ of their original amplitude (shown in Figure 19), leaving throughput unchanged, although with increased variance. For larger messages the distributions broaden by a factor less than n, depending on the extent to which message-passing startup times and packetisation losses are amortised, and reduce in amplitude appropriately.

Another interesting aspect of the performance of `MPI_Isend` on Orion is obvious in Figures 18 and 20. This is the degraded performance for messages between 16 Kbytes and

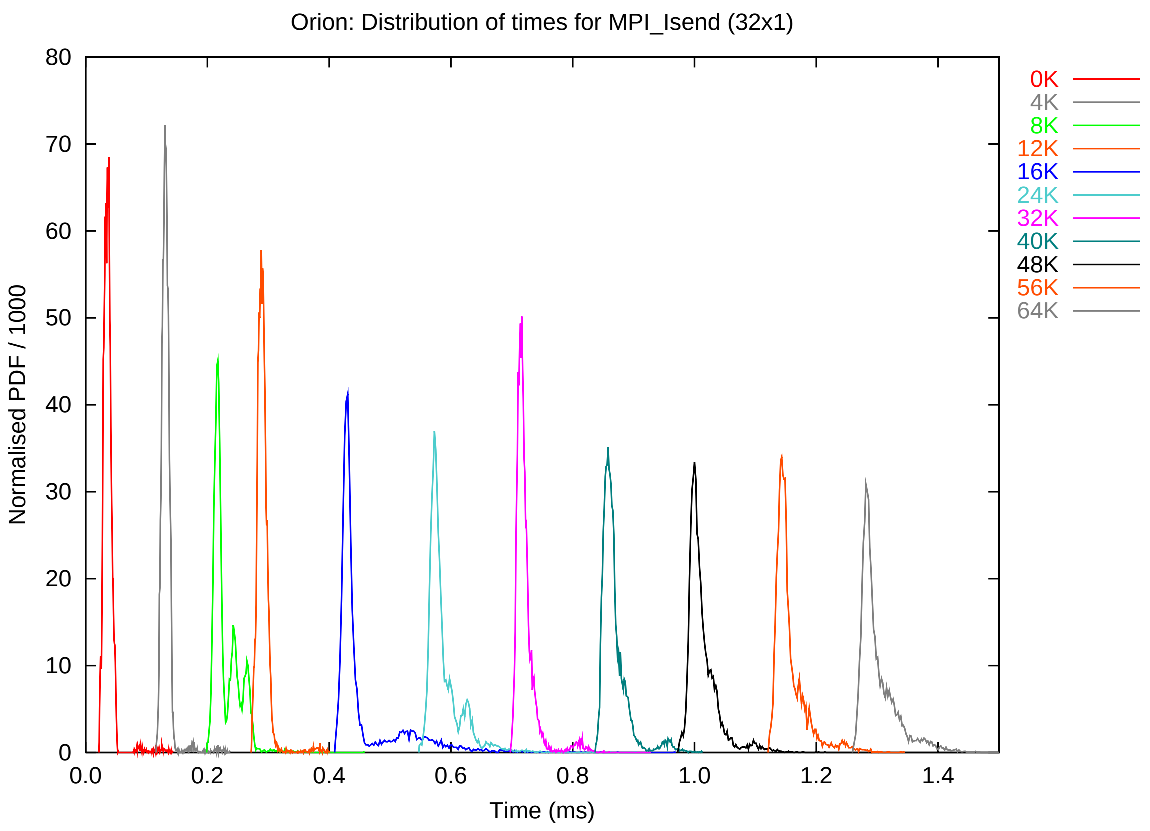


Figure 20: Sampled performance profiles for `MPI_Isend` using large message sizes with 32x1 communicating processes (moderate network contention) on Orion.

32 Kbytes. In Figure 18 this is clearly indicated by a distinct rise in the time taken to send messages in that size range in comparison with the trend for either smaller or larger messages. Discontinuities such as these in performance graphs are indicative of either a change in the message-passing protocol of an MPI implementation, or of the effect of packetisation on messages by the operating system or interconnect network. In this case, documentation for the Myrinet Protocol Module in ClusterTools 4.0 [251] reveals that it is due to a change in message-passing protocol. For messages up to 16 Kbytes minus 96 or 112 book-keeping bytes for 32-bit or 64-bit applications respectively, GM uses an eager message delivery protocol where messages are always sent immediately and buffered by the MPI implementation at the receiver. Messages larger than this but less than 32 Kbytes minus 8 bytes are sent using a rendezvous protocol, where the sender does not transmit any data until the receiver acknowledges that it has buffer space to receive it. Finally, messages larger again are transmitted using a multiphase-rendezvous protocol, where data is split into chunks, the first of which is sent using a rendezvous protocol while the rest follow in a pipelined manner. It is intriguing that messages sent using the rendezvous protocol perform so poorly, and in fact worse than messages sent with the multiphase-rendezvous protocol, which should theoretically take longer. However, the unwarranted

poor performance of message-passing using the rendezvous protocol using ClusterTools 4.0 and GM 1.4 disappeared in brief tests with GM 1.6, so it reasonable to dismiss the measured poor performance as a bug in the GM 1.4 implementation on Orion.

The final set of `MPI_Isend` results were obtained by running MPIBench on the APAC NF. A subset of these results (corresponding to those covered for Perseus and Orion) are plotted in Figures 21-24. Broadly speaking, these figures confirm the results that were expected, namely that the QsNet communication network in the APAC NF behaves in a very similar manner to the Myrinet network in Orion, but with lower latency and higher bandwidth. Looking at the results in more detail, Figure 21 reveals that the latency of `MPI_Isend` over the QsNet in the APAC NF was about 3.5 μs in the very best case. For small messages the average message-passing time of the QsNet in the APAC NF was 6-8 times faster than the Myrinet in Orion and 30-45 times faster than the Fast Ethernet in Perseus, depending on the number of processes per node. For large messages, a bandwidth of 262 MByte/s (2.1 Gbit/s) was achieved, which was appreciably lower than the 340 MByte/s of peak bandwidth promised by the hardware specifications. Note that this is still 2.5-5 times the (effective) bandwidth provided by the Myrinet network in Orion (for messages up to 64 KByte, depending on the number of simultaneous communicating processes per node) and 20 times the bandwidth of the Fast Ethernet in Perseus.

Contrasting the width of the PDFs relative to the average value of the PDFs in Figure 23 with those for Perseus in Figure 14 and Orion in Figure 19 shows that the variation in message-passing time (roughly) decreases in proportion to the average decrease in message-passing time of each of the technologies. One curiosity of the performance of the QsNet is the distinct jump in message-passing time at a message size of 288 bytes. The reason for this particular jump is due to protocol optimisations for very small messages. Messages up to 288 bytes are inlined in the underlying Elan message envelope, which makes them available to the receiver as soon as the message arrives. In contrast, messages larger than 288 bytes are delivered synchronously by the underlying hardware, and require an acknowledgement to be sent to the sender before the receiver may process a messages contents. There is also a barely perceptible change in the slope of the curve for messages larger than 65536 bytes. Up until this size, messages are buffered directly by the communication system at remote processes, whereas after this size the receiver must acknowledge that it has buffers ready to receive the data before the sender transmits the data.

Worthy of note is a slight difference between the the tests on Orion and Perseus compared with those on the APAC NF. Because of a large allocation of compute-time on the APAC NF it was possible to run tests on larger message sizes (up to 256 KBytes instead of 64 KBytes for the large message tests) and also in smaller increments along the way (32 bytes instead of 128 bytes for the small message tests). On top of this, a much

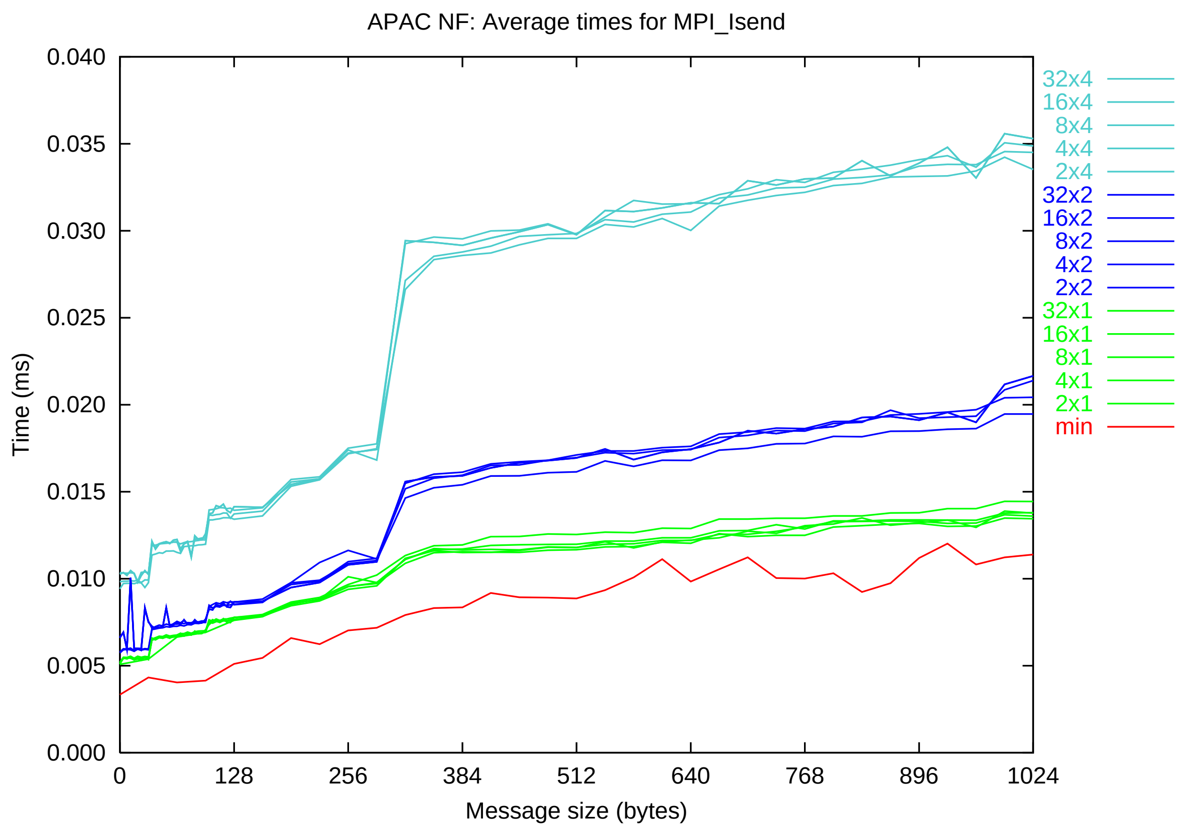


Figure 21: Average times for `MPI_Isend` using small message sizes with various numbers of communicating processes on the APAC NF.

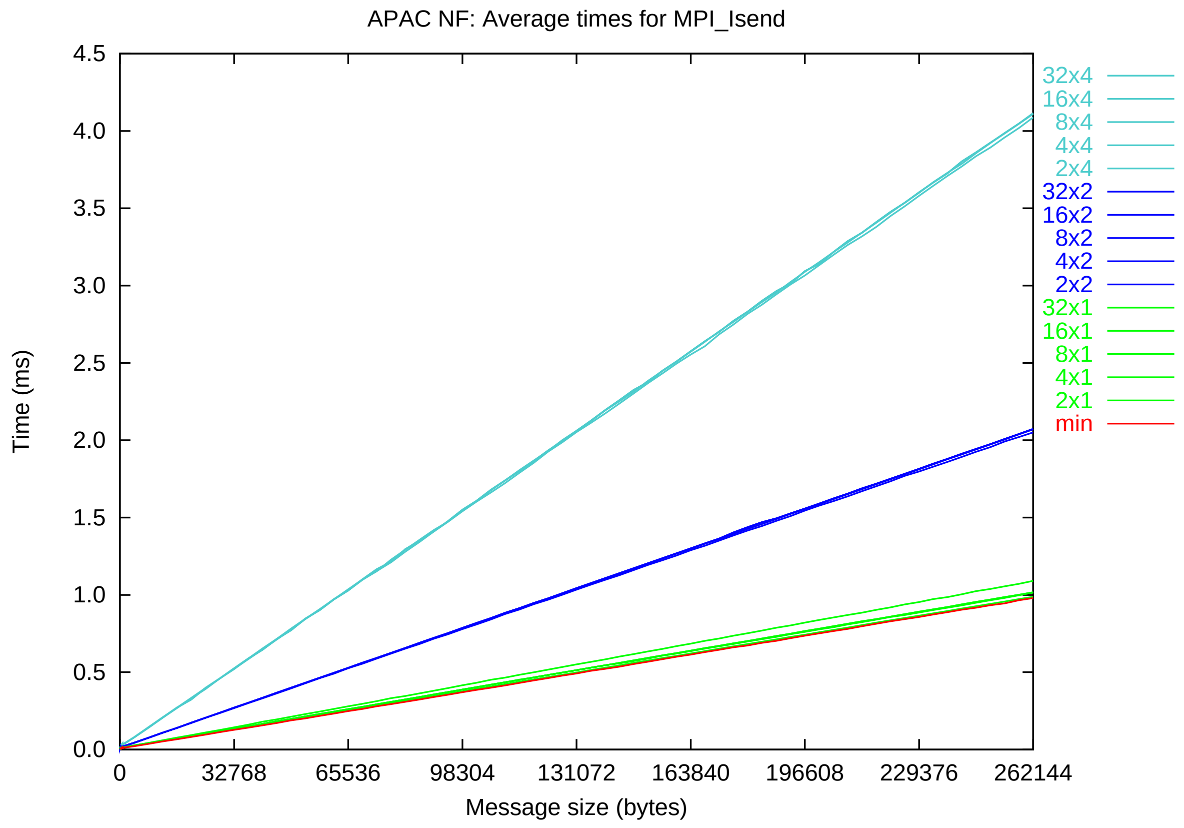


Figure 22: Average times for `MPI_Isend` using large message sizes with various numbers of communicating processes on the APAC NF.

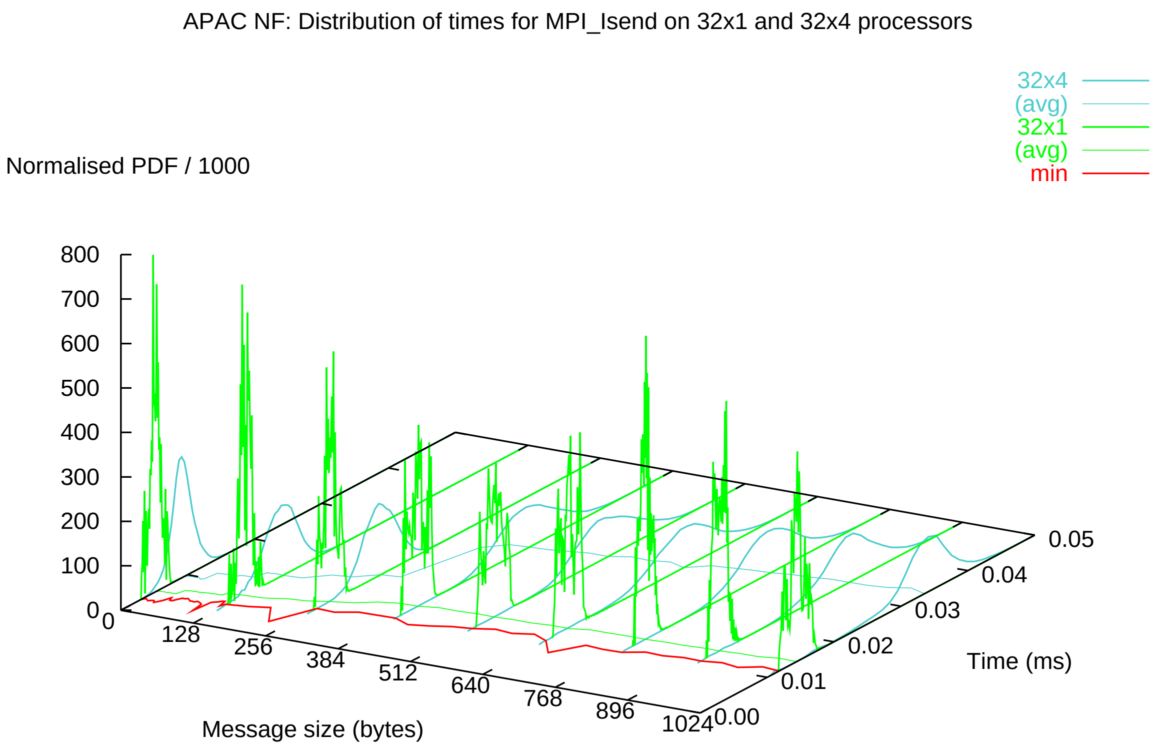


Figure 23: Sampled performance profiles for `MPI_Isend` using small message sizes with 32x1 processes (low contention for the local network interface) and 32x4 processes (high contention for the local network interface) on the APAC NF.

higher repetition count for each test was possible - 100,000 instead of the 10,000 that were used for most of the tests on Perseus and Orion. Using a higher repetition count produces smoother PDFs of the performance of each test, which is qualitatively apparent in the results of Figure 24, compared with those in Figures 15, 16 and 20. These latest results show that, with enough tests, very stable PDFs can be produced for the performance of MPI operations. This assertion, however, raises a question about the cause of the less-smooth performance profiles for smaller message sizes across 32x1 processes of the APAC NF that are apparent in Figure 23. There are two factors at play in these results. Firstly, because the histogram bin-width for any distribution is chosen automatically in proportion to its variance, along with the fact that the variance for these less-smooth results is less than for any of the other tests, the histogram bin-width is smaller in this case than in any of the others. This tends to reduce the appearance of smoothness for a fixed number of tests, even though the significance of statistical variation from a smooth distribution remains unchanged. In particular, despite the fine detail of the distribution, its broad shape is indeed similar to the other distributions observed. Secondly, however, the fine details are in fact indicative of a physical phenomenon: fundamentally, digital

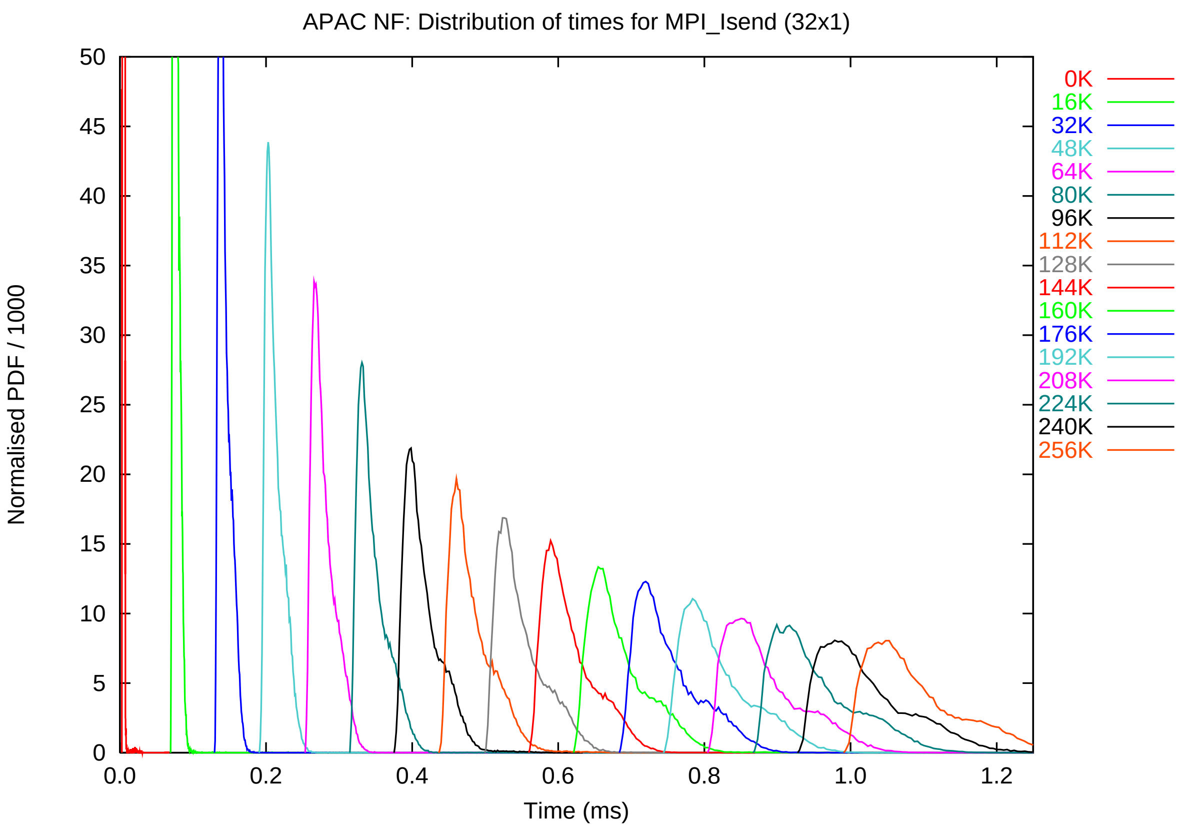


Figure 24: Sampled performance profiles for `MPI_Isend` using large message sizes with 32x1 communicating processes (moderate network contention) on the APAC NF.

computers built from synchronous logic perform operations at discrete points in time. These results were recorded using a histogram bin-width of approximately 50 clock cycles on the APAC NF, and at this granularity the true discrete nature of computational events is actually becoming clear. (A more vivid example of this phenomenon is discussed near the end of Section 5.2, where Figure 57 highlights the effect of the operating system's scheduling policy on the completion of an `MPI_Bcast` operation). With this caveat in mind, the implications of being able to obtain smooth timing distributions for low level communication events are explored further in Section 4.7.

4.5.2 Intra-node, end-to-end completion time

The `MPI_Isend` tests just described were conducted on the inter-node communication networks in Perseus, Orion and the APAC NF. In contrast, this subsection presents the results of similar `MPI_Isend` tests; the only difference being that this time, the intra-node communication networks connecting local processors in each SMP node of the same parallel machines were tested. Figures 25 and 26 show the average times that were recorded for end-to-end `MPI_Isend`-based communication for message sizes between a pair of processes running on one node of Perseus (labelled perseus 1x2), between pairs

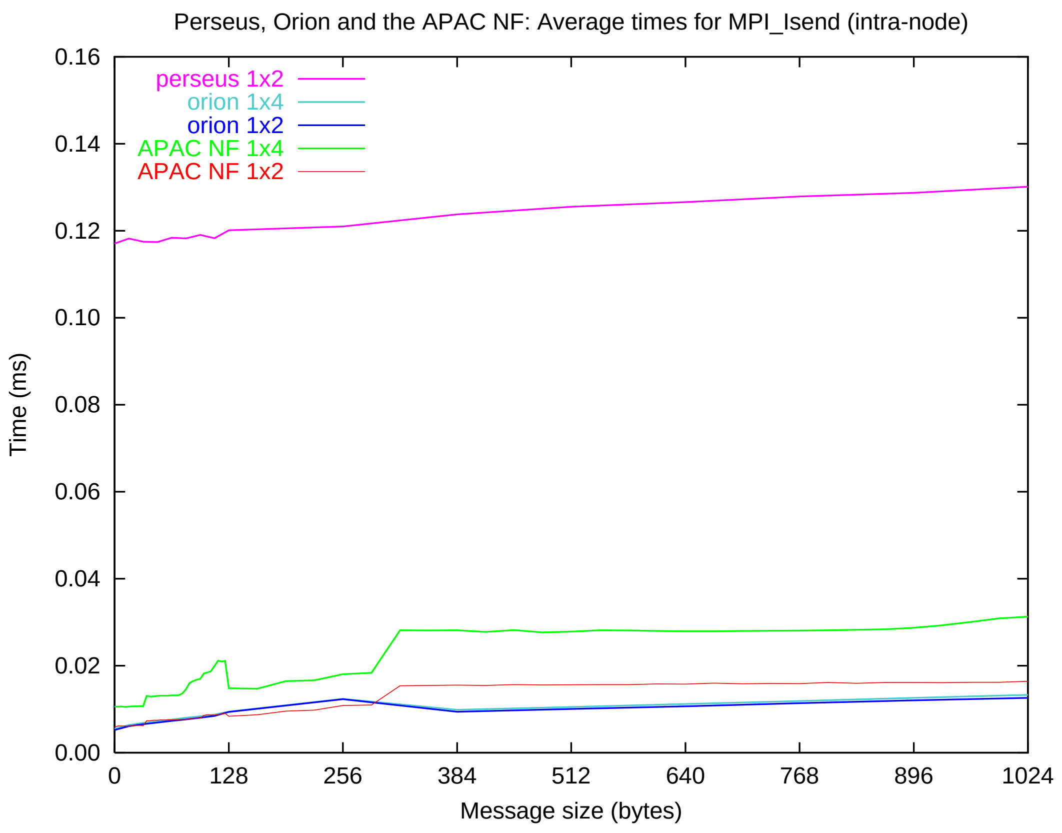


Figure 25: Average times for `MPI_Isend` (intra-node) using small message sizes with 1x2-4 processes on Perseus, Orion and the APAC NF.

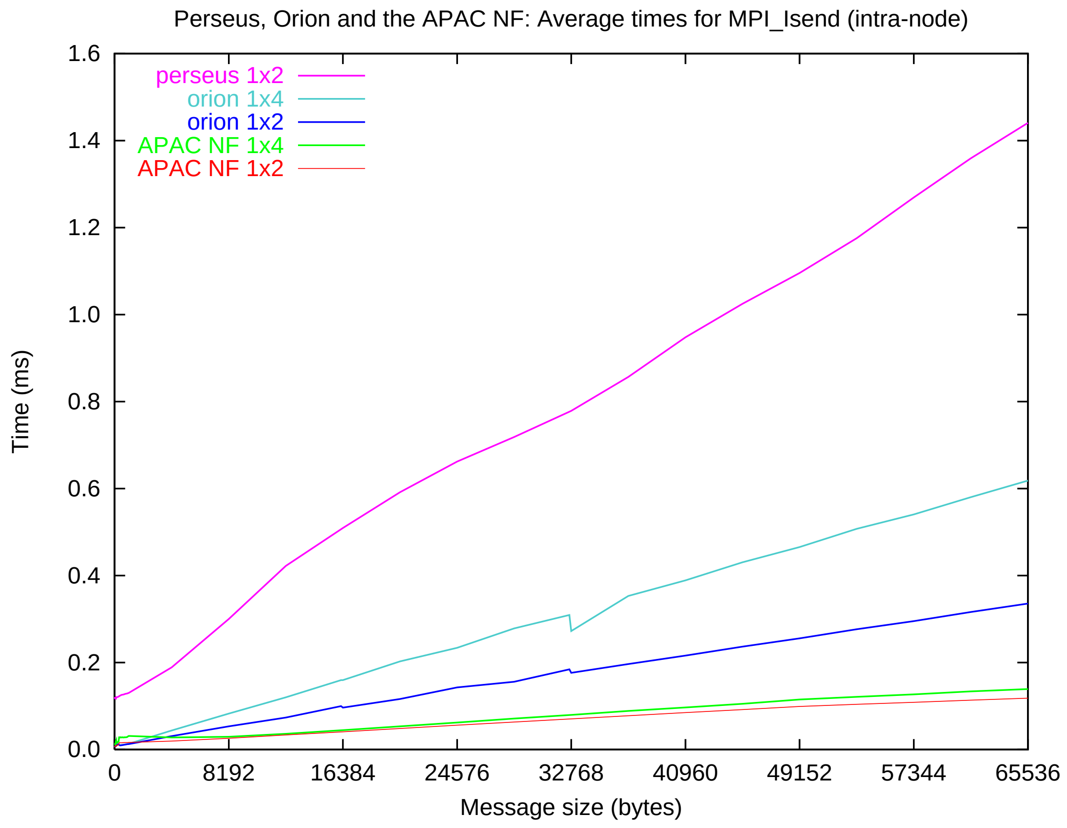


Figure 26: Average times for `MPI_Isend` (intra-node) using large message sizes with 1x2-4 processes on Perseus, Orion and the APAC NF.

of processes running on one node of Orion and one node of the APAC NF respectively (labelled orion 1x2 and APAC NF 1x2) and between two pairs of processes each, running on one node of Orion and one node of the APAC NF respectively (labelled orion 1x4 and APAC NF 1x4).

Roughly-speaking, the intra-node (1x2) message-passing performance on Perseus is described by a latency of 118μs and a bandwidth of 400 Mbit/s; compared to an inter-node (2x1) message-passing performance (see Figures 12 and 13) described by a latency of 160μs and a bandwidth of 91 Mbit/s. Clearly, the intra-node communication performance is superior, as expected. The extent of this superiority, however, is not as great as might be expected. In theory, based on the node's Pentium III memory bus, a bandwidth approaching 100MHz$*$32bit/2 (since there are two processors sharing the bus) or 1600 Mbit/s could be expected; along with a significantly lower latency than that observed. (After all, the latency of the same memory bus is of the order of 10 nanoseconds; and the overhead of the MPI implementation is likely to be just a few microseconds). The reason for slow intra-node performance in this case is that the MPICH version on Perseus was not compiled with shared memory extensions enabled (due to stability issues; shmem support had only just been introduced and did not work reliably), so all communication is forced through the node's normal TCP/IP networking subsystem. Although the data is routed via the loopback address and so is never transmitted via the relatively slow Ethernet interface (as is inter-node communication), the TCP/IP processing must still be done, and this introduces significant performance penalties.

In contrast to this, the intra-node message-passing on nodes of Orion and the APAC NF was carried out directly through shared memory, with clear performance benefit: significantly lower latencies and higher bandwidths. For example, the intra-node message-passing performance on Orion can be loosely characterised by a latency of about 5μs (both 1x2 and 1x4) and bandwidths of 1560 Mbit/s (1x2) and 850 Mbit/s (1x4). It is quite interesting that for small message sizes (i.e. up to 1024 bytes as shown) nodes in Orion can sustain either one or two pairs of communicating processes with almost no performance difference. However, as message size is increased, the 1x2 and 1x4 cases clearly diverge, the latter taking roughly twice as long as the former. This shows that the shared memory part of the MPI implementation on Orion is very well-optimised for small messages - probably taking advantage of caching behaviour and memory interleaving - but that for larger data transfers the (fixed) total memory bandwidth collectively available to all SMP processors (because of the shared memory bus used by Orion's E420R nodes) holds back performance. In fact, bus contention becomes so severe that the ability of the extra SMP processors to improve the overall performance of a parallel code becomes highly questionable: adding extra processors merely starves existing processors of the memory bandwidth required to achieve more than a fraction of their potential.

The AlphaServer ES45 nodes in the APAC NF, however, are equipped with dual memory subsystems, so two pairs of processes can communicate concurrently without interfering with each other. The increased performance that this affords can be clearly seen in Figure 26, which shows that the 1x4 results are almost as good as the 1x2 results. The slight performance drop for using 1x4 processes over 1x2 processes is almost certainly due to increased costs of overseeing extra MPI processes.

4.5.3 Inter-node, local completion time

This subsection is closely related to Section 4.5.1; in fact, the results presented here were gathered from the same benchmarking runs. However, a different facet of the `MPI_Isend` performance was measured. Instead of measuring the time for an end-to-end data transfer, the results here portray the local completion time for an `MPI_Isend` operation. Remember, the `MPI_Isend` operation is asynchronous, which means that when a process sends data to another process, it does not wait for that process to receive the data before carrying on with other work. This subsection examines the amount of time that it takes for the sending process to queue the outgoing data before it can get on with other work.

Figures 27 and 28 show the local completion time for `MPI_Isend` operations as a function of message size on Perseus. Consider first the nx1 results for message sizes up to 14 Kbytes. These results show a very stable linear trend, where the number of communicating processes has little effect on performance. The linear trend, characterised by a bandwidth of about 500 Mbit/s, is basically determined by the rate at which data from the `MPI_Isend` can be processed into TCP/IP packets by Perseus' underlying network subsystem. The reason a slightly higher bandwidth was observed here in comparison with the similar intra-node communication in the last subsection is that the receive part of the communication does not occur here. (It will eventually occur, but that is not what is being timed in this case). With regard to the stable performance measured regardless of the number of communicating processes: because this part of the `MPI_Isend` operation is entirely local to a node, there is no contention for shared network resources, and hence very little performance variability. Now, consider the nx2 results over the same range of message sizes. The basic linear trend remains, but now the number of communicating processes has a small effect on performance. The clue to what is happening is that over this range of message sizes the times recorded for 64x2 communicating processes are the fastest, while those recorded for 2x2 communicating processes are the slowest (note the legend ordering in Figure 27). For the 2x2, 4x2 and 8x2 results, there is clearly a significant overhead when compared with x1 results on the same number of nodes. This is because of intra-node contention for the memory bus as each process attempts to feed data into the network subsystem. The 16x2, 32x2 and 64x2 results fare better because

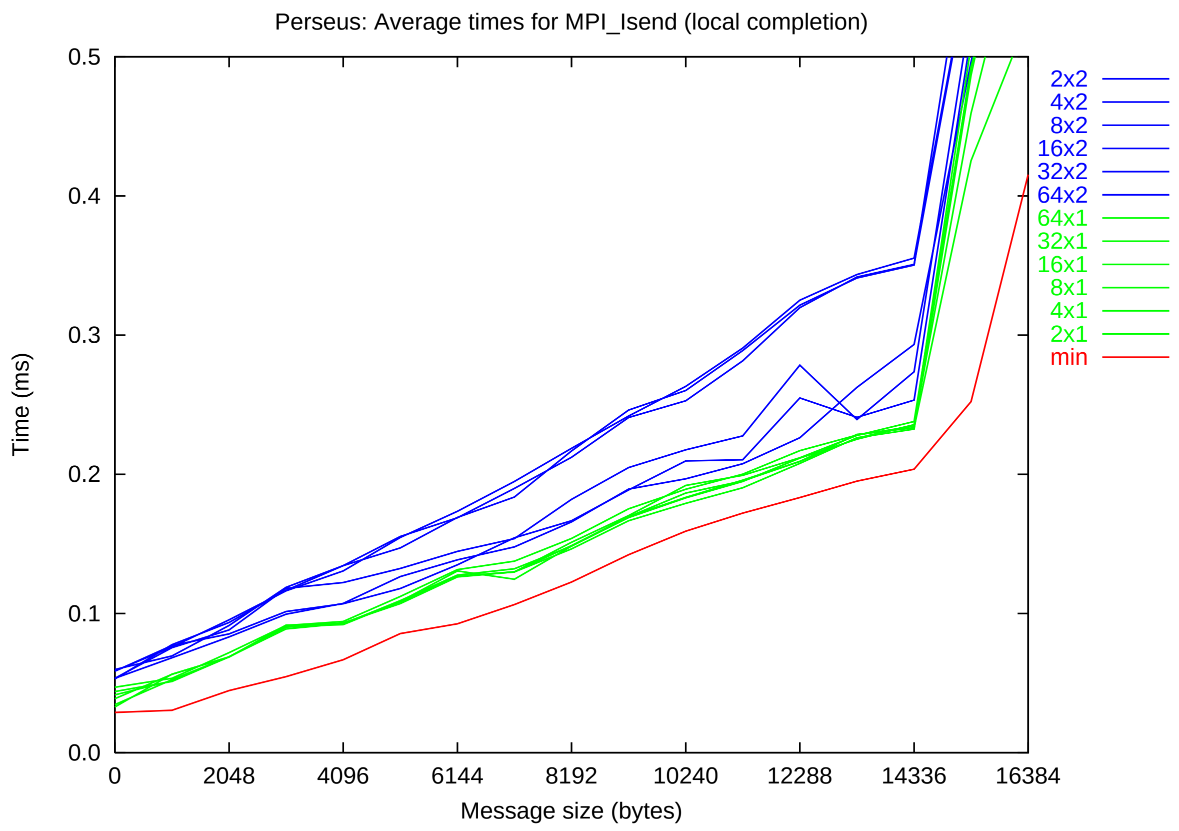


Figure 27: Average times for `MPI_Isend` (local completion) using medium message sizes with various numbers of communicating processes on Perseus.

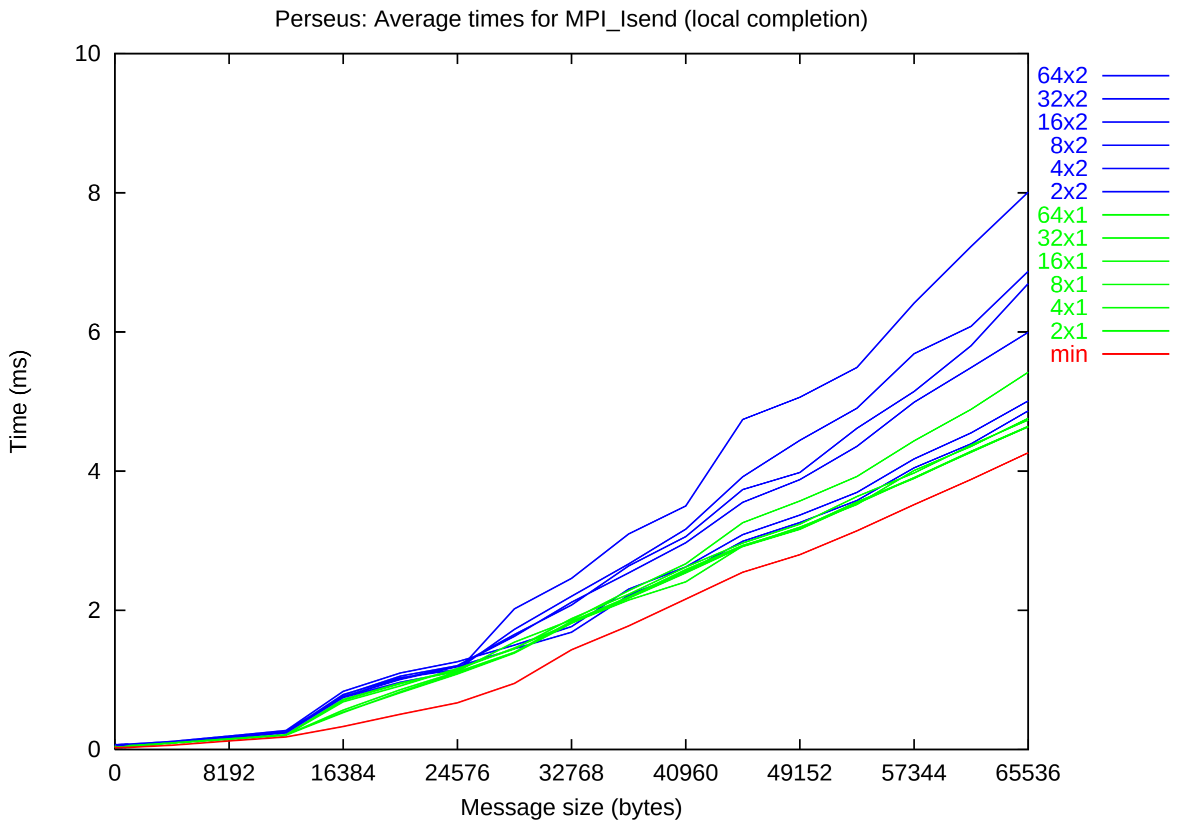


Figure 28: Average times for `MPI_Isend` (local completion) using large message sizes with various numbers of communicating processes on Perseus.

there is less intra-node contention for the memory bus; the subtle reason behind this is (ironically) that inter-node contention does play a small part in this scenario. Because the two individual processes running on each node are not explicitly synchronised and also because they are affected by the variance in inter-node communication, they are not always sending and receiving data in lock-step. In fact, as more nodes communicate, inter-node contention increases, the variance in inter-node message-passing increases, and thus processes spend more time sitting idle. If one process is waiting idle, the other process in the node can get exclusive access to the memory bus, and hence complete sooner itself. The counter-intuitive overall effect of this is that while end-to-end time for an MPI_Isend goes up as contention increases, the time for the average local completion of the operation decreases.

Something is obviously occurring for messages greater than about 14 Kbytes that is greatly affecting the local completion times of MPI_Isend operations on Perseus. This significant jump in completion time is caused by a protocol change. While messages up to 16 Kbytes minus one IP packet are queued for transmission completely asynchronously, messages larger than this are not. Instead, because of buffer space limitations, the first (message size - (16 Kbyte + IP packet size)) bytes are pipelined from sender to receiver, and only the last part of the message is asynchronously queued. Thus, the local completion time for messages larger than this breakpoint is roughly equivalent to the end-to-end time for an MPI_ISend of a message 14 Kbytes smaller than the original plus the local completion time for a 14 Kbyte MPI_Isend. One very practical point of interest arises from this: codes using asynchronous messages to overlap communication and computation will not perform well unless enough buffer space is provided for messages.

Figures 29 and 30 show the results for the local completion times tests on Orion. There are three distinct protocols used, depending on message size, as described in Section 4.5.1. From 0 to 16 Kbytes (minus 96 or 112 book-keeping bytes), the eager message delivery protocol is used, and the results indicate that an MPI_Isend does not return control to a local MPI process until the data to transmit has been copied to an intermediate transmission buffer. From about 16 Kbytes to 32 Kbytes the rendezvous message delivery protocol is used, and for messages larger again the multiphase-rendezvous protocol is used. Both of these behave almost the same, although messages sent with the plain rendezvous protocol suffer from the small performance hiccup that has already been observed. Putting this detail aside, the most important point to make is that the local completion time of messages transmitted using these protocols does not vary with message size. This indicates that the true intent of the MPI_Isend call is being delivered: that the outgoing message is simply queued for delivery by the user-level MPI process; the underlying communication hardware deals with message delivery concurrently in the background.

The most fascinating features of Figures 29 and 30 are the differences in completion

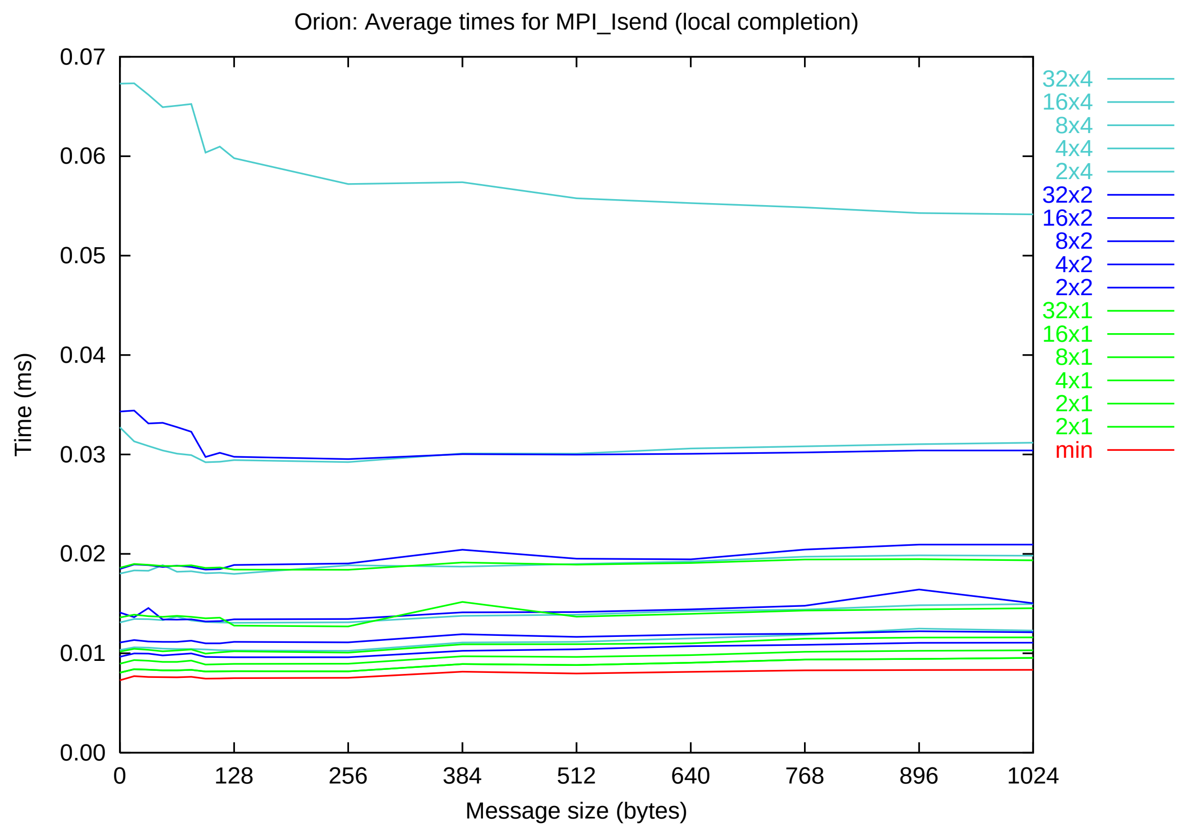


Figure 29: Average times for `MPI_Isend` (local completion) using small message sizes with various numbers of communicating processes on Orion.

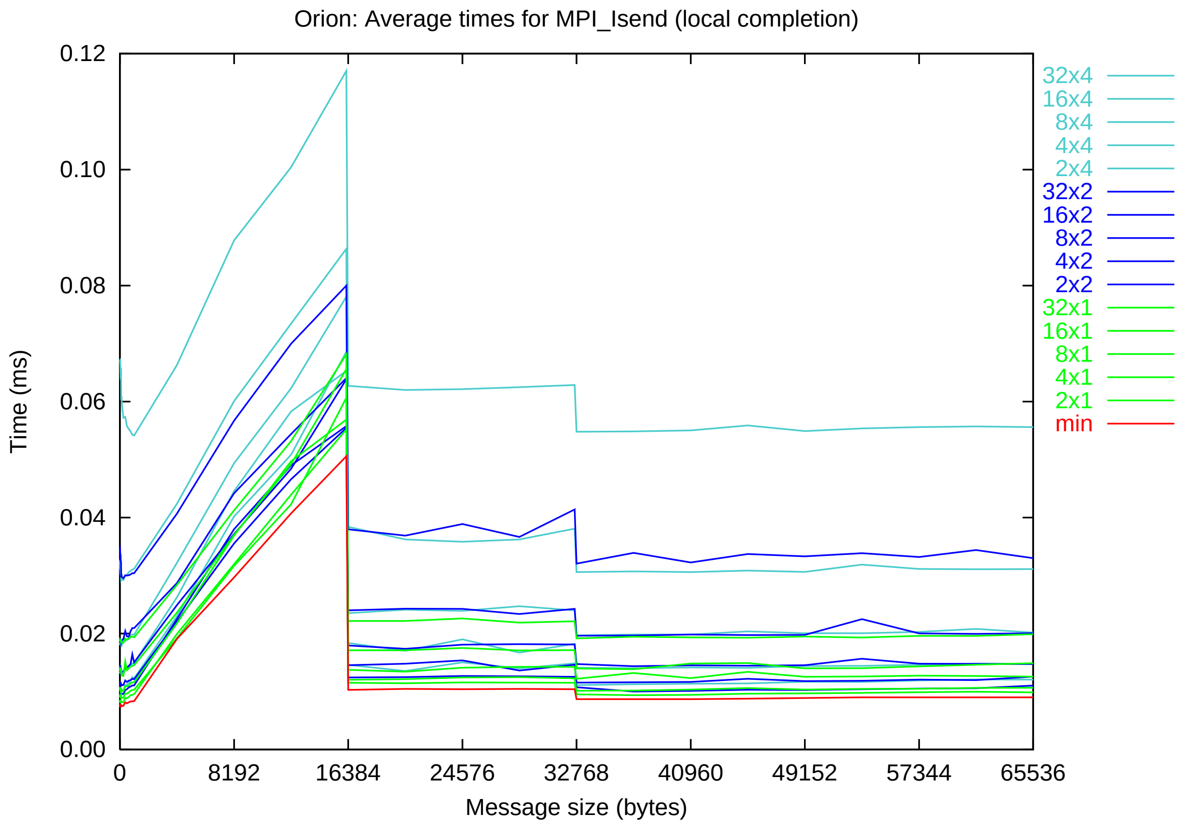


Figure 30: Average times for `MPI_Isend` (local completion) using large message sizes with various numbers of communicating processes on Orion.

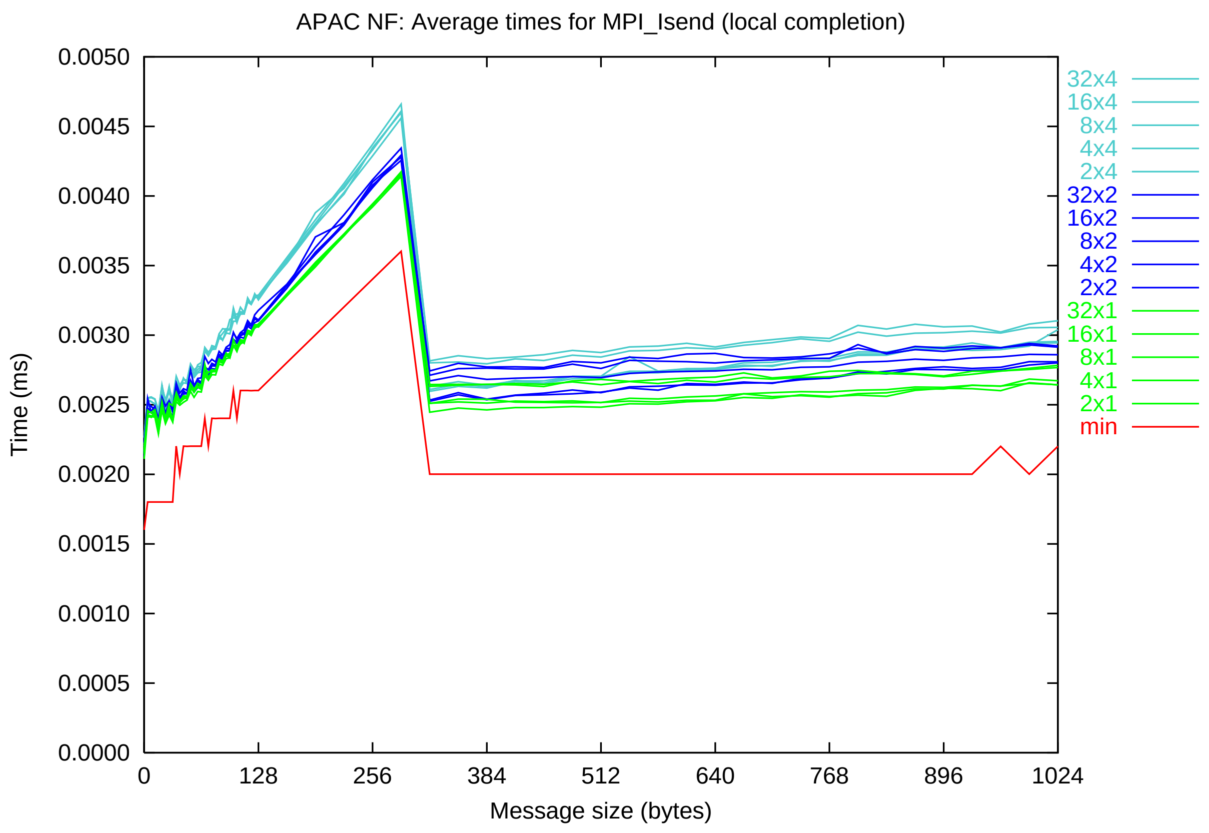


Figure 31: Average times for MPI_Isend (local completion) using small message sizes with various numbers of communicating processes on the APAC NF.

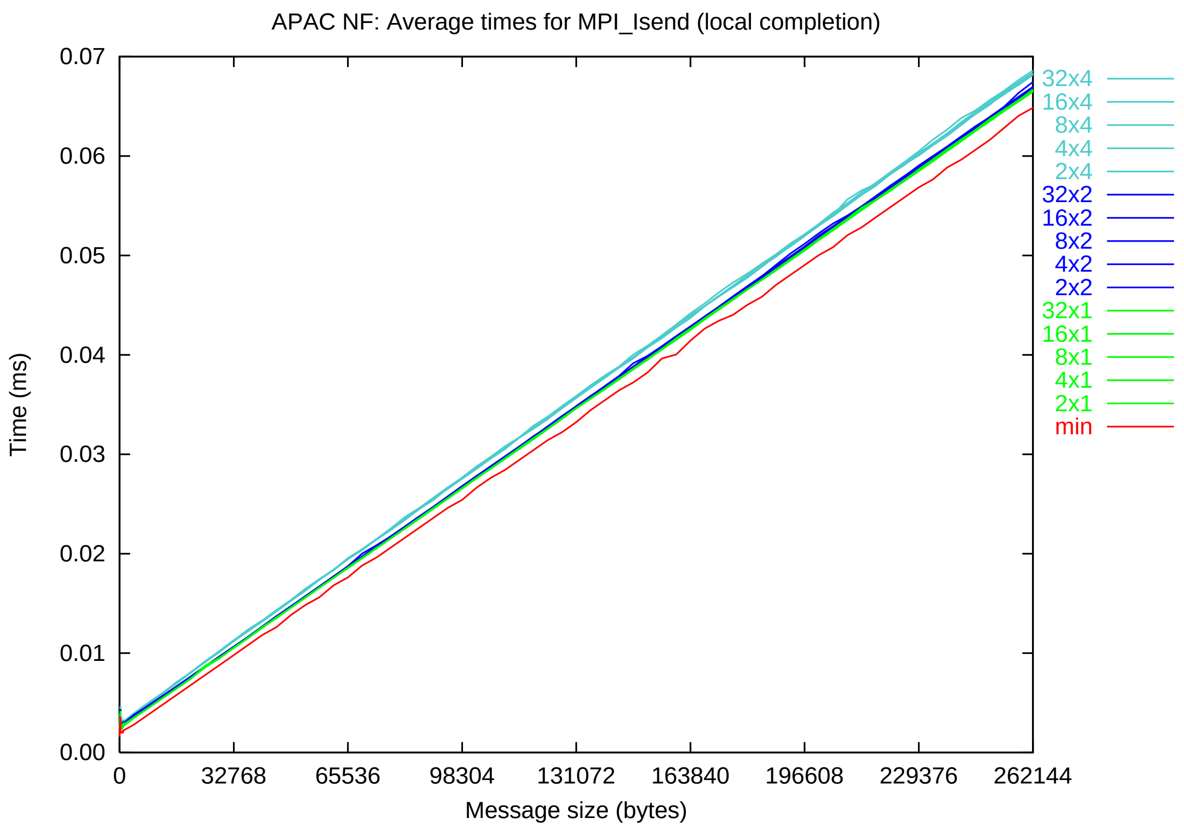


Figure 32: Average times for MPI_Isend (local completion) using large message sizes with various numbers of communicating processes on the APAC NF.

time observed for different numbers of communicating processes. Unlike the results for Perseus (and those of the APAC NF, which will be discussed shortly), the times observed for Orion are heavily dependent on the total number of communicating processes. For example, the 16x4 and 32x2 results are almost the same, the 8x4, 16x2 and 32x1 results are almost the same, etc. Also, there is an almost linear relationship between the number of processes and the time required for the operation. While this overhead is clearly caused by the MPI implementation, it is hard to imagine why it exists – it seems completely unnecessary. What is clear is that this overhead (which appears to run through all other MPI operations on Orion too) will have increasingly detrimental effects for large parallel programs.

Finally, the average local completion times for `MPI_Isend` operations with different message sizes on the APAC NF are shown in Figures 31 and 32. The trend for results using message sizes between 0 and 288 bytes resembles that for messages sent using the eager message delivery protocol on Orion, and a similar process is indeed occurring. The local completion times for messages in this size range take about as long to complete as end-to-end messages in the same size range in Figure 21 minus the latency for a zero byte message. Those end-to-end results represent a round-trip message, because of the synchronous nature of QsNet. Therefore, the results for local completion of an `MPI_Isend` show that return of control to the user-level MPI process is synchronously delayed until the data have been completely transmitted, but returns before the data reception has been acknowledged. This small expense in local completion time due to immediate processing is more than offset by the improved performance in end-to-end `MPI_Isend` performance, which can be seen in Figure 21. Messages larger than 288 bytes are queued for separate processing by the communication subsystem, and control is returned directly to the user-level MPI process. Figure 32 shows that this queueing time depends on message size, however the very small slope of results in that figure mean that very little time is spent preparing a message for subsequent delivery. Indeed, comparison of Figure 32 with Figure 22 shows that the local completion time for large `MPI_Isend` messages on the APAC NF is typically much less than 1% of the complete end-to-end transfer time.

4.6 Results for `MPI_Sendrecv`

The same tests described in Section 4.5 were repeated, except that this time the performance of `MPI_Sendrecv` operations was measured. From the perspective of a user program, an `MPI_Sendrecv` operation involves the simultaneous exchange of two messages between a pair of processes – one in each direction. Physically, many machines (including all three of the machines benchmarked here) have full-duplex network links so there is the potential for MPI_Sendrecv messages to actually transit those interconnect networks simultaneously. Note, however, that message-handling by the operating system at each end cannot proceed concurrently (at least, not if only one processor is available for that task). This subsection examines the time taken for MPI_Sendrecv operations on Perseus, Orion and the APAC NF to see how well these machines actually perform when instructed to carry out simultaneous, bidirectional message-passing.

Firstly, contrast the nx1 curves plotted in Figure 12 with the corresponding curves in Figure 33. Even though twice as much data is being transmitted in the bidirectional MPI_Sendrecv test, the time taken to achieve this for any particular message size is considerably less than twice that of the MPI_Isend test, especially when a large number of nodes are used. This characterises the situation where the full-duplex nature of the interconnect network is able to support the simultaneous communication of the MPI_Sendrecv operation without much performance degradation.

Note, however, that part of the reason for the small difference between the performance of the two operations in this case is that the work done by the operating system of processing the incoming and outgoing messages can be divided amongst the two processors in each node. Although MPIBench binds each MPI process to a separate processor, there is no such binding of systems routines, and of particular interest in this case, network I/O. Therefore the effect of message-handling overhead on simultaneous bidirectional message-passing is more pronounced in the comparison of the x2 curves plotted in the same figures. In these cases, each processor is busy servicing both departing and arriving messages. Notice, for these cases, that time taken to complete an MPI_Sendrecv routine using small messages is roughly twice as long as the time taken to complete a unidirectional, but otherwise identical, MPI_Isend routine. This effect becomes less noticeable as message size increases, because the greater part of a message's life time is spent traversing the network link rather than the operating system's network stack. Hence, medium-sized MPI_Sendrecv messages perform almost as well as equivalently sized MPI_Isend messages. However, a third effect comes into play for even larger message sizes and increased contention levels (i.e. the number of participating processes), which is therefore almost certainly caused by exhaustion of network switching resources. This effect causes performance for MPI_Sendrecv messages to degrade noticeably, and they once more approach half of the performance of MPI_Isend messages.

A final point concerning the performance of the MPI_Sendrecv routine on Perseus relates to the frequent lack of results for message sizes above 32 Kbytes in size (see Figure 34). It was found that the MPICH 1.2.0 libraries would fail when asked to perform a large number of MPI_Sendrecv operations with message sizes larger than twice the underlying socket buffer space allocated with P4_SOCKBUFSIZE. It was discovered, however, that this problem did not recur in the results of further tests that used an updated

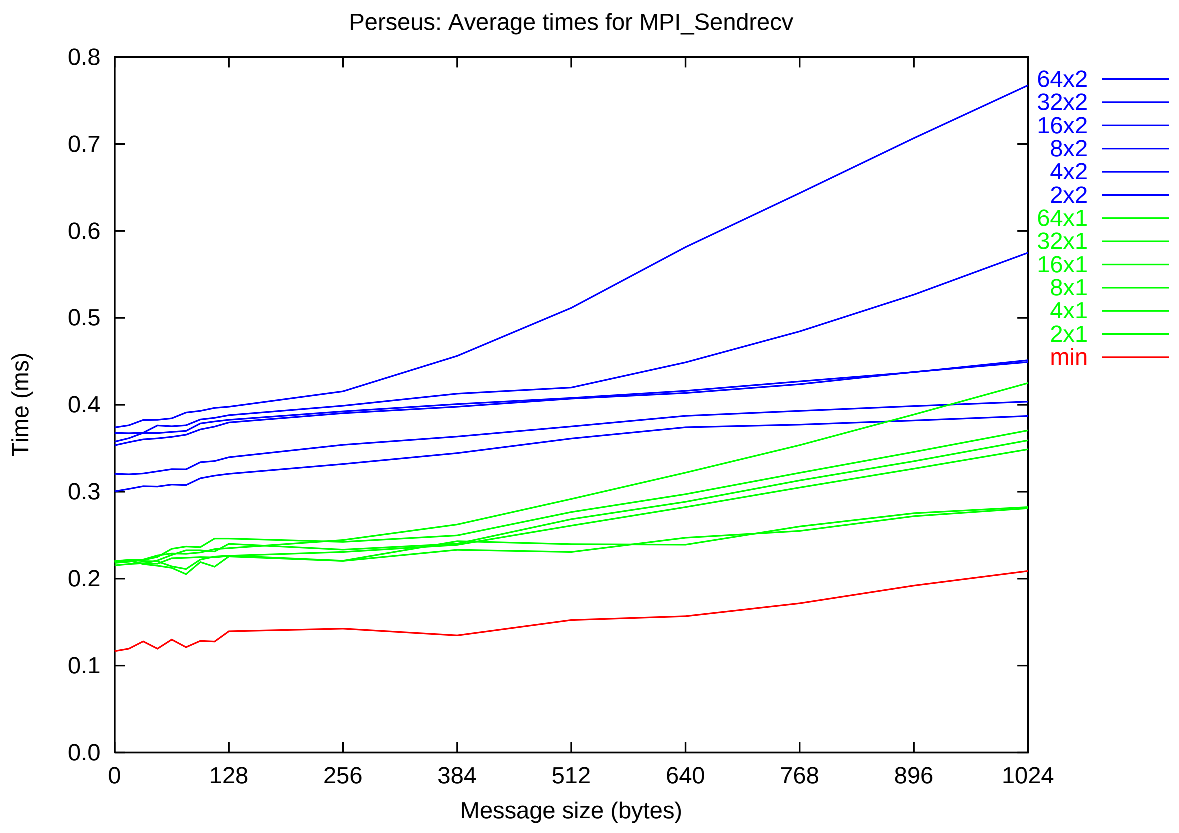


Figure 33: Average times for `MPI_Sendrecv` using small message sizes with various numbers of communicating processes on the Perseus.

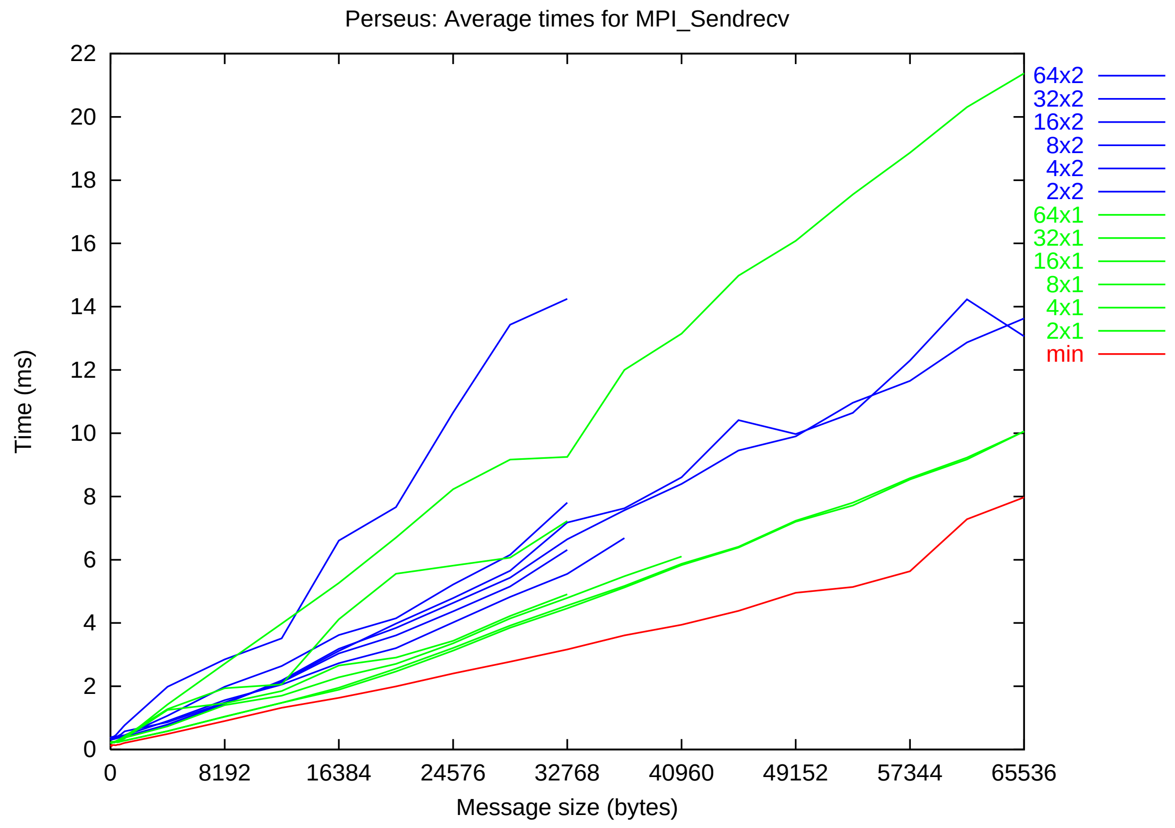


Figure 34: Average times for `MPI_Sendrecv` using large message sizes with various numbers of communicating processes on Perseus.

version (1.2.4) of MPICH, so it appears that the bug has been corrected. Unfortunately, it was impractical to rerun other tests using the updated MPICH library or an enlarged `P4_SOCKBUFSIZE` - there are almost always going to be problems with whichever version and configuration of a software package is being analysed. Therefore, in the interest of maintaining uniform experimental conditions for all tests, the buffer space was not changed from its default setting nor was a different MPICH version benchmarked to fill in the small gaps in the results, which are of little practical concern anyway.

Surprisingly, the performance of the `MPI_Sendrecv` operation does not appear to benefit from the full-duplex nature of Orion's Myrinet network. This can be seen by comparing the completion time of the `MPI_Sendrecv` operations in Figures 35-36 with the `MPI_Isend` operations in Figures 17-18. This reveals that (for both large and small messages) the `MPI_Sendrecv`/x1 curves track the `MPI_Isend`/x2 curves, and the `MPI_Sendrecv`/x2 track the `MPI_Isend`/x4 curves, where two or four messages are simultaneously queued for transmission at each node respectively. Furthermore, the `MPI_Sendrecv`/x4 curve tracks two-times the `MPI_Sendrecv`/x2 curve (for large messages), which, because of the high bandwidth efficiency of the x4 test (that was discussed earlier), indicates that serialisation of the `MPI_Sendrecv` is indeed occurring. Because the underlying Myrinet hardware provides full-duplex communication channels, the serialisation cannot occur there. Furthermore, because GM is accessed from user-space, the serialisation cannot be occurring at the operating system level. This leaves the MPI implementation or the GM implementation upon which it relies as the culprit. In order to further investigate the source of the serialisation, the `MPI_Sendrecv` results were compared with results from an `MPI_Isend` test that had been modified to measure the performance of two simultaneous and out-of-phase but essentially independent ping-pong streams (rather than a single ping-pong stream). The results for these bidirectional `MPI_Isend` tests were essentially indistinguishable from the results of the (in terms of data transfer, identical) `MPI_Sendrecv` test. Because of the independent nature of the data streams, the most reasonable conclusion that can be drawn from this is that bidirectional message-passing is serialised in the GM layer implementation on Orion, although the MPI implementation itself can not be categorically ruled out. (Unfortunately, only access to the source code would lead to conclusive answers). Whatever the cause, the effect of the serialisation is clearly demonstrated in Figure 37. This figure shows the performance profile of a `MPI_Sendrecv` operation between two processes using 512 byte messages. Given full-duplex operation, there should be one spike, since data travelling to each process could occur concurrently and so the processes should be expected to finish at the same time. The fact that there are two well-separated spikes, where closer analysis revealed that the completion of both processes were always represented by values in separate peaks for any individual test, demonstrates the serialisation of the send/receive process.

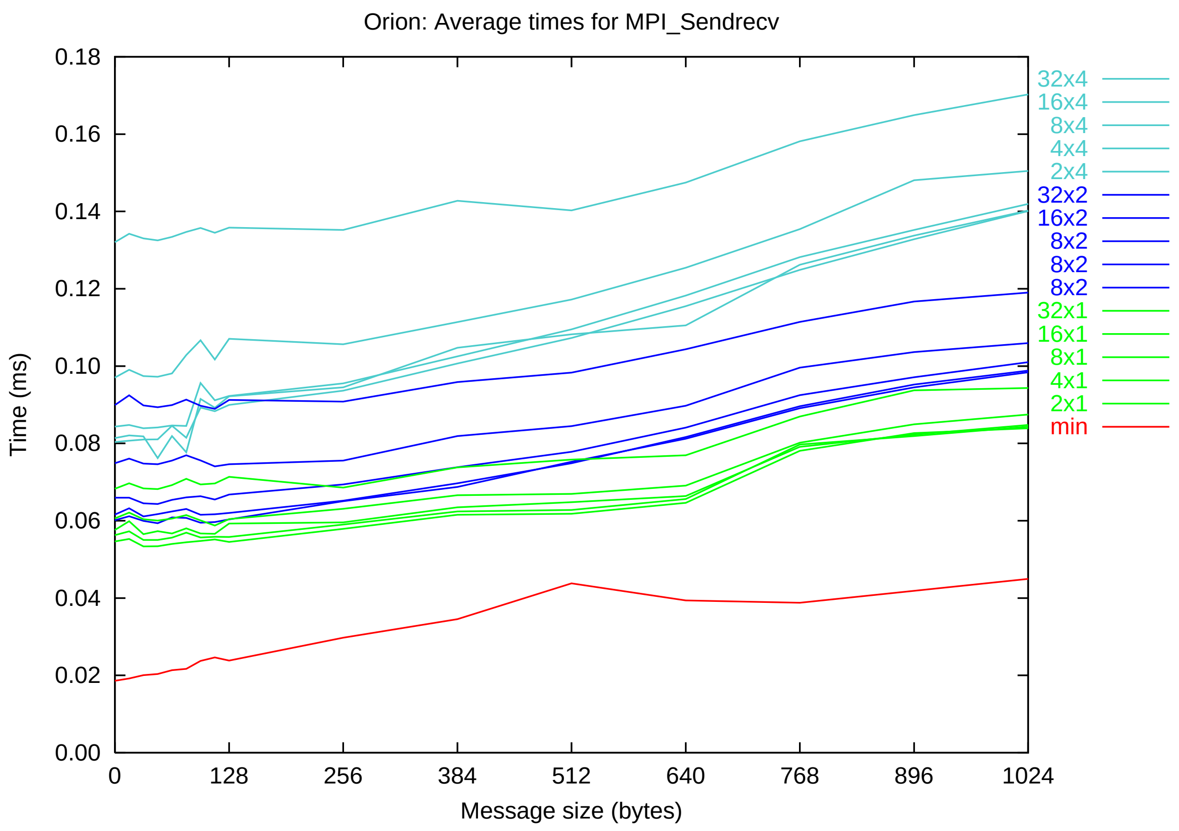


Figure 35: Average times for `MPI_Sendrecv` using small message sizes with various numbers of communicating processes on Orion.

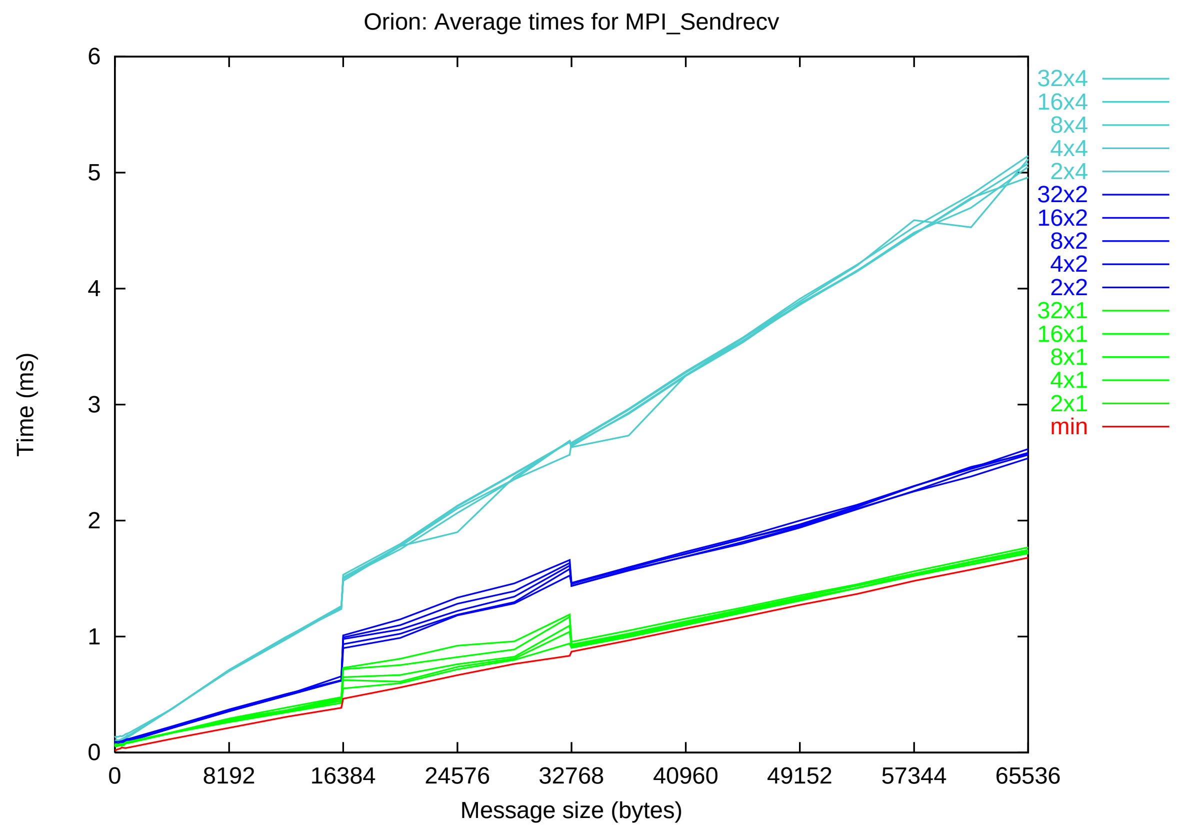


Figure 36: Average times for `MPI_Isend` using large message sizes with various numbers of communicating processes on Orion.

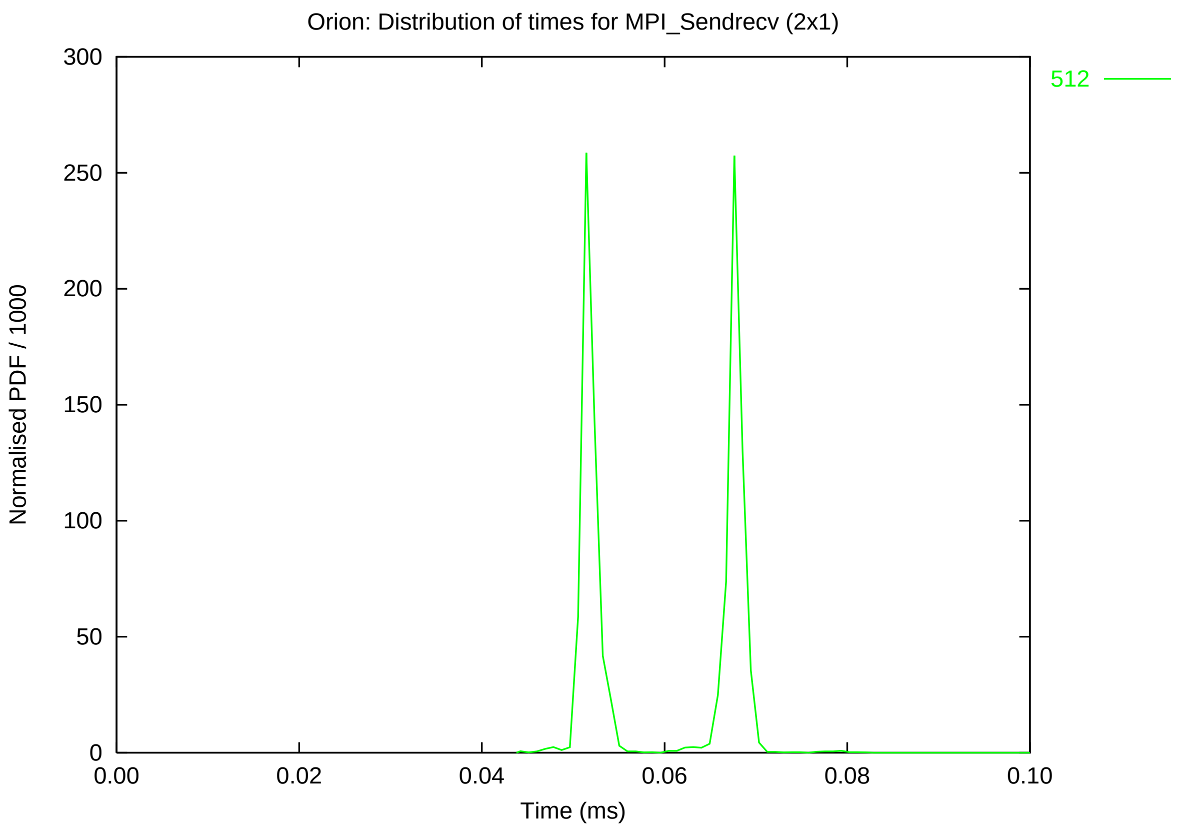


Figure 37: Performance profile showing the serialisation of bidirectional message-passing for an `MPI_Sendrecv` operation using 512 byte messages with 2x1 processes on Orion.

The differences between the performance of `MPI_Sendrecv` (see Figures 38-39) and `MPI_Isend` (see Figures 21-22) on the APAC NF are very similar to the the differences between the performance of the same operations on Orion that have just been discussed. Once again, it seems that the `MPI_Sendrecv` operation does not appear to derive overall advantage from the full-duplex nature of the underlying network. Other research [277] on identical hardware, which obtained commensurate values for peak `MPI_Sendrecv` bandwidth at various message sizes, has suggested that PCI bottlenecks and DMA contention between system memory and the network interface are the cause of the unexpectedly poor performance.

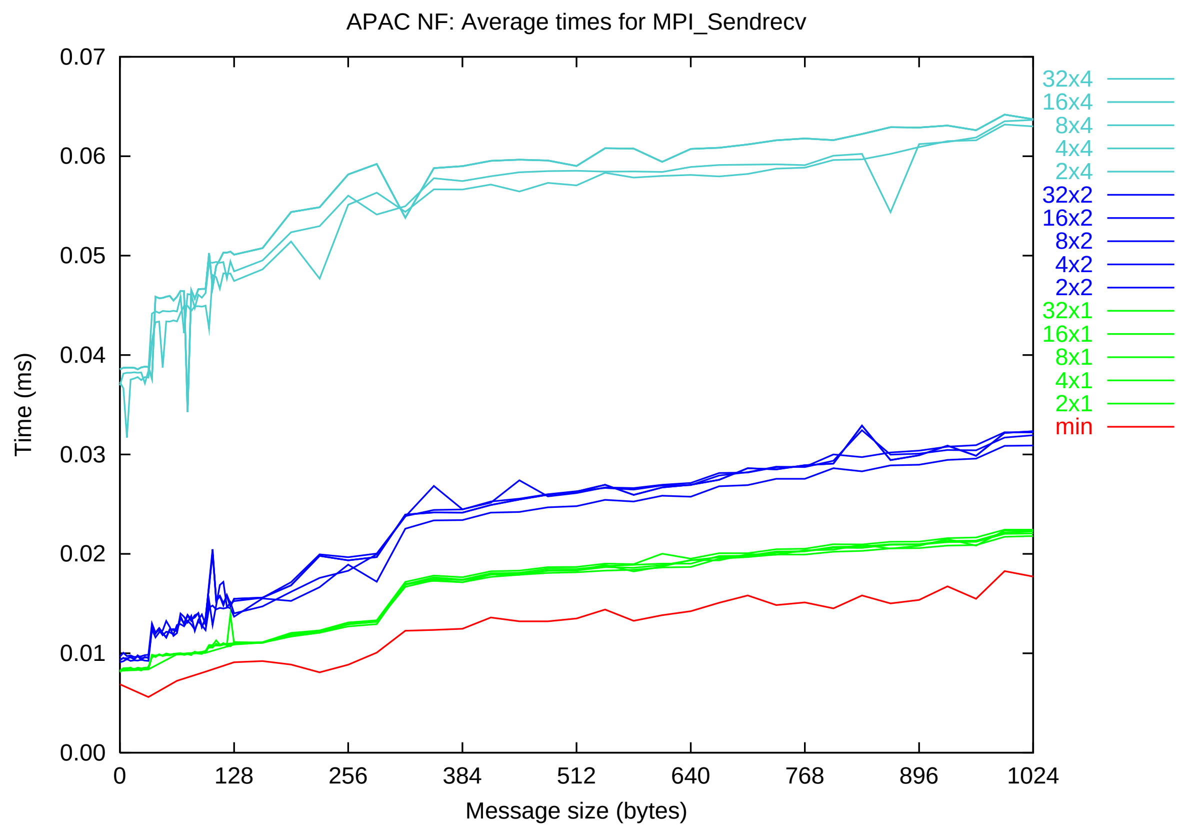


Figure 38: Average times for `MPI_Sendrecv` using small message sizes with various numbers of communicating processes on the APAC NF.

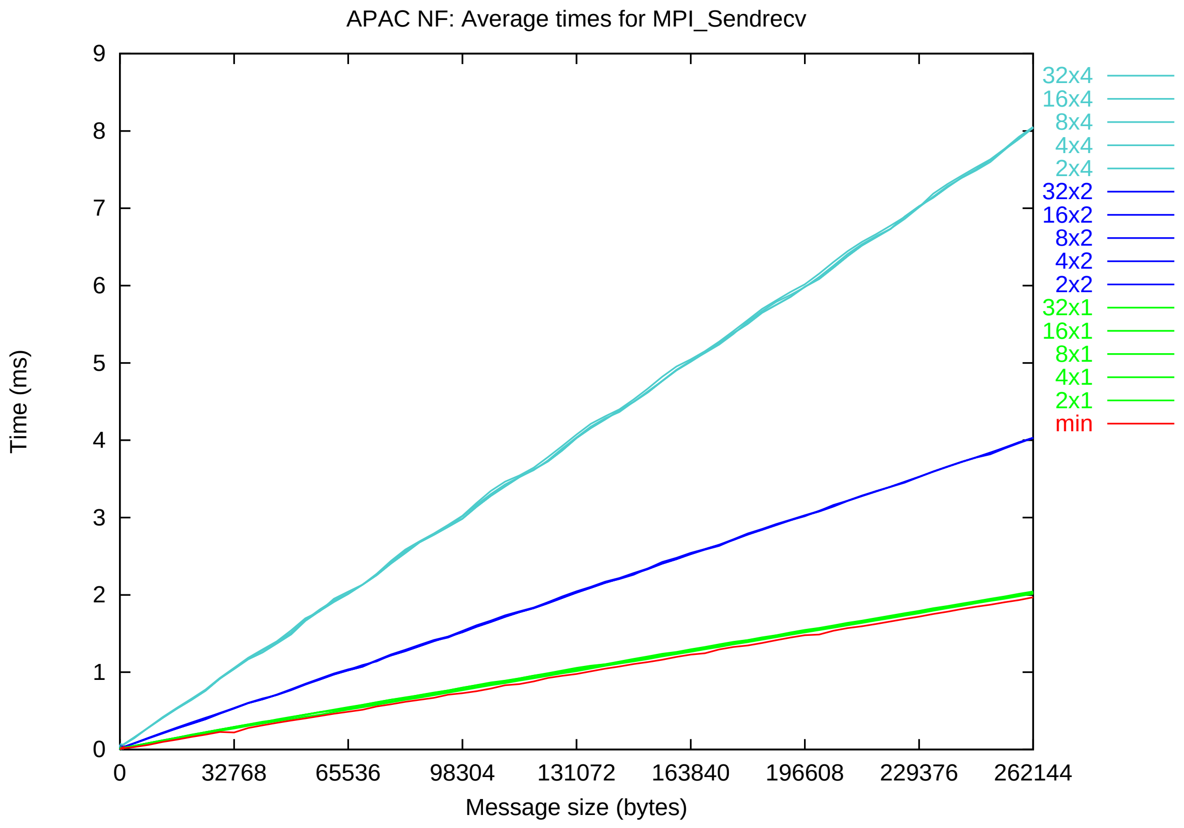


Figure 39: Average times for `MPI_Sendrecv` using large message sizes with various numbers of communicating processes on the APAC NF.

4.7 Analytical Models

Many of the results and related discussions presented so far have been heavily focussed on the PDFs of the measured performance data; and in particular with their empirical qualities. In contrast, this section examines the quantitative nature of the observed distributions to try and determine if and how they can be characterised by analytical models. Good analytical models would be very useful indeed, for both PEVPM modelling and a deeper general understanding of the performance characteristics of communication in parallel programs (or even communication in computer networks in general).

In terms of the PEVPM, being able to randomly select communication time from a simple analytical expression instead of from an empirical distribution has several significant advantages. Firstly, it reduces the PEVPM's storage requirements by potentially orders of magnitude. Instead of having to store complete histograms of the observed performance of an operation for each message size, only a handful of parameters describing the distribution need to be recorded. Secondly, because of this, the time-cost of evaluating a PEVPM model using analytical models for communication performance will be far less. Using empirical models of communication performance requires the PEVPM to read a comparatively large amount of data from disk (which it is sensible to cache if sufficient memory is available) for each specific communication model; this is relatively costly. In contrast, using analytical models of communication performance would allow the PEVPM to perform individual selections from the modelled distribution on the fly (which may also be sensible to cache; there is a space-time trade-off for each selection), which is relatively cheap. Thirdly, it is quite possible that using analytical models for communication performance will lead to better simulation results, especially in cases where benchmarks were conducted with only a small number of iterations and using coarse histogram bin sizes. On the assumption that the real performance distributions are in fact smooth and follow an analytic form (which will be shown to be reasonable in the next paragraph) then using that analytic form to model communication time would be superior to using the approximate empirical data. Finally, it is easier to interpolate or extrapolate for gaps in measured performance data or even speculate on hypothetical performance data using analytical rather than empirical models of communication speed, especially in the case that parameters for an analytical model can be derived from physical parameters such as network latency and bandwidth as well as communication stack processing overheads.

As noted in Section 4.5.1, the true performance distributions that occur are actually discrete on a small enough time scale. However, as pointed out in Section 3.4.2, using continuous distributions to gloss over these very fine-grained characteristics still produces very accurate macroscopic performance results. Therefore, with this caveat in mind, it is reasonable to assume that the PDFs describing the performance of individual communication operations are in fact smooth. The assumption that these PDFs can be adequately described by analytical expressions, however, still needs to be examined.

For now, consider any of the PDFs that have been presented (for example Figures 14, 16, 19, 20, 23 or 24 from Section 4.5.1). Roughly speaking, each of these distributions have a hard lower bound, usually a normal-shaped middle and taper out with an unbounded tail. The lower bound is determined by the minimum message latency that is possible under perfect conditions. The shape of the middle-part of the curve is determined by contention effects. In reality, the right-hand tail does not actually extend to infinity because of the true discrete nature of the distribution and protocol timeouts (which will be discussed in the next section). It is convenient, however, to assume an infinite tail and so fit a continuous analytical function to the observed results. This approximation affects overall accuracy very little because the probabilities associated with the tail become astronomically small very quickly.

A number of common distribution functions exhibit these broad properties. These include exponential, Erlang, gamma, Pearson 5, lognormal and Weibull distributions [191]. Unlike the normal distribution, these distributions are asymmetric in general and cannot be distinguished by their mean and variance alone. In addition to mean and variance, which are also known as the first and second moments of a distribution, these distributions must be differentiated by their third and fourth order moments, known as skewness and kurtosis respectively. The skewness statistic describes the degree of symmetry of a distribution. A positively skewed (right-skewed) distribution rises rapidly, reaches its maximum and falls slowly with a pronounced right-tail. A negatively skewed (left-skewed) distribution rises slowly reaches through a pronounced left-tail, reaches its maximum and falls rapidly. The kurtosis statistic describes the peakedness/flatness of a distribution near its mode, relative to the normal distribution.

The distribution functions listed above, and hence their moments, are defined by at most three parameters, usually known as the scale parameter, the shape parameter and the location parameter. The scale parameter defines where the bulk of the distribution lies, or how stretched out the distribution is. In the case of the normal distribution, for example, the scale parameter is the standard deviation. Unsurprisingly, the shape parameter defines the shape of a distribution. Some distributions, for example the normal distribution, do not have a shape parameter because they have a predefined shape that does not change. Finally, the location parameter shifts the origin of a distribution either left or right.

Without a location parameter (or with a location parameter of zero) all of the distributions listed above have a domain of $(0,\infty]$ so the location parameter can be used to model the lower bound on message latency. Determining which scale and shape parameters should be used to model the PDF of communication performance is less clear. Rather than blindly trying to fit observed data to known analytical distributions, it is more useful to first examine how the assumptions of those analytical expressions mesh with the underlying traffic patterns and contention that are fundamental to parallel programs.

Historically, the most frequently used model for the time instants at which events are observed has been the Poisson process (also known as an M/M/1 queue in Kendall notation [201]). In particular, this model has been (and still is) heavily used in the telecommunications industry to model the interarrival and service times of telephone calls. From these roots, it has been commonly applied to modelling data transmission in computer networks. A Poisson process is characterised by a sequence of randomly spaced events, where the (event) arrival behaviour after an event is independent of, but probabilistically like, the original behaviour. This *memoryless* property means that the fact an event has not happened yet provides no information about how much longer it will be before it does happen. Slightly more formally, a Poisson process must meet three conditions: 1) that in a sufficiently short time, only 0 or 1 events can occur; 2) that the probability of exactly 1 event occurring in an interval is proportional to the the duration of the interval; and 3) that non-overlapping intervals are independent Bernoulli trials. Mathematically, this implies that the interarrival times between events are exponentially distributed with a mean of $1/\lambda$, where λ is the average arrival rate.

Under these conditions, the Poisson distribution gives the probability that a given number of events will occur within a certain time interval. In relation to a communication network, when a large number of packet arrival events occur in a short period of time (due to the inherent randomness of interarrival times) communication buffers will become very full. Hence the time that a packet can spend waiting for transmission can be large. With this in mind, a Poisson process provides a model of network contention that can be used to determine the interarrival time and service time (i.e. end-to-end latency) of message-passing operations. Like the distribution of interarrival times, the service times of a Poisson process are exponentially distributed. The exponential distribution function is defined by a location parameter γ and scale parameter λ, but no shape parameter because it has a fixed shape – it begins at $T = \gamma$ with a value of λ and decreases exponentially and monotonically with T.

The Poisson process provides an attractive modelling formalism because it has a number of properties that greatly simplify its evaluation. Unfortunately, however, it fails as a realistic model for network traffic, and in particular message-passing traffic for parallel programs, because in these cases condition 3 (above) is not true. Data communication is often very bursty and is self-similar in nature [125, 214, 268, 371, 372]. Message-passing programs, due to their frequent synchronisation (either explicit or implicit), are even more so. This means that contention between seemingly unrelated processes is not truly independent. For this reason, researchers have suggested that Poisson processes are inappropriate for modelling data communication; a number of recent studies, mostly focussed on wide-area networks, have found that service times for data traffic are much better modelled by heavy-tailed distributions such as Erlang, lognormal or Weibull distributions [119, 120, 121, 232, 270, 272, 378].

The Erlang distribution was first described in 1917 [52, 111] by the famous Danish telephone engineer of the same name, who is considered to be the founder of queueing theory. It was specifically designed to model the situation where the likelihood of immediate process completion increases with the amount of processing that has already been done. In particular it describes the waiting time until the m^{th} event of a process that occurs randomly over time. This makes the Erlang distribution particularly good at modelling transmission times in the face of contention (or more generally, survival data). Erlang distributions are defined by their location parameter x, positive integer shape factor m and scale parameter β. The case of $m = 1$ reduces an Erlang distribution to an exponential. For $m > 1$, the Erlang distribution is 0 at x, peaks at a value that depends on both m and β and decreases monotonically thereafter.

The Erlang distribution is actually a special case of the gamma distribution, which is identical, except that the shape factor m may take on non-integer values. Also related to the gamma distribution is the Pearson 5 distribution, which is sometimes called the inverse gamma distribution: there is a reciprocal relationship between a Pearson 5 random variable and a gamma random variable. The Pearson 5 distribution is particularly useful for modelling time delays where some minimum delay value is almost assured and the maximum time is unbounded and variably long [213]. This makes it an attractive candidate for modelling message-passing time.

The lognormal distribution, first described by Kolmogorov in 1941 [206], results from the product of many independent random variables, where overall distribution values are based on the cumulative effect of many small perturbations in those variables. This theoretical underpinning also fits well with the idea of contention, where mutually excluded access to shared resources can increase the chance of further contention, thus causing increasingly lengthy delays. Mathematically, the lognormal distribution is described by a random variable whose logarithm is normally distributed. It is a three-parameter distribution, defined by the parameters: μ', which represents the mean of the logarithms of the random variable; $\sigma^{T'}$, which represents the variance of the logarithms of the random variable; and T, which can be used to shift the distribution left or right. The distribution looks like a normal curve that has been right-skewed. For $\sigma^{T'}$ significantly greater than 1 the PDF rises and falls very sharply, looking almost like an exponential distribution.

Both the gamma family (including Erlang and Pearson 5) and lognormal processes provide (different) potential theoretical explanations for the effects of random contention on message-passing service times. However, a lack of strict randomness in the underlying process being modelled (in this case contention) could lead to negatively skewed data, which cannot be fit by either gamma family or lognormal models. The Weibull distribution, first described by Weibull in 1939 [365] and then to a wider audience in 1951 [364], is a very versatile, general-purpose distribution that can be used in these cases [191]. It is defined by three parameters: a shape parameter β, scale parameter η and location parameter γ. Depending on the values of the parameters, the Weibull distribution can be used to model a variety of behaviours. For example, using $\beta = 1$, the Weibull distribution reduces to an exponential distribution; $\beta < 1$ produces a exponential-like curve, except that it begins higher and diminishes faster. Using $1 < \beta < 3.6$ results in a distribution that looks much like a gamma or lognormal, i.e. monotonically rising until the mode, and then monotonically decreasing with a pronounced right-tail. For $\beta = 3.6$ the coefficient of skewness approaches zero, and the curve approximates a normal distribution. Uniquely, for $\beta > 3.6$ the distribution is negatively skewed, i.e. most data is found in the right-hand side of the distribution, despite a left-bounded tail.

Interestingly, all of these distributions can be mimicked by a special distribution called the generalised gamma function $f(t)$, which is a three-parameter distribution that can be written as:

$$f(t) \;=\; \frac{\beta}{\Gamma(k)\theta}\left(\frac{t}{\theta}\right)^{k\beta-1} e^{-(\frac{t}{\theta})^{\beta}}$$

where $\theta > 0$ is a scale parameter, $\beta > 0$ and $k > 0$ are shape parameters and $\Gamma(x)$ is the gamma function of x, defined by:

$$\Gamma(x) \;=\; \int_0^\infty s^{x-1}e^{-s}ds$$

In 1974, Prentice showed that this can be parameterised to a form more suitable for (automatic) computation [284], by setting:

$$\begin{aligned}
\lambda &= \tfrac{1}{\sqrt{k}} \\
\sigma &= \tfrac{1}{\beta}\sqrt{k} \\
\mu &= ln(\theta) + \tfrac{1}{\beta}ln(\tfrac{1}{\lambda^2})
\end{aligned}$$

for $-\infty < \mu < \infty$, $\sigma > 0$ and $-\infty < \lambda < \infty$, giving:

$$f(t) \;=\; \begin{cases}
\frac{|\lambda|}{\sigma t}\frac{1}{\Gamma(\frac{1}{\lambda^2})}e^{\left[\frac{\lambda-\frac{ln(t)-\mu}{\sigma}+ln(\frac{1}{\lambda^2})-e^{\lambda-\frac{ln(t)-\mu}{\sigma}}}{\lambda^2}\right]} & \text{if } \lambda \neq 0 \\[2ex]
\frac{1}{t\sigma\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{ln(t)-\mu}{\sigma})^2} & \text{otherwise}
\end{cases}$$

The generalised gamma function includes exponential, gamma (and hence Erlang and Pearson 5), lognormal and Weibull distributions as special cases: if $\lambda = 1$ and $\sigma = 1$ the distribution becomes an exponential; if $\lambda = \sigma$ the distribution becomes a gamma (and Erlang if this value is integral, or Pearson 5 with the appropriate inversion); if $\lambda = 0$ the distribution becomes lognormal; if $\lambda = 1$, (Weibull parameter) $\beta = 1/\sigma$ and $\eta = ln(\mu)$ then the distribution behaves like a Weibull distribution for $\sigma > 1 \equiv \beta < 1$, $\sigma = 1 \equiv \beta = 1$ and $\sigma < 1 \equiv \beta > 1$. While the generalised gamma function is not often used in final-stage models because of its complexity, its ability to behave like other more common distributions can be helpful in determining which distribution should be used for a particular set of data.

Consider once more the PDFs in Figures 14, 16, 19, 20, 23 and 24. The data from a broad selection of those measured distributions were input into a statistics program called Stat::Fit [149] and analysed to determine which of the analytical distribution(s) listed above could best describe them. Stat::Fit is very simple to use, in particular via its Auto::Fit function which automatically fits data to different distributions, provides an absolute measure of each distribution's acceptability and ranks the results. Stat::Fit uses Maximum Likelihood Estimation (MLE) [101] for parameter estimation, which determines the parameter values that maximise the probability of obtaining the sample data. MLE is considered the most accurate parameter estimation method for 100 or more samples, although it suffers from convergence problems for 3-parameter fits of nearly exponential distributions [226, 346]. Stat::Fit uses χ^2, Kolmogorov-Smirnov and Anderson-Darling tests to provide goodness of fit measures. Notably, the Kolmogorov-Smirnov test provides the best metric over a wide range of distributions and the Anderson-Darling test provides the best metric for heavy-tailed distributions [18]. Importantly, it is known that all of these tests can become too sensitive for a large number (say more than 1000) data points and thus occasionally reject proposed distributions that in reality provide useful fits [149].

Firstly, Stat::Fit was used to perform independence tests on the input data using its runs test and autocorrelation test facilities. At a significance level of 0.01, which means that there was a 1% chance of incorrectly rejecting the correct hypothesis, these tests indicated that the input data was gathered from a stationary process. This is an important result because it validates the crucial assumption (made in Section 3.4.2) that the performance of individual communication operations can be treated independently. Then, for each investigated distribution, 500 (i.e. greater than 100 but less than 1000) data points were randomly selected from the 10,000 or 100,000 measured samples available (see Table 1) and used for fitting. Auto::Fit MLE and goodness of fit analyses were performed with a significance level of 0.01. Once potential fits were identified (by the Kolmogorov-Smirnov test for approximately normally distributed data and the Anderson-Darling test for heavily skewed data) their p-values (i.e. the probability that another sample would be as unusual given that the fit under consideration was appropriate) were manually examined to determine the most promising models. Samples of the fits that were obtained for the three machines being examined in this thesis, covering small to large messages being communicated between 32x1 processes, are shown in Figures 40-45.

Significantly, the results support a common interpretation for the behaviour of point-to-point message-passing performance on all three of the machines that were examined. The performance distributions observed for small messages (for example the 512 byte results shown in Figure 40 for Perseus, Figure 42 for Orion and Figure 44 for the APAC NF) are essentially normal-shaped, although they necessarily have a bounded lower limit. The best fit in each of these cases was provided by a Weibull distribution with a shape parameter near 3.6. Importantly, however, the standard deviation for each of these distributions, i.e. 58μs on Perseus, 5μs on Orion and 1.5μs on the APAC NF, is comparatively small: they are all about half of the minimum message latency for a zero byte message on the same system. These normal-shaped distributions are consistent with random rather than contention delays, for example caused during context switching, polling for message arrivals or physical transmission. Also worth noting, for reasons that will be made clear shortly, is that the Pearson 5 distribution provides a usable, if somewhat crude, alternative to the Weibull distribution for these small messages.

For larger messages (for example for the 16 Kbyte results for Perseus in Figure 41, the 28 Kbyte results for Orion[3] in Figure 42 and the 16 Kbyte results for the APAC NF in Figure 44), where contention is more prevalent, the Pearson 5 distribution provided by far the best fit for message-passing time on all three machines. In comparison, Weibull (or other) distributions could not be used because they were too heavy-tailed and lacked the peakedness to fit the observed data well. It seems, therefore, that (for the communication networks examined in this thesis) the best quantitative explanation for performance variation in message-passing time under a normal contention level lies in the roots of the Pearson 5 distribution; i.e. performance variation occurs as the result of a transmission process that has a high chance of succeeding in minimum time, yet has a small chance of being continually delayed.

Where multiple processes on an SMP node are competing for a shared network interface, as discussed in Section 4.5.1, access contention to the shared network interface causes increased variance in message delivery times. For smaller messages, the normalised PDFs for message transmission time broaden by the number of processes per node n and reduce in amplitude proportionally. For larger messages, the distributions broaden in duration and reduce in amplitude by a factor less than n, depending on the extent to which message-passing startup times and packetisation losses are amortised. Because the

[3]Results for 28 Kbyte messages were used instead of 16 Kbytes to avoid a noisy distribution caused by Orion's poorly tuned rendezvous transmission protocol at that size, as described in Section 4.5.1.

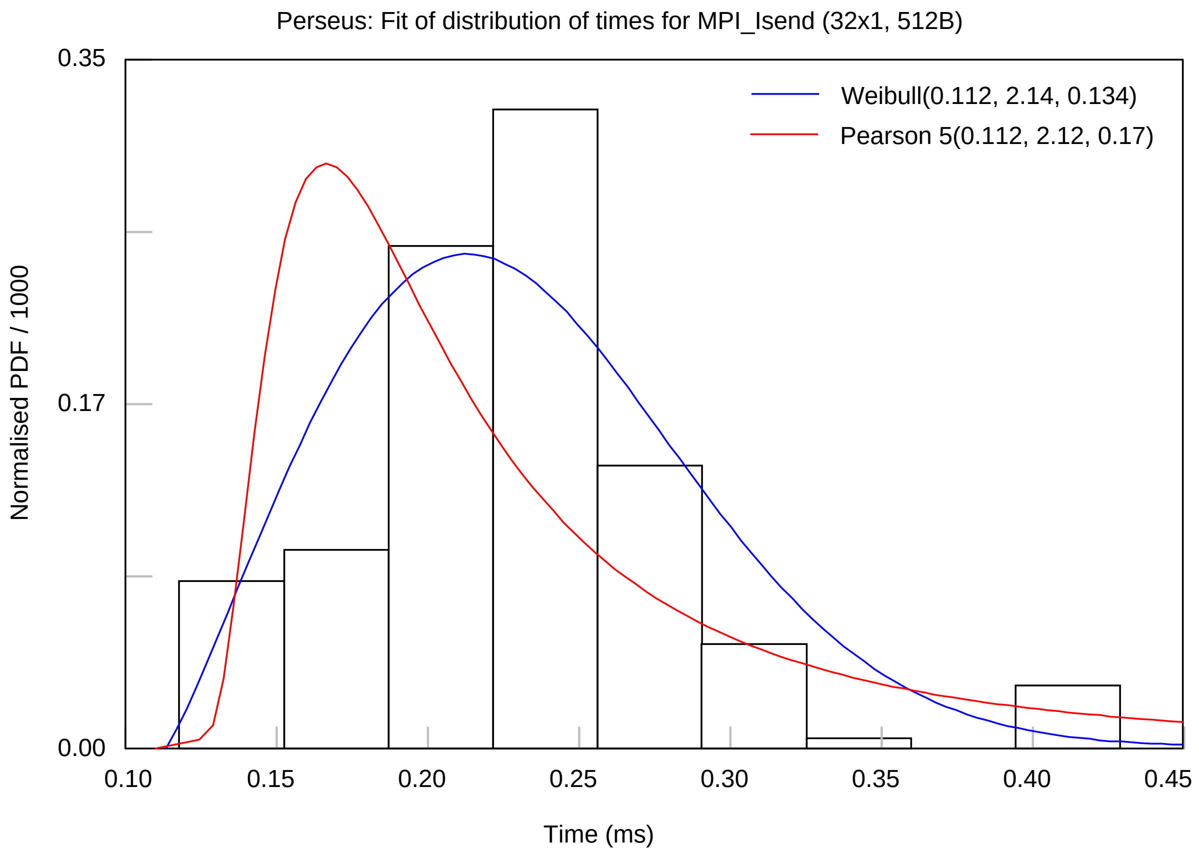


Figure 40: Pearson 5- and Weibull-fitted performance profiles for 512 byte `MPI_Isend` messages with 32x1 processes on Perseus.

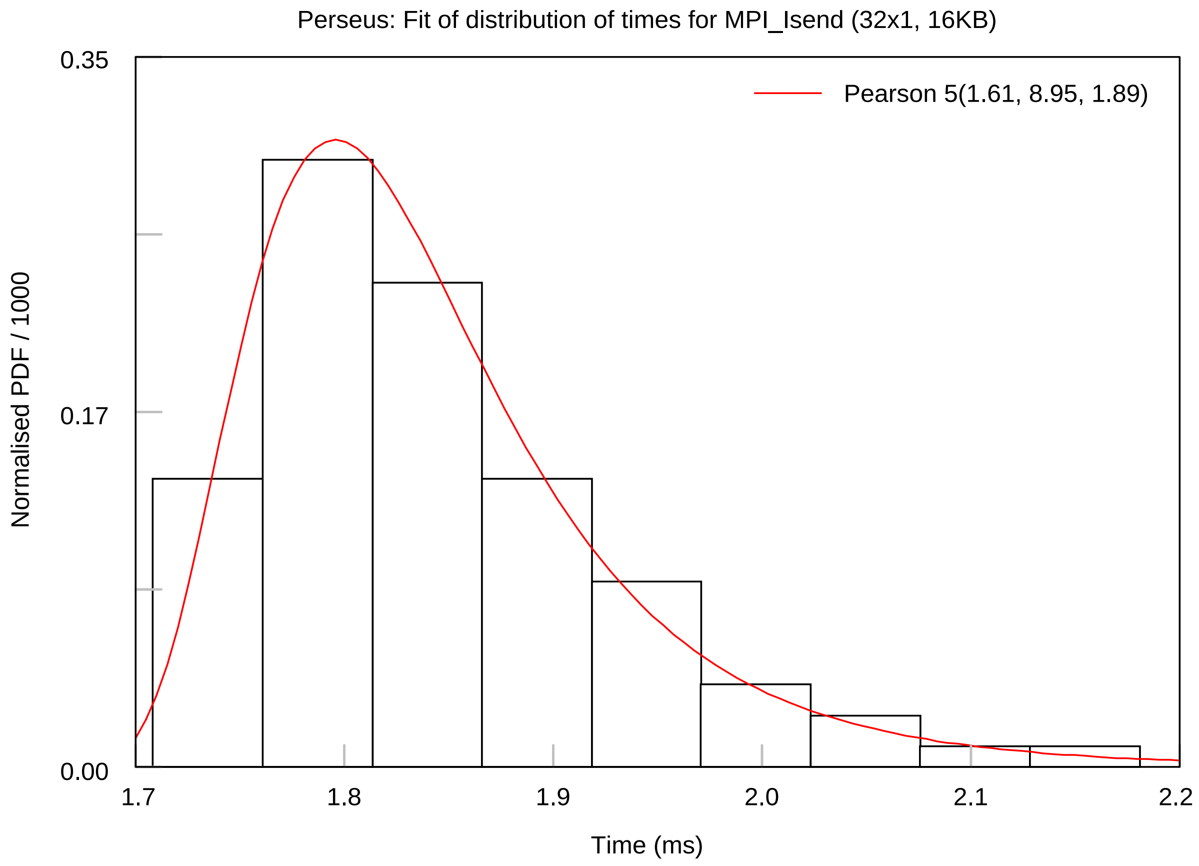


Figure 41: A Pearson 5-fitted performance profile for 16 Kbyte `MPI_Isend` messages with 32x1 processes on Perseus.

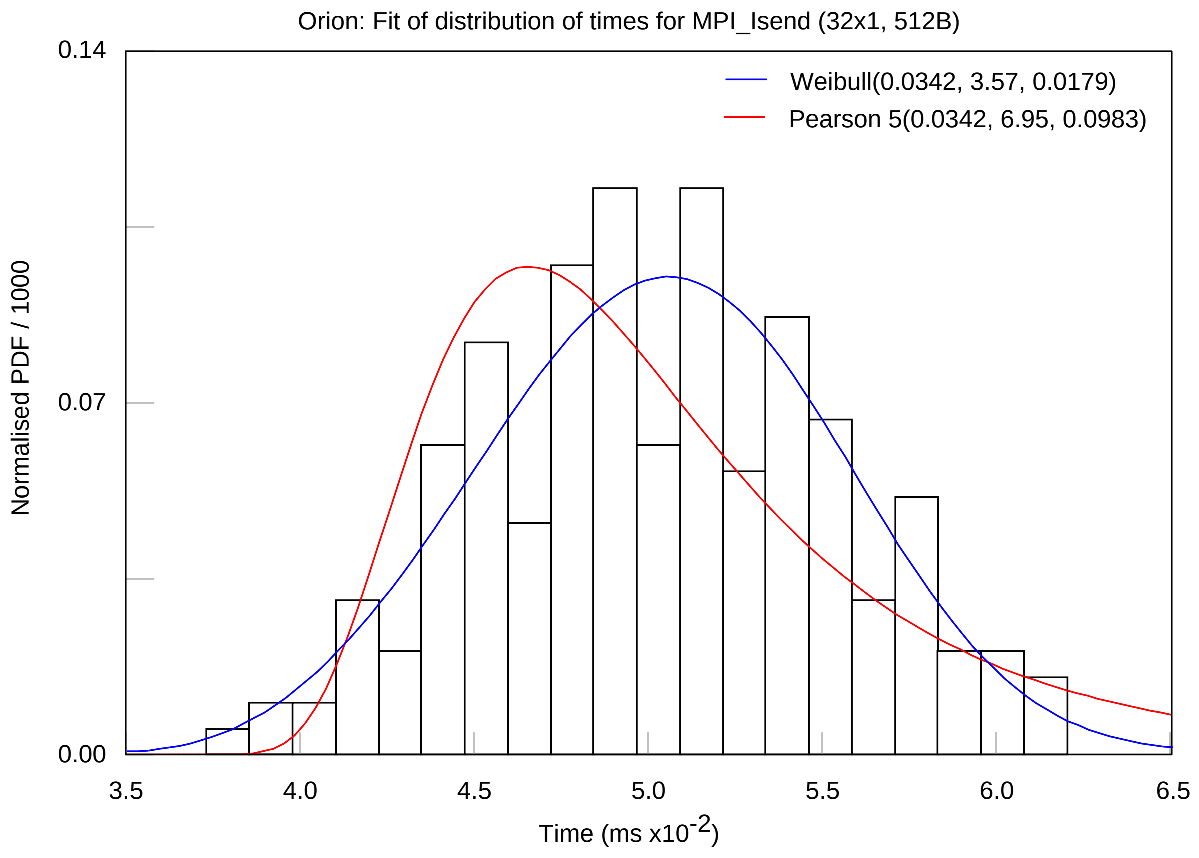


Figure 42: Pearson 5- and Weibull-fitted performance profiles for 512 byte `MPI_Isend` messages with 32x1 processes on Orion.

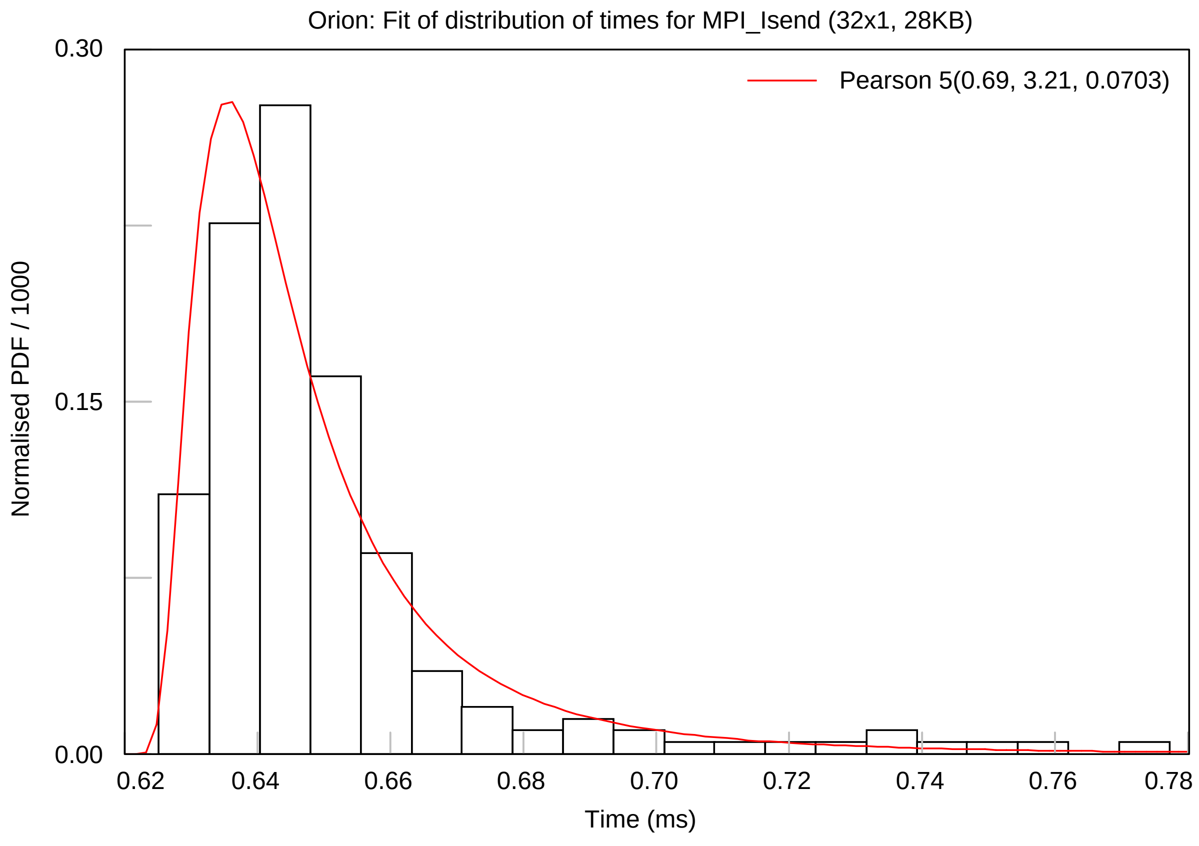


Figure 43: A Pearson 5-fitted performance profile for 28 Kbyte `MPI_Isend` messages with 32x1 processes on Orion.

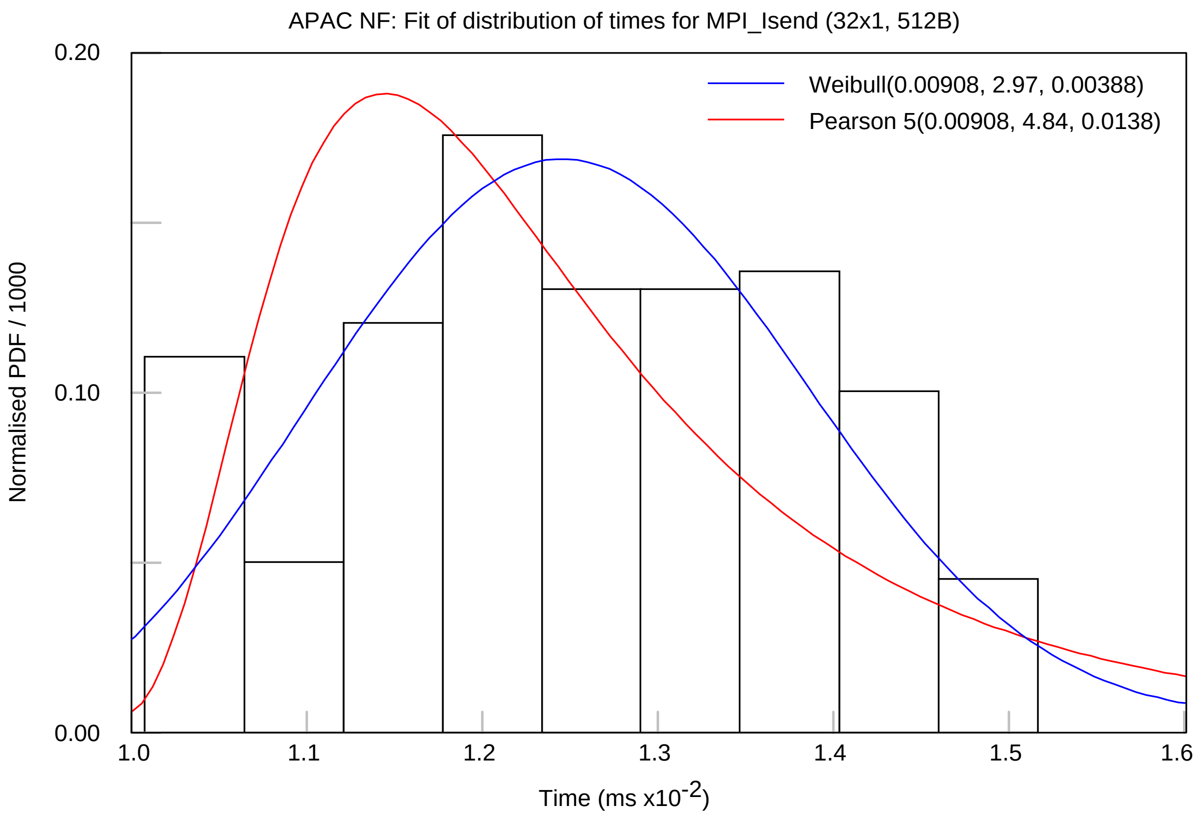


Figure 44: Pearson 5- and Weibull-fitted performance profiles for 512 byte `MPI_Isend` messages with 32x1 processes on the APAC NF.

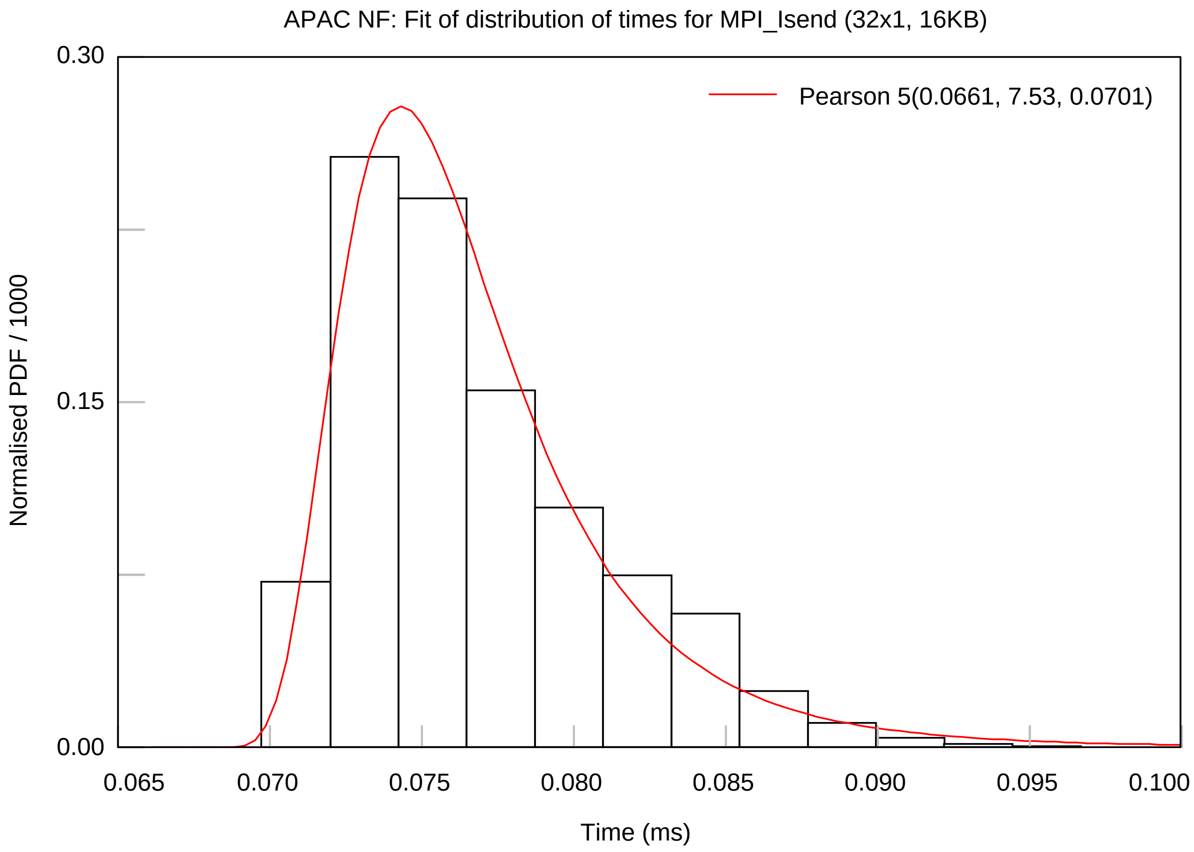


Figure 45: A Pearson 5-fitted performance profile for 16 Kbyte `MPI_Isend` messages with 32x1 processes on the APAC NF.

amount of amortisation that will occur is very difficult to predict quantitatively, benchmark results for different numbers of communicating processes per SMP node should be conducted and analytically modelled separately. Qualitatively, however, the overall effect of this is that, in the case of multiple communicating processes per node, the (quasi normal-shaped) Weibull model remains more accurate up to larger message sizes.

Although it is difficult to determine exact bounds on the message sizes and contention levels for which Pearson 5 distributions provide superior fits to the observed data (compared with a Weibull distribution), it is certainly true that the Pearson 5 distribution provides by far the most accurate fit over the broadest spectrum of conditions. Even in the few cases where a Weibull distribution would provide a better fit, the Pearson 5 remains (in the context of the PEVPM) a passable alternative. Therefore, the Pearson 5 process provides the most usable analytical model of point-to-point message-passing time for parallel programs. Of course, for an overall communication performance model, its parameters will need to be determined based on network type, the number of communicating processes, message size and acknowledgement policy.

In general, increasing the message size or the number of contending processes will decrease the Pearson 5 distribution's shape parameter, creating a more heavily skewed distribution. With this knowledge, a useful extension to MPIBench's functionality can be envisaged. If a robust automatic distribution fitting mechanism could be included in MPIBench, then when presenting results across message size, rather than only plotting minimum or average times (for example as done in Figure 12), or using 3D plots of complete performance profiles (for example as done in Figure 14), 3-parameter plots of location, shape and scale factors could be made. The location parameter plot would look the same as traditional minimum latency plots. Trends in the shape and scale parameters, however, would provide valuable, easy to understand information about how contention effects scale with message size and the number of communicating processes. The extension of MPIBench in this way was outside the scope of this thesis, but would be a good avenue for further research. Even further afield, a model that could predict appropriate Pearson 5 parameters directly from machine characteristics rather than by measurement would be very valuable. Then, PEVPM communication submodels could be simply obtained through computation, rather than via extensive benchmarking. Obtaining such a mapping between machine characteristics and Pearson 5 parameters, however, is likely to be very difficult. It would require developing accurate models of a parallel machine's communication performance from potentially very fine-grained sources of hardware design information, together with extensive simulations over a wide range workload descriptions. At the extreme, VHDL-level descriptions of network hardware may be required, although simpler models based on more coarse-grained hardware characteristics may be usable under certain circumstances. In either case, accurate hardware-based models of a parallel machine's communication performance will increase the level of simulation detail for the communication submodels *within* the PEVPM framework, in the same way that the PEVPM model has provided an increased level of simulation detail for message-passing programs.

4.8 Stability and Interference

As explained in Section 4.3.4, MPIBench records outlying time measurements for the performance of individual message-passing operations. These outliers are indicative of two qualitatively different phenomena from the normal point-to-point message-passing processes examined earlier in this chapter. The main causes of these outliers are: 1) communication protocol timeouts; and 2) spurious interference from unrelated operating system services. Both can result in extremely long delays compared to the normal time required for message-passing operations. Importantly, as the number of processes that are involved in a parallel computation is increased, the chance that one of those processes will be delayed also increases (according to the binomial distribution). Hence, these long delays have a multiplicatively large effect for parallel programs, especially those with frequent synchronisation, because delaying any one process will also retard the progress of all the others (thus introducing load imbalance on a small scale). This can be a major cause of slow-down in large parallel programs. In particular it is, in large part, responsible for the characteristic tail off of parallel application speedup plots where it is not expected by Amdahl's Law (see Section 2.2) – especially on Beowulf-type clusters, where using TCP/IP for communication and running an unnecessary gamut of (standard) system services is common. A secondary reason for this tail off, noticeable only in extremely large parallel systems, is variance in normal message-passing time, which introduces load imbalance on an even smaller scale.

Relatively little research has been published about outliers in message-passing times, probably due to the lack of sufficiently powerful measurement tools like MPIBench. Notably, however, Mraz [248] and Tabe *et al.* [340] (see Section 2.16) traced the presence of outliers in message-passing times on an IBM SP2 to the time required by the operating system to process interrupts and page faults. As a result of Mraz's work, a number of common AIX operating system interrupts were ganged together so they would occur simultaneously on all processors, thereby removing the performance degradation caused by unsynchronised stalls. It is worth noting that no similar efforts appear to have been undertaken to synchronise interrupts or other system services on Beowulf-type clusters. Schaubschlager [309] (see Section 2.26) recorded a large number of slow message-passing times on nCUBE-2 and Origin 2000 hardware, especially in the presence of heavy network load, which he attributed to contention effects. Finally, Loncaric [220] reported poor message-passing performance using Linux 2.2.12's TCP/IP stack, which he traced to a flaw in its message acknowledgement mechanism.

As explained in Section 4.4.1, the experiments carried out in this thesis were done on machines that were running the bare minimum of system services. As a result of this, almost all of the spurious interruptions that were measured can be traced to communication protocol timeouts. Most communication protocol timeouts are required in order to deal with flow control, communication over an unreliable channel and congestion control. The Myrinet and QsNet protocols are highly-tuned to provide good message-passing performance, hence the number of outliers in message-passing times on Orion and the APAC NF was extremely small. The TCP/IP/Fast Ethernet combination, however, provides a general purpose communication network that is not as highly-tuned for message-passing in parallel programs and thus substantial number of outliers in message-passing times were observed on Perseus. Therefore, it is appropriate to briefly examine TCP/IP's acknowledgement policy, retransmit timeout mechanism and congestion avoidance/slowstart algorithm – which collectively affect how TCP provides reliable communication, controls data flow and copes with congestion – to determine which TCP/IP protocol timeouts significantly affect message-passing performance.

In order to provide reliable communication, TCP/IP requires that all data messages (also known as segments in TCP/IP parlance) must be acknowledged by the receiver, so that the sender is aware of their safe delivery. To facilitate this, every data message is augmented with an ordinal sequence number (and some other information; the complete package is called an IP datagram). Then, upon receipt of a data message, the receiver sends a special acknowledgement message containing that sequence number back to the original sender. A number of techniques, however, are used to reduce the overheads associated with sending these acknowledgement messages. Firstly, an acknowledgement can be piggy-backed on to any data message that the receiver is itself sending to the sender. Thus, only one message header must be constructed by the sender, processed by network hardware and evaluated by the receiver. Secondly, the receiver may choose to delay an acknowledgement message, in the the hope that it can be piggy-backed on to a later data message. Thirdly, acknowledgements can be grouped together into a cumulative acknowledgement, so a receiver may also decide to delay sending an acknowledgement message until several are pending from the receipt of data messages.

If, however, it is not possible to piggy-back or cumulatively acknowledge a message that has arrived, an acknowledgement message must eventually be sent anyway. Modern versions of the TCP specification mandate that the delayed acknowledgement interval must be less than 500ms and also recommend that the receiver should not delay an acknowledgement if two or more unacknowledged, full-sized data messages have been received [47, 338]. Otherwise, a TCP stack may send acknowledgements as it sees fit, and a number of different TCP acknowledgement strategies are in common use [10, 273]. Finally, most MPI implementations (including MPICH 1.2.0, used on Perseus) set a special `TCP_NODELAY` socket option, which, in theory, forces immediate acknowledgements. Thus, synchronous message-passing operations (for example) should only be delayed by the time required to transfer the message data plus the time required to transmit a (minimum-sized) acknowledgement message. However, Loncaric [220] showed that the TCP/IP stacks in some versions of Linux suffer from regular stalls that delay every n^{th} "small" message by 1-2 jiffies (which are defined by the 100Hz system clock). For example, he showed that Linux 2.2.2 delays every 41^{st} message smaller than 125 bytes by 10-20ms. Therefore, while acknowledgement delays should not theoretically play a major part in message-passing performance, some imperfectly implemented TCP/IP stacks can cause substantial acknowledgement delays in practice.

For transmission over a physical network, each internet-layer IP datagram must be split into a number of link-layer packets, each up to Maximum Transmission Unit (MTU) bytes in size, and encapsulated along with a link-layer header into what is called a frame. For example, in the case of MPI/TCP/IP/Ethernet communication, MPI messages are eventually split into a number of Ethernet frames, which are transmitted through the network and reassembled at the destination host. The transmission of Ethernet frames is not guaranteed; frames can be dropped by network switches if the switches run out of intermediate buffer space or packet processing resources. In an attempt to prevent this congestive loss, most modern TCP/IP stacks use a pair of interrelated algorithms that limit the amount of data that they inject into a network.

The first of these is the *slow start* algorithm [11, 189, 338], which is applied at the beginning of a TCP connection or after a retransmit timeout (discussed in next paragraph) has occurred. TCP senders maintain a congestion window of *cwnd* bytes, which constrains the amount of unacknowledged data that a sender may have injected into the network. Slow start initialises this window to one (Maximum Sized) Segment (MSS), and increases it by one (maximum sized) segment for every segment that gets acknowledged. This causes the window to increase exponentially in size over time because, as the window opens up, more and more data are permitted to be sent. This happens until the receiver's allowable window size (which is advertised during connection establishment) is reached, congestive loss is detected or measures to preempt congestive loss are invoked. In the case that a message is lost, a slow start threshold *ssthresh* is set to half of the current congestion window, then the congestion window is reset to one (maximum sized) segment and slow start is performed until the slow start threshold is reached. At this point, the *congestion avoidance* algorithm [11, 189, 338] is set in motion. During this phase, the congestion window is increased more conservatively than during slow start; for each acknowledged segment, the congestion window is increased by $MSS^2/cwnd$ bytes (up to the receiver's advertised window size), which results in a linear increase in the window over time. This slower increase in window size prevents congestive loss from being reached too quickly and thus reduces the amount of time spent transmitting with a small (poorly performing) window size. Ideally, a segment will never be dropped by the underlying network and hence the congestion window will increase to, and remain stable at, the receiver's advertised window size – thus maximising performance. Unfortunately, however, as mentioned above, message delivery cannot always be guaranteed.

When a frame is dropped, it becomes impossible to reconstruct its parent IP datagram at the receiver. Consequently, the receiver will not generate an acknowledgement for the data message that the IP datagram contained. Eventually, based on the non-arrival of this (never created) acknowledgement message, the sender will infer that its original message was lost and will retransmit it. The length of time that a TCP/IP stack will wait before retransmitting a (supposedly) lost message (since the message may only have been lengthily delayed) is governed by its Retransmit Time Out (RTO) value, or more precisely, by an algorithm that computes that value. If the retransmit timeout is too short, segments will timeout prematurely, leading to unnecessary retransmissions. If it is too long, the TCP/IP stack will be slow to respond to lost messages.

The original TCP definition [281] suggested setting the retransmit timeout value as a function of the (estimated) Round Trip Time (RTT) required to send a segment and receive an acknowledgement for that segment. The round trip time should be estimated by keeping a running average of observed round trip times (not including any that involved retransmission [197]), giving a Smoothed Round Trip Time (SRTT) at time $i+1$:

$$SRTT_{i+1} = \alpha.SRTT_i + (1-\alpha)RTT$$

where the smoothing factor $\alpha \in (0,1)$ should be set at about 0.8 or 0.9. The retransmission timeout should be set according to:

$$RTO_{i+1} = min\{UBOUND, max\{LBOUND, (\beta.SRTT_{i+1})\}\}$$

where $UBOUND$ is an upper bound on the timeout, $LBOUND$ is a lower bound on the timeout and $\beta > 1.0$ is a delay variance factor (with a recommended value of 2.0). In 1988, Van Jacobson [189] proposed replacing $\beta.SRTT_{i+1}$ in the above formula with $SRTT_{i+1} + 4D$, where D represents the measured variance in round trip times, smoothed in the same manner as round trip times; this improvement has become standard in modern TCP/IP stacks. In addition to the above calculations, if retransmission is unsuccessful, hence leading to further retransmission, the last $SRTT_{i+1} + 4D$ value that was used must be doubled; this causes exponential backoff when multiple successive retransmissions are required. Finally, the current TCP/IP standard suggests using a $LBOUND$ of 1 second and an $UBOUND$ of 60 seconds [271]. While this is essential for avoiding congestive collapse in the Internet, it is inappropriately long for modern local area networks, which have sub-millisecond round trip times. Notably, Linux 2.2 kernels use a $UBOUND$ of 200ms, although this is still far too long on local area networks [307]. The performance of parallel programs using standard TCP/IP communication protocols over reasonably lossy (i.e. Ethernet-based) local area networks could be significantly improved by substantially reducing the lower bound on retransmit timeout values; this is being trialled in some experimental clusters [221] (but is not done on Perseus).

Obviously, any retransmission timeouts that occur will have a significant impact on communication performance because of the retransmit timeout itself (especially if exponential backoff is required) as well as the time required to retransmit the lost data. More insidious still, is that every time a retransmit timeout occurs, the congestion window is reset and the slow start algorithm is started. This is particularly problematic for parallel programs, whose bursty communication can cause frequent timeouts [357]. When this is the case, the congestion window will constantly cycle, unable to settle on or just below the size that maintains optimal throughput and avoids congestive packet loss. In order to combat these problems additions have been made to TCP/IP over the years, resulting in a number of compatible (but with varying performance) TCP/IP implementations [48,117,129,130,224,225,244]. The most standard additions are *fast retransmission* for when multiple segments are lost from one congestion window and *fast recovery*, which advances the congestion window more aggressively (but also safely) after a retransmission. The details of these algorithms are beyond the scope of this thesis.

Regardless of the exact acknowledgement and retransmission policies of any particular TCP/IP dialect, what is particularly important for modelling the performance of parallel programs is under what circumstances and how frequently messages are lost, and what length of delay is caused. Usually, it should be possible to easily distinguish the number of messages that have suffered retransmit timeouts and the duration of those timeouts (for any particular MPIBench test) by examining the distribution of observed message-passing times. Messages that were transmitted successfully the first time will be grouped in a large primary distribution, messages that had one retransmit timeout will be grouped in a smaller secondary distribution, messages that had two retransmit timeouts will be grouped in an even smaller tertiary distribution (according to exponential backoff), etc.

The quantitative effects of slow start, congestion avoidance, fast retransmission and fast recovery will be far more difficult to observe; quite likely, this could only be achieved through detailed examination of particular TCP/IP stacks and packet-level traces of specific communication patterns. Once again, this is outside the scope of this thesis, but it is an area of active research (although usually from the perspective of, for example, Internet-wide, bulk or satellite data transfers, rather than from the perspective of the extremely bursty communication that occurs in parallel programs). In terms of this thesis, however, these effects are secondary in importance compared to the retransmit timeouts themselves, and will only become significant when a very large percentage of messages are affected by retransmit timeouts. This should not occur in a parallel computing environment; if it does, there are serious problems afoot that should be dealt with to avoid the excessive message loss. Finally, therefore, only the frequency and rough duration of delays caused by retransmission timeouts need to be examined in this thesis; this information can be graphed for specific operations. Although a small number of retransmission timeouts were observed during the point-to-point operations carried out for this chapter, their effect only becomes truly significant during group communication, and in particular during `MPI_Bcast` and `MPI_Alltoall` operations. Therefore, a quantitative analysis of retransmit timeouts is left until those operations are discussed in the following chapter.

4.9 Summary

The work described in this chapter grew out of the PEVPM's need for extremely accurate characterisations of the communication performance of message-passing operations on parallel computers. Because of the deficiencies of existing techniques, described in Section 4.2, a new tool for benchmarking low-level MPI operations called MPIBench was constructed, according to the design described in Section 4.3. In addition to providing the standard functionality available in existing benchmarks, namely the ability to test the performance of many operations using different message sizes and in some cases using different communication patterns, MPIBench provides extra functionality to overcome some important inadequacies of these existing techniques. Firstly, MPIBench is topology-aware, and is specifically designed to ensure meaningful results on clusters of SMP nodes. Secondly, MPIBench uses an accurate global clock to measure the performance characteristics of all of the processes in an MPI program rather than simply measuring the time required for round-trip messages or collective operations at at a single process. This is especially important for measuring the complex performance characteristics of collective operations, as will be seen in the next chapter. Thirdly, and crucially, the extremely fine resolution of the global clock in MPIBench allows timing data on individual MPI operations to be obtained, rather than the average time over a large number of repetitions of an MPI operation. This gives MPIBench the unique ability to accurately quantify the performance variability of MPI operations due to contention, which it does by producing probability distributions of the performance of everything that it measures. Performance characterisations of MPI operations at this level of detail are central to the PEVPM performance prediction technique that was described in the previous chapter. In addition to this, such detailed performance information could also be of great benefit during the implementation and testing of MPI libraries, systems software, and hardware for parallel computers.

In order to obtain empirical data for the validation process of the PEVPM system in Chapter 6, MPIBench was used to benchmark the MPI communication performance of three large parallel computers, as described in Section 4.4. The performance results for point-to-point communication were presented in Sections 4.5 and 4.6, while the results for collective communication will be discussed in the next chapter because of their substantially different nature. In summary of the major results obtained for point-to-point operations, it was demonstrated that performance variability due to contention can be very significant; these effects are especially prevalent when large numbers of processes on distinct nodes are communicating concurrently and/or when processes in the same SMP are communicating concurrently, and even more so when transmitting large messages and/or using bidirectional communication.

Finally, an analysis of the probability distributions describing point-to-point message-passing performance in the presence of contention was carried out in Section 4.7, which showed that the Pearson 5 distribution can best explain the results observed earlier in the chapter. In the future, it would be valuable to extend MPIBench so that it can automatically fit measured data from point-to-point tests to Pearson 5 distributions and plot the resultant fit parameters across a range of message sizes and contention levels. While plots of the Pearson 5 location parameter will closely resemble existing (minimum) latency graphs, the accompanying shape and scale parameters will present important yet easily digestible information about performance in the face of contention. Section 4.8 extended Section 4.7 via a short study of systematic outliers that were observed in performance measurements. In particular, the causes of these very slow message-passing times were identified – mainly operating system interruptions and retransmission of lost messages – and the adverse performance effects that they impose on parallel programs were discussed.

Chapter 5

Benchmarking Collective Communication

5.1 Introduction

This chapter builds on the work in the previous chapter, which used MPIBench to accurately characterise the performance of MPI-based point-to-point communication on three parallel computers. This chapter presents more MPIBench results, this time to accurately characterise the performance of MPI-based collective communication on the same three machines. As the discussion in Section 3.4.2 detailed, most MPI implementations create collective communication operations using a number of point-to-point operations. It is clear, then, that "it is important to understand the behaviour of the lower-level communication primitives upon which a higher-level communication is based in order to gain insight into the behaviour of the higher-level communication primitives" [340]. In addition to this, however, the performance of collective communication is also heavily dependent on how point-to-point operations are combined to provide the collection operation. While the previous chapter showed how different protocols could be used to optimise the performance of point-to-point messages for different message sizes, there is far more scope for using different algorithms to optimise the performance of collective communication. Indeed, optimising the performance of collective communication remains a very active field of research [184, 198, 318, 341, 343, 353].

Since the performance of point-to-point messages was already examined in detail in the previous chapter, the results and analyses in this chapter are mainly aimed at determining how various collective operations are constructed from point-to-point messages on each of the three machines. This information is vital for simulating the performance of collective operations, and consequently complete MPI programs, using the PEVPM from Chapter 3. In fact, each of the collective operations examined in this chapter can be considered as a miniature MPI program. Hence, explanation of the performance of collective operations using the PEVPM approach provides a useful validation of the PEVPM modelling system.

In addition to determining how point-to-point operations are combined to form collective operations for each of the three machines, the results outlined in this chapter also serve as a fall-back resource, similar to the point-to-point results from the previous chapter. In cases where underlying point-to-point models for collective operations are either nonsensical (for example for hardware-based collective operations) or unable to be determined, the benchmark results in this chapter can be used as empirical performance models for larger PEVPM simulations.

5.2 Results for `MPI_Bcast`

The most conceptually simple MPI collective routine is `MPI_Bcast`, which broadcasts data from a designated root process to all the others. It is an important operation, because many parallel algorithms rely on propagating copies of certain data to all processes involved in a computation. Portable (software-based) versions of the `MPI_Bcast` operation are almost always implemented using a number of point-to-point operations. However, because of the importance of the `MPI_Bcast` operation in almost all parallel algorithms, many high-end parallel machines provide special hardware and custom `MPI_Bcast` routines to achieve superior performance.

The MPICH MPI implementation that was used for benchmark tests on Perseus does not (with one exception noted below) take advantage of the hardware broadcast capability of Fast Ethernet, but provides only a software-based `MPI_Bcast`. This software-based broadcast mechanism employs a collection of point-to-point routines that are arranged in a binomial tree, rooted at the process that initiates the broadcast. Sending the data to be broadcast using point-to-point messages by traversing the tree as shown in Figure 46 provides all processes with a copy of the data. Because message-passing at each level of the tree can proceed concurrently, the longest communication path is $\lceil \log_2 n \rceil$ messages. Furthermore, for the tests here, where the number of processes used was a power of two, all broadcast paths have this length. (If this were not the case, $2^{\lfloor \log_2 n \rfloor}$ processes would have that length $\lfloor \log_2 n \rfloor$, and the remaining $n - 2^{\lfloor \log_2 n \rfloor}$ processes would have length $\lfloor \log_2 n \rfloor + 1$). This has led to the conventional approximation (for example by Georgitsis [141]) that the message-passing time for a software-based broadcast can be modelled by $\lceil \log_2 n \rceil t_{MPI\_Send}$, where t_{MPI_Send} is itself approximated by a single-valued quantity composed of startup latency and message-size multiplied by bandwidth. This approximation has traditionally been supported by comparing `MPI_Bcast` times using large messages and a small number of processes (see Figure 48) with `MPI_Isend` times using large messages and a small number of processes (see Figure 13). Doing so shows that, for large messages, the average completion time of a broadcast very closely matches

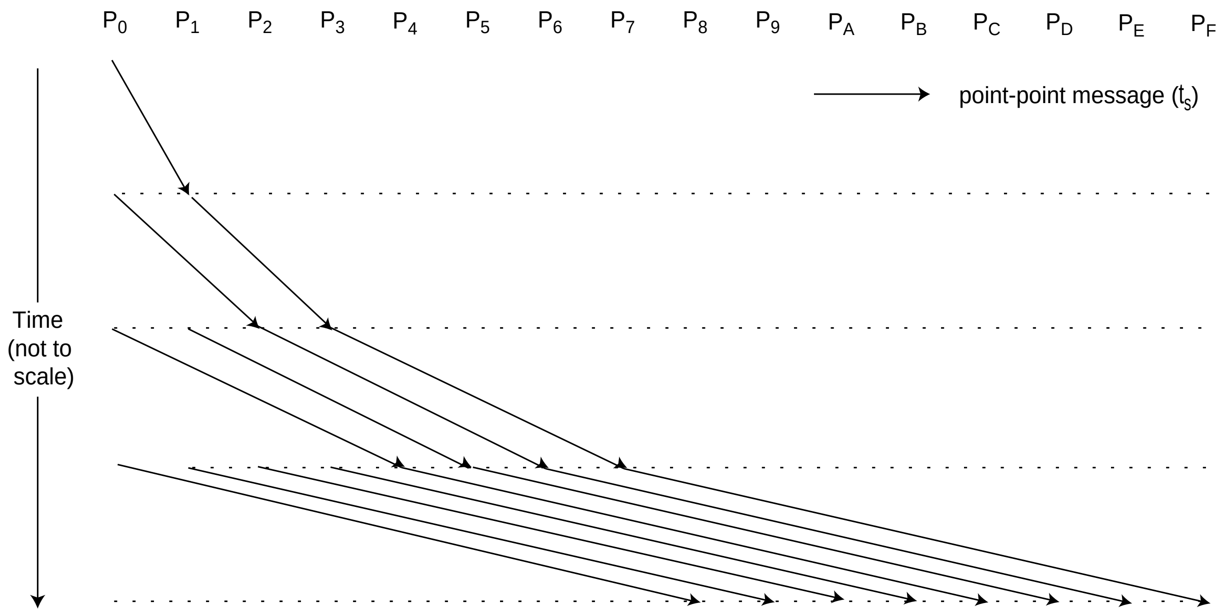


Figure 46: The binomial tree that constructs a 16 process software-based `MPI_Bcast` from a collection of point-to-point messages.

that predicted by the simple model above. For example, an `MPI_Isend` of 64 Kbytes takes almost exactly 6ms (see Figure 13), which predicts that a 4x1 broadcast of 64 Kbytes should take 12ms, an 8x1 broadcast of 64 Kbytes should take 18ms, a 16x1 broadcast of 64 Kbytes should take 24ms, all of which match the measured values.

However, Figure 48 also shows that the average time for a 32x1 broadcast of 64 Kbytes is appreciably longer than the expected 30ms – but this is slightly misleading. In that case, congestion in the network caused packet loss that resulted in a substantial number of 200ms retransmit timeouts (i.e. retransmit timeouts occurred in more than 1% of all measurements – see Section 4.3.4). This was enough to cause MPIBench's automatic outlier detection mechanism (explained in Section 4.3.4) to not identify them as such. Thus, these extremely slow `MPI_Bcast` measurements, which would not normally be treated as systematic results, led to the higher than expected average broadcast time. The more detailed results in Figure 49 shows that most broadcasts of 64 Kbytes across 32x1 nodes do in fact take roughly 30ms. The other results in Figure 48 that deviate from the expected linear trends (i.e. 16x2, 32x2, 64x1 and 64x2) can be similarly explained. However, given that the outlier events are not rare in those cases, they will have a significant impact on the performance of `MPI_Bcast` operations in practice. It is therefore important to quantify how often outliers occur for various `MPI_Bcast` operations. This is done in Figure 50, which shows that a substantial number of `MPI_Bcast` operations on Perseus incur retransmit timeouts, due to congestive packet loss.

With the important caveat of retransmit delays caused by packet loss, the average performance results for software-based `MPI_Bcast` operations on Perseus measured using

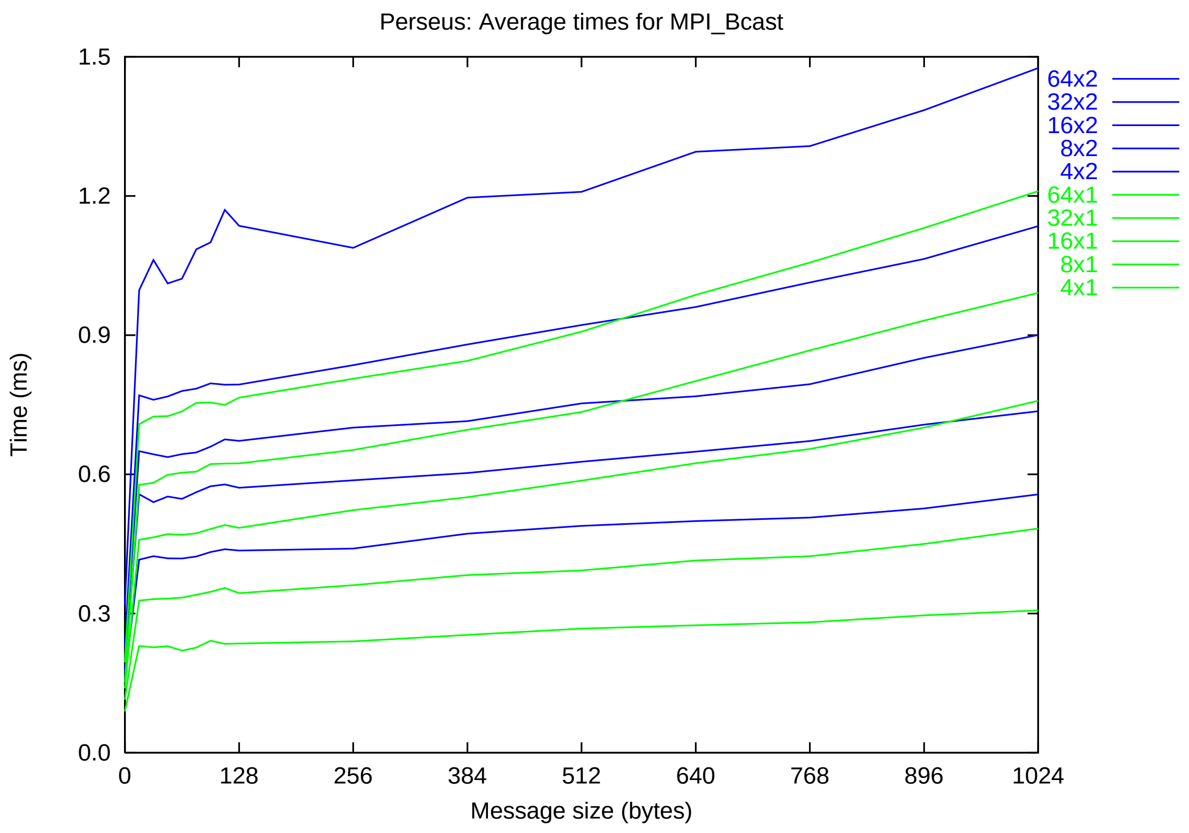


Figure 47: Average times for `MPI_Bcast` using small message sizes with various numbers of communicating processes on the Perseus.

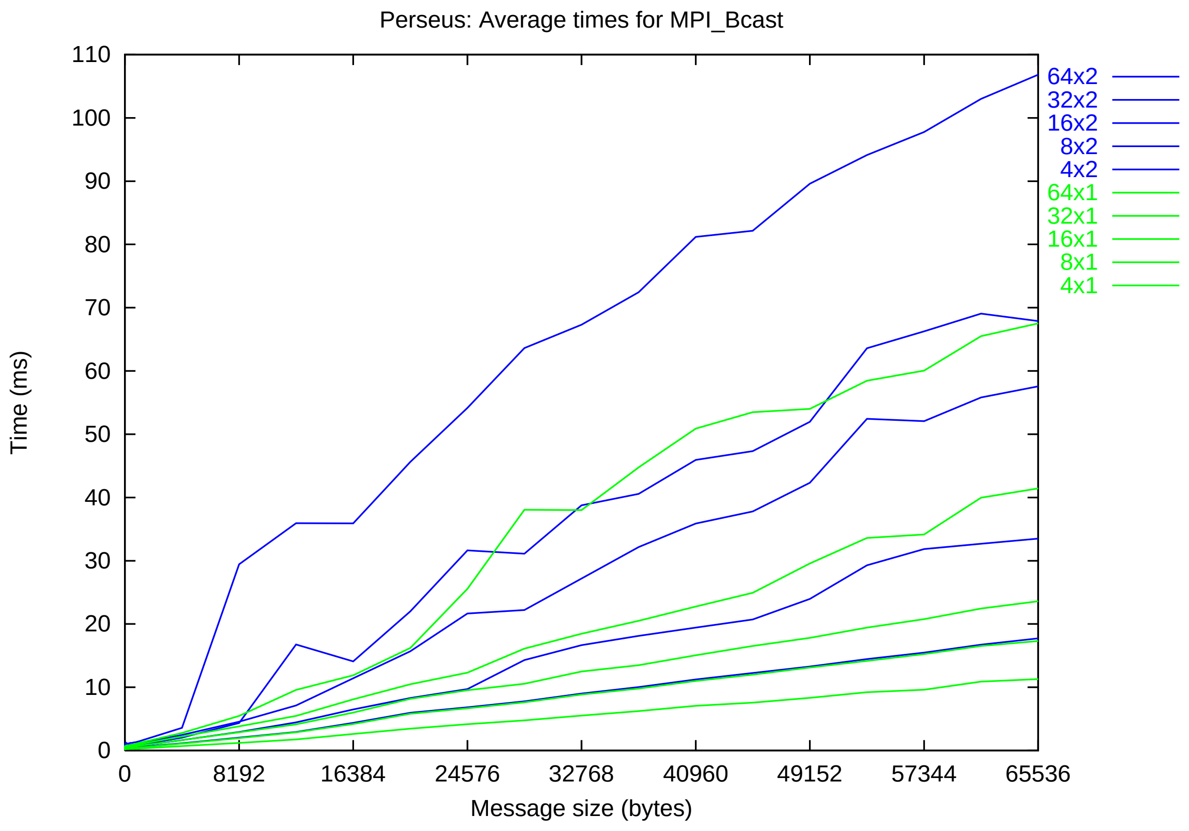


Figure 48: Average times for `MPI_Bcast` using large message sizes with various numbers of communicating processes on Perseus.

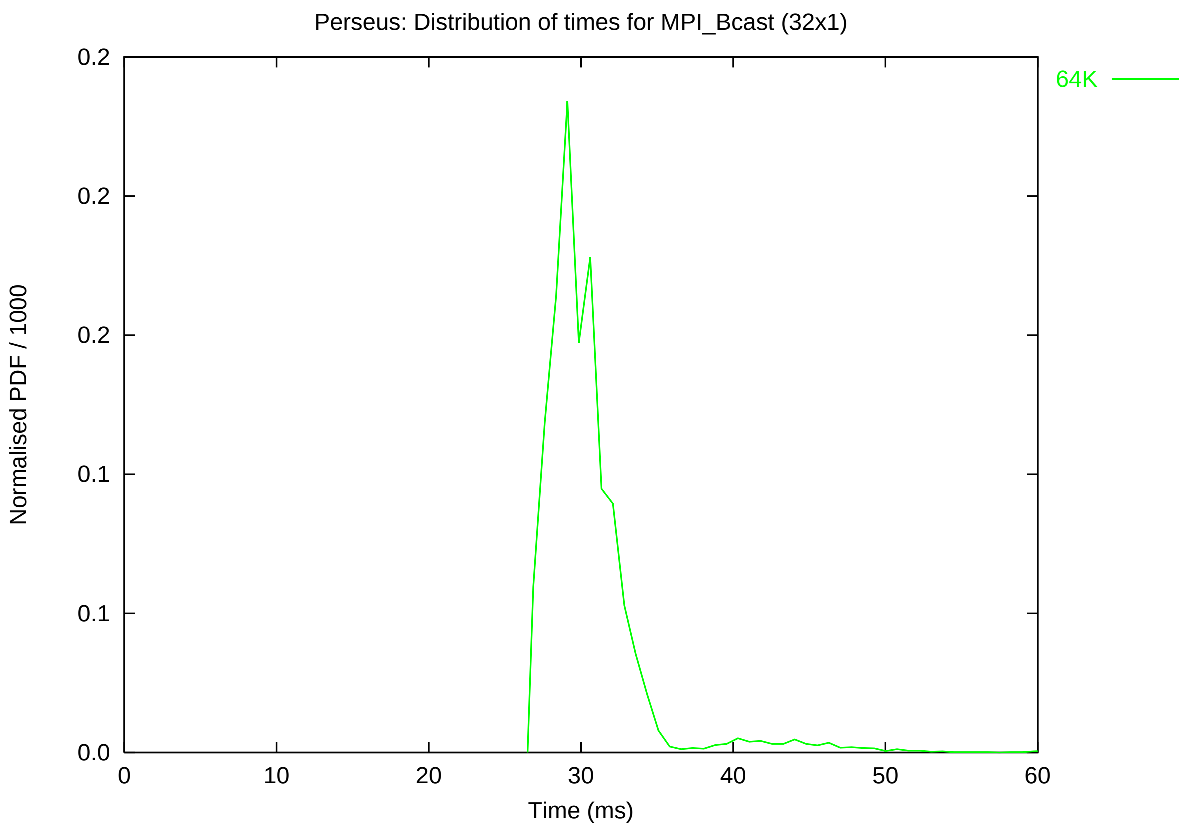


Figure 49: Sampled performance profile for `MPI_Bcast` using a 64 Kbyte message with 32x1 processes on Perseus.

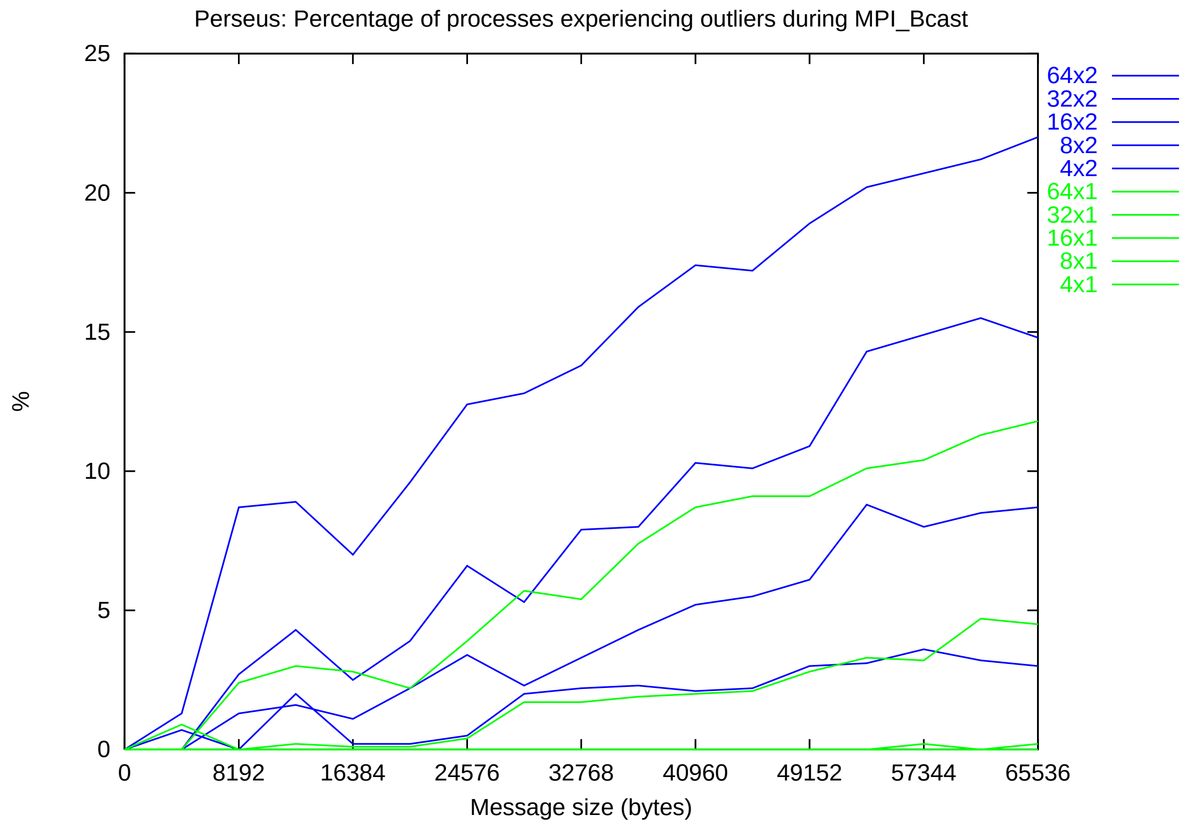


Figure 50: Percentage of processes experiencing a TCP/IP retransmit timeout during an `MPI_Bcast` to various numbers of processes on Perseus.

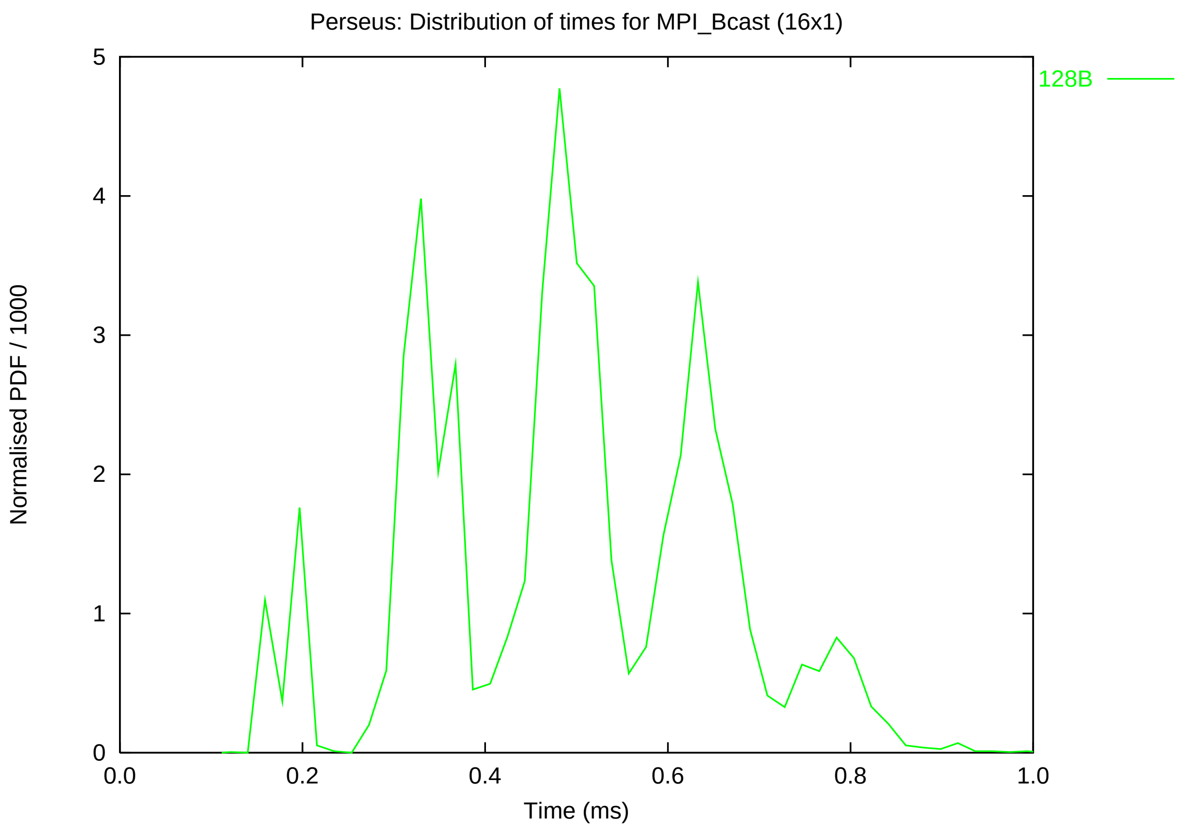


Figure 51: Sampled performance profile for `MPI_Bcast` using a 128 byte message with 16x1 processes on Perseus.

MPIBench basically agree with conventional models. However, now consider the distribution of completion times for a 16x1 process broadcast of a small message, shown in Figure 51. Clearly this complex performance profile cannot be modelled using a single-value, nor even a simple distribution like that of a point-to-point operation. Doing so would have a severe impact on the accuracy of any simulation using that model. For example, even using this relatively small number of processes (16x1), there is a two-fold difference between the minimum and average times observed, and another two-fold difference between the average and maximum times observed. Such discrepancies become more significant as the number of processes involved in a broadcast increases. Fortunately, a good understanding of the cause of this complex performance profile can be achieved, and used to construct a far more accurate model.

There are two clues about the origins of these interesting results. The first is that for the 16x1 case shown in Figure 51 there are 5 peaks rather than the more intuitively expected $\log_2 16 = 4$. Furthermore, other results showed that for 4x1 processes there were 3 peaks and not $\log_2 4 = 2$, for 8x1 processes there were 4 peaks and not $\log_2 8 = 3$, for 32x1 processes there were 6 peaks and not $\log_2 32 = 5$, etc. The second clue is in the distribution of area under each peak. In this case the areas are in the ratio 1:4:6:4:1.

Furthermore, there is a total area of $1 + 4 + 6 + 4 + 1 = 16$ units under all of the peaks, which matches the number of processes involved in the broadcast. Remembering that the results show the distribution of completion times of the `MPI_Bcast` call at all processes, these facts indicate a systematic difference in the completion time of the `MPI_Bcast` call for individual processes, based on their position in the broadcast tree. A relatively thorough investigation of the literature revealed that this effect has only been previously studied in a paper by Supinksi and Karonis [94]. In their study, Supinksi and Karonis used a LogP-based model (see Section 2.8) to demonstrate how previous benchmarking techniques mistakenly measure `MPI_Bcast` performance because of a pipelining effect that occurs when those operations are timed in series (which is necessary without an accurate global clock). Therefore they proposed an elaborate means of scheduling tests so that the true performance of `MPI_Bcast` routines could be made clear. Despite their method's ability to systematically evaluate the completion times of `MPI_Bcast` operations at individual processes by using many repetitions of their benchmark, they did not present any results about the distribution of completion times; instead they reported only the maximum completion time of any process. Furthermore, because (like other benchmarks) their benchmark only measured the average time for an `MPI_Bcast` over a large number of repetitions, it was also unable to account for stochastic delays and overheads; hence it could not obtain results as revealing as those obtained by MPIBench and shown in Figure 51.

The MPICH MPI implementation that was used to create these benchmark results employs non-blocking communication to pipeline the broadcast process, as shown in Figure 52. Therefore, provided that adequate buffer space can be obtained, the completion time for any particular process involved in a broadcast can be calculated by summation of the time for each point-to-point message (modelled by t_{MPI_Send}, but labelled t_s in the figure for brevity), local completion delay (modelled by $t_{MPI\_Isend:local\_completion}$ but labelled t_{lc} in the figure for brevity), and finalisation delays on its path to completion. Consider the path to completion of any process in the broadcast tree, and count the number of point-to-point messages and local completion delays that it requires. For example, the path "0-0-2-6-14" (distinguished in green in the figure) comprises of local completion time delay at P_0 while it initiates a message to process P_1 (i.e. the "0-0" part of the path), followed by three point-to-point communication times (i.e. the "0-2-6-14" part of the path). In addition, before any process can complete its involvement in the `MPI_Bcast` it must `MPI_Wait` for all of the non-blocking messages that it has issued to complete. This delays the completion of the process until the latest response of a zero byte acknowledgement message from any of the `MPI_Isend`s that it issued (but note that these acknowledgement messages – which are simply in direct response to those already shown in Figure 52 – are not included in that figure to preserve its clarity). For example, in the case shown in

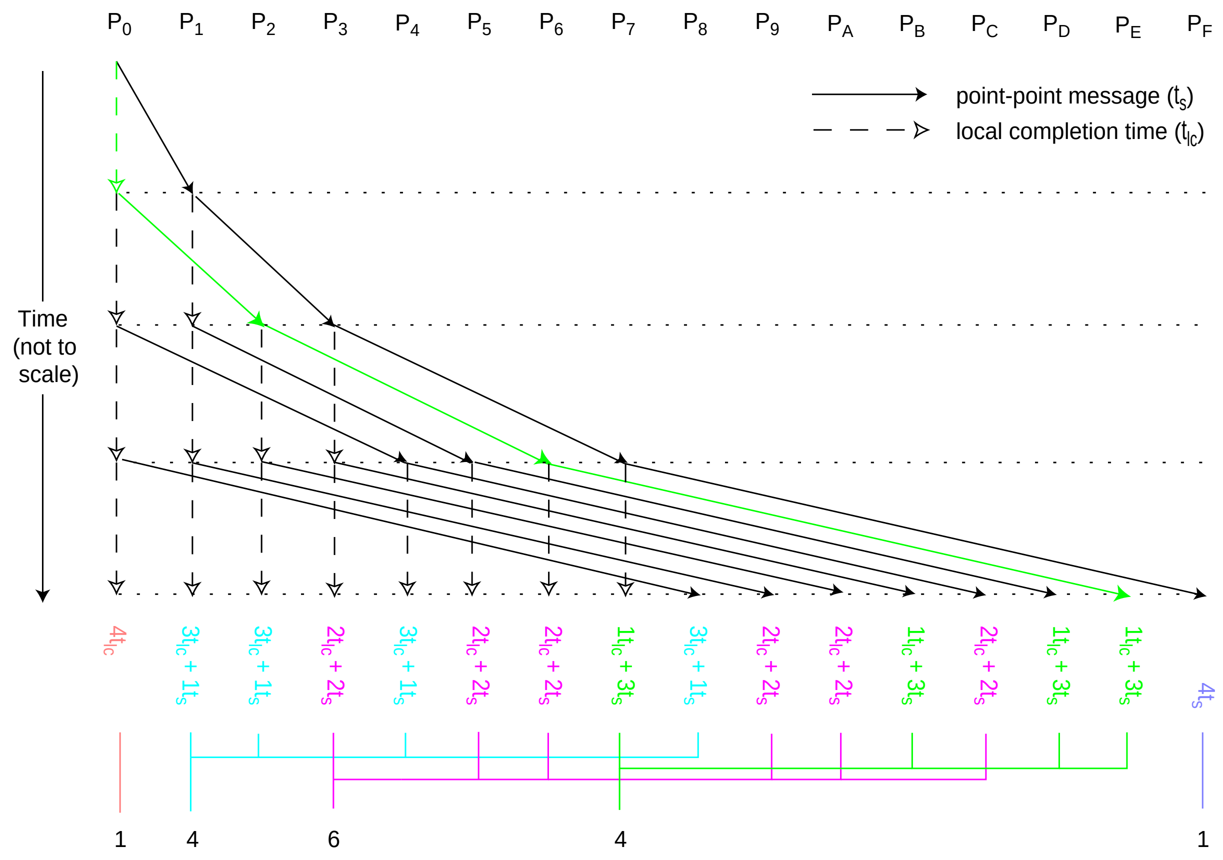


Figure 52: An augmented version of Figure 46, which includes the number of point-to-point message-passing times (t_s) and local completion times (t_{lc}) on each process's broadcast path.

Figure 52, process P_0 will have to wait until processes P_1, P_2, P_4 and P_8 have all acknowledged receipt of the messages they were sent from process P_0. Applying this procedure for all processes in the broadcast tree – labelling point-to-point messages as t_s and local completion times as t_{lc}, and then grouping like terms - creates a table of the number and type of communication events in the broadcast path to each process. For example, consider the 16x1 process broadcast tree shown in Figure 52. This shows one process with four t_{lc}s (P_0), which corresponds to the left-most peak of Figure 51 (at approximately 0.2ms), four processes with one t_s and three t_{lc}s (P_1, P_2, P_4 and P_8), which correspond to the next peak (at approximately 0.35ms), six processes with two t_ss and two t_{lc}s (P_3, P_5, P_6, P_9, P_{10} and P_{12}), and one process with four t_ss (P_{15}), which corresponds to the right-most peak (at approximately 0.8ms). Hence, the number of t_{lc}s and t_ss in any given broadcast path can be used to accurately determine the completion time of any process taking part in an `MPI_Bcast` operation: it is the sum of any local completion times, the times for each point-to-point message transmission, and any delay incurred while `MPI_Wait`ing for point-to-point messages to be acknowledged. Finally, the number of processes completing in any peak of the broadcast distribution can be elegantly determined by application of

Pascal's Triangle: a broadcast to one process has peaks distributed in the ratio :1; to two processes in the ratio 1:1; to 4 processes in the ratio 1:2:1; to 8 processes in the ratio 1:3:3:1; to 16 processes in the ratio 1:4:6:4:1 (as shown in this example); etc. Although this final curiosity offers little direct applicability to performance simulation of software-based broadcast trees (because it does not identify which processes are in particular peaks), it provides valuable insight into how software-based broadcasts scale with the number of processes involved.

Now that the structure of the performance profile of a software-based broadcast process is more understandable, it is instructive to briefly revisit the results shown in Figure 49. Earlier, it seemed that this profile was basically the same as that for a point-to-point operation, with a few difficult to explain sub-peaks, which could possibly be dismissed as noise. However, it is now clear that these sub-peaks are indeed systematic, and that the overall profile is structurally equivalent to the profile with the obvious peaks that is shown in Figure 51. The reason that the peaks are difficult to see in Figure 49 is because the variance of each of the peaks is roughly the same size as the separation between them. In turn, the reasons for this are two-fold. Firstly, the performance variance of point-to-point messages increases with message size due to greater contention (for example see Figure 15). Secondly, the separation of peaks, which is governed by the difference $t_s - t_{lc}$, does not (in this case) increase as rapidly. Incidentally, the slowly growing disparity between t_s and t_{lc} occurs because both are linear functions dependent on message size, where each has a startup latency and bandwidth. Because both the startup latency of local completion is less than that of a complete point-to-point `MPI_Isend` call and the bandwidth of local memory is greater than that of the network, the difference between them slowly grows.

In addition to the main peaks, a number of sub-peaks can be observed in Figure 51. These extremely fine details are related to completion times for individual processes, rather than the groups of processes with the same number of t_s and $t_{lc}s$. These subtle differences in completion time are, in large part, caused by the non-uniform contention levels to which individual point-to-point messages are subject. In each stage g of the broadcast, there are $\log_2 g$ messages transitting the network. These increased contention levels lead to more message-passing delays, which cascade through the broadcast process. As confirmation of this, the variance of the peaks (for example see Figure 51) increases as the broadcast progresses, which is (on average) represented towards the right-hand side of a broadcast performance distribution.

Predicting these extremely fine details is beyond the abilities of the software-based broadcast model presented in this subsection. However, it is not beyond the ability of the PEVPM model presented in the previous chapter. While the model developed here is essential to building a correct PEVPM model of a software-based broadcast, once that is done, the ability of the PEVPM to take into account contention levels makes it capable of reproducing even the extremely fine performance details that have been observed in practice. Furthermore, this also makes the PEVPM capable of predicting the performance of broadcasts using SMP nodes that do not provide shared memory support based on the performance of `MPI_Isend` tests for the required number of processes per node. Alternatively, the performance of broadcasts using SMP nodes that do provide shared memory support can be reasonably well modelled using inter-node and intra-node `MPI_Isend` performance results for constituent point-to-point messages as appropriate.

Finally, there is one important note that must be made about the performance of `MPI_Bcast` on Perseus, regarding the note made earlier about a special case where hardware-assistance is (possibly) improving performance. The completion time of zero byte broadcasts, shown in Figure 47, did not exhibit the characteristic profile of a software-based `MPI_Bcast`. In fact its performance profile far more closely resembled that of a typical point-to-point operation, although with a significant number of even shorter completion times. Although the MPICH implementation makes no exception in its broadcast technique for zero byte messages, it is clear that hardware-assistance is somehow being applied. Although the mechanism for this has not yet been conclusively determined, the significant number of very short completion completion times suggests that some strange optimisation may be occurring at the TCP/IP level, and in particular in its acknowledgement processing subsystem. Lastly, although the usefulness of broadcasting no actual data may seem quite pointless, a zero byte broadcast plays an important part in the `MPI_Barrier` procedure, which will be discussed in the next section.

Based on the discussion so far, analysing the results of the `MPI_Bcast` test for Orion is a simple matter, because it appears to use essentially the same software-based broadcast mechanism as Perseus. The performance profiles shown in Figure 54 are virtually identical in structure to those of Perseus (shown in Figure 51), i.e. the peaks are in the ratio 1:4:6:4:1 as expected for a 16x1 process software-based broadcast. Unlike on Perseus, however, where the peaks blur together for `MPI_Bcast`s of large messages sizes, the peaks remain defined for the 32 Kbyte `MPI_Bcast` on Orion. This is simply a consequence of the smaller variance in performance of point-to-point messages on Orion than on Perseus. What is of further interest in this figure, however, is the relationship between the performance profiles for the 16x1 and 16x4 process broadcasts. The small performance offset between these profiles shows that `MPI_Bcast` is optimised to take advantage of the fast internal shared memory network. A discussion of these optimisations can be found in Sistare *et al.* [318]. For completeness, an overview of the performance of the `MPI_Bcast` operation on Orion is also provided in Figure 53. Note the unstable performance of the broadcast operation for messages between 16 Kbytes and 32 Kbytes in size, which reflects the instability of the point-to-point operations (over the same range of message sizes) from which it is constructed.

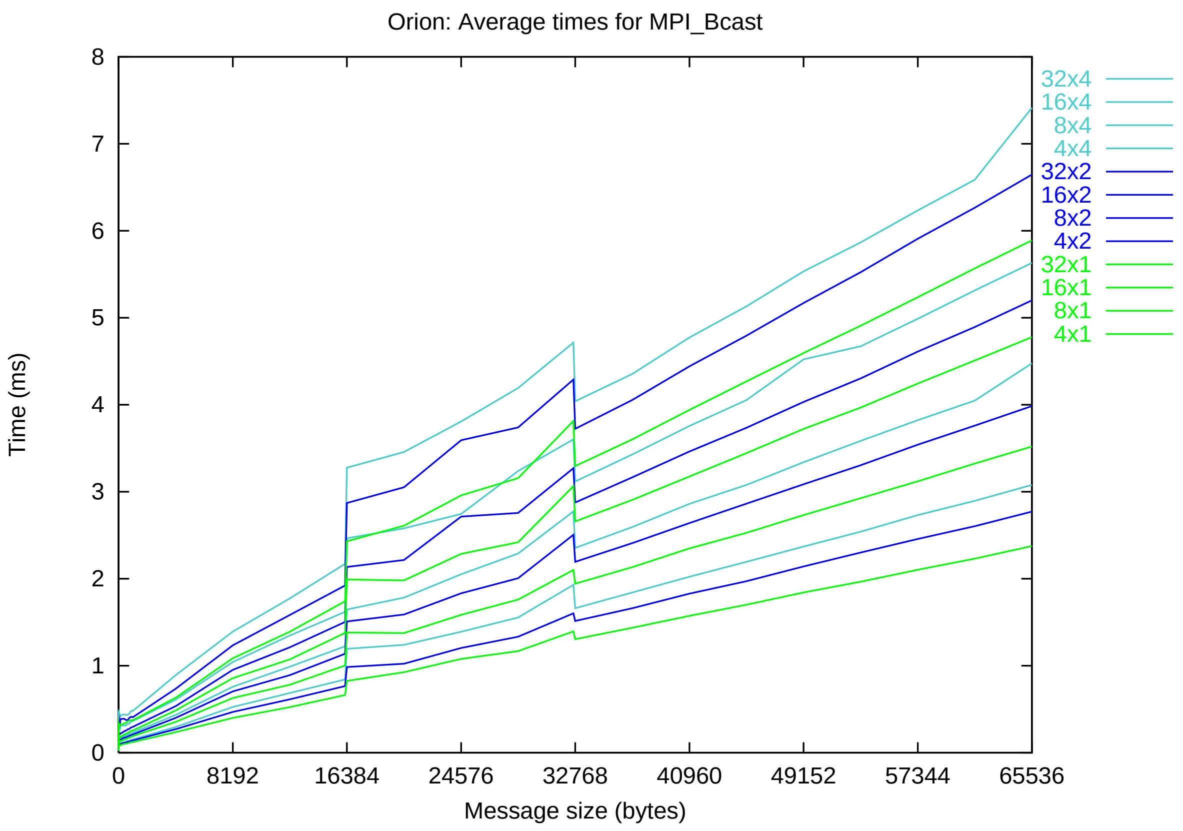


Figure 53: Average times for `MPI_Bcast` using large message sizes with various numbers of communicating processes on Orion.

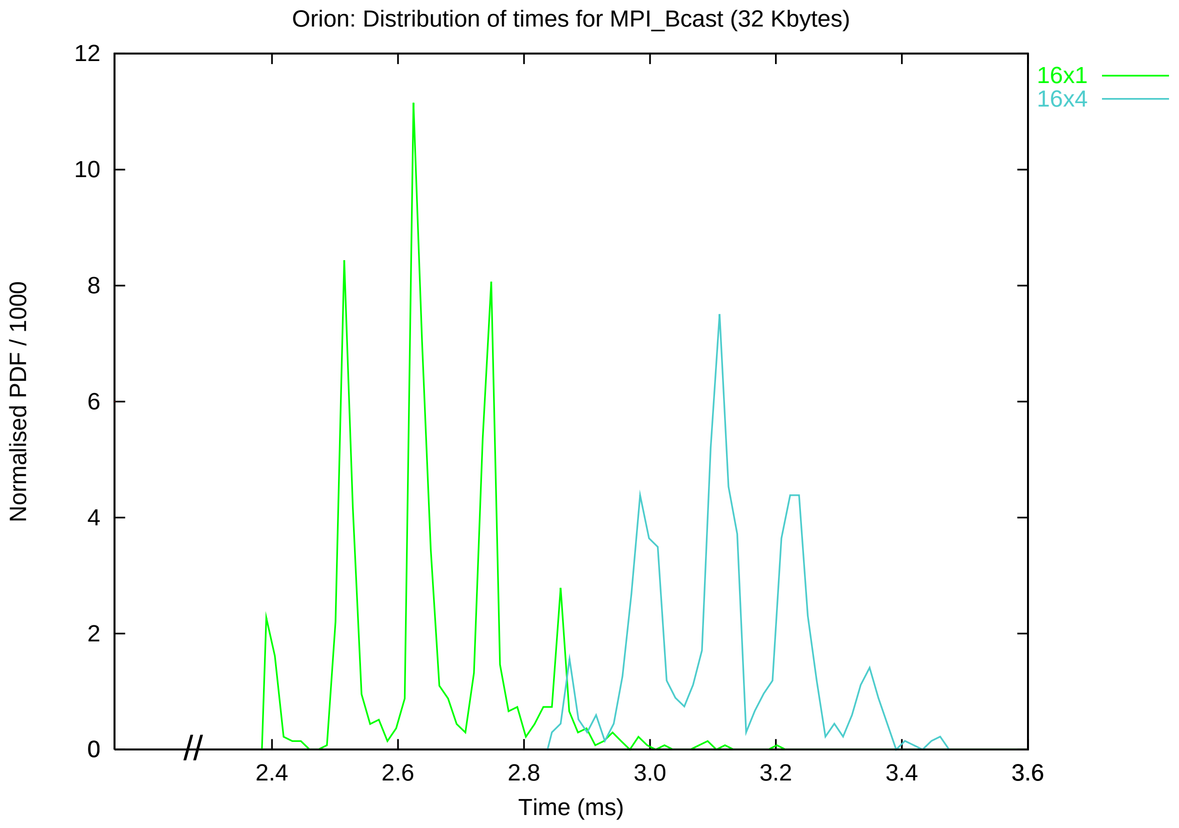


Figure 54: Sampled performance profile for `MPI_Bcast` using a 32 Kbyte message with 16x1-4 processes on Orion.

The results for `MPI_Bcast` on the APAC NF are completely different than those observed for Perseus or Orion. This is because the QsNet in the APAC NF enables a hardware-based `MPI_Bcast` when MPI processes are allocated to physically contiguous nodes. With the exception of one test, which is noted, all results conducted on the APAC NF used consecutive nodes, hence testing the hardware broadcast capability of the machine. QsNet provides a hardware-based broadcast mechanism that is capable of sending a broadcast message in essentially the same time as a point-to-point message (given that no unrelated messages are in transit on required network links). Broadcast messages are first sent to the top of the of the (physical) QsNet tree, where the switch at that level copies the message to every output-link, and it thus cascades down the entire network tree to all nodes. The average time for an `MPI_Bcast` test on the APAC NF, shown in Figure 55, when compared with the results for an `MPI_Isend` (see Figure 22), does indeed confirm that the hardware broadcast completes in essentially the same time as a point-to-point operation. Furthermore, comparison of the performance profile for a 16 Kbyte `MPI_Bcast` (using one process per node), shown in Figure 56, with the performance profile for a 16 Kbyte `MPI_Isend` (see Figure 15), underscores the essentially equivalent performance of the two operations; at least, as long as only one process is being run on any node. Unfortunately, however, the QsNet hardware broadcast only supports one process per node, and (like on Orion) the internal shared memory network is used to propagate the broadcast to any other processes within an SMP node. This can be seen in the small performance offset between the x1, x2 and x4 cases. As was explained earlier, this offset can be modelled using `MPI_Isend` results of the relevant message size for 1x2 or 1x4 processes, which basically represent the time required for an intra-node memory copy.

There are two further details about the performance of `MPI_Bcast` on the APAC NF that are quite interesting. First, notice the small peaks that precede the main peak for each of the distributions in Figure 56. A magnified version of the small initial peak in the 32x1 case is shown in Figure 57. This peak accounts for 1/32 of the total completion times recorded, and more particularly, it accounts for all of the completion times recorded at the root process. This figure shows that the root process completes participation in the `MPI_Bcast` asynchronously and thus finishes marginally sooner than the other processes taking part in the broadcast. More strikingly, although of marginal performance importance, it also shows the effect of (what is almost certainly) the operating system's scheduling quanta on when user level processes are notified of message completion. The second interesting detail, shown in Figure 58 is that a systematic serialisation of completion time at different processes still occurs during broadcasts on the APAC NF, although to much less extent than the software-based broadcasts on Perseus or Orion; this serialisation is only observable for very small broadcasts over a large number of processes.

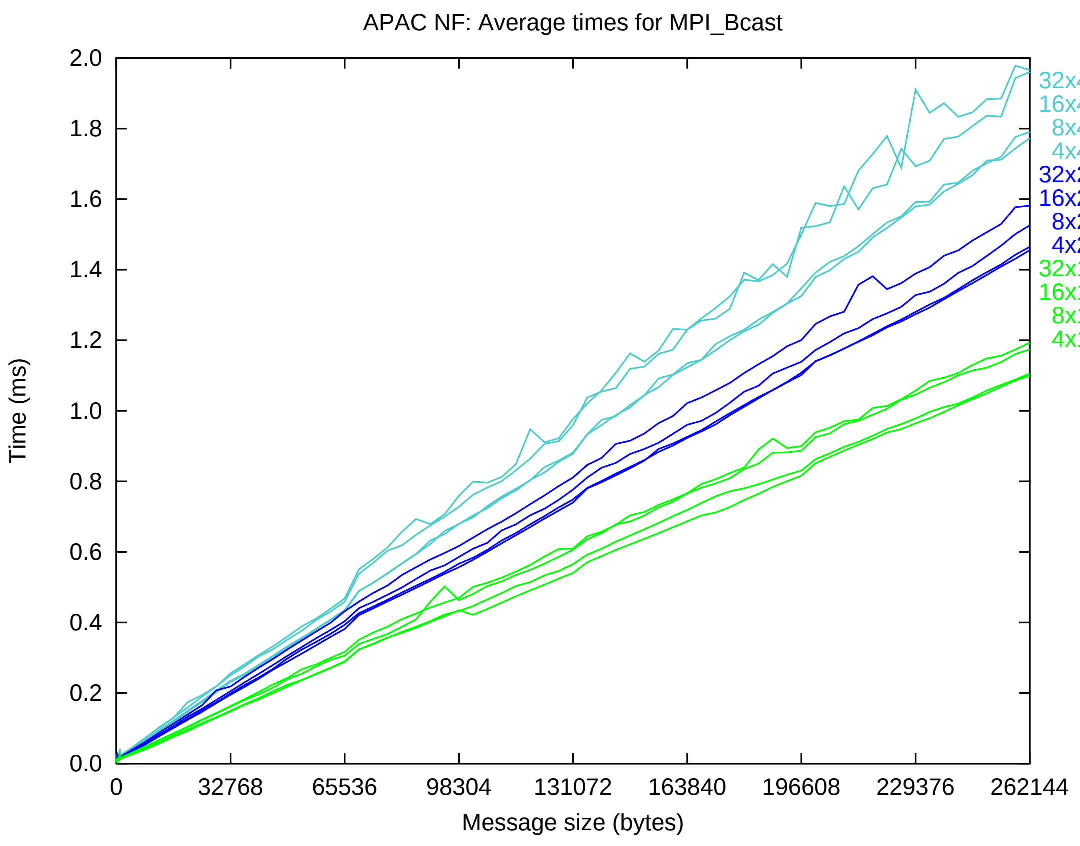


Figure 55: Average times for `MPI_Bcast` using large message sizes with various numbers of communicating processes on the APAC NF.

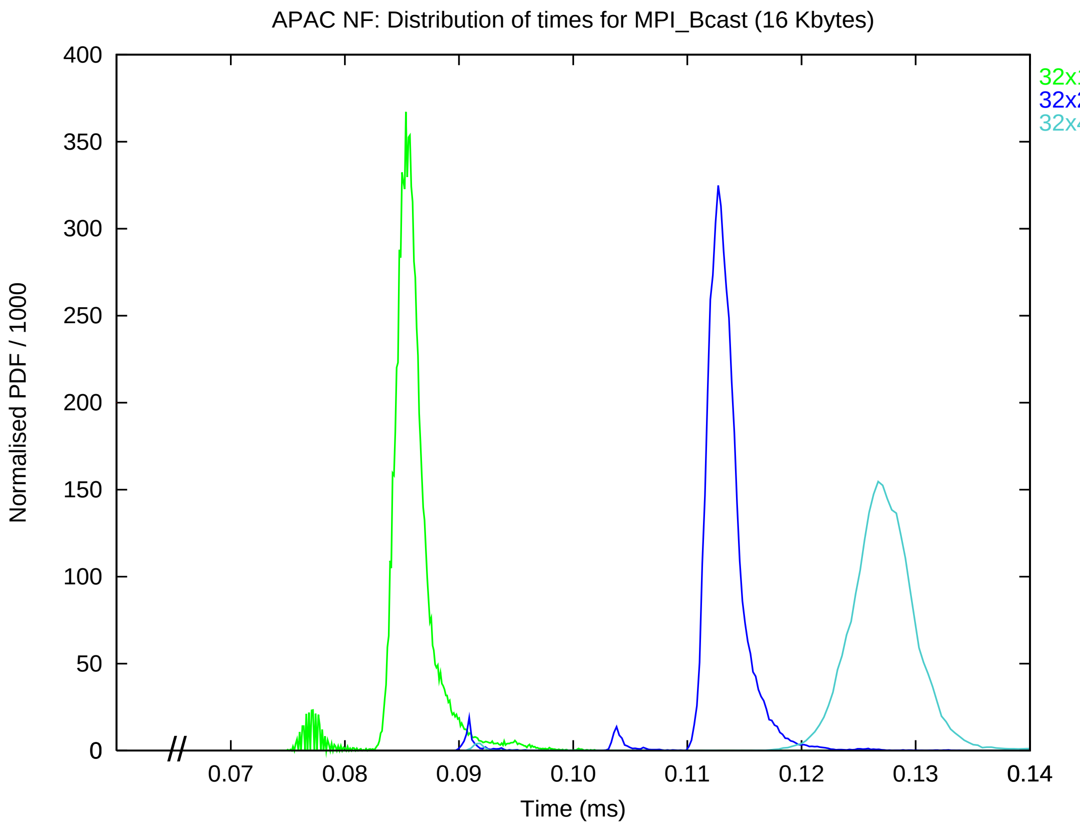


Figure 56: Sampled performance profiles for `MPI_Bcast` using a 16 Kbyte message with 32x1-4 processes on the APAC NF.

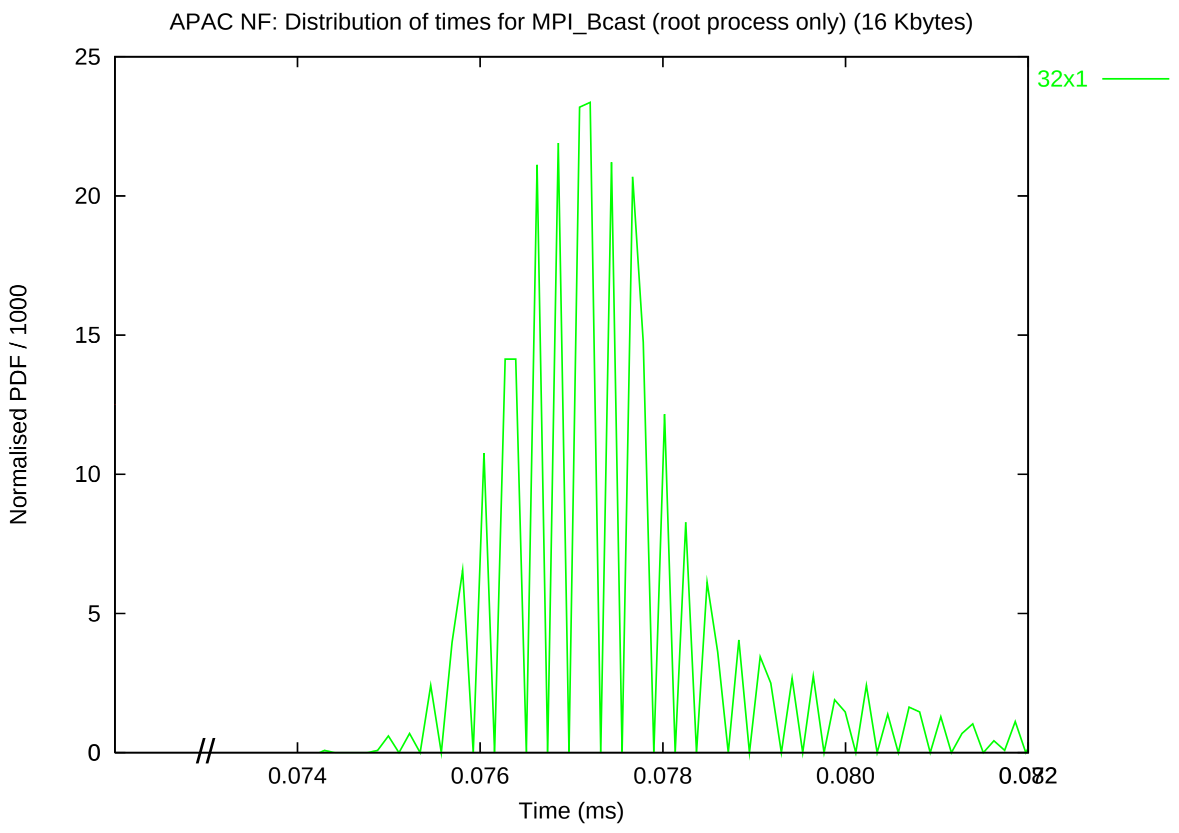


Figure 57: Sampled performance profile for `MPI_Bcast` using a 16 Kbyte message with 32x1 processes on the APAC NF, measured at the root process.

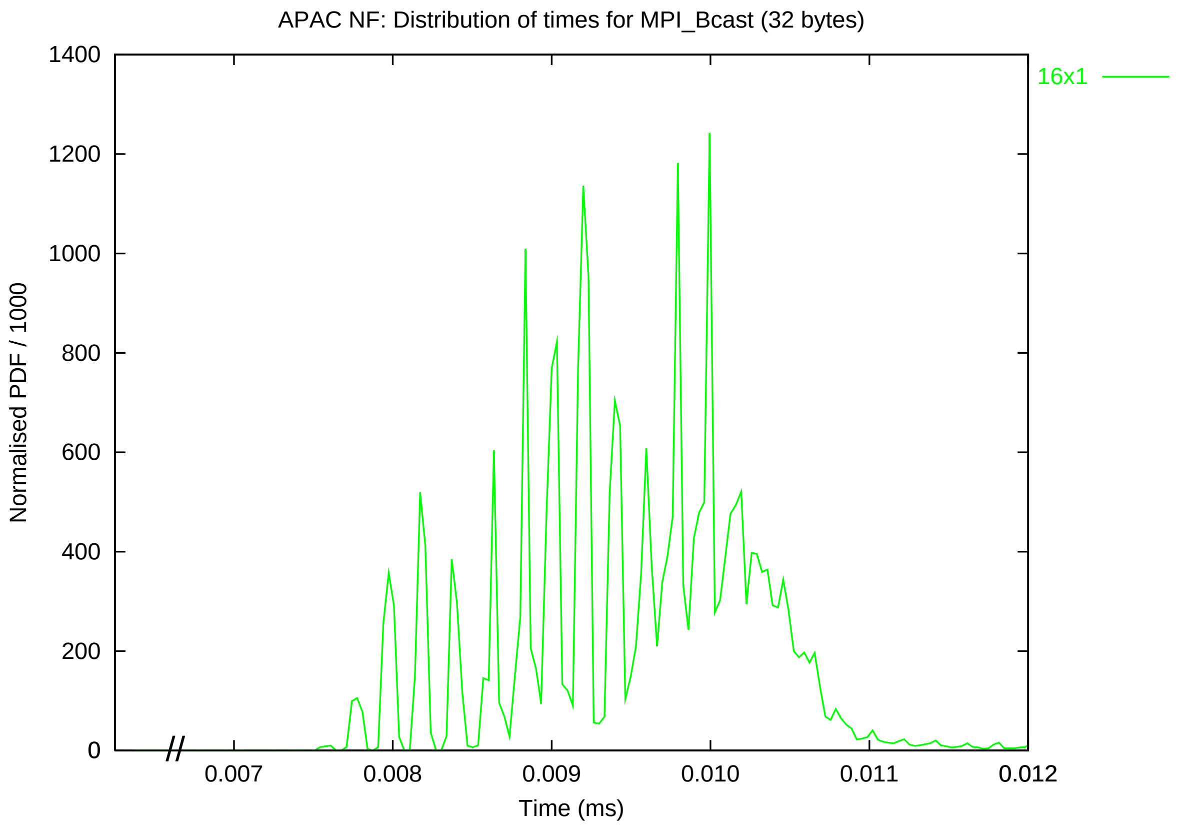


Figure 58: Sampled performance profile for `MPI_Bcast` using a 32 byte message with 16x1 processes on the APAC NF, which shows serialisation by a QsNet switch.

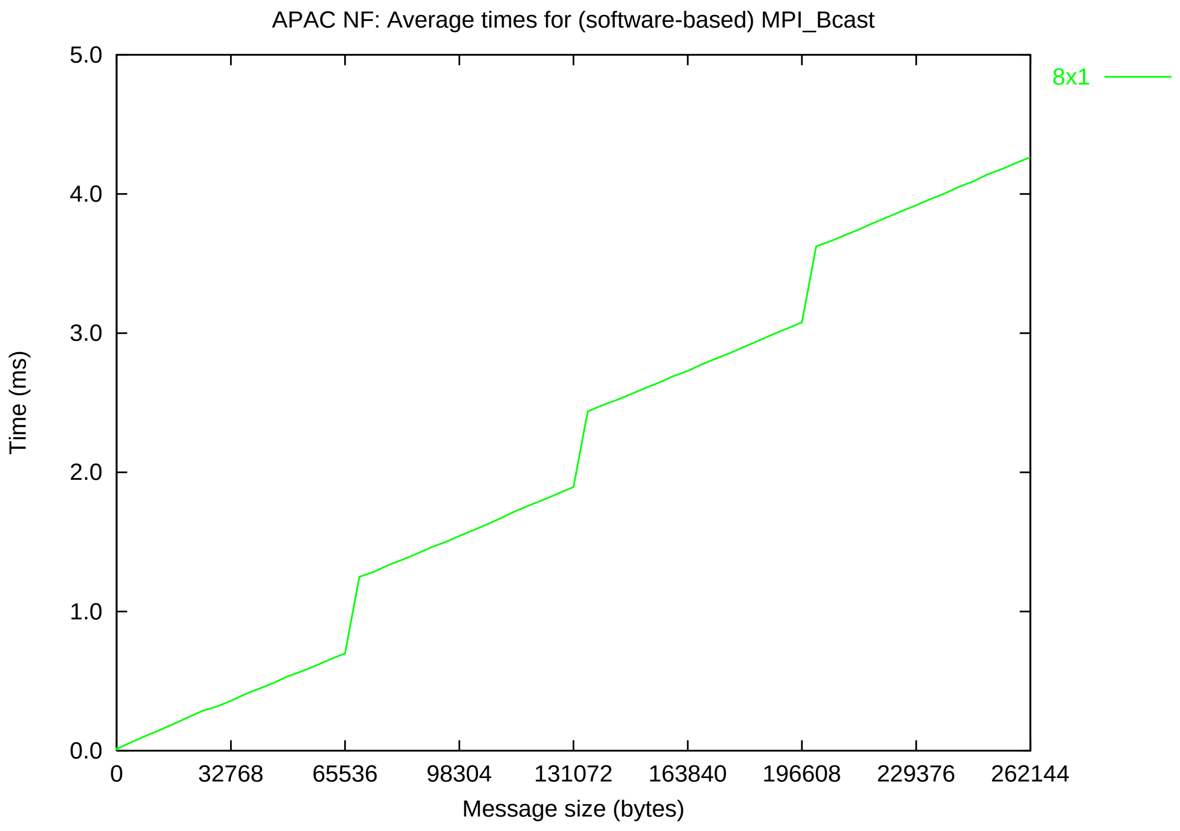


Figure 59: Average times for software-based `MPI_Bcast` using large message sizes with 8 processes running on nodes 69, 77, 78, 80, 81, 86, 88 and 99 of the APAC NF.

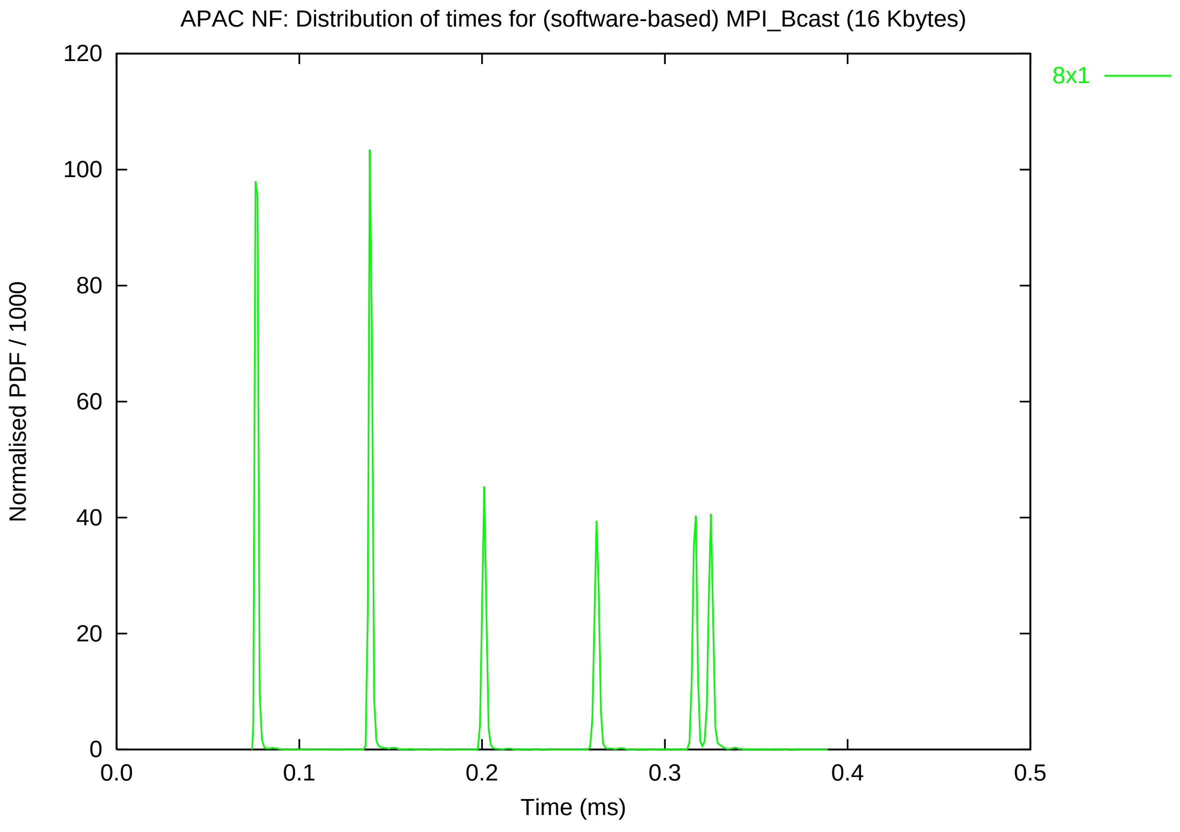


Figure 60: Sampled performance profile for the software-based `MPI_Bcast` in Figure 60 using a 16 Kbyte message.

The very small separation of peaks in this figure, corresponding to approximately 150ns, shows the serialisation that occurs in a QsNet switch when copying a broadcast message to each output link.

Finally, for comparison with the hardware-assisted `MPI_Bcast`, the performance of a software-based `MPI_Bcast` on the APAC NF was measured by running the test across a non-contiguous block of nodes. The average completion time of an `MPI_Bcast` using one process in each of nodes 69, 77, 78, 80, 81, 86, 88 and 99 is shown in Figure 59, with its associated performance distribution for a 16 Kbyte message in Figure 60. The jumps in average completion time at 64 Kbyte intervals coincide with the value of a user-defined shell variable, `LIBELAN_TPORT_BIGMSG`; messages larger (in bytes) than this value are sent only when a matching receive has been posted, which causes the jump in completion time at each interval. The distribution of peaks, which, from left to right, belong to the processes on nodes 77 and 78, 80 and 81, 86, 88, 69, and 99 respectively, show that (in this case, at least) the hardware broadcast capability of the QsNet is still being made use of where possible. In particular, the results show that the root process broadcasts data to other processes using a linear (rather than binomial tree) software-based broadcast algorithm, but where hardware-assistance is called upon to deliver the data to subsets of consecutively numbered nodes. (Overall, this algorithm is of questionable usefulness from a performance stand-point: note that in this case the broadcast has five messages of 64 Kbytes on the critical path, yet a binomial tree algorithm that ignored the hardware broadcast capability would have only three; the overall algorithm could be improved by combining the hardware broadcast capability for intra-subset communication with a binomial tree approach for inter-subset communication). Because of the QsNet's synchronous packet delivery protocol, the root process does not terminate until all other processes have acknowledged the receipt of the data. (Actually, for a subtle reason, the root process terminates second-last by the narrowest of margins; the last process to be sent broadcast data from the root cannot complete until it receives an acknowledgement of its acknowledgement message from the root, thus completing a three-way handshake).

5.3 Results for `MPI_Barrier`

The purpose of an `MPI_Barrier` operation is to synchronise a group of processes within an MPI communicator so that two stages of a computation can be isolated from each other. To achieve this, no individual process is allowed to progress past an `MPI_Barrier` operation until all of the processes in the group are also ready to do so. Like `MPI_Bcast` portable (software-based) versions of the `MPI_Barrier` are constructed of point-to-point operations. Also like `MPI_Bcast` the importance of the `MPI_Barrier` operation encourages high-end parallel machines to provide hardware-assisted barrier synchronisation.

The MPICH implementation of MPI that was used for benchmark tests on Perseus provides a software-based `MPI_Barrier` operation. The execution of an `MPI_Barrier` operation proceeds in two stages. In the first stage, which is basically the software-based broadcast tree from Figure 46 applied in reverse, a tree structure with $\lceil \log_2 n \rceil$ levels is created, where n is the number of processes in the communicator, such that half of the processes at each level of the tree have arcs to the other half of the processes at the next level. This defines a precedence hierarchy, and sending messages according to a bottom-up traversal of the tree results in a notional synchronisation of the processes. Because this message-passing can occur concurrently at each level of the tree, this first stage of the process is completed in $\lceil \log_2 n \rceil$ steps. The second stage is required to ensure the semantics of the `MPI_Barrier` operation. Because no process is allowed to progress until all other processes are also ready to do so, none of the processes may continue until the process at the root of the tree has received what must necessarily be the last message of the first stage. Hence, in the second stage, all processes must wait to receive an acknowledgement message from the root process, which is usually achieved using a broadcast mechanism. When this message has been received by any process, that process may continue. It is worth noting that this means that different processes will have to wait at different times during the `MPI_Barrier` routine. For example, the root process only becomes involved at the very end of the first stage, and is the first process to exit the broadcast stage (as explained in the previous section) and hence is the first process to exit entire barrier synchronisation. The PEVPM model described in Chapter 3 is capable of taking this into account, but for the the purpose of result verification and subsequent model construction, a simpler model will be evaluated here.

The performance of the `MPI_Barrier` operation on Perseus is shown in Figures 61-62. The results towards the tip of the left tail of the distribution are likely to be associated with the root process, while those towards the tip of the right tail are likely to be associated with the slowest processes to exit the broadcast phase of the barrier synchronisation. The remaining variability visible in these distributions is caused by the inherent uncertainty in the performance of the constituent message-passing routines that construct the `MPI_Barrier` operation (which in turn is due to network contention as discussed in Section 3.4.2).

A simple model for the performance of the `MPI_Barrier` operation on Perseus can be obtained by ignoring the variability in message-passing time and considering only average times (for zero byte messages):

$$\hat{\bar{t}}_{MPI\_Barrier} = \lceil \log_2 n \rceil \bar{t}_{MPI\_Isend} + \bar{t}_{MPI\_Bcast}$$

The accuracy of this simple model is examined in Table 2, which compares times for

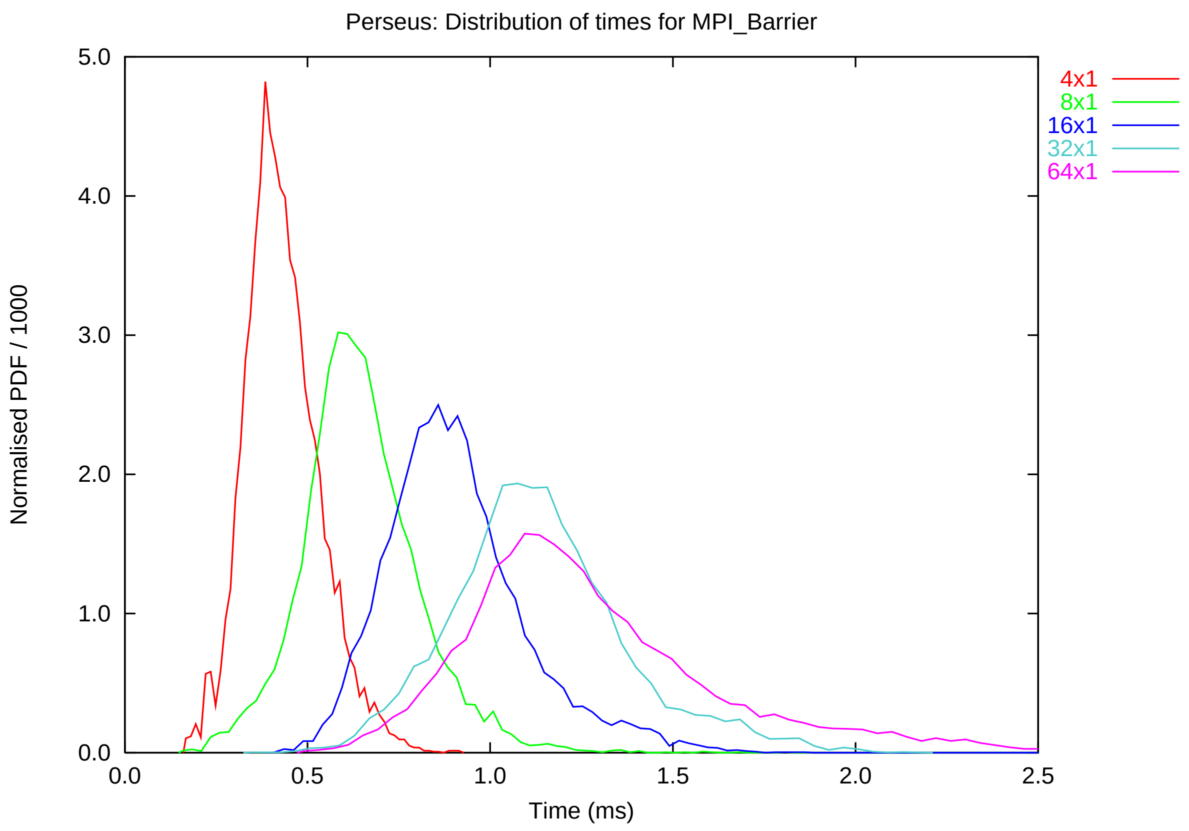


Figure 61: Sampled performance profiles for `MPI_Barrier` using various numbers of processes (one per node) on Perseus.

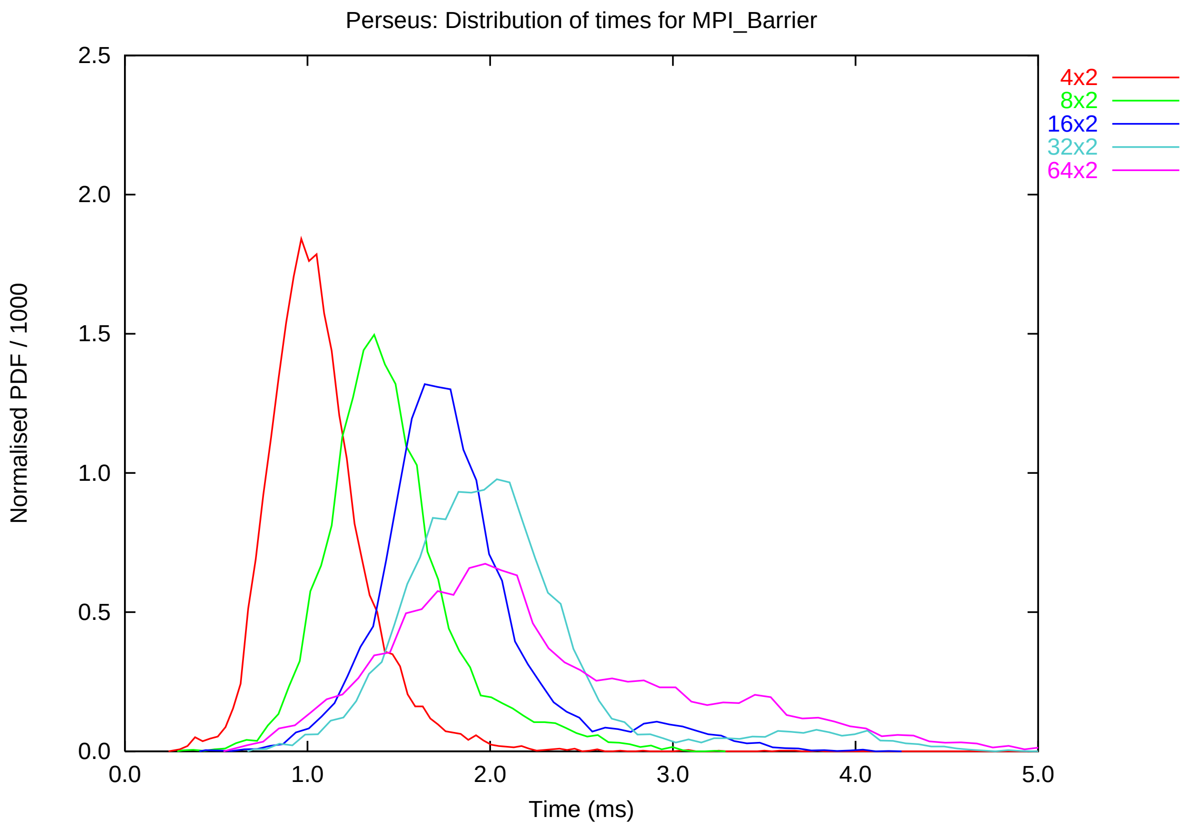


Figure 62: Sampled performance profiles for `MPI_Barrier` using various numbers of processes (two per node) on Perseus.

MPI_Barrier generated using the simple model with actual measurements. The re-

Table 2: A comparison of the measured average times for MPI_Barrier on Perseus ($\bar{t}_{MPI\_Barrier}$) with average times generated by the simple model for the same operation ($\hat{\bar{t}}_{MPI\_Barrier}$).

n	$\bar{t}_{MPI\_Isend}$	$\bar{t}_{MPI\_Bcast}$	$\hat{\bar{t}}_{MPI\_Barrier}$	$\bar{t}_{MPI\_Barrier}$	$\delta x(\%)$
4x1	0.165	0.089	0.419	0.432	-3.0
8x1	0.175	0.114	0.639	0.646	-1.1
16x1	0.176	0.139	0.843	0.924	-8.8
32x1	0.178	0.153	1.043	1.136	-8.2
64x1	0.183	0.197	1.295	1.275	1.6

sults in Table 2 show that the simple model is quite accurate for all of the process sizes measured. However, it cannot account for the large variations in performance that are shown in Figures 61-62. The simple model does not take into account the effect of the constituent messages of the MPI_Barrier call that complete faster or slower than average. Because of the precedence relationships present in these constituent messages, any delays will cascade to higher levels in the synchronisation tree, thus causing performance variation. This simple case is a good example of why it is important to consider the performance distribution of individual message-passing operations when constructing a macroscopic performance model. However, the detailed PEVPM model that would take into account performance distributions for the constituent messages of an MPI_Barrier is not examined here. Such detailed models are left until the next chapter, where case studies are undertaken in order to analyse the accuracy of the PEVPM approach. The simple model presented here merely serves to assert the structural correctness of the precedence relationships that form the basis of those PEVPM models. In particular, the reasonably good correlation between the actual results and those predicted using the simple model provides good evidence that the MPI_Barrier routine is being decomposed (for PEVPM modelling) to its constituent point-to-point messages in the correct manner.

The widening of the performance distribution in Figure 62 by a factor of two with respect to Figure 61 shows that the same synchronisation algorithm is used in both cases. In particular, because MPICH was not configured with shared memory support (see Section 4.5.2), serialisation of messages from both processes at each node through its one network interface caused those messages to take, probabilistically, twice as long. If MPICH had been configured to use shared memory for intra-node communication, synchronisation of nx2 processes should perform almost as well as the nx1 results that were observed.

Unlike MPICH's MPI_Barrier on Perseus, Sun MPI's MPI_Barrier on Orion was optimised for SMP nodes [318]. The Sun MPI implementation synchronises all processes on a node using shared memory, which is faster than the inter-node network, and then

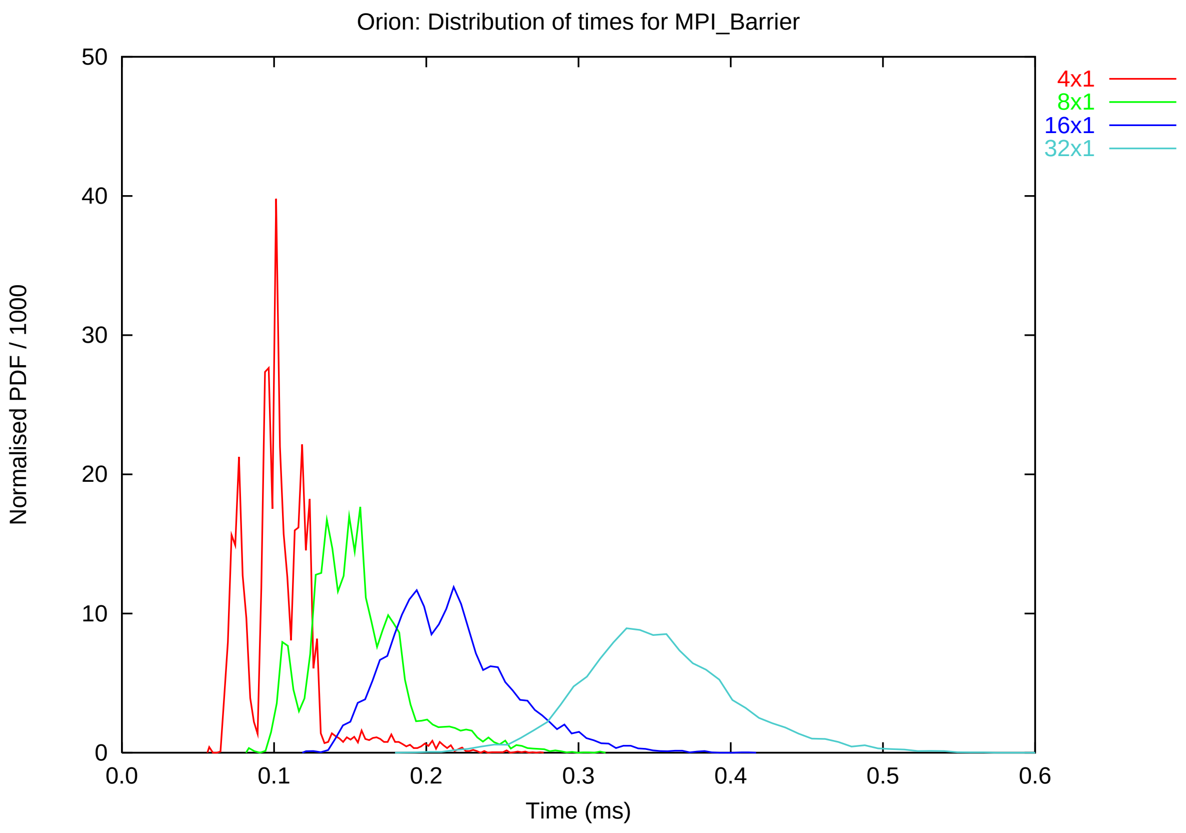


Figure 63: Sampled performance profiles for `MPI_Barrier` using various numbers of processes (one per node) on Orion.

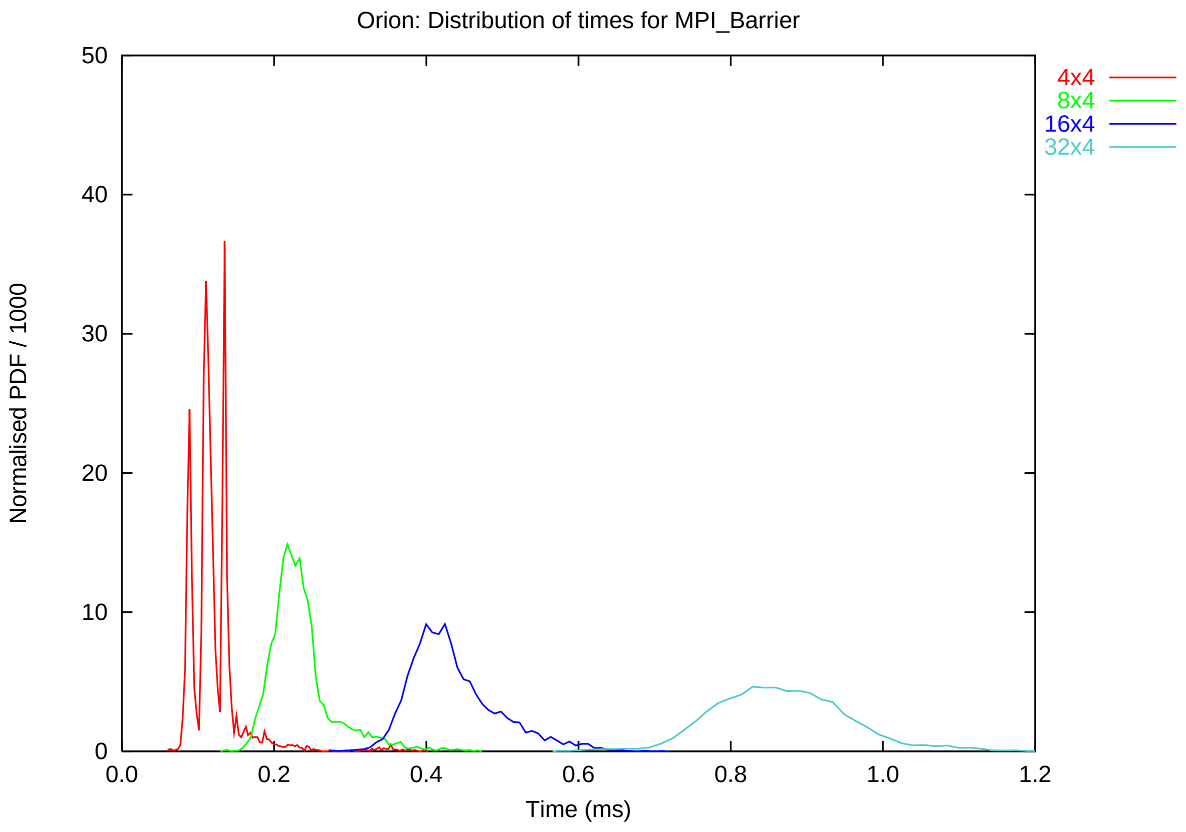


Figure 64: Sampled performance profiles for `MPI_Barrier` using various numbers of processes (four per node) on Orion.

synchronises nodes using the same technique described above. The result of this is that the synchronisation time for any given number of nodes is not supposed to be as affected by the number of processes running on each node as would otherwise be the case. Although the results in Figures 63-64 do show a performance improvement of about a factor of two when four processes per node are used, there seems to be room for improvement, especially given the results for the APAC NF below. As a side note, the characteristic effect on the performance distribution of the `MPI_Bcast` phase of the synchronisation process is quite clear in the 4x1 process case, because the performance variance of the constituent point-to-point messages in the first phase of the `MPI_Barrier` is small enough that the peaks do not merge together. Finally, although they are not plotted, the results of synchronisation using two processes per node fell between the nx1 and nx4 results.

The APAC NF can achieve very low `MPI_Barrier` times by using the hardware synchronisation capability of the QsNet interconnect, which is available as long as processes are run on physically contiguous nodes. A slightly slower, software-based `MPI_Barrier` technique is used when processes are running on nodes that are not physically contiguous. The software-based `MPI_Barrier` call on the APAC NF takes advantage of the synchronous packet delivery protocol of QsNet and is implemented as two consecutive software-based `MPI_Bcast`s (see Section 5.2) of zero bytes; therefore there is no need to investigate it further here. Disregarding the variance caused by contention effects, the APAC NF's hardware-based `MPI_Barrier` synchronises one process in each node in constant time regardless of the number of nodes involved, in contrast to the $O(\log_2 n)$ time required for the software-based synchronisation technique on Perseus or Orion. However, as mentioned in Section 5.2, the QsNet hardware synchronisation only supports one processor per node, and the internal shared memory network is used to propagate the barrier to any other processors within an SMP node, leading to a small performance offset dependent on the number of processes run on a node. The distribution of completion times of the `MPI_Barrier` operation on the APAC NF is almost completely flat (ignoring the noise caused by the operating system's scheduling quanta) and spans a very short window of duration; the best way to present the measured results is in tabular format.

Table 3: Performance of `MPI_Barrier` the APAC NF using various numbers of communicating processes.

n	$t_{MPI\_Barrier}$
nx1	$4.0 \pm 1.2\mu s$
nx2	$5.0 \pm 1.2\mu s$
nx4	$6.5 \pm 2.0\mu s$

Table 3 shows the impressive performance of the hardware-based `MPI_Barrier` on the APAC NF, which is orders of magnitude faster than a software-based `MPI_Barrier` on Perseus or Orion.

5.4 Results for `MPI_Scatter` and `MPI_Gather`

An important pair of collective communication routines in MPI are `MPI_Scatter` and `MPI_Gather`. The `MPI_Scatter` operation allows one process to divide a copy of specified local data into equal portions and distribute those portions to other processes. The `MPI_Gather` operation is the reverse of this, and allows one process to collect equal portions of data sent from other processes into a single copy. For example, the `MPI_Scatter` operation is often used to distribute large arrays among many processes, which act on the data they have received, and the `MPI_Gather` operation is used to collect the results at one process (probably the same one that scattered the data).

The `MPI_Scatter` operation can be fashioned from n point-to-point transmissions of s/n bytes, where n is the number of processes involved and s is the total size of the data to be scattered, for example like this:

```
if (procnum == r) {
  // code to execute at the root process, r
  for (i = 0; i < n; i++) {
    MPI_Send(sendbuf + i * s/n, s/n, MPI_BYTE, i, ...);
  }
}
else {
  // code to execute at the other processes
  MPI_Recv(recvbuf, s/n, MPI_BYTE, r, ...);
}
```

Similarly, the `MPI_gather` operation can also be fashioned from n point-to-point transmissions of s/n bytes. This time, however, one process receives the n messages from other processes and stores them in rank order, for example like this:

```
if (procnum == r) {
  // code to execute at the root process, r
  for (i = 0; i < n; i++) {
    MPI_Recv(recfbuf + i * s/n, s/n, MPI_BYTE, i, ...);
  }
}
else {
  // code to execute at the other processes
  MPI_Send(sendbuf, s/n, MPI_BYTE, r, ...);
}
```

Several optimisations can be made to these simple implementations. Firstly, as with `MPI_Bcast` in the last section (and in particular given adequate buffer space), the blocking sends in both the `MPI_Scatter` and `MPI_Gather` routines can be replaced with non-blocking ones. In the case of `MPI_Scatter` this allows the root process to quickly queue the required outgoing messages (at a rate governed by their local completion) and therefore complete itself sooner, as well as drive the other processes to complete as soon as possible because of the subsequent efficient utilisation of the root process's output link. In the case of `MPI_Gather` this allows the $n-1$ sending processes to complete according to their local completion time and potentially carry on with other useful work. Secondly, but applicable only to `MPI_Gather`, the receive operation at the root process can be modified to accept messages from other processes in any order, thereby minimising serialisation delays (and as a useful side-effect, reducing buffering requirements). Finally, messages can be grouped together and transmitted using a similar pattern to the software-based broadcast tree described in the last section. The only difference in the pattern is the amount of data transmitted by processes at each stage of the tree. While in an `MPI_Bcast` the same data is transmitted at each stage, in an `MPI_Scatter` a reduced amount of data is transmitted by processes at each stage. In the first stage, half of the total data to be scattered is transmitted by the root process to another process, then in the second stage both of these processes send half of that data (i.e. one-quarter of the original data) to two further processes, and so on, until the $\lceil \log_2 n \rceil^{nd}$ stage, in which all processes have received their allotted $1/n$ bytes of the total data (as well as any other intermediate data). In the first stage of an `MPI_Gather` (which is the procedural inverse of an `MPI_Scatter`) half of the processes send their $1/n$ bytes of the total data to the other half of the processes, which the receiving processes concatenate with their $1/n$ bytes of the data and send to a further half of their number, and so on, until the $\lceil \log_2 n \rceil^{nd}$ stage which sees the total data collected at the root process.

Implementing `MPI_Scatter` and `MPI_Gather` operations using the final approach explained above is especially common for hypercube networks, because data must be routed through intermediate nodes anyway. On single-hop networks, using this pattern trades off an increased amount of total traffic for fewer messages on the critical path [208, 261]. The increased total traffic does not usually have a direct performance impact, because it can be sent in parallel, but indirectly it usually increases contention and hence causes message-passing delays throughout the network. The benefit of fewer (in particular $\lceil \log_2 n \rceil$ instead of n) messages on the critical path is a reduction in serialised point-to-point startup costs, especially for small messages where these costs dominate total message-passing time.

Despite these optimisations, however, serialisation remains a central characteristic of the performance of `MPI_Scatter` and `MPI_Gather` routines, simply because of their respective one-many and many-one natures. If the network interface at the root process is the same speed as the network interface at other processes (which is almost always the case; and is the case for the machines benchmarked in this thesis) then it becomes a throughput bottleneck. For example, consider the case where none of the above optimisations have been applied to either `MPI_Scatter` or `MPI_Gather`. In the case of `MPI_Scatter`, this means that data scattered to different processes will be serialised at a rate determined by the speed of the output link at the root process, and the processes receiving the scattered data will (barring contention delays) complete in accordance with this serialisation. In the case of `MPI_Gather`, data sent to the root process will be serialised through its input link, hence determining the completion time of the gathering process. This can easily be seen in the following MPIBench results.

The average completion times (measured at the first process to complete) for an `MPI_Scatter` of a total of up to 64 Kbytes across various numbers of processes on Perseus are shown in Figure 65. This allows a direct comparison between the scattering time of a selection of data against a point-to-point operation transmitting the same amount of data (i.e. compare Figures 65 and 13). There is clearly a massive overhead when scattering small messages across a large number of processes compared to an `MPI_Isend` of the same amount of data, but the overhead becomes insignificant for large enough message sizes. As described in Section 4.3.3, because the total amount of data scattered in the `MPI_Scatter` test remains constant, this comparison clearly highlights the communication inefficiencies that are introduced by using more processes, and can therefore provide valuable insight into the performance scalability of fixed-size problems as processing resources are increased.

The simple structure of the performance distribution for an `MPI_Scatter` on Perseus (shown in Figure 66) makes it quite simple to estimate the completion time of any individual process from the average completion time measured at the last process to complete. The completion time for a process p out of n processes can be reasonably well modelled by p/n multiplied the average completion time (of the last process to complete) of the `MPI_Scatter`; with the exception that the root process should be modelled by nt_{lc}, where t_{lc} is the local completion time of the constituent messages of size s/n. Also, noting the fact that the root process does not actually need to send data to itself (because it is already there) raises a subtle point. When comparing the performance difference between `MPI_Scatter`s of a constant amount of data across a varying number of processes, the amount of data transferred via the network is not actually constant. Rather, the amount of data transferred is $(n-1)/n$ of the total data to be scattered, while the remaining data simply remains *in situ* at the root process. This makes the performance of `MPI_Scatter`s in comparison with `MPI_Isend`s of the "same" amount of data look better than they actually are. The PEVPM model for `MPI_Scatter` takes the basic approach described above, but by its nature provides a more refined model because it naturally accounts for local

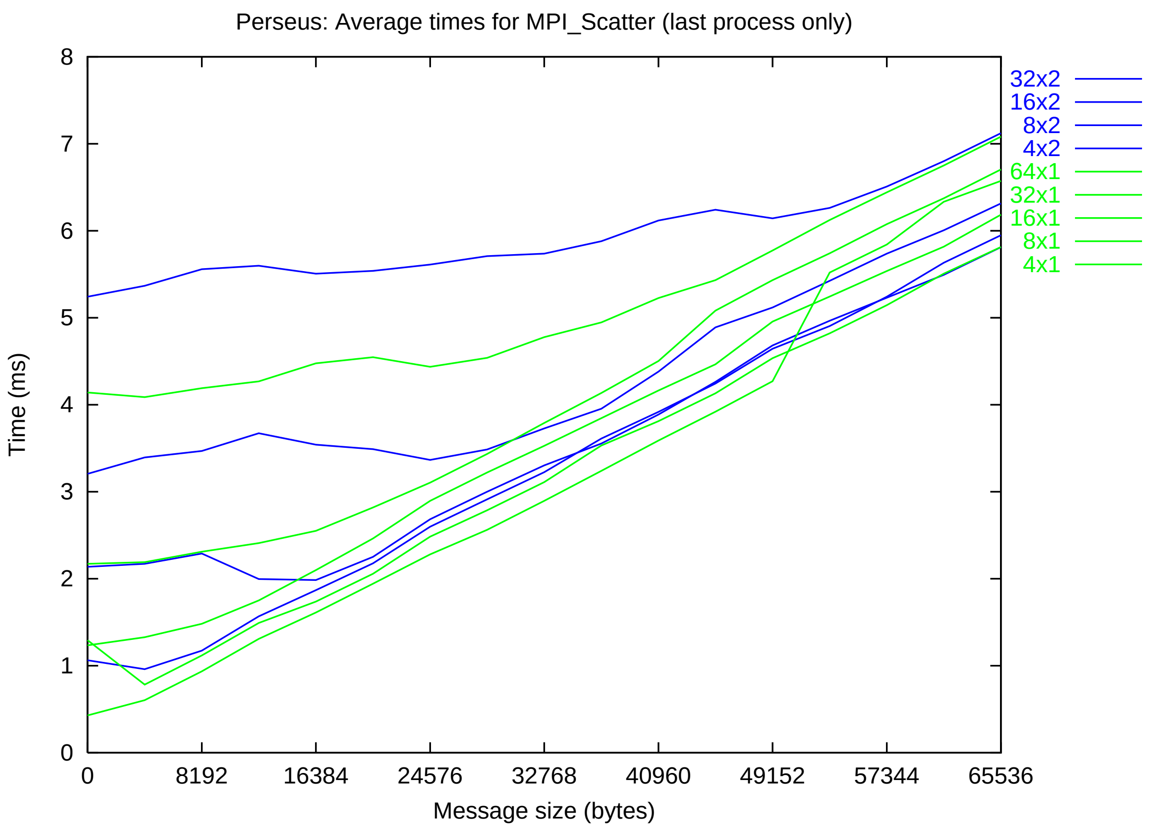


Figure 65: Average times for `MPI_Scatter` using large message sizes with various numbers of communicating processes on Perseus, measured at the last process to complete.

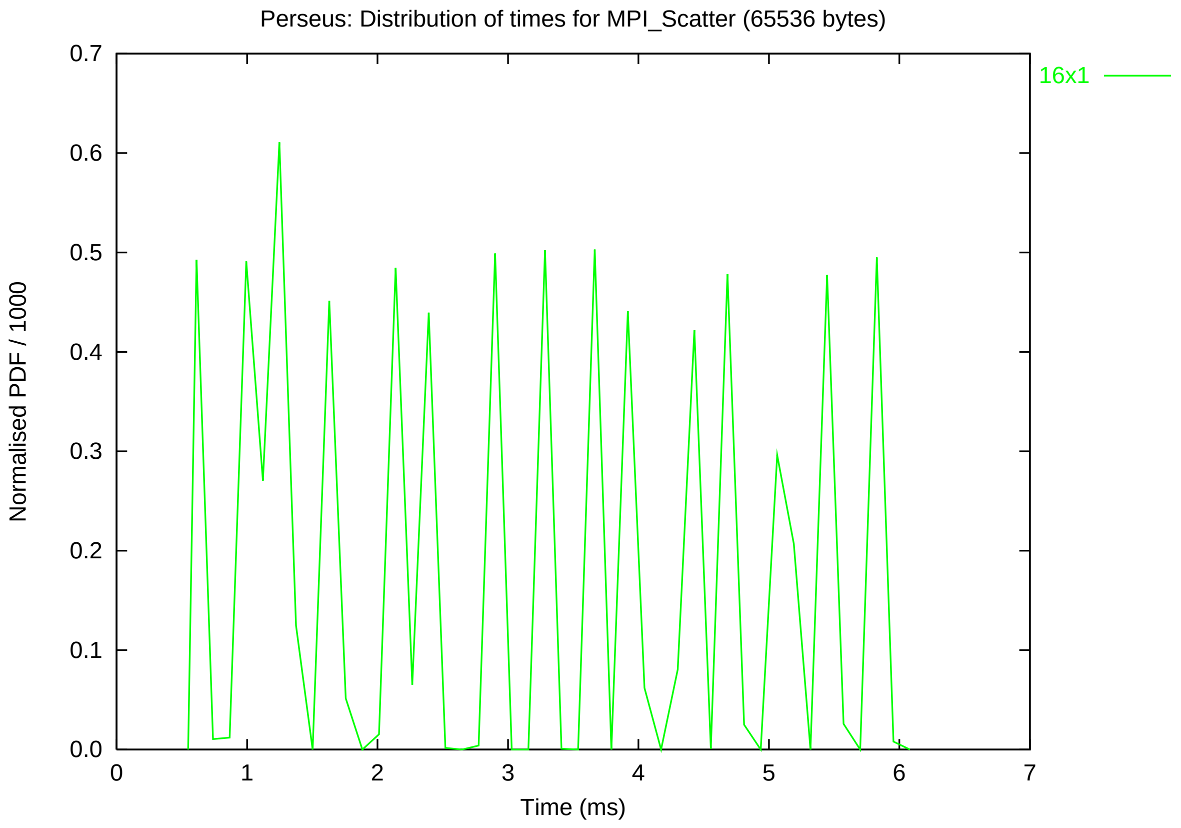


Figure 66: Sampled performance profile for `MPI_Scatter` using a 64 Kbyte message with 16x1 processes on Perseus.

completion time, the amount of data actually transferred via the network interface, and network contention.

The performance distributions for an MPI_Gather on Perseus are more complex than those for an MPI_Scatter. For example, consider the performance distribution of an MPI_Gather of 64 Kbytes using 8x1 processes shown in Figure 68. In this example, there are three peaks with areas in the ratio of 4:2:1 at low completion times, and one peak with area in the ratio of 1 at a higher completion time. For now, consider this as a bimodal distribution, with a low peak and a high peak with areas in the ratio of 7:1. The low peak represents the completion time of the seven processes that are sending their data to the root process, which are completing asynchronously (i.e. they are using MPI_Isends). The high peak represents the completion time of the root process in gathering all of that data. The root process takes longer to complete because of the serialisation that occurs at its network interface as the result of the many-one gather process.

The average completion times of both the $n-1$ sending processes and one gathering process for MPI_Gather tests using various numbers of processes on Perseus are shown in Figure 67. Comparing the average completion times of the root process in an MPI_Gather (see Figure 67) with the completion time for an MPI_Isend of the same amount of data (see Figure 13) shows a surprising amount of difference. Because the root process of an MPI_Gather can conceivably receive data from other processes in any order, the overall effect of delays experienced by any particular message should be reduced. If this were the case, it would be reasonable to expect that an MPI_Gather would perform similarly to an MPI_Isend of the same amount of data. However, this is clearly not the case. In fact, the performance of an MPI_Gather operation (see Figure 67) scales worse than its relative MPI_Scatter (see Figure 65), which has no potential to amortise delayed messages. Examination of the source code shows that the reason for this is that the MPI_Gather on Perseus is implemented in such a way that data must be gathered in rank order, thereby serialising delays rather than amortising them.

The sub-peaks at low completion times occur systematically based on process number, in the ratio ...:8:4:2:1:1, depending on how many processes are involved in the MPI_Scatter. These are simply a carry-over effect of the MPIBench measurement process that was used. As explained in Section 4.3.1, MPI_Barrier operations were used to isolate MPI_Gather operations so that they could be individually timed. The time difference between peaks is characteristic of the broadcast stage at the end of each of those MPI_Barrier operations, similar to the way the broadcast pattern shows through for the MPI_Barrier results on Orion (in Figure 63). Use of the alternate and more accurate synchronisation method described in Section 4.3.1 would avoid incurring this small systematic error. However, using that method, the tests would have taken approximately 50-100 times longer to run (because the of the large retransmit timeout on Perseus). Since

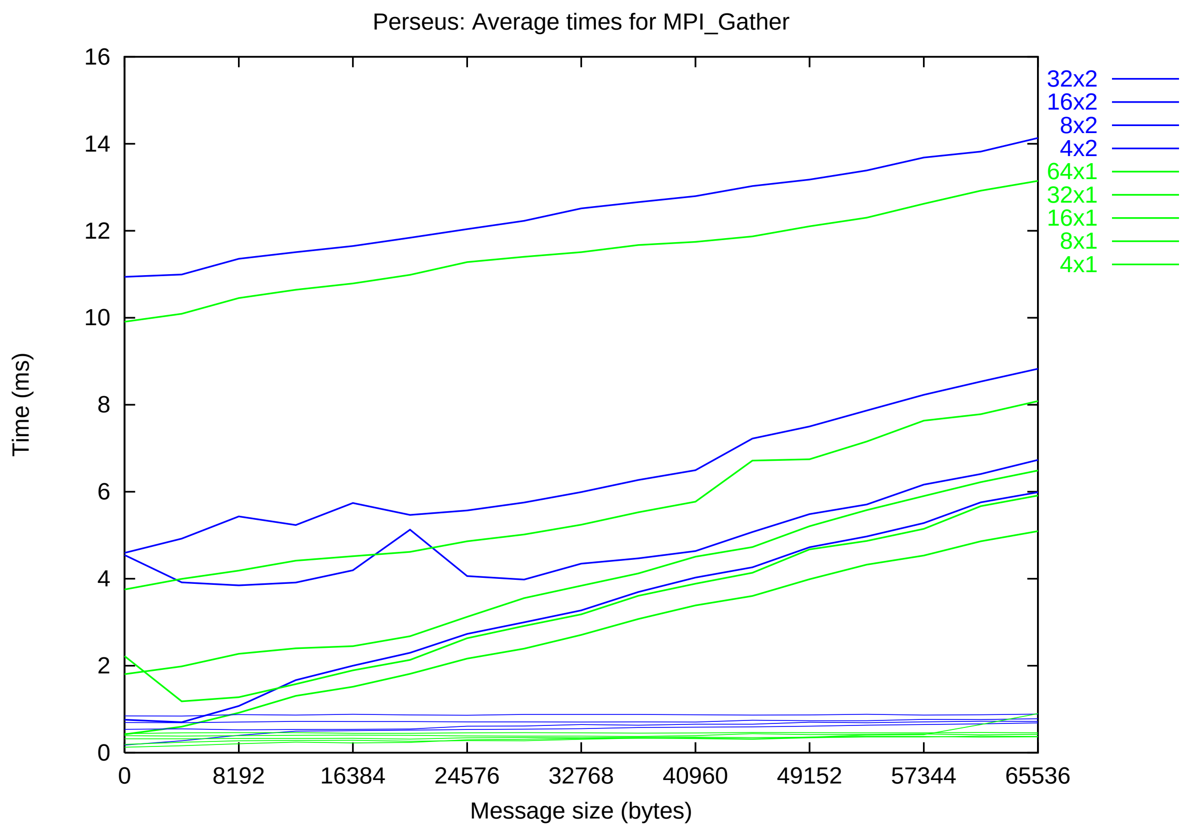


Figure 67: Average times for `MPI_Gather` using large message sizes with various numbers of communicating processes on Perseus, measured at both gathering and sending processes.

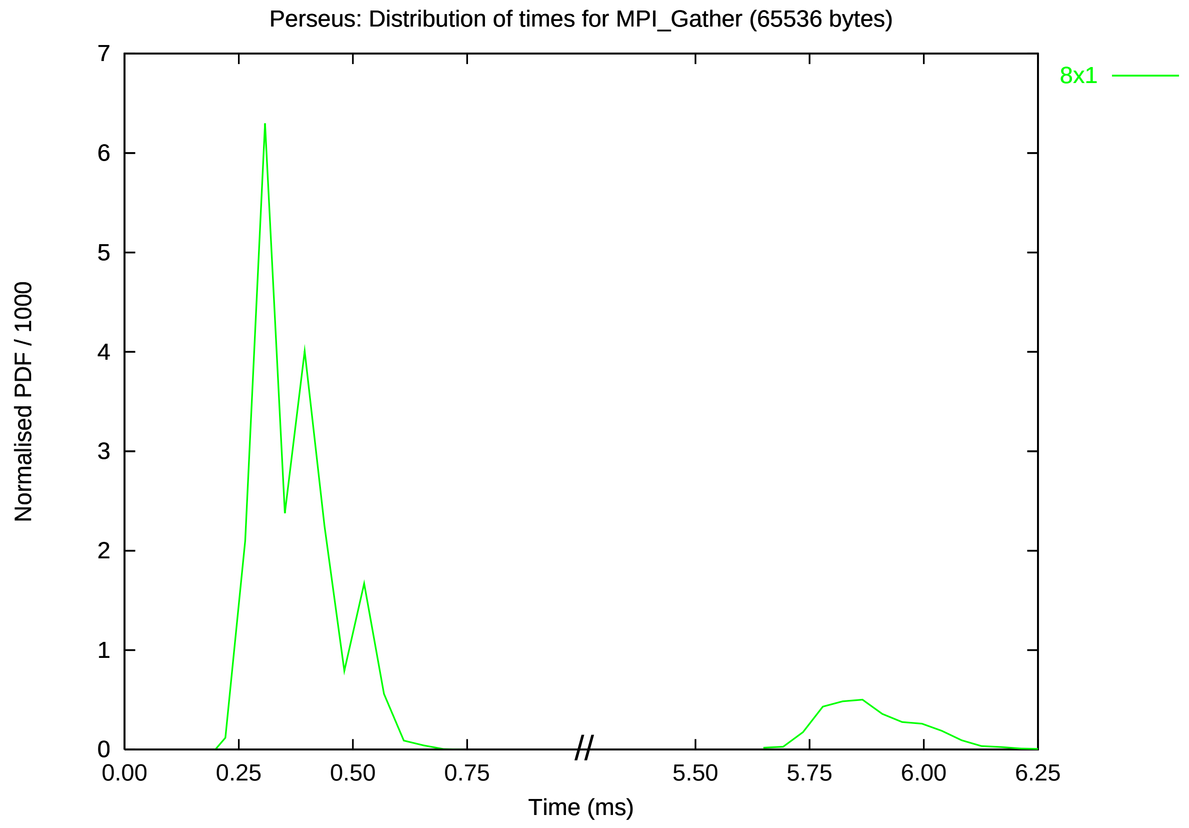


Figure 68: Sampled performance profile for `MPI_Gather` using a 64 Kbyte message with 8x1 processes on Perseus.

the results obtained here were perfectly sufficient for constructing an accurate model of MPI_Gather performance, those more expensive tests were not run.

Finally, therefore, an MPI_Gather on Perseus is modelled by the PEVPM as a collection of point-to-point messages using a simple linear process. As usual this takes into account message-ordering and network contention; and finally the concept of limited bandwidth into an individual processing node is applied to rate-limit the serialising effect of many-one communication, according to the contention scoreboard described in Section 3.5.

In the light of the previous discussion, the performance of MPI_Scatter operations on Orion (in Figures 69-70) are largely self-explanatory, because, with one exception, the same algorithms appear to be used. The exception occurs when processes are run on 32 (and presumably more) distinct nodes. When this is the case, the saw-tooth performance profile is replaced with a normal-like distribution (see Figure 69). This suggests that synchronous sends are used as the constituent point-to-point message, and the root process does not send any acknowledgements to other processes until it has successfully scattered all of the data. In any event, the average time at which the last process completes does not vary from what is expected (for example, compare the similarity between the 32x1 and 16x2 curves in Figure 69). The only significant effect of this protocol change, therefore, is that processes complete (approximately) synchronously with the slowest process, but this will usually make little overall performance difference.

The performance of MPI_Gather routines on Orion is plotted in Figure 71. Unlike the stable performance of the MPI_Scatter operation, the average completion time of individual processes engaging in MPI_Gather operations on Orion can be quite unstable, which can be seen in Figure 71. The reason for this unpredictability seems to lie either in buffering problems or possibly in an unfortunate interplay caused by the use of synchronous constituent point-to-point messages. The saw-toothed distribution for the 8x1/32 Kbyte case shown in Figure 72 suggests that synchronous messages are used to gather the data to the root process; this case behaves as expected. The 8x1/16 Kbyte case, however, shows a massive spike in completion times at approximately 0.03ms. Interestingly, this spike accounts for almost exactly 50% of total completion times, and contains a mixture of completion times from all processes. It seems that the completion of the processes signified by this spike may be expedited by the timely output of an acknowledgement from the root process in preference to other data transfers to be scheduled. Unfortunately, in these cases it seems impossible to analytically predict the completion time of an MPI_Gather at any individual process. However, because a complete performance distribution has been measured, that distribution can still be used in conjunction with the PEVPM to make useful performance predictions where MPI_Gather operations are used.

The structural characteristics of the performance of MPI_Scatter routines on the APAC NF (in Figures 73-74) are almost identical to those of Perseus and Orion, with

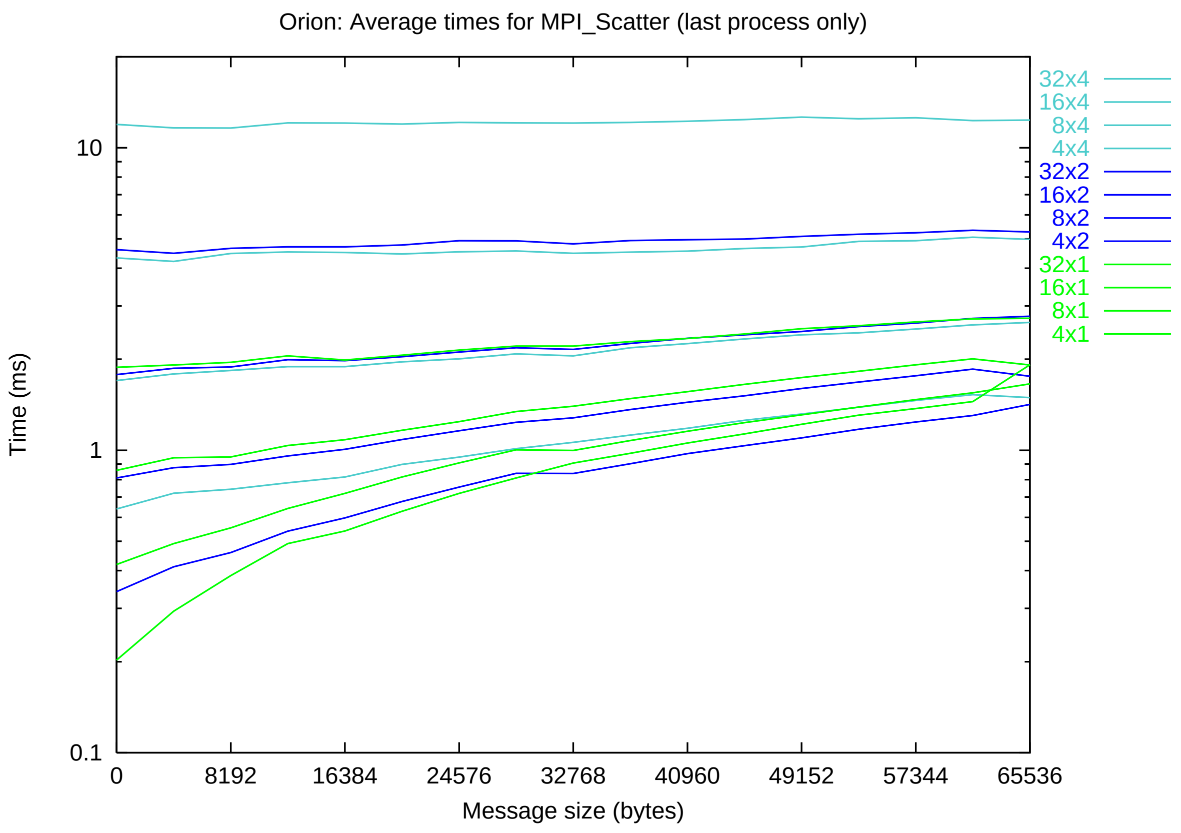


Figure 69: Average times for `MPI_Scatter` using large message sizes with various numbers of communicating processes on Orion, measured at the last process to complete.

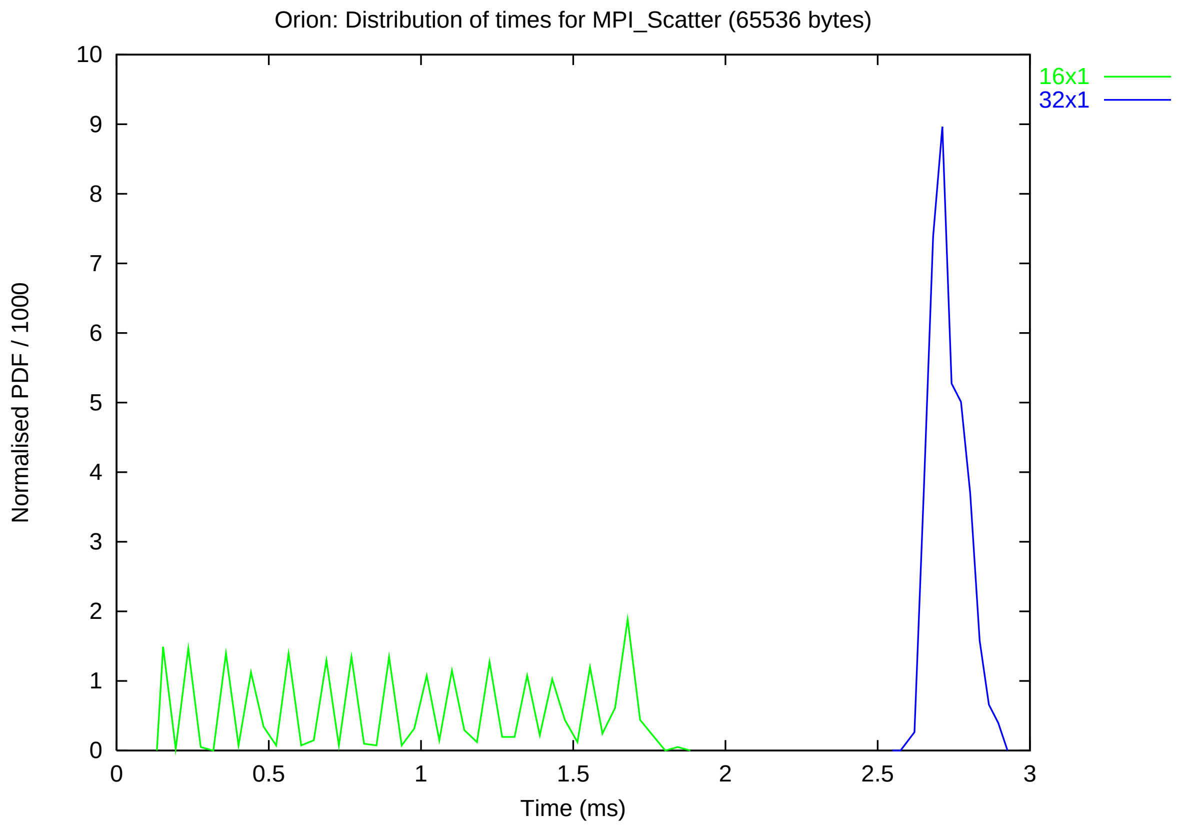


Figure 70: Sampled performance profile for `MPI_Scatter` using 64 Kbyte messages with 16x1 and 32x1 processes on Orion.

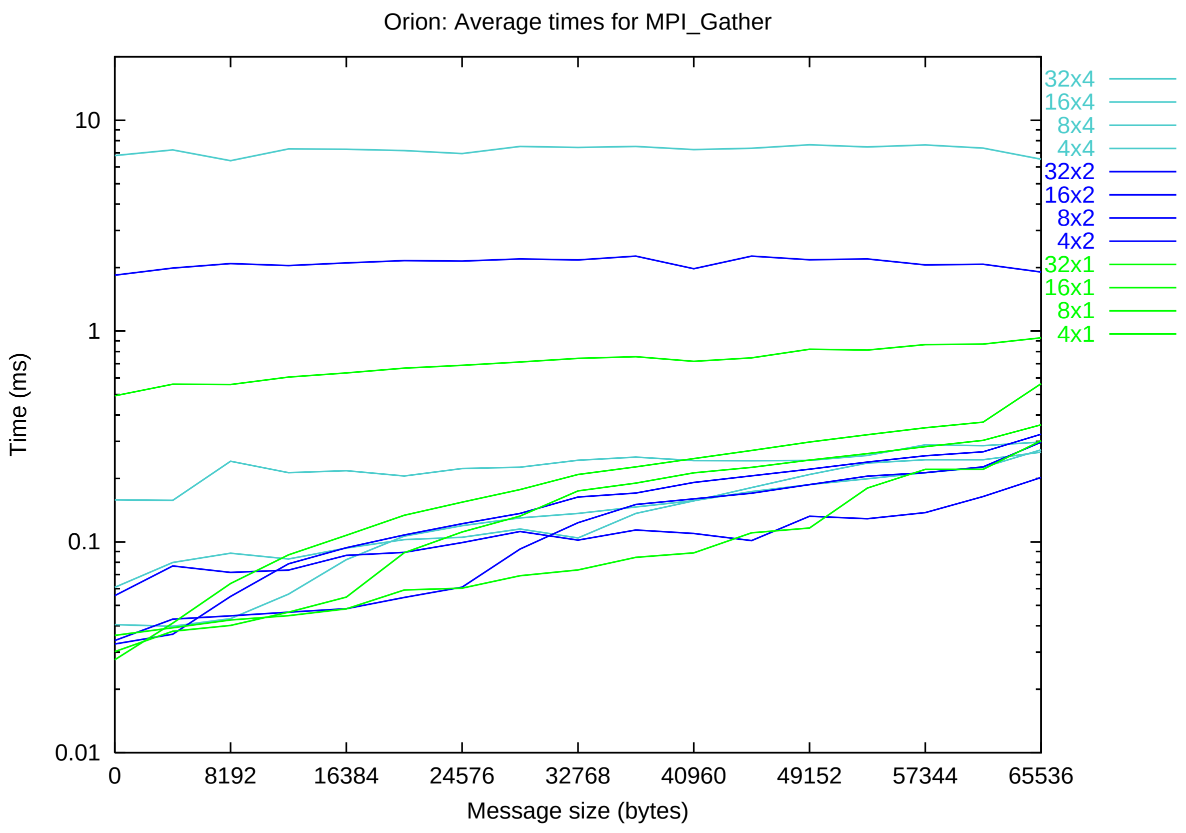


Figure 71: Average times for `MPI_Gather` using large message sizes with various numbers of communicating processes on Orion, measured at both gathering and sending processes.

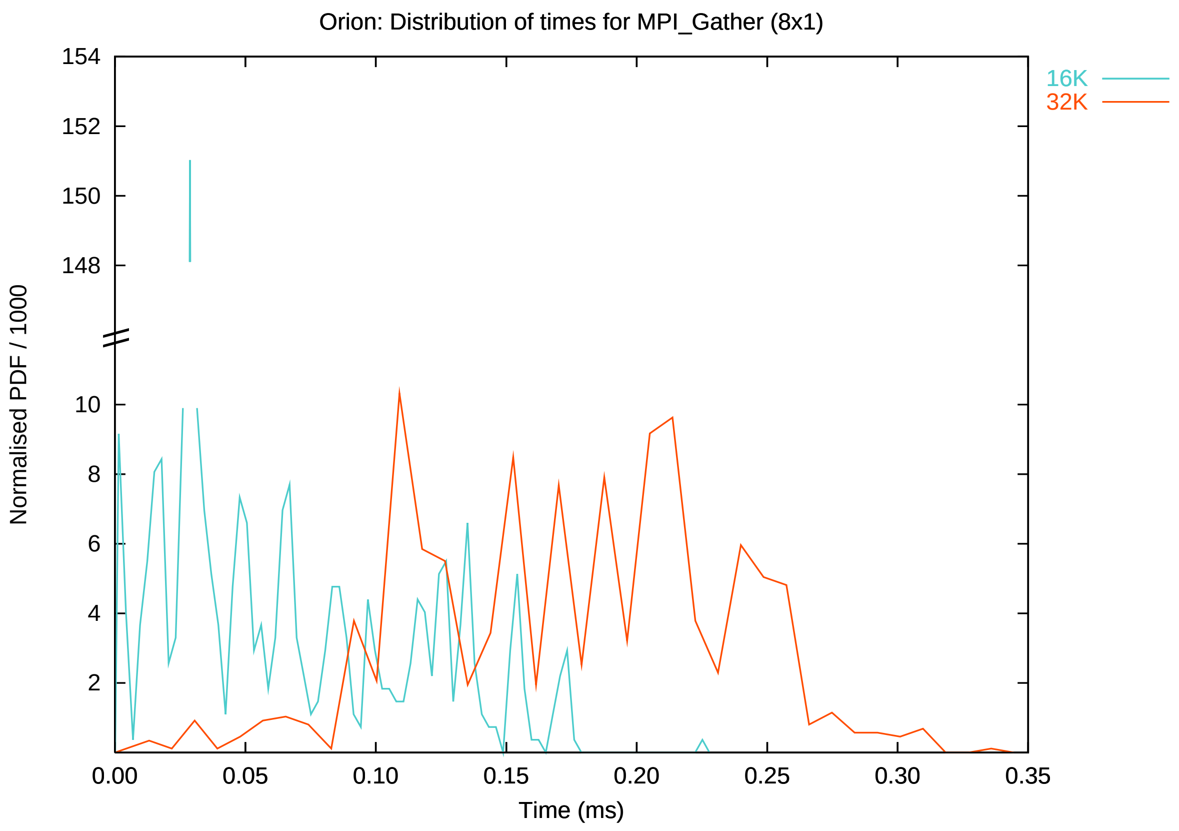


Figure 72: Sampled performance profile for `MPI_Gather` using 32 KByte and 64 Kbyte messages with 8x1 processes on Orion.

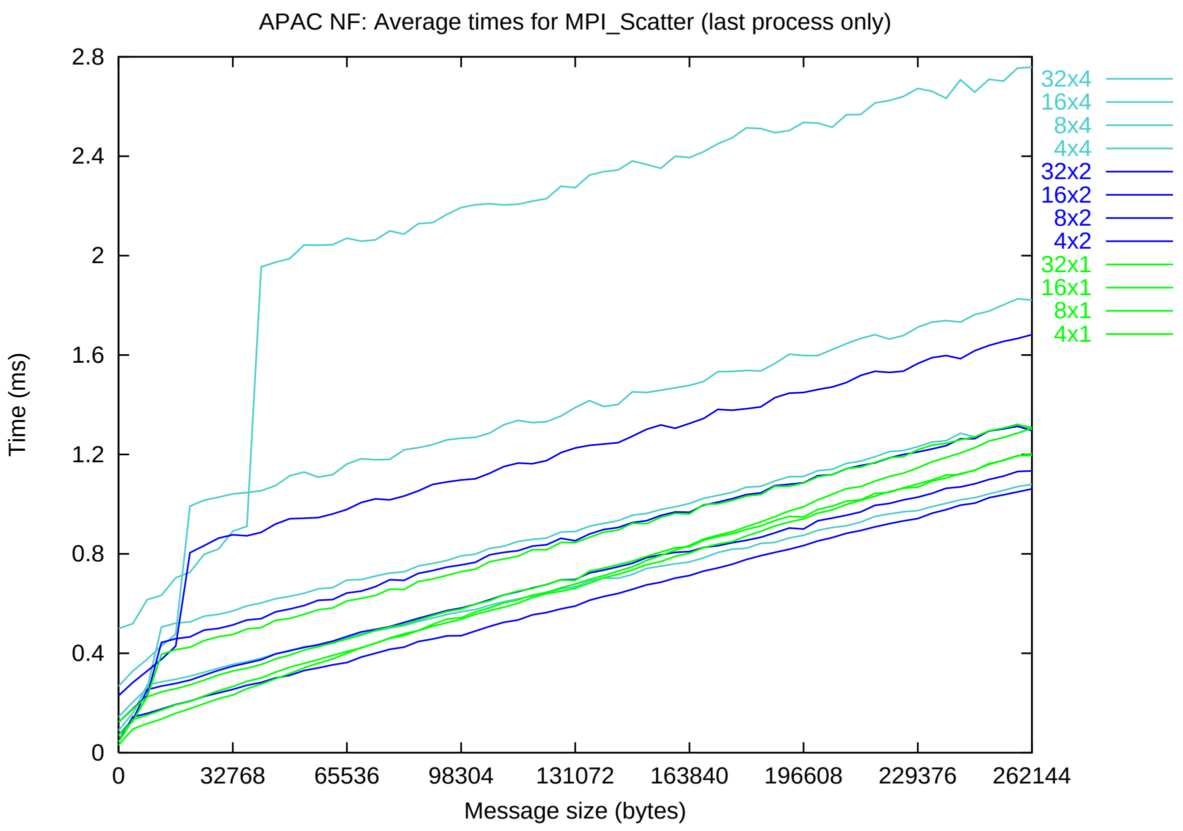


Figure 73: Average times for `MPI_Scatter` using large message sizes with various numbers of communicating processes on the APAC NF, measured at the last process called.

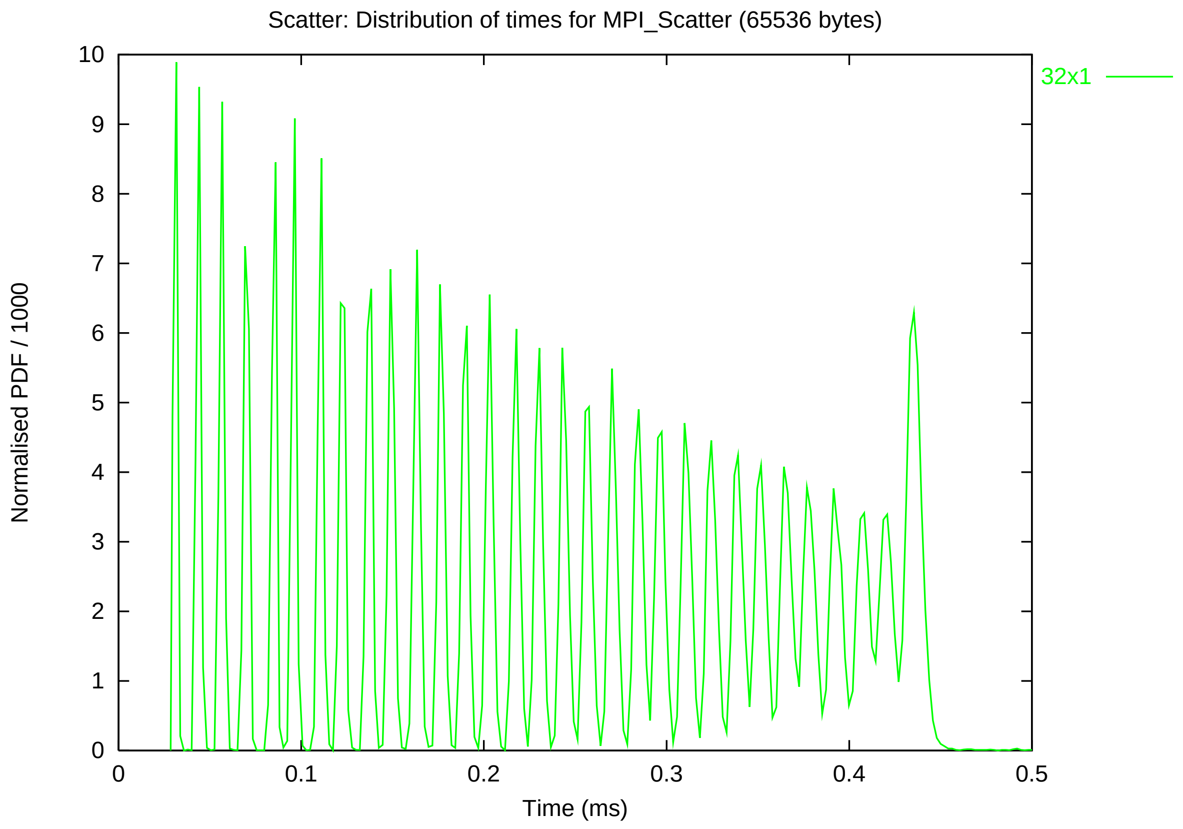


Figure 74: Sampled performance profile for `MPI_Scatter` using 64 Kbyte messages 32x1 processes on the APAC NF.

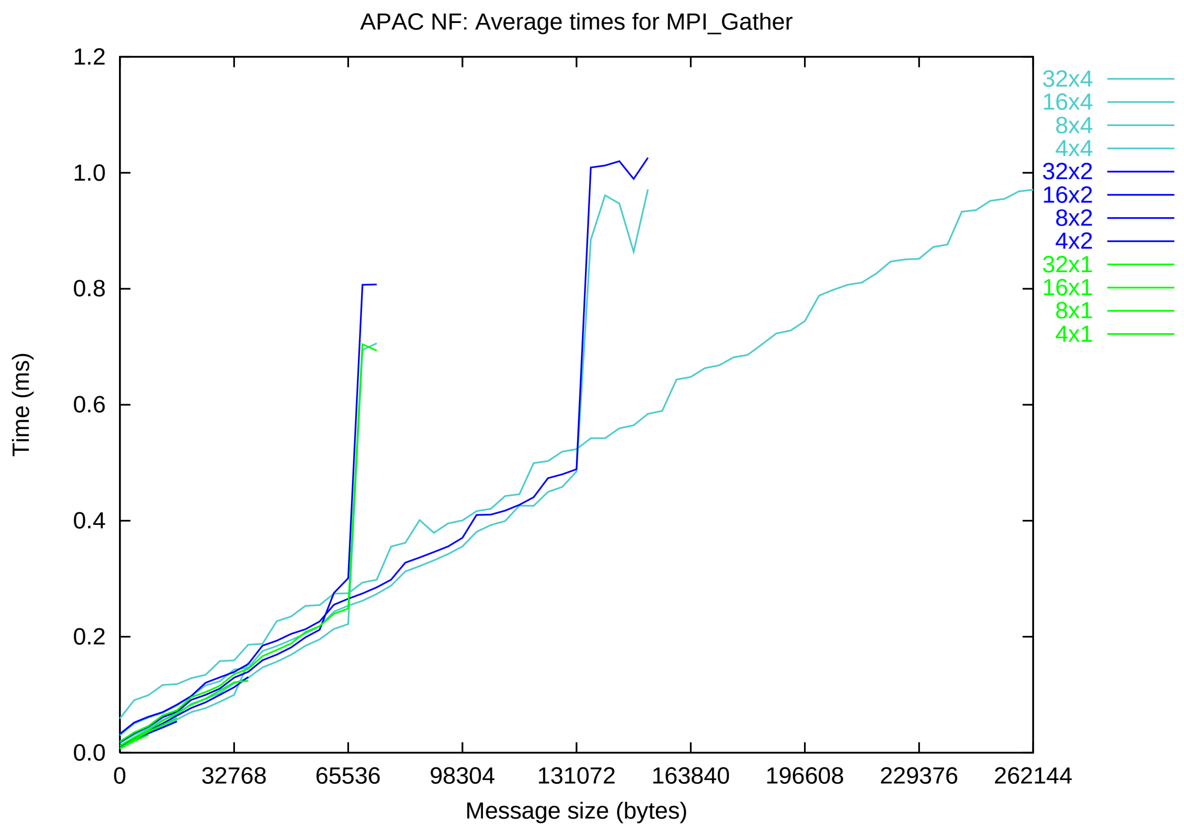


Figure 75: Average times for `MPI_Gather` using large message sizes with various numbers of communicating processes on the APAC NF, measured at both gathering and sending processes.

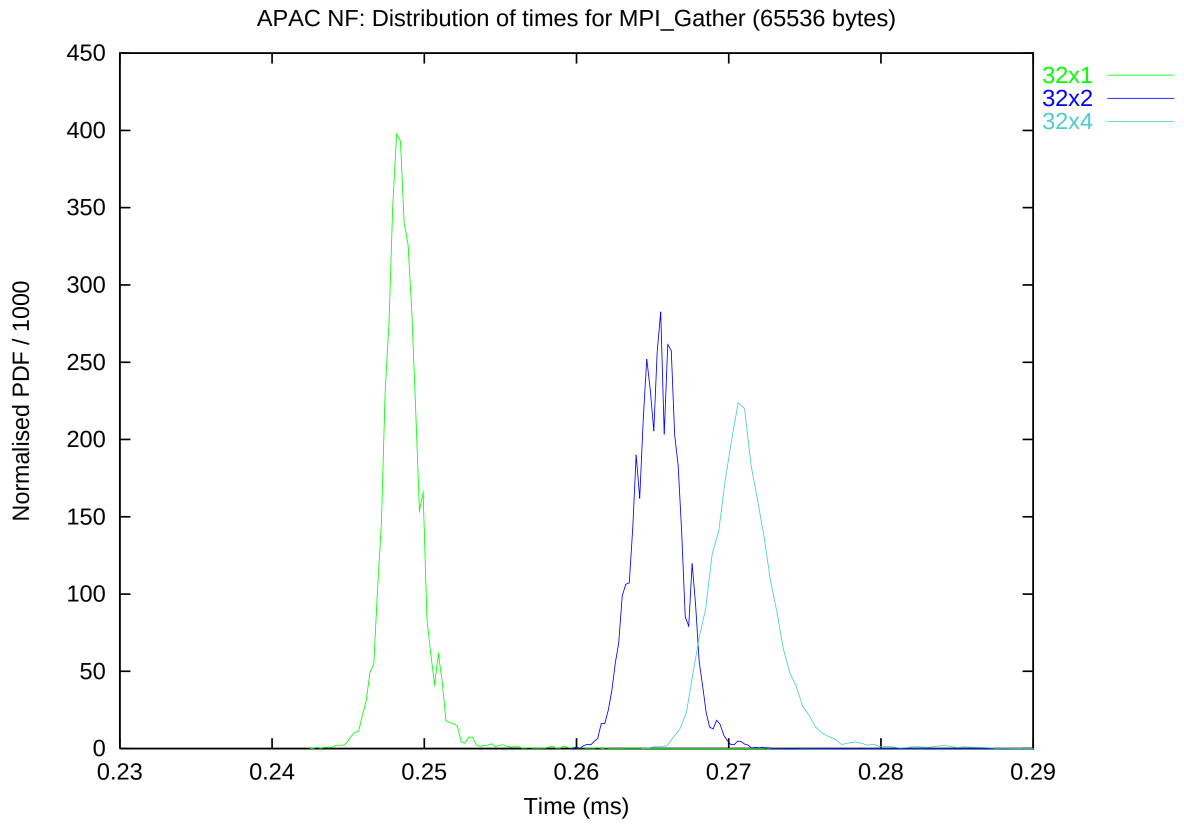


Figure 76: Sampled performance profile for `MPI_Gather` using a 64 Kbyte message with 32x1, 32x2 and 32x4 processes on the APAC NF.

several minor exceptions. Firstly, there are large jumps in performance at various sizes for various numbers of processes. These jumps occur where message sizes of 288 bytes per process are encountered, coinciding with similar jumps in performance of the MPI_Scatter's constituent point-to-point messages at that size (due to packetising effects; see Section 4.5.1). Secondly, the distribution of performance results on a per-process basis is far more stable than on Perseus or Orion, which attests to the quality of the hardware and MPI implementation. It is also interesting to note that the variability in completion time for a particular processes increases slightly in proportion with its position in line to receive data. This is because the APAC NF's underlying QsNet hardware uses synchronous message delivery, and therefore any small delays are accumulated throughout an MPI_Scatter. This property also results in the final note; that the last peak encompasses twice as many results as any of the others. This is because it accounts for the completion time of not only process number n but also the root process.

The APAC NF appears to use a similar MPI_Gather technique to the one Orion employs when 32 or more nodes are used, in that all processes involved in a gather finish synchronously (indicated by the distribution shown in Figure 76). However, in contrast to Perseus or Orion, where significant performance penalties are incurred as more processes are involved, the performance of MPI_Gather operations on the APAC NF are far less dependent on the number of processes used, attested to by the similarity of the curves in Figure 75. Once again, this highlights how well the hardware and MPI implementation on the APAC NF have been tuned to achieve maximum performance; with the exception of several obvious non-linearities in the figure which will now be explained. The abrupt increase in completion times seen at 64 Kbytes and 128 Kbytes are caused by some sort of buffering problem, which has not yet been tracked down. At the point of these abrupt changes, it seems that buffer space is being exhausted, and hence large delays are incurred while earlier messages are finalised and the buffers are reclaimed. Worse, shortly after these buffers are exhausted, the MPI implementation issues an internal error and crashes; thus explaining the lack of results after these abrupt changes. Regardless of this bug (the source of which is still under investigation – a bug report has been submitted to the manufacturer), the stable performance of the MPI_Gather operation before the onset of buffer shortages portends how it is likely to perform in the absence of such problems.

Finally, there are several variants of MPI_Scatter and MPI_Gather that can be easily modelled using trivial modifications to the models just discussed. Firstly, the MPI_Scatterv and MPI_Gatherv routines, which respectively scatter and gather varying amounts of data from each process, can be modelled by varying the size of the point-to-point messages that construct them. Secondly, the MPI_Allgather routine, which is essentially an MPI_Gather but where all processes receive the result, can be modelled as either (depending on how it is implemented) n MPI_Gathers or an MPI_Gather followed by an MPI_Bcast.

5.5 Results for MPI_Alltoall

The most intensive MPI communication routine is MPI_Alltoall, which sends distinct data of size s/n from each process in a communicator to every other process in that communicator, where s is the total size of the data on all processes and n is the number of processes involved. More precisely, MPI_Alltoall performs a transposition of data stored across a set of processes, redistributing it so that the j^{th} block of data from process i is moved to the i^{th} block of data on process j, for example like this:

```
// scatter part executed at all processes
for (i = 0; i < n; i++){
  MPI_Isend(sendbuf + i * s/n, s/n, MPI_BYTE, i, ...);
}
// gather part executed at all processes
for (i = 0; i < n; i++){
  MPI_Recv(recfbuf + i * s/n, s/n, MPI_BYTE, i, ...);
}
```

The actual implementation of an MPI_Alltoall operation must deal with buffering issues and completion testing for asynchronous sends, which are not considered in the example implementation above, but these do not affect the essential communication structure of the call. As the comments in the example code hint, an MPI_Alltoall is actually just a collection of MPI_Scatter-like and MPI_Gather-like communication patterns performed simultaneously by every process. The scatter-like part of the MPI_Alltoall implementation above uses asynchronous point-to-point routines to avoid deadlock and to maximise the performance of outgoing traffic at each process. Like an MPI_Gather, the gather-like part of an MPI_Alltoall can be optimised by allowing the receive operations at each process to accept messages from other processes in any order, thereby avoiding serialisation delays and reducing buffering requirements.

There are, of course, many other ways of implementing the MPI_Alltoall operation. It could be implemented using MPI_Scatter and MPI_Gather routines. Because of its balanced and systematic pattern of data exchange, it could also be implemented using a collection of MPI_Sendrecv operations in a statically determined permutation pattern. This is a common approach, usually utilising a circular neighbour communication pattern with $s \in \{1, ..., n-1\}$ steps in which all processes r send to process $(r + s \bmod n)$ and receive from process $(r - s \bmod n)$ [184]. Note that the circular neighbour approach does not match processes $(r+s \bmod n)$ and $(r-s \bmod n)$ to each other in each step, although that can be achieved using a butterfly-like communication pattern if the number of processes is a power of two, and doing so may improve performance slightly. Regardless of which MPI_Alltoall algorithm is used, the overall result of the operation remains the same: each of n processes exchanges a unique block of data of size s/n with each of the other $n-1$ processes; of these total $n(n-1)$ exchanges, n can be made to proceed concurrently in any network with sufficient parallelism (i.e. switched networks up to their backplane limit, for example in Perseus; fat trees, for example in Orion or the APAC NF; etc). This amount of data movement, approximately equivalent to concurrent sequences of $n-1$ `MPI_Sendrecv` calls of s/n bytes at every process, provides a yardstick by which the performance of any `MPI_Alltoall` implementation can be critiqued.

The measured completion times of `MPI_Alltoall` operations on Perseus were extremely variable, making them difficult to present in a succinct way. Even more so than the `MPI_Bcast` operation (examined in Section 5.2), the execution of `MPI_Alltoall` operations was plagued by huge numbers of very slow completion times, especially when large numbers of processes or large data sizes were used. Like the slow completion times observed for the `MPI_Bcast`, these slow completion times were due to packet losses and their associated TCP/IP retransmit timeouts, caused by extreme network load. Similar catastrophic packet loss under extreme network load was noted by Carns *et al.* [57] in an all-to-all-like communication pattern that occurs during initial connection setup in the LAM 6.1 version of MPI [345].

Because of these extraordinarily long completion times, simply presenting the average completion times of `MPI_Alltoall` operations on Perseus is very misleading. Firstly, it is useful to present how often retransmit timeouts occurred during the benchmarking tests. It is easy to distinguish results that were subject to retransmit timeouts on Perseus because the version of the Linux kernel that was used during the tests has a minimum retransmit timeout value of 200ms (±10ms due to the resolution of timeout clock). In the absence of retransmit timeouts, `MPI_Alltoall` operations of up to 64 Kbytes across up to 64x2 processes on Perseus should not take much more than 40ms or so. Therefore, it is safe to assume that any measurements of more than 190ms have suffered a retransmit timeout. The percentages of measurements for various numbers of participating processes and message sizes that were affected by retransmit timeouts, according to this criterion, are shown in Figure 77. This shows that no retransmit timeouts were experienced by 4x1 and 4x2 process `MPI_Alltoall` operations for any message size up to 64 Kbytes. However, as the number of participating processes was increased, the chance of a retransmit timeout occurring during an `MPI_Alltoall` operation increased rapidly, especially for `MPI_Alltoall` operations on large amounts of data. For example, the chance that a process would suffer a retransmit timeout became more likely than not in the 16x1 process case for data sizes greater than 24 Kbytes, in the 32x1 process case for data sizes greater than 12 Kbytes and in the 64x1 process case for data sizes greater than 4 Kbytes. The situation was actually even worse than this: higher-numbered processes were more likely to suffer a retransmit timeout than lower-numbered processes (for reasons that will

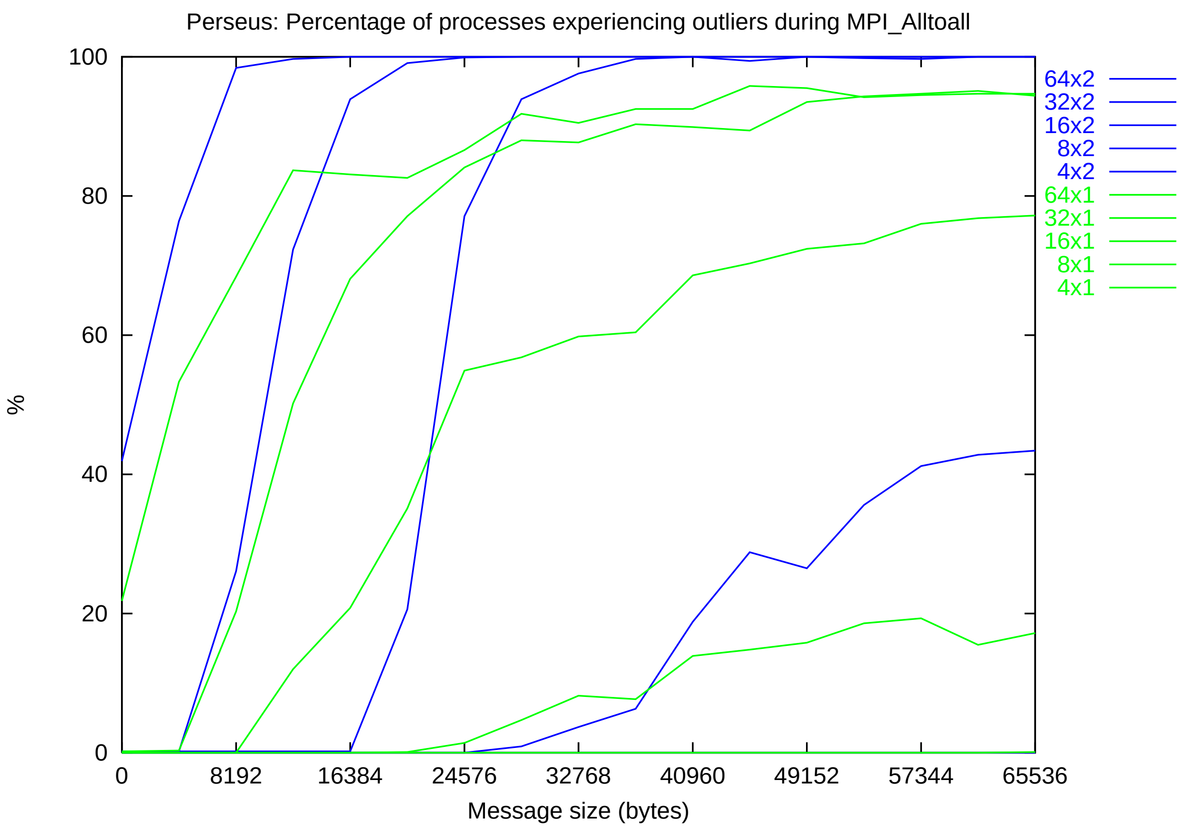


Figure 77: Percentage of processes experiencing a TCP/IP retransmit timeout during an `MPI_Alltoall` between various numbers of participating processes on Perseus.

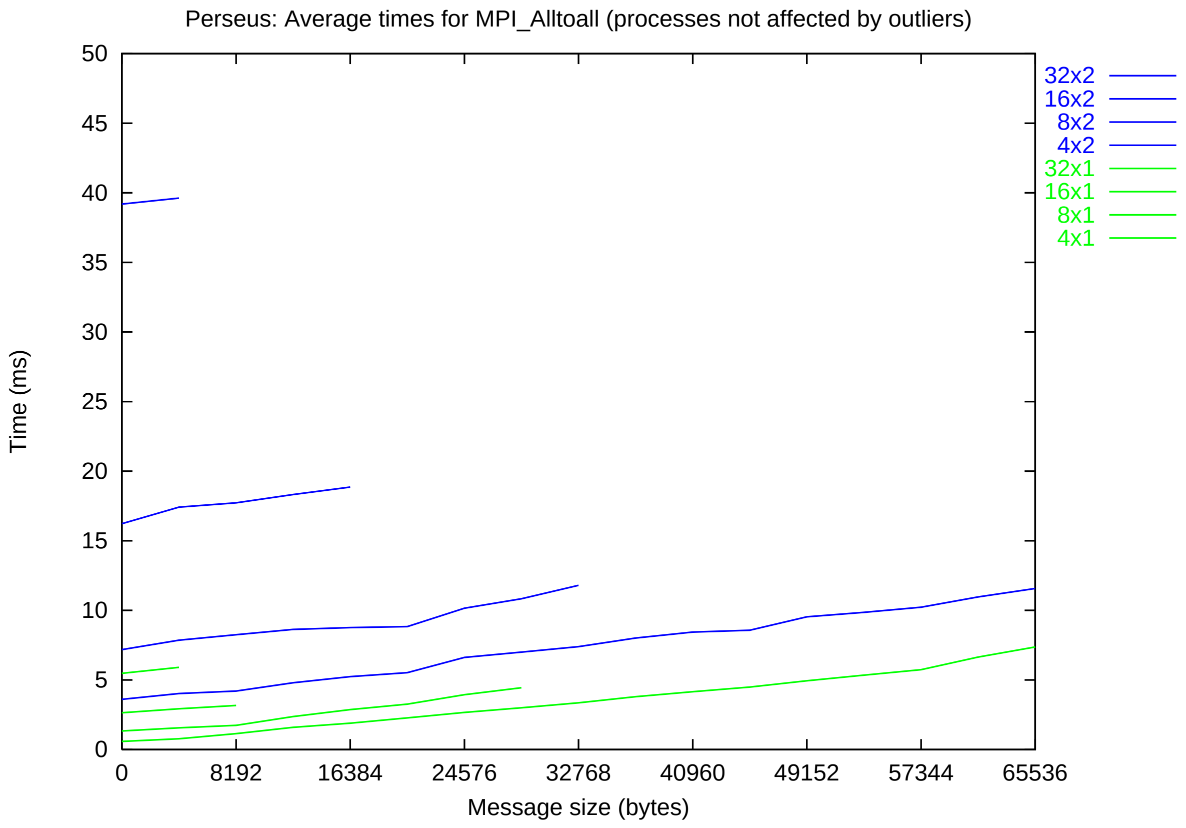


Figure 78: Average times for `MPI_Alltoall` operations of various message sizes with various numbers of participating processes on Perseus when no TCP/IP retransmit timeouts occurred.

be explained shortly). Therefore, in any one MPI_Alltoall operation, the chances of at least one of the participating processes being delayed approached 100% even more quickly. This would devastate parallel efficiency in many real applications, because performance is often bounded by the maximum rather than average time for any process to complete each step in a sequence of computations.

The average completion times for MPI_Alltoall operations on Perseus in the few tests where no retransmit timeouts occurred are shown in Figure 78. In these cases, the MPI_Alltoall operations performed as expected. For example, consider the case where 32x1 processes participated in a 4 Kbyte MPI_Alltoall (i.e. where each of the 32 concurrent processes exchanged 31 messages in sequence, with each message containing 64 bytes of data). Based on the total amount of data transferred and using the results in Figure 33, this was expected to take (roughly) $31 * 0.22 = 6.82$ms. As shown in Figure 78 and Figure 79, this matches quite well with the measured values. The measured average completion time of 5.91ms was probably slightly faster than the rough prediction of average completion time because messages could be processed as they arrived rather than in a predefined order, thereby minimising random serialisation delays. Clearly, the performance characteristics shown in Figure 79 are commensurate with how MPICH's MPI_Alltoall operation was intended to behave – performance scaled linearly with data size, and the number of participating processes affected completion time only through latency serialisation and increased contention.

However, as the network load was increased, the performance of MPI_Alltoall operations on Perseus degraded markedly. Consider, for example, the distribution of completion times for the 64x1 process MPI_Alltoall of 4 Kbytes shown in Figure 79 (take note of the scales in the figure, which were chosen to show the two parts of the distribution with reasonable magnifications, yet with a constant aspect ratio to allow a visual comparison of the area under each part). In this case, only 47% of processes completed according to performance expectations (i.e. in the part of the curve from 11-16ms), while the other 53% suffered a 200ms retransmit timeout. Performance deteriorated even more under further network load, as shown by the results for the 64x2 process MPI_Alltoall of 64 Kbytes in Figure 80, the most taxing case that was measured. In that case, no processes completed without experiencing a retransmit timeout. About 42% of processes were delayed by 600ms of retransmit timeouts while the remainder were hindered by 1400ms of retransmit timeouts.

In Figure 79, the peaks near the expected completion times (i.e. those between 5-7ms and 11-16ms) represent the completion times of processes that had none of the communications in which they were involved affected by retransmit timeouts. Processes that did have data dropped by the network retransmitted the data after a timeout in the TCP/IP layer. The completion times of processes that finished successfully after that

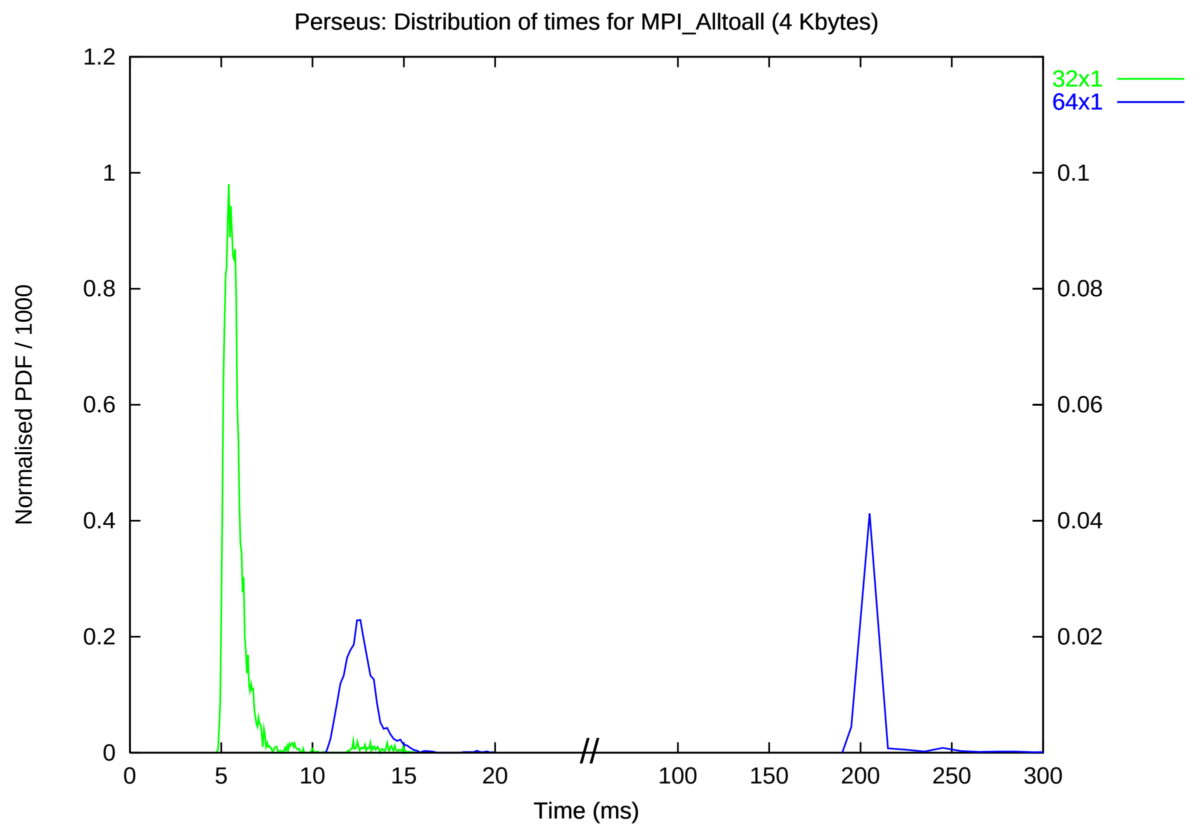


Figure 79: Sampled performance profile for an `MPI_Alltoall` of 4 Kbytes with 32x1 and 64x1 processes on Perseus.

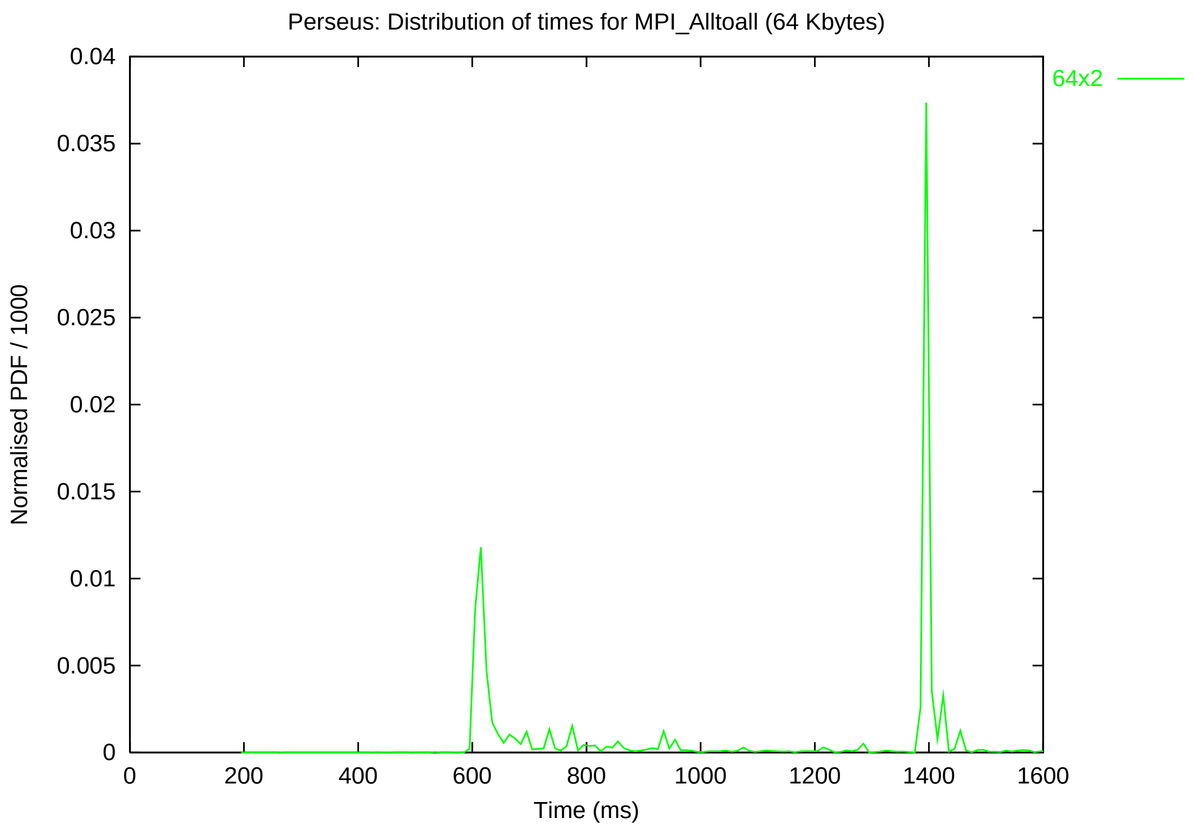


Figure 80: Sampled performance profile for an `MPI_Alltoall` of 64 Kbytes with 64x2 processes on Perseus.

first retransmission are represented by the peak at approximately 200ms. Note that the packets that were dropped in the initial transmission were all dropped at about the same time, timed-out at about the same time, were retransmitted at about the same time and completed at about the same time. Unfortunately, however, in the case of the more taxing `MPI_Alltoall` operation that was mentioned earlier (see Figure 80), all of the retransmitted messages were dropped again because of the instantaneous load placed on the network during retransmission. In line with TCP/IP's exponential backoff of retransmit timeout in the face of repeated packet loss (see Section 4.8), a 400ms delay was imposed before a second retransmission, hence the spike at approximately $200 + 400 = 600$ms. Similarly, the spike at approximately $200 + 400 + 800 = 1400$ms represents the completion times of processes that suffered three successive packet losses. Thus, the unusual distribution of `MPI_Alltoall` completion times that were observed on Perseus were the result of a cascading sequence of network overload, packet dropping and retransmission after exponentially increasing TCP/IP timeouts. Somewhat ironically, this situation arose despite the fact that the network was only overloaded for very short periods of time; the network remained basically idle during the timeout periods.

To further investigate the reasons behind the packet dropping and subsequent TCP/IP retransmit timeouts, the raw completion time data were manually investigated. This showed that completion times represented by the first peak in any particular test iteration belonged to low numbered processes, up to some small but variable limit, where the completion times jumped to a value commensurate with the next timeout level. The reason for this can be traced to MPICH 1.2.0's implementation of the `MPI_Alltoall` operation. Examination of the source code revealed that each process simply executes a loop containing `MPI_Isend`s of the data it needs to transmit and `MPI_Irecv`s of the data it needs to receive, and finally tests for the successful completion of those asynchronous calls with an `MPI_Wait`. Crucially, each process begins by sending data to process 0 first, then process 1, etc, and finally to process $n - 1$ (i.e. in the same fashion as the skeleton code presented at the beginning of this section). At the beginning of this communication pattern, n processes each send a message containing s/n bytes (for a total of s bytes) to process 0, which can only receive one message (s/n bytes) in the same time. The rest of the data remains stored in buffers in the network. Then, the n processes each send another s/n byte message (for a total of s bytes) to process 1. During this time, processes 0 and 1 each receive one message (s/n bytes) of the data *en route* to them. Now even more data is stored in buffers in the network. This continues until every process has transmitted the messages they are required to, at which point there are $n^2/2 - n$ messages (containing a total of $ns/2 - s$ bytes) stored in the network. After this, data is drained from buffers at the rate of n messages per iteration. Furthermore, one acknowledgement message lags every data message, because the systolic nature of the communication pattern makes it impossible to piggy-back acknowledgement messages onto data messages. In the case of a 64x1 process MPI_Alltoall, for example, there will be a peak of approximately $64^2/2 - 64 = 1984$ data messages of size $s/64$ bytes and $64^2/2 = 2048$ acknowledgement messages of size 0 bytes buffered in the network.

Interestingly, the results show a clear correlation between the number of packets and/or amount of data buffered in the network and the number of timed-out message transfers due to packet loss. The Intel 510T switches in Perseus have a 4 MByte shared buffer space. For the MPICH 1.2.0 MPI_Alltoall algorithm (and not accounting for the small overheads contributed by Ethernet framing, acknowledgement messages or unrelated system traffic) this is just enough to support a 64x1 process, 64 Kbyte MPI_Alltoall without exhausting the buffer space. Therefore, the high number of packet losses recorded during much smaller MPI_Alltoall operations (shown in Figure 77) indicates that exhausted buffer space was not the main cause of the packet losses that were observed. However, as more packets and more data accumulated in the switches' buffers, the switches had to undertake more processing to determine which packets should be scheduled for output first. It seems likely, therefore, that increased contention for buffer access by the switches' input queueing processes, packet processing engines and output transmission processes was causing incoming packets to be dropped on arrival.

The performance of MPICH's MPI_Alltoall operation could be vastly improved by staggering the destinations of each processes' MPI_Isends, using the circular neighbour approach described above. The benefits of this would be twofold. Firstly, it would halve the critical path for an MPI_Alltoall operation, thus improving performance in the normal case (i.e. in the absence of retransmit timeouts) by a factor of two. Secondly, it would allow all input and output links to be constantly utilised throughout the operation, and hence prevent data from accumulating in network buffers. In turn, this would drastically reduce the occurrence of retransmit timeouts, improving performance by an order of magnitude. While the incorporation of this improved MPI_Alltoall algorithm into MPICH was beyond the scope of this thesis, it has been suggested to the MPICH development team and is being evaluated for possible inclusion in MPICH 1.2.5.

The average performances of MPI_Alltoall operations on Orion for various numbers of communicating processes and a range of data sizes, shown in Figure 81, fit the expected trends that were mentioned above. The performance scaled linearly with data size, and the number of participating processes affected completion time only through latency serialisation and increased contention. Consider the same example case for Orion that was discussed for Perseus, i.e. the case where 32x1 processes participated in a 4 Kbyte MPI_Alltoall operation. Based on the total amount of data transferred and using the results in Figure 35, this was expected to take (roughly) $31*0.07 = 2.17$ms, which matches well with the measured value of 2.29ms shown in Figure 81. Similarly, a 32x1 process

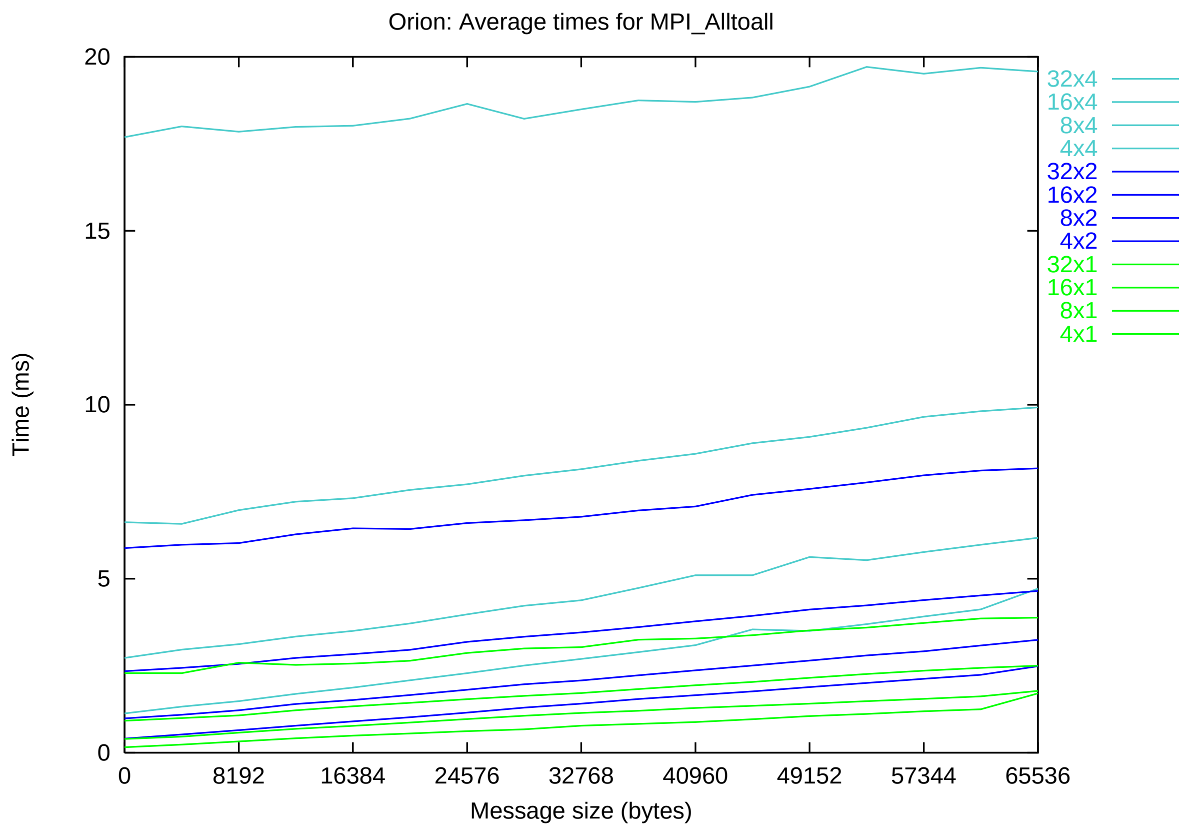


Figure 81: Average times for `MPI_Alltoall` using large message sizes with various numbers of communicating processes on Orion.

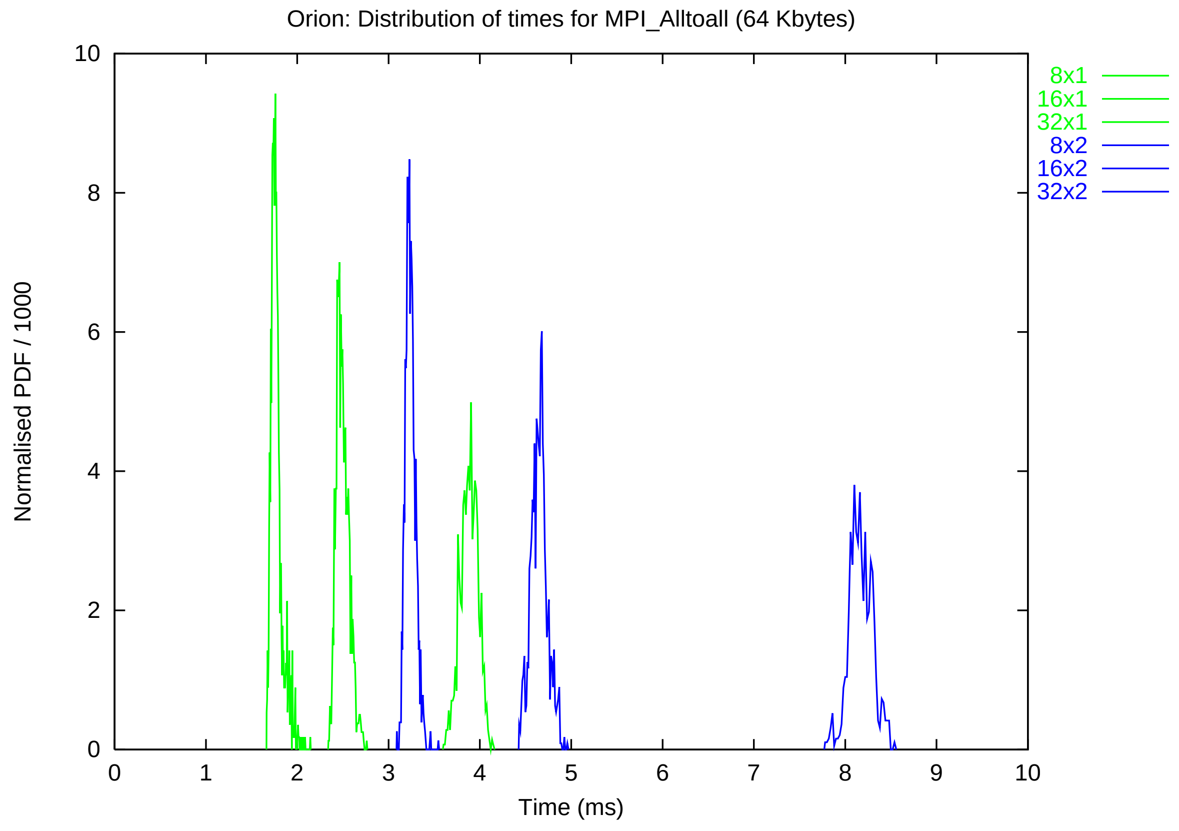


Figure 82: Sampled performance profiles for an `MPI_Alltoall` of 64 Kbytes with 8-32x1-2 processes on Orion.

MPI_Alltoall of 64 Kbytes was expected to take (roughly) $31 * 0.12 = 3.72$ms but actually took 3.88ms. The fact that the measured completion times took slightly longer than the rough predictions of average completion time (rather than the other way around, as was the case for Perseus) suggests that Orion's MPI_Alltoall implementation uses a systolic communication pattern (rather than processing whichever messages arrive first). In particular, the results indicate that Orion almost certainly uses a circular neighbour communication pattern with synchronous MPI_Sendrecv-like operations, although this could not be verified for certain because of the lack of access to the source code for Orion's MPI library. While using synchronous point-to-point operations as the constituent messages of a circular neighbour communication pattern imposes some very short precedence delays, it allows the MPI_Alltoall operation to proceed in an orderly fashion that is not prone to degraded performance from increased contention if the communication pattern becomes skewed [184] (which is part of what went wrong in MPI_Alltoall operations on Perseus). The orderly execution of MPI_Alltoall operations on Orion leads to dependable performance in those operations, which can be seen in the stability and relatively low variance of the performance profiles shown in Figure 82.

Figures 81 and 82 also capture two other important performance characteristics of the circular neighbour implementation of the MPI_Alltoall operation. Firstly, it is obvious that the completion time of this MPI_Alltoall algorithm scales very poorly as the number of participating processes is increased, especially for small amounts of data; it scales as $O(n)$ rather than $O(\log n)$, like many of the group operations that have been investigated so far. This is caused by the circular neighbour algorithm's serialisation of $n-1$ latency overheads. Similarly to the optimised MPI_Scatter and MPI_Gather algorithms discussed earlier, a more advanced MPI_Alltoall algorithm could group messages (especially small ones) together in order to reduce the number of messages on the critical path (albeit at the expense of more data transfer overall), thereby reducing the number of costly latency overheads. Secondly, it is clear that the MPI_Alltoall algorithm has not been optimised for SMP nodes. If it had been, 8x2 process MPI_Alltoalls (for example) would perform about as well an 8x1 process MPI_Alltoalls with the same amount of total data. This could be achieved in the 8x2 process case by concatenating the two half-sized messages (relative to the 8x1 process case) from each node to any other and transmitting them as one message. Instead, the results show that the circular neighbour approach used on Orion treats all processes equally, regardless of which node they reside on, and this results in non-optimal performance. A further consequence of this is that the nx2 process MPI_Alltoall is slightly slower than the $2n$x1 process MPI_Alltoall (and likewise for the nx4 process case versus the $4n$x1 or $2n$x2 process cases) despite the same total number of communicating processes in each case. This is because of the increased contention delays that occur when more processes per node are involved in simultaneous communication.

The average performance of `MPI_Alltoall` operations on the APAC NF under various conditions (see Figure 83) show that the APAC NF's `MPI_Alltoall` operation scales much better than Perseus' or Orion's. Despite the lack of access to the source code for the APAC NF's highly tuned MPI implementation [77], some guesswork and a careful investigation of the measured performance data were able to uncover most of the implementation details for the APAC NF's `MPI_Alltoall` operation. Firstly, Figure 83 shows abrupt performance jumps for `MPI_Alltoall` operations at various total data sizes. In particular, these jumps occur when the size of constituent messages between participating processes s/n cross 2 Kbyte and 8 Kbyte boundaries. In addition, the slopes of the average performance trends also change at these points. Together, these facts suggest that the APAC NF's `MPI_Alltoall` operation makes use of one of three different algorithms, depending on the size of the data being sent from each process to every other. Performance profiles for 16x1 process `MPI_Alltoall`s of various data sizes on the APAC NF can be seen in Figure 84, which also highlights the situations where each of the three different `MPI_Alltoall` algorithms have come into play.

When s/n is less than or equal to 2 Kbytes, a latency tolerant algorithm is used to minimise the effects of latency serialisation. The performance of the `MPI_Alltoall` operation for this range of data sizes is consistent with a multiply-trunked tree-like algorithm, where the data scattered by each process is recursively transferred through intermediate processes in the same way as for the optimised `MPI_Scatter` routine discussed in Section 5.4. This communication pattern requires $\lceil \log_2 n \rceil$ stages, where $n/2$ bidirectional, $s/2$ byte messages are required in each stage. Thus, the time for a 0 byte `MPI_Alltoall` reduces from 31 times the minimum message latency (using the circular neighbour approach) to 5 times the minimum message latency. Curiously, the results indicate that the constituent messages should be modelled using `MPI_Isend` messages rather than `MPI_Sendrecv` messages, despite the duplex nature of the communication. This suggests that the bottlenecks in bidirectional message passing, described in Section 4.6, are not being encountered in this case. With this caveat, a 32x1 process, 32 Kbyte `MPI_Alltoall` would require 5 stages, each with 16 pairs of processes simultaneously exchanging 16 Kbyte messages.

Note, however, that in the case above the total amount of data sent and received by each process has increased from (the intrinsic) 31 Kbytes to $5 * 16 = 70$ Kbytes; the extra 39 Kbytes of data is transferred as a result of each process acting as an intermediary for data that is ultimately destined for other processes. As the total data size for an `MPI_Alltoall` is increased, the overhead incurred in transmitting extra (intermediate) data using this latency-tolerant algorithm will increasingly nullify the performance advantage gained from having less messages on the critical path. At some point these trade-offs will completely negate each other. Beyond this point the circular neighbour approach will perform better than the latency-tolerant algorithm. In order to leverage this

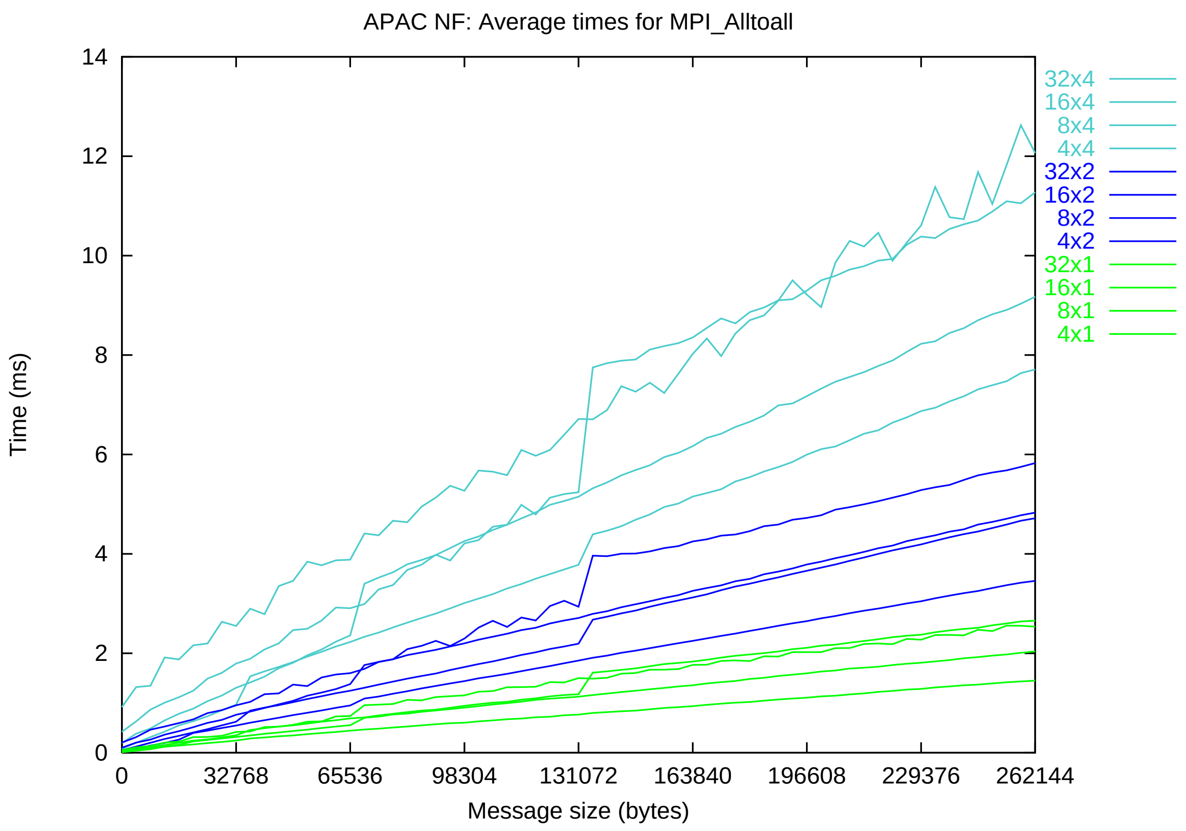


Figure 83: Average times for `MPI_Alltoall` using large message sizes with various numbers of communicating processes on the APAC NF.

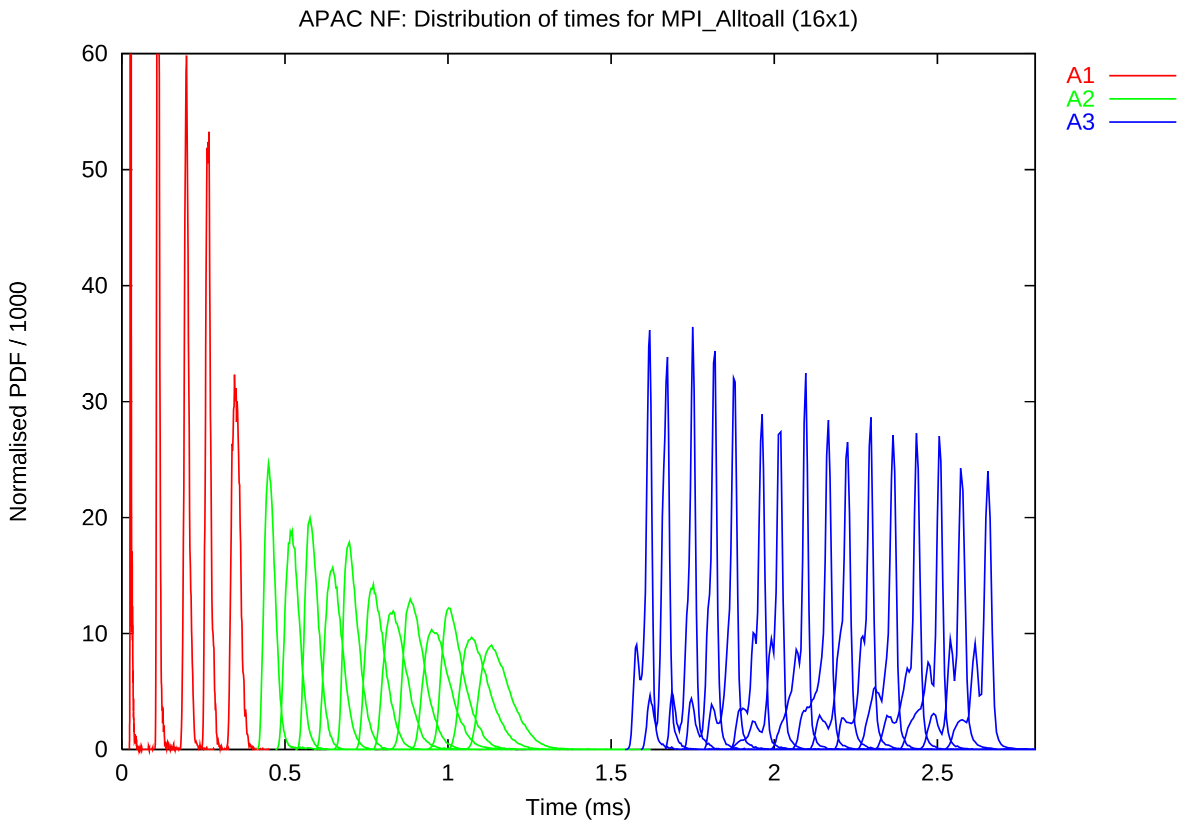


Figure 84: Sampled performance profiles for `MPI_Alltoall`s with 16x1 processes on the APAC NF, using algorithms A1 (0-512-2048 bytes per constituent message), A2 (2560-512-8192 b.p.c.m.) and A3 (8704-512-32768 b.p.c.m.).

fact, the APAC NF's `MPI_Alltoall` operation uses a circular neighbour approach when s/n is more than 2 Kbytes but less than or equal to 8 Kbytes. While the jumps in the curves (in Figure 83) at points where the constituent message sizes hit 2 Kbyte imply that the change-over point has not been perfectly optimised, the small value of those jumps indicates that it has at least been quite well optimised; and furthermore, 2 Kbytes is a neat message size at which to switch between algorithms.

As the (intrinsic) amount of point-to-point data exchanged between each pair of processes increases beyond 8 Kbytes, a third `MPI_Alltoall` algorithm seems to come into play; although it is also possible that this change is triggered by some sort of buffering problem, but that seems quite unlikely given that no commensurate jumps were observed for 8 Kbyte messages in earlier point-to-point performance measurements. Surprisingly, this third algorithm performs worse than a circular buffer algorithm (theoretically) would. Therefore it is difficult to imagine the reasoning behind this third algorithm, and deducing its mechanism is almost impossible without access to the source code. Consequently, the raw performance measurements of the `MPI_Alltoall` operation will have to serve as the performance model for the `MPI_Alltoall` operation in any case that this third algorithm would be used.

Finally, despite the attention that has been paid to optimising the `MPI_Alltoall` operation on the APAC NF, it is worth noting that, like on Orion, the APAC NF's `MPI_Alltoall` operation does not appear to have been optimised for SMP nodes. When multiple processes per node are involved in an `MPI_Alltoall`, all processes fight for access to the node's local network interface and the increased contention delays that result from this retard the completion of the operation.

5.6 Discussion of Collective Computation

There are two basic types of collective computation routines: reduce and scan; both of which are very similar. A reduction operation combines data from all processes to either one process (with an `MPI_Reduce` operation), all processes (with an `MPI_Allreduce` operation) or cyclicly across the communicator (with an `MPI_Reduce_Scatter` operation). The basic `MPI_Reduce` operation over n processes, where $D(i, j)$ represents the j^{th} data item at process i, combines all data items to the root process r according to:

$$D(r, j) \; = \; D(0, j) * D(1, j) * \; ... \; * D(n-1, j)$$

where $*$ represents a reduction function, which is always assumed to be associative. MPI provides a number of predefined reduction functions, which can be used to find the maximum or minimum data value, the location of that data value, the sum or product of all data values, or the bitwise or logical `and`, `or` or `xor` of all data values; note that these are assumed to always be commutative. It is also possible for user-defined reduction functions to be supplied, which need not be – although they can be – either associative and/or commutative. The scan operation is a prefix-reduction operation that only partially combines data each process, according to:

$$D(k,j) \ = \ D(0,j) * D(1,j) * \ ... \ * D(k,j)$$

for $k = 0,\ 1,\ ...\ ,\ n-1$.

Essentially, both of these operations work by gathering data to processes and applying some data reduction operator (and then optionally broadcasting or scattering the result). Because the computation (data reduction) part of the operation is usually of insignificant cost compared to the communication (data gathering) part of the operation (at least, for predefined reduction operations), the performance of these collective computation routines can be reasonably accurately modelled by the time taken for data movements alone: in other words, by simply reusing the performance models already constructed for `MPI_Gather` operations in Section 5.4. Even if the computation part were not insignificant, for example for some custom reduction operation, it would not be hard to insert delays at appropriate points of the PEVPM model of an `MPI_Gather` operation to create a consonant performance model.

5.7 Summary

This chapter described how MPIBench was used to accurately characterise the performance of collective communication on the same three parallel machines for which point-to-point communication was examined in the previous chapter. In particular, the results in this chapter were analysed in detail to determine either the means by which `MPI_Bcast`, `MPI_Barrier`, `MPI_Scatter`, `MPI_Gather`, `MPI_Alltoall` and their variants were constructed from constituent point-to-point messages, or whether those operations were hardware-assisted. In the case of collective operations constructed from point-to-point operations, performance models of those operations were built from the bottom-up using performance models for their constituent point-to-point operations. Each performance model of a collective operation was used to analyse the performance of the algorithm, and was also used as a small validation of the PEVPM modelling approach (see chapter 3). In the case of hardware-assisted collective operations, the benchmark results were obtained to serve as stand-alone empirical performance models. Many of the results obtained in this chapter, along with results from the previous chapter on point-to-point message-passing performance, will be used to validate some complete MPI programs in the next chapter.

It is also worth summarising the major results of this chapter, which are interesting in their own right. It was graphically demonstrated that there can be vast differences in the time that individual processes take to complete their part in collective operations, attested to by the wide and complicated performance profiles that were observed; for example the Pascal's Triangle-shaped PDFs for software-based `MPI_Bcast`s or the saw-tooth-shaped PDFs for `MPI_Scatter`s. While this behaviour has been alluded to in other studies, none have made a significant attempt to quantify it. The work presented here, however, provided in-depth analyses, and in particular demonstrated that the standard MPI performance benchmarking technique of measuring average completion time for collective operations at one process is completely inadequate. Once a thorough understanding of the general shape of the performance profile for an operation is gained, however, average completion times can provide valuable summary information for the wary. Sometimes, however, these performance profiles can be very complicated, and even chaotic. This was demonstrated for the `MPI_Alltoall` operation on Perseus; in this case, the PDF-approach to MPI performance benchmarking made it possible to explain that the very poor performance that was observed was due to massive contention causing packet loss and subsequent network timeouts. Uncovering the source of this deficiency would have been very difficult, if in fact it was not completely overlooked, by performance benchmarking tools other than MPIBench.

Chapter 6

Case Studies

6.1 Introduction

As explained in Chapter 1, parallel computing is essential for solving very large problems in a reasonable amount of time. The vast array of problems that can benefit from parallel computing come from all fields of science and engineering, and include Computational Fluid Dynamics (CFD) simulations of fluid flow and heat transfer, Finite Element Analysis (FEA) techniques in structural analysis and for modelling wave propagation, weather forecasting and climate modelling, *ab initio* quantum chemistry calculations for determining the electronic structure of molecules, N-body simulations in astronomical modelling and molecular dynamics computations, the simulation of protein folding, the simulation of quantum field theories about the fundamental forces of nature and the basic structure of matter, simulated annealing in optimisation problems, and signal processing in the Search for Extra-Terrestrial Intelligence (SETI). The main computations that must be performed in all of these applications usually reduce to common mathematical techniques (albeit on large amounts of data) such as matrix manipulations, the numerical evaluation of differential equations, Fourier transformations, data sorting and the like. A range of parallel algorithms have been devised over the years to carry out each of these mathematical functions.

All of these parallel algorithms can be distinguished, from a performance perspective, by the computation/communication pattern that they employ. Although a parallel program's computation/communication pattern must be defined in terms of individual computation and communication operations, it is possible to classify any particular pattern into one of three broad types. Parallel codes exhibit either *regular-local* communication, *regular-global* communication or *irregular* communication (which may be either local or global); interspersed computation is implicit in each case. The regularity of a communication pattern refers to its temporal and spatial repetitiveness, and in particular its temporal periodicity and spatial symmetry. The spatial extent of a communication pattern, either local or global, describes whether processes communicate only with their immediate neighbours, or with any processes, no matter how distantly they are connected. In general, the broad performance properties of parallel codes depend closely upon the class of computation/communication pattern to which they belong. Thus, studying the performance of parallel programs drawn from each of the three classes of communication/computation pattern can lead to a good understanding of many of the general performance characteristics of all parallel programs. More importantly in terms of this thesis, all performance modelling systems for parallel programs can be prudently assessed by their ability to cope with cases from each of the three classes of parallel code.

All of the previous techniques for the performance modelling of parallel programs that were described in Chapter 2 are inadequate for one or more of the following reasons:

1. Most are not general enough to model arbitrary parallel code, especially code with irregular or, even more so, nondeterministic computation/communication patterns;

2. Some are so inflexible that unique models must be created every time a single code is run with different input parameters (such as problem size) or under different environmental conditions (such as the number of processors available);

3. Many are prohibitively expensive, in terms of the resources required for model creation and/or model evaluation;

4. Others are too inaccurate to be useful.

In contrast, the PEVPM performance modelling technique that was developed in Chapter 3 is completely general, arbitrarily flexible, very cost-effective and extremely accurate. This chapter presents nine case studies that substantiate these claims. The nine case studies are comprised of three example codes, each coupled, in turn, to each of the three parallel machines that were benchmarked in Chapters 4 and 5. The three example codes are representative of the three classes of parallel programs, which were described above. The first example code performs a Jacobi Iteration (JI), using a regular-local communication pattern constructed from a number of `MPI_Send` and `MPI_Recv` operations. The second example code uses a master-slave (or task-farm) approach to process a randomly generated Bag of Tasks (BOTs), which has both irregular and nondeterministic communication and computation requirements. The third example code calculates a two-dimensional Fast Fourier Transform (FFT) using the `MPI_Alltoall` operation, which has a regular-global communication pattern. The three parallel machines – Perseus, Orion and the APAC NF – span a wide performance range and exemplify low-end, middle-of-the-range and high-end parallel supercomputers. This matrix of test cases provides a good general reflection of all of the ways in which parallel computing can be used, and hence serves as a trustworthy basis on which to evaluate the quality of the PEVPM performance modelling system.

6.2 Jacobi Iteration

Differential equations are arguably the most important mathematical tools for describing complex physical systems. They are central in a huge number of mathematical models, ranging across the scientific and engineering problems described in the last section and beyond. Unfortunately, it is often difficult to find exact solutions to differential equations, especially if they are non-linear, have coefficients that vary with time, are of high order or must be solved for many inputs and/or initial conditions – all of which are commonplace in problems of real-world interest. Fortunately, however, there are a number methods for finding approximate solutions to differential equations. One of the simplest of these is Jacobi Iteration, which belongs to a family of algorithms known as relaxation methods. All of these methods begin by discretising whichever differential equation is under consideration. This leads to a system of linear equations that can be solved by iteration. Discretisation of a two-dimensional differential equation, for example, will translate it into a two-dimensional grid of data that can be solved using some form of nearest-neighbour updating scheme, or stencil. Although Jacobi Iteration is rarely used in practice because it converges more slowly that other (more complex) relaxation methods, such as Gauss-Sidel Iteration (GSI) or Successive Over Relaxation (SOR), it serves as a fine example for analysis because, in addition to its simplicity, it shares the same communication pattern as all relaxation methods. More generally, its communication pattern is also similar to that of all parallel algorithms with regular-local communication, including the large and important class of algorithms that perform stencil calculations on regular meshes of data.

The program listing in Figure 85 shows the skeleton code for the Jacobi Iteration that will be examined in this section. For readability, the listing does not include the code for memory allocation and cleanup, loading initialised values of a discretised differential equation into the `grid` array, any MPI call's tag, communicator or status/request variable, or auxiliary functions. Conceptually, every point in the 256x256 `grid` is iteratively updated using a 4-point stencil, such that each point is equal to 0.25 times the sum of the values of its neighbours (excepting boundary values, which do not change). Parallelism is introduced by a one-dimensional data decomposition that splits the `grid` into n subgrids, one for each of the processes involved in the computation. During each iteration, every process transfers the edge of its subgrid to any immediate neighbours in a regular-local communication phase and computes the 4-point stencil on its subgrid in conjunction with any edge data that it received. A one-dimensional data decomposition was chosen instead of a two-dimensional data decomposition because it is easy to implement. Note, however, that it also happens to optimise the performance of the Jacobi Iteration for the case of a small `grid` because it leads to fewer (but larger) message transfers than a two-dimensional data decomposition would, hence minimising latency requirements.

```
int i, j, k, procnum, numprocs;
int iterations = 100000;
int xsize = 256;
int ysize = 256/numprocs+2;
float grid[size][size];
float griddash[size][size];

MPI_Comm_rank(MPI_COMM_WORLD, &procnum);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

for (i = 0; i < iterations; i++){

  // communication part
  if (procnum%2 == 0){
    if (procnum != 0){
      MPI_Send(grid[1], xsize, ... , procnum-1, ... );
    }
    MPI_Send(grid[ysize-2], xsize, ... , procnum+1, ... );
    MPI_Recv(grid[ysize-1], xsize, ... , procnum+1, ... );
    if (procnum != 0){
      MPI_Recv(grid[0], xsize, ... , procnum-1, ... );
    }
  }
  else{
    if (procnum != (numprocs-1)){
      MPI_Recv(grid[ysize-1], xsize, ... , procnum+1, ... );
    }
    MPI_Recv(grid[0], xsize, ... , procnum-1, ... );
    MPI_Send(grid[1], xsize, ... , procnum-1, ... );
    if (procnum != (numprocs-1)){
      MPI_Send(grid[ysize-2], xsize, ... , procnum+1, ... );
    }
  }

  // computation part
  for(j = 1; j < ysize-1; j++){
    for(k = 1; k < xsize-1; k++){
      griddash[j][k]=0.25*
        (grid[j][k-1]+grid[j-1][k]+grid[j][k+1]+grid[j+1][k]);
    }
  }
  swap_ptr(grid, griddash);

}
```

Figure 85: Skeleton code for the Jacobi Iteration.

Conversely, a two-dimensional data decomposition would optimise for a larger `grid` by minimising the amount of edge data that must be transferred (but at the expense of requiring more messages to do so), hence minimising bandwidth requirements. In a real problem, this iteration of communication and computation phases would terminate when some desired level of convergence was obtained between `grid` and `griddash`. Because that termination condition is data-dependent, it could only be determined by actual execution of every computation, which would defeat the purpose of performance modelling. Therefore the example code simply terminates after 1000 iterations. This is perfectly reasonable, as performance comparisons between different Jacobi Iteration implementations on specific parallel machines make far more sense on a per iteration basis.

A PEVPM-annotated version of the Jacobi Iteration from Figure 85 is listed in Figure 86. Only a few minutes were required to manually insert these PEVPM directives, which was achieved by simply applying the rules detailed in Sections 3.4.4 and 3.4.5. This mostly trivial process could easily be carried out by an automated compiler (like that envisaged at the end of Section 3.4.5) with little or no human intervention. Because PEVPM directives are so self-evident, redundant (in that they may be easily generated from the skeleton code) and rather lengthy, they will not be comprehensively layed out for the two other example codes that are discussed in the remaining sections of this chapter.

The most noteworthy PEVPM directives for the Jacobi Iteration shown in Figure 86 are those pertaining to the serial segment of computation at the end of the program listing. As explained in Section 3.4.1, there are a number of traditional methods that are capable of accurately estimating the run-time of a serial segment of computation, especially one as structured as the stencil computation being considered here. For simplicity, however, an empirical method was used to determine appropriate parameters for the PEVPM `Serial` directives governing the estimation of completion time of the stencil calculations. One iteration of the stencil computation was actually run on Perseus, Orion and the APAC NF, and the execution times were measured. Because the per-processor amount of computation required for each iteration of the stencil calculations varies inversely with the number of processors available, the `Serial` computation time for each iteration of stencil computations was modelled by the measured execution time divided by *numprocs*.

A small amount of finesse was required to: 1) ensure the viability of this simple model, which is only reasonable given constant cache and memory system access patterns; and 2) make certain that the times required for computation and communication were not so disproportionate as to render either one unimportant. This firstly required choosing a problem size such that the Jacobi Iteration code and its data structures, most importantly `grid` and `griddash`, would always fit completely in level 2 cache *or* core memory – not a mixture of the two – no matter how many processors were used. Each of the 64 nodes in Perseus had 128MB of RAM split between 2 Pentium III processors, each of those with

```
   int i, j, k, procnum, numprocs;
   int iterations = 100000;
   int xsize = 256;
   int ysize = 256/numprocs+2;
   float grid[size][size];
   float griddash[size][size];
   MPI_Comm_rank(MPI_COMM_WORLD, &procnum);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
// PEVPM Loop iterations = 100000
// PEVPM {
   for (i = 0; i < iterations; i++){
// PEVPM Runon c1 = procnum%2 == 0
// PEVPM &     c2 = procnum%2 != 0
// PEVPM {
     if (procnum%2 == 0){
// PEVPM Runon c1 = procnum != 0
// PEVPM {
       if (procnum != 0){
// PEVPM Message type = MPI_Send
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum
// PEVPM &         to = procnum-1
         MPI_Send(grid[1], xsize, ... , procnum-1, ... );
       }
// PEVPM }
// PEVPM Message type = MPI_Send
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum
// PEVPM &         to = procnum+1
       MPI_Send(grid[ysize-2], xsize, ... , procnum+1, ... );
// PEVPM Message type = MPI_Recv
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum+1
// PEVPM &         to = procnum
       MPI_Recv(grid[ysize-1], xsize, ... , procnum+1, ... );
// PEVPM Runon c1 = procnum != 0
// PEVPM {
       if (procnum != 0){
// PEVPM Message type = MPI_Recv
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum-1
// PEVPM &         to = procnum
         MPI_Recv(grid[0], xsize, ... , procnum-1, ... );
       }
// PEVPM }
     }
// PEVPM }
```

Figure 86 (part 1/2): Skeleton code for the Jacobi Iteration with PEVPM annotations.

```
// PEVPM {
     else{
// PEVPM Runon c1 = procnum != numprocs-1
// PEVPM {
       if (procnum != (numprocs-1)){
// PEVPM Message type = MPI_Recv
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum+1
// PEVPM &         to = procnum
         MPI_Recv(grid[ysize-1], xsize, ... , procnum+1, ... );
       }
// PEVPM }
// PEVPM Message type = MPI_Recv
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum-1
// PEVPM &         to = procnum
       MPI_Recv(grid[0], xsize, ... , procnum-1, ... );
// PEVPM Message type = MPI_Send
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum
// PEVPM &         to = procnum-1
       MPI_Send(grid[1], xsize, ... , procnum-1, ... );
// PEVPM Runon c1 = procnum != numprocs-1
// PEVPM {
       if (procnum != (numprocs-1)){
// PEVPM Message type = MPI_Send
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum
// PEVPM &         to = procnum+1
         MPI_Send(grid[ysize-2], xsize, ... , procnum+1, ... );
       }
// PEVPM }
     }
// PEVPM }
// PEVPM Serial on perseus time = 3.24/numprocs
// PEVPM Serial on orion time = 5.70/numprocs
// PEVPM Serial on sc time = 1.582/numprocs
     for(j = 1; j < ysize-1; j++){
       for(k = 1; k < xsize-1; k++){
         griddash[j][k]=0.25*
           (grid[j][k-1]+grid[j-1][k]+grid[j][k+1]+grid[j+1][k]);
       }
     }
     swap_ptr(grid, griddash);
   }
// PEVPM }
```

Figure 86 (part 2/2): Skeleton code for the Jacobi Iteration with PEVPM annotations.

512KB of level 2 cache. Each of the 32 nodes in Orion had 4GB of RAM split between 4 UltraSparc II processors, each of those with 4MB of level 2 cache. Each of the 32 nodes in the APAC NF had 4GB of RAM split between 4 Alpha EV68 processors, each of those with 8MB of level 2 cache. A 256x256 (or smaller) `grid` of (4 byte) `float`s (plus its identically sized shadow `griddash`), totalling 512KB of data, represents the only problem size that could fit in level 2 cache when using anywhere between 1 and 128 processors of either Perseus, Orion or the APAC NF; one processor of Perseus being the limiting case. Similarly, a 11585x11585 (or larger) `grid` (and `griddash`), totalling 1GB of data, would be required to ensure that the problem would never drop entirely into level 2 cache on any machine; 128 processors of the APAC NF being the limiting case. Since this larger problem size could not be run on any less than 8 (and more realistically 16) processors of Perseus because of memory limitations, the smaller 256x256 problem size was chosen. This problem size was also admirably suited to the second requirement above, because it led to appreciable amounts of both computation and communication time given the use of any number of processors on either Perseus, Orion or the APAC NF.

Three of the four criteria that were slated (in Section 6.1) to be used for evaluating the utility of the PEVPM modelling system have already been touched upon. Firstly, the generality of the PEVPM has been partially demonstrated by its applicability to a regular-local code, one of the three possible (general) types of parallel program. The applicability of the PEVPM to the two other types of parallel program will be demonstrated in the following two sections. Secondly, the flexibility of the PEVPM has been implicitly demonstrated; because important program and machine parameters (such as *procnum*, *numprocs* and ostensibly data size arguments, by using appropriate compiler techniques) are retained symbolically in PEVPM models, those models can be easily re-evaluated under different input and environmental conditions. Thirdly, the demonstrated simplicity of adding PEVPM annotations to existing code is testament to the low-cost of PEVPM model creation.

The costs associated with model evaluation, and the accuracy of PEVPM models are far more quantifiable. A driver program was manually derived from the PEVPM model of the Jacobi Iteration listed in Figure 86. Because the driver program merely reflected the control structure defined by the PEVPM directives, it only took several minutes to generate; note, however, that this process could also be automated by using appropriate compiler techniques. This driver program was linked with prototype implementations of the PEVPM process sweep/match sweep algorithms (which were detailed in Section 3.5) and hence used to predict the performance of the Jacobi Iteration for many configurations of Perseus, Orion and the APAC NF. These performance predictions are plotted as dashed (or dotted) lines in Figures 87-92. The times required to actually execute the same Jacobi Iterations on the real machines in corresponding situations were also measured and those

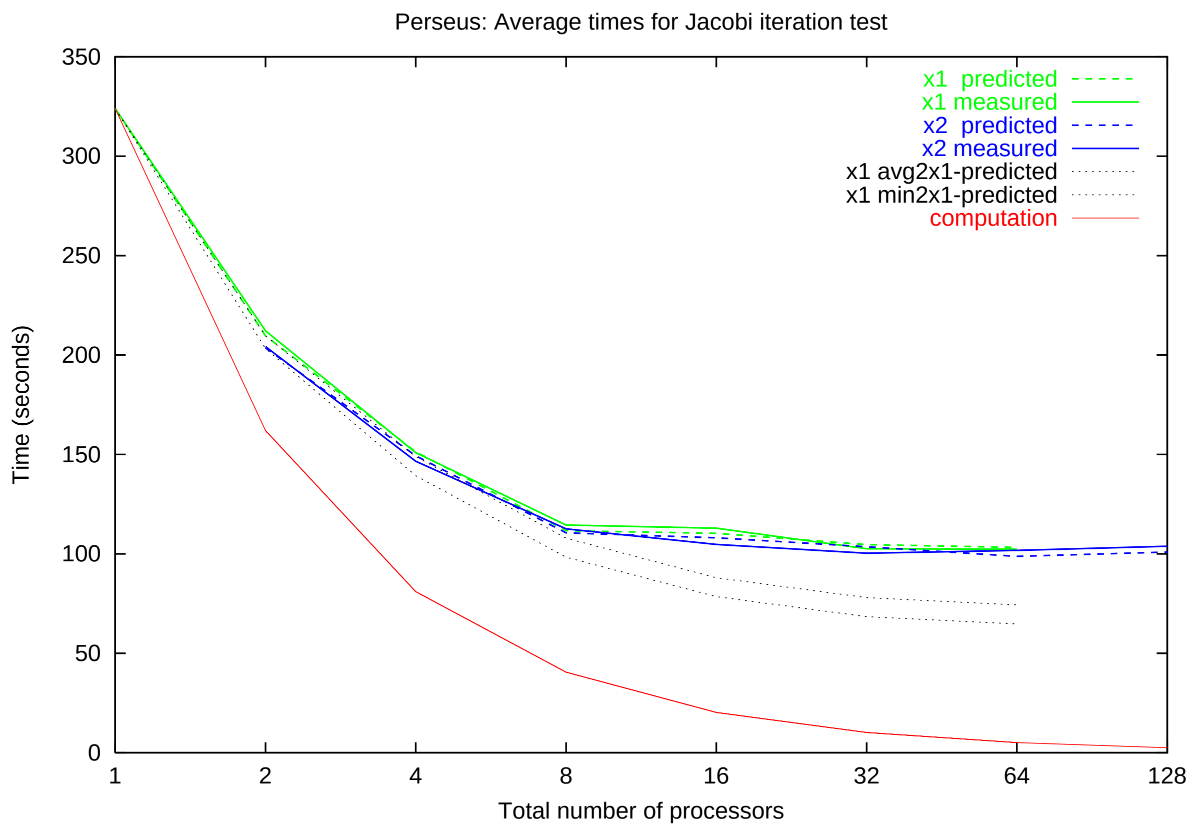


Figure 87: PEVPM-predicted average times and measured average times for the Jacobi Iteration test using 2-64x1-2 processes on Perseus.

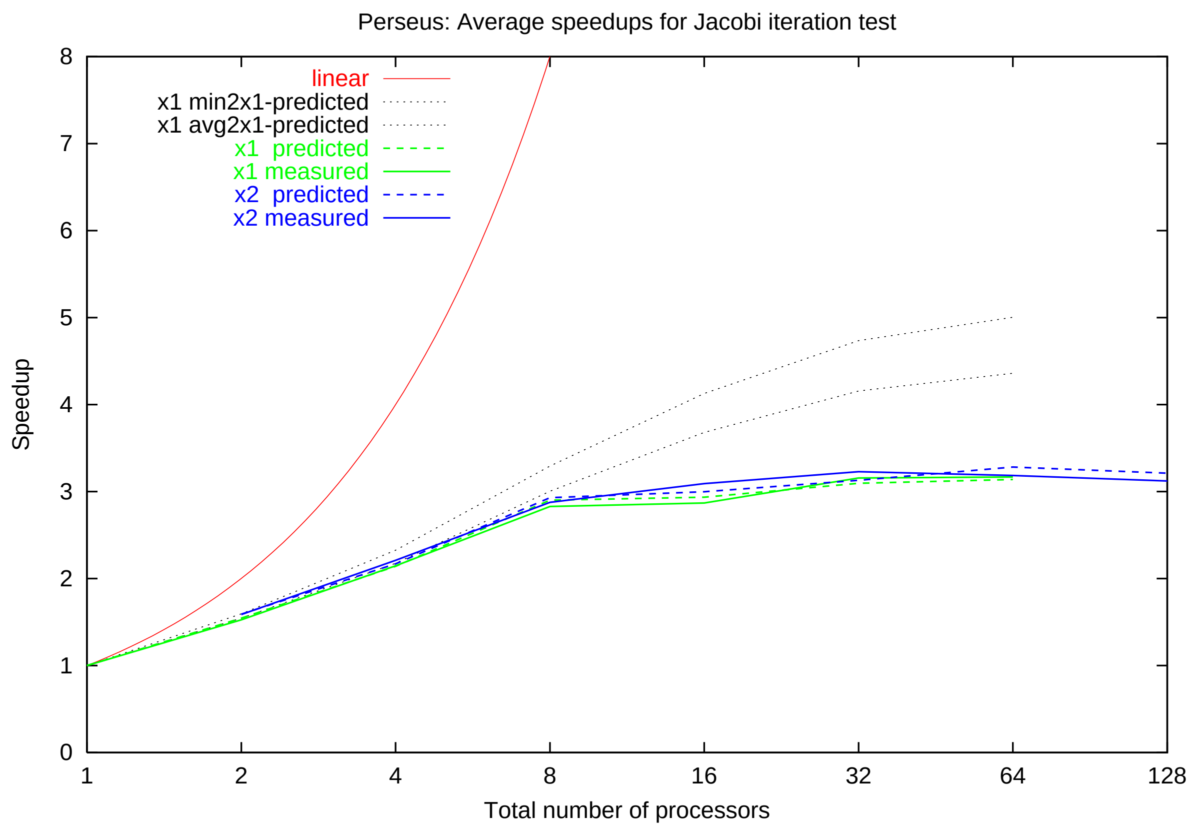


Figure 88: PEVPM-predicted average speedups and measured average speedups for the Jacobi Iteration test using 2-64x1-2 processes on Perseus.

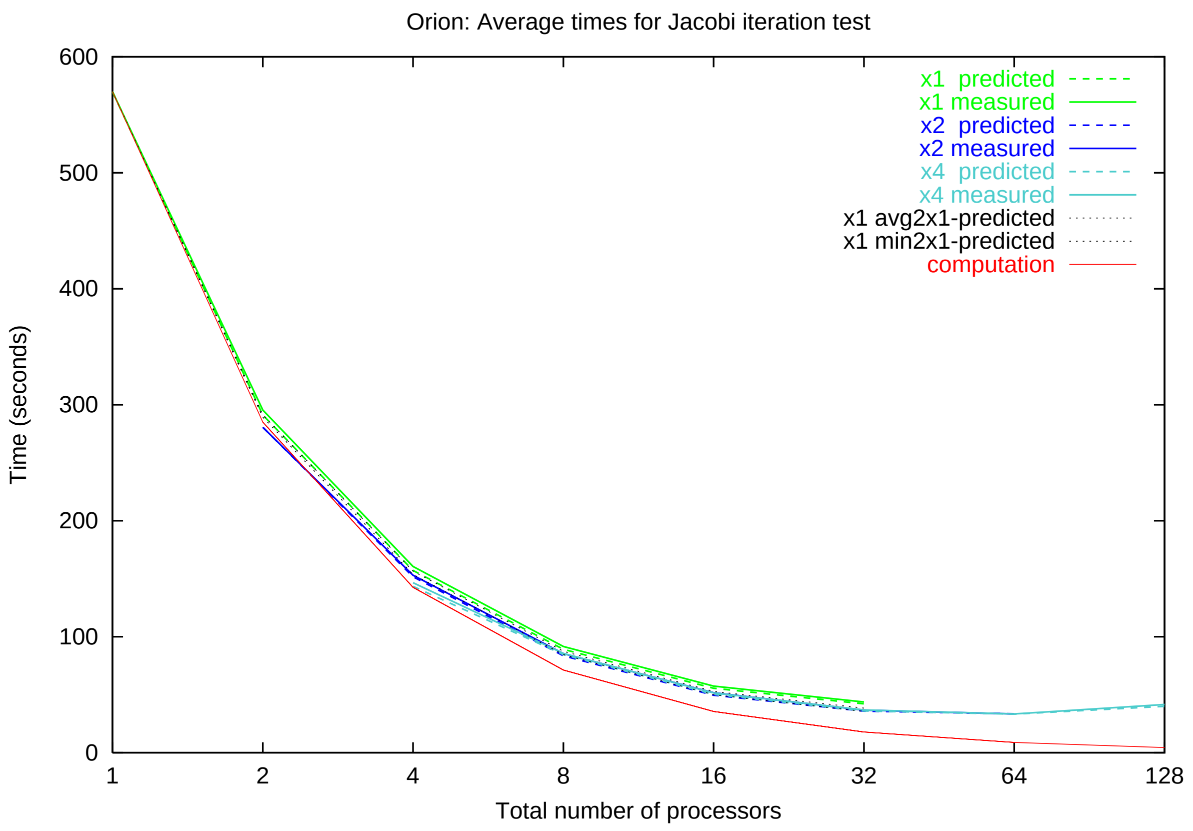


Figure 89: PEVPM-predicted average times and measured average times for the Jacobi Iteration test using 2-32x1-4 processes on Orion.

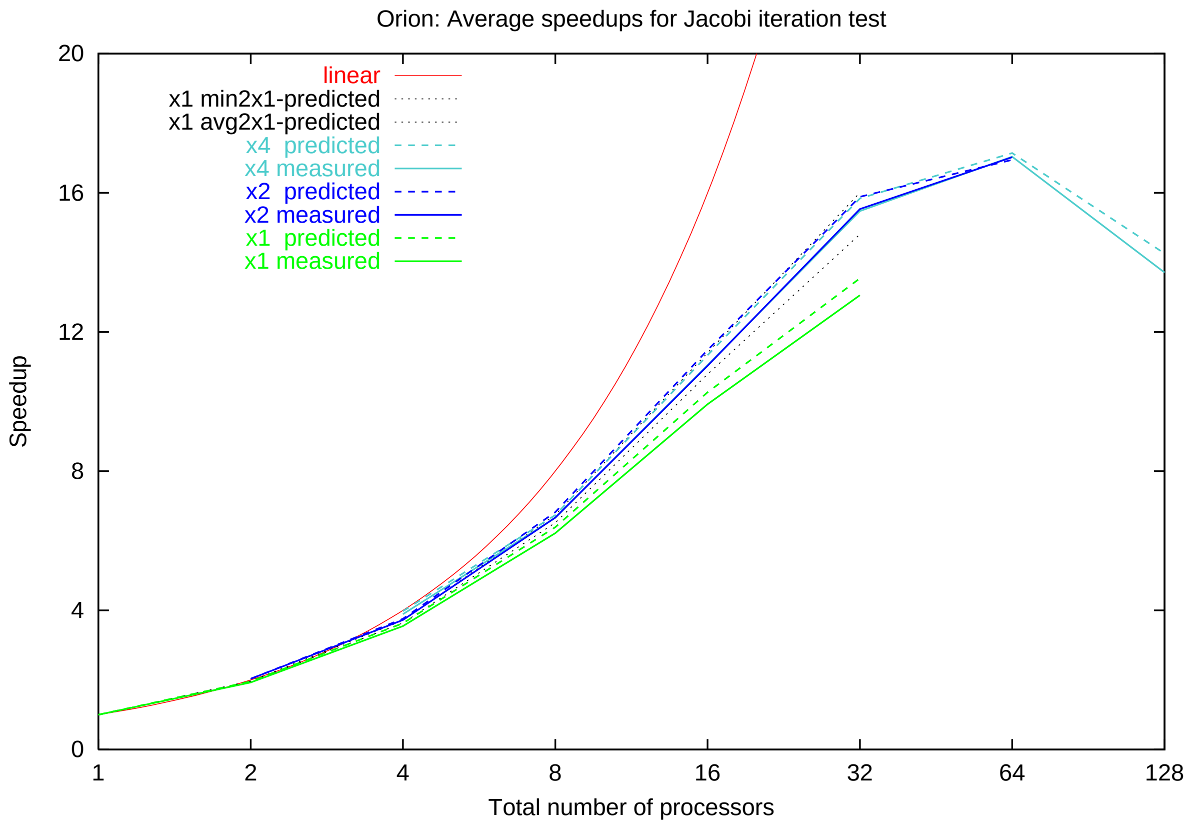


Figure 90: PEVPM-predicted average speedups and measured average speedups for the Jacobi Iteration test using 2-32x1-4 processes on Orion.

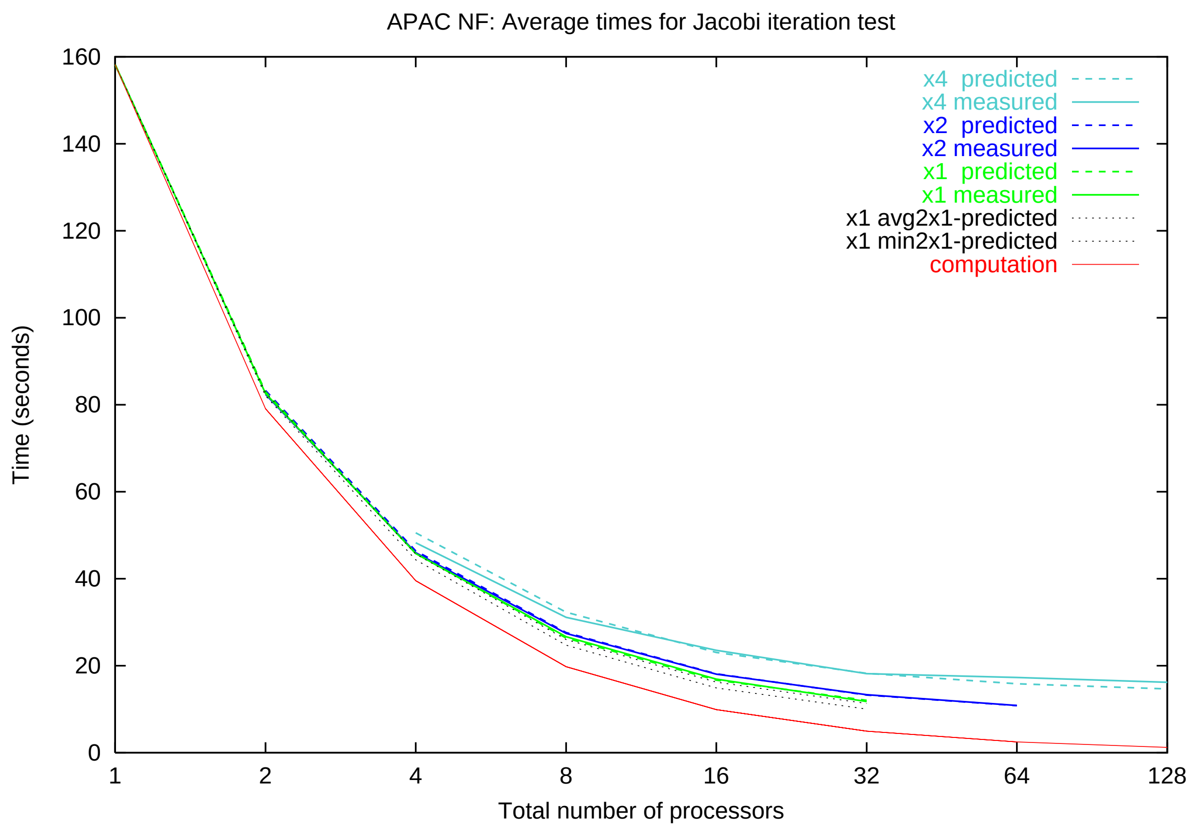


Figure 91: PEVPM-predicted average times and measured average times for the Jacobi Iteration test using 2-32x1-4 processes on the APAC NF.

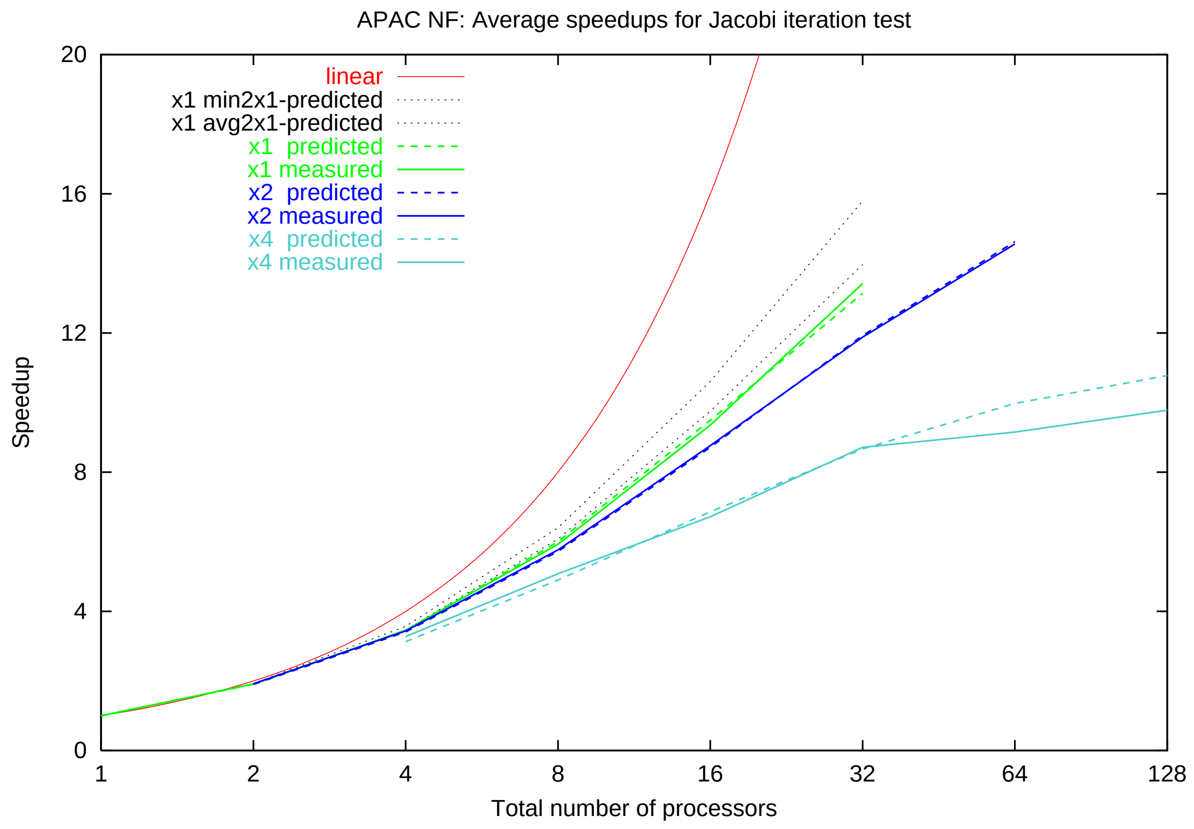


Figure 92: PEVPM-predicted average speedups and measured average speedups for the Jacobi Iteration test using 2-32x1-4 processes on the APAC NF.

results are plotted as solid lines in the same figures. Of those six figures, the first two relate to the Jacobi Iteration's performance on Perseus, the next two to the Jacobi Iteration's performance on Orion, and the final two to the Jacobi Iteration's performance on the APAC NF. The first figure in each of these three pairs of figures plots the completion time of the Jacobi Iteration against the number of processors used, while the second plots the speedup attained against the number of processors used. All of these curves were plotted according to conventional guidelines [84], and include the perfect (linear) speedups that would be achieved under ideal conditions.

The accuracy of the PEVPM predictions is readily apparent in each of Figures 87-92. In particular, there are two classes of performance predictions in each figure. Firstly, shown in green, blue or light blue dashes (corresponding to x1, x2 or x4 processor configurations) are the PEVPM predictions that were made using complete performance profiles of constituent message-passing operations (which were measured with MPIBench). These are clearly very accurate, always predicting completion time to within 5%[1] and usually to within 1%. Importantly, these predictions were consistently accurate, regardless of the number of processors used. This suggests that the small prediction errors that were observed were mainly due the granularity (i.e. histogram bin size) of the benchmark results that were input into the PEVPM simulation (which, if desired, could be made finer still at the price of greater solution time requirements) rather than any underlying deficiency in the PEVPM approach. These constant and high accuracy predictions across such a wide range of machine sizes are unprecedented; previously devised performance modelling techniques (see Chapter 2) rapidly lose the ability to make accurate predictions for parallel programs running on a large number of processors.

This tendency towards inaccurate predictions when a large number of processors are involved can be seen in the second class of more simplistic PEVPM predictions, shown by dotted black lines in Figures 87-92. These performance predictions were made by inputting minimum or average benchmark results into the PEVPM evaluations instead of complete performance distributions. In particular, these minimum and average times were garnered from MPIBench 2x1 process benchmarks, i.e. simple ping-pong benchmarks, like those produced by traditional MPI benchmarking tools. This was done on purpose to highlight the most inaccurate predictions that could result from using simple benchmark results. Using minimums and averages from MPIBench *n*x*p* process benchmarks, which better match the actual machine configurations that were used, produced results of intermediate quality (but these are not shown to avoid excessive clutter in the figures). Also note, however that traditional MPI benchmarking tools do not typically provide *n*x*p* process benchmarks; and having to use simple uniprocessor ping-pong benchmark results to model

[1]Except for two anomalous cases, found at 16x4 and 32x4 process configurations of the APAC NF. The discrepancies between the predicted and measured times in these two cases appear to be due to the performance saturation of some unknown (and hence unmodelled) subsystem of the APAC NF.

communication performance on clusters of SMPs would provide very poor results.

Even for this trivial regular-local code, the more simplistic prediction methods described in the last paragraph are prone to large errors, which tend to grow in proportion with the total number of processors utilised. In particular, these simplistic methods will always overestimate performance, because they do not account for the flow-on effects of contention in a parallel system. Hence, while they may possibly be useful for modelling the performance of parallel programs running on a small number of processors, they are inadequate for modelling large parallel programs, where contention effects become very important. For parallel programs running on a large number of processors, accurate performance models must take the complete performance distributions of constituent message-passing operations into account, instead of just their ideal or average performances. This is especially apparent on Perseus, where contention for network resources quickly limits performance. Although it is less evident on Orion and the APAC SC because of the scalability provided by their fat-tree networks, it is certainly noticeable.

Briefly returning to the discussion about the cost-effectiveness of the PEVPM approach, consider the actual execution time versus the simulation time for all of the Jacobi Iterations that were carried out on (for example) Perseus. The 11 hours and 15 minutes of processor time consumed by actually running the Jacobi Iterations on Perseus were simulated in just under 10 minutes by a prototype (i.e. unoptimised) PEVPM implementation running on just one processor of Perseus. This comparison shows that the PEVPM simulated the Jacobi Iterations on Perseus at about 67.5 times their actual execution speed. Similar "speedups" were achieved in the simulation of the Jacobi Iterations on Orion and the APAC NF – although strict comparisons are impossible to make, because those simulations were also run on Perseus rather than on Orion or the APAC NF themselves. Interestingly, the PEVPM algorithms themselves are close to embarrassingly parallel; assuming a parallelised PEVPM implementation exhibiting perfect speedup could be constructed, the 11 hours and 15 minutes worth of Jacobi Iterations on Perseus could be simulated in just under 5 seconds of compute time using all 128 processors of Perseus. While the speedup achieved by any given PEVPM simulation will depend heavily on model granularity, it seems reasonable to conclude that the great majority of PEVPM models would be relatively cheap to evaluate.

Finally, there are a number of general comments to be made about the performance characteristics shown in Figures 87-92. The most obvious observation is that the Jacobi Iteration code scales much better on Orion and the APAC NF than on Perseus, mainly because of the low latency and low performance variability of their communication networks. Also, judging by the performance of a Jacobi Iteration on one processor, the 1GHz Alpha EV68 processors in the APAC NF provide the greatest computational speed. The

500MHz Pentium III processors in Perseus are only half as fast, and the 450MHz UltraSparc II processors in Orion are only half as fast again. These mixtures of network performance and processor performance give rise to complex performance trends. For example, a 4 processor Beowulf-class machine like Perseus would outperform a 4 processor Sun TCF like Orion on a Jacobi Iteration of this size. However, an 8 processor Sun TCF would outperform an 8 processor Beowulf-class machine on the same problem. Beyond this, adding more processors to the Beowulf-class machine would achieve little extra performance, while adding processors to the Sun TCF (up to at least 32 in total) would continue to improve overall performance. Maximal performance on the Sun TCF would be achieved with 64 processors (either 32x2 or 16x4), but this would probably change if the problem size were altered. As another example, the absolute best performance for the Jacobi iteration was obtained using 32x2 processors on the APAC NF, although that performance was almost matched using only 32x1 processors on the same machine. Clearly, without a tool like PEVPM it would be very difficult to determine the size of a problem or parallel machine that should be used to make the most effective use of available resources – on the basis of either raw speed, efficiency, or economy, etc.

The number of processors per SMP node used in a machine obviously plays an important part in determining overall performance. On Perseus it does not seem to matter greatly whether 2nx1 (i.e. single processor) or nx2 (dual processor) configurations are used for this Jacobi Iteration implementation – performance scales equally well. On Orion, however, SMP nodes slightly outperform single processor nodes; on the APAC SC single processor nodes significantly outperform SMP nodes (given, in both cases, the same total number of processors). There has been much debate about the merits of multiprocessor versus uniprocessors nodes, especially in regards to their performance/price ratios. Capello *et al.* [58, 59, 60, 61], Gustafson [157] and Hsieh [182] found (empirically) that current (at the time of their publications) dual processor nodes outperformed single processor nodes on the NAS Parallel Benchmarks [28], in both raw performance and performance/price. They also found that while quad processor nodes outperformed dual processor nodes in terms of raw performance, they had less performance/price than even the single processor nodes. While these empirical discoveries were indeed useful, a first-principles approach to understanding the reasons for these outcomes is far more valuable. The exact performance implications of using single processor or SMP nodes depend on many factors, including the speed of the external an internal communication networks in a cluster of SMPs, the contention effects that are experienced when multiple processors within a node try and communicate with other processors (either local or remote), and the communication pattern of the program being executed. Only with tools like MPIBench and PEVPM is it possible to accurately and methodically study the effects of these factors.

6.3 Bag of Tasks

This section evaluates the capacity of the PEVPM to cope with parallel programs that exhibit irregular computation/communication patterns. This study could arguably have been left until last, because of the extreme demands that irregular codes place upon the generality of a performance modelling system. It is presented now, however, because the (irregular) Bag of Tasks code described here is less taxing on the network (during actual program execution) than the (regular-global) FFT code described in the following section. Hence, the accuracy of the PEVPM modelling system is tested under moderate network stress in this section and under extreme network stress in Section 6.4. In addition, this ordering of case studies more closely mirrors the ordering of the benchmarking tests described in the last two chapters. The communication pattern of the Jacobi Iteration examined in Section 6.2 was defined by point-to-point operations (c.f. the results in Section 4.5); the Bag of Tasks problem discussed here makes use of scatter/gather-like communication (c.f. the results in Section 5.4); and the FFT code described in Section 6.4 revolves around the `MPI_Alltoall` operation (c.f. the results in Section 5.5).

The computation/communication patterns and hence performance properties of irregular codes are interrelated in complex and often nondeterministic ways. This makes those properties very difficult to understand even in hindsight, let alone determine *a priori*. Although these difficulties are overcome by the PEVPM's ability to simulate the execution structure of any irregular code and hence predict its performance, a bothersome problem arises out of the need to feed input and/or condition data into that simulation. While the condition data will almost always be determinable by automatic means[2], the input data must always be explicitly specified. In pathological cases, input data may depend upon the execution history of the parallel program, which would make specifying the possible streams of input data exceptionally difficult. Fortunately, although such pathological programs may be contrived, there is almost no practical need for parallel codes to be developed in such a way. The Bag of Tasks code developed for study in this section, for example, utilises a pseudorandom input stream to guarantee fixed input data, but follows an irregular and nondeterministic execution path. Thus, the ability of the PEVPM to simulate nondeterministic and irregular codes with (statistically) random input data is demonstrated, yet experimental repeatability is ensured so that performance comparisons can be made between different runs (either actual or simulated) of that code.

The skeleton code listed in Figure 93 describes a program that will solve a collection of independent subproblems, such as may occur during transaction processing, when conducting parametric studies, or in a large number of other applications. The first page

[2]Note that the condition data are, theoretically, always determinable by automatic means (through simply executing the actual code). In some cases, however, this may not be computationally feasible and other solution techniques (such as calling upon programmer insight) will be required.

```
#define DEFN 3001 // MPI communicator tags
#define WORK 3002
#define DIE  3003
#define SIZE_MAX 65536 // per job in/out data size
#define COMPUTE_MAX 25000000 // per job clock cycles

typedef struct{
  long sendsize;
  long computeops;
  long recvsize;
} workdef;

workdef out, in;
int *sbuf, *rbuf;
int rank, procnum, numprocs;
int njobs = 10000
MPI_Status s;

// ranf_start and ranf_arr_next were snarfed from Donald Knuth's
// portable random number generator described in The Art of Computer
// Programming, Volume 2, 3rd Edition, Section 3.6, 9th printing (2002)
void ranf_start(long seed){
   ...
}
double ranf_arr_next(void){
   ...
}

workdef getwork(void){
  workdef this;
  if (njobs>0){
    this.sendsize   = (long)(ranf_arr_next()*(double)SIZE_MAX);
    this.computeops = (long)(ranf_arr_next()*(double)COMPUTE_MAX);
    this.recvsize   = (long)(ranf_arr_next()*(double)SIZE_MAX);
    njobs--;
  }
  else{
    this.sendsize = this.computeops = this.recvsize = -1;
  }
  return this;
}

MPI_Comm_rank(MPI_COMM_WORLD, &procnum);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
ranf_start(310952);
```

Figure 93 (part 1/2): Skeleton code for the Bag of Tasks.

```
// the master
if (procnum == 0) {

  for (rank = 1; rank < numprocs; rank++) {
    out = getwork();
    MPI_Send(&out, 3, MPI_LONG, rank, DEFN, ... );
    MPI_Send(&sbuf[0], out.sendsize, MPI_BYTE, rank, WORK, ... );
  }
  out = getwork();
  while (out.sendsize!=-1) {
    MPI_Recv(&in, 3, MPI_LONG, MPI_ANY_SOURCE, DEFN, ... , &s);
    MPI_Recv(&rbuf[0], in.recvsize, MPI_BYTE, s.MPI_SOURCE, WORK, ... , &s);
    MPI_Send(&out, 3, MPI_LONG, s.MPI_SOURCE, DEFN, ... );
    MPI_Send(&sbuf[0], out.sendsize, MPI_BYTE, s.MPI_SOURCE, WORK, ... );
    out = getwork();
  }
  for (rank = 1; rank < numprocs; rank++) {
    MPI_Recv(&in, 3, MPI_LONG, MPI_ANY_SOURCE, DEFN, ... , &s);
    MPI_Recv(&rbuf[0], in.recvsize, MPI_BYTE, s.MPI_SOURCE, WORK, ... , &s);
  }

  for (rank = 1; rank < numprocs; rank++) {
    MPI_Send(&out, 3, MPI_LONG, rank, DIE, ... );
  }

}

// the slaves
else {
  for (;;) {

    MPI_Recv(&in, 3, MPI_LONG, 0, MPI_ANY_TAG, ... , &s);
    if (s.MPI_TAG == DIE) {
      break;
    }
    MPI_Recv(&rbuf[0], in.sendsize, MPI_BYTE, 0, WORK, ... , &s);

    wait_cycles(in.computeops); // simulate actual computation

    MPI_Send(&in, 3, MPI_LONG, 0, DEFN, ... );
    MPI_Send(&sbuf[0], in.recvsize, MPI_BYTE, 0, WORK, ... );

  }
}
```

Figure 93 (part 2/2): Skeleton code for the Bag of Tasks.

of the skeleton code defines a number of data types, variables and auxiliary functions, most of which are geared towards describing and generating a pseudorandom sequence of tasks. Those tasks, or jobs, are defined by `workdef` structures, which specify (simulated versions of) each job's input data (`sendsize`), processing requirements (`computeops`) and output data (`recvsize`). (The `COMPUTE_MAX` and `SIZE_MAX` values offer rough control over the computation/communication ratio of the Bag of Tasks code; the values listed in Figure 93 were chosen because they defined a computation/communication ratio that led to interesting speedups – i.e. neither perfect nor completely saturated – on each of Perseus, Orion and the APAC NF). A new job can be created by calling the `getwork` function, which generates a `workdef` item with pseudorandom input data, processing and output data requirements. The second page of the skeleton code lists the main computation/communication part of the Bag of Tasks program, where a master process generates `workdef` items, farms them out to any available slave processes for solution, and collects the results. Initially, all of the slaves are idle, so the master generates jobs as fast as it can and sends them out to the slaves using a `MPI_Scatter`-like process. When this computational pipeline is filled, the master enters a new phase of operation, where it waits to receive the results of a completed job from any slave; upon doing so the master will immediately issue that slave a new task. When all of the (10,000) jobs have been dispatched, the master drains the remaining results from the computational pipeline and instructs the slaves to terminate.

A good feature of the task-farming approach to parallel computation is that it is inherently load-balanced. Because all of the slaves work independently, it does not matter if some tasks take longer than others to perform – when a slave completes a task, it simply asks the master for another, which it begins to process while the other slaves are still busy working on their own jobs. Hence, given a random Bag of Tasks to process, it is difficult to predict the order in which computation and communication will proceed, and thus the problem is irregular. If computation and communication times are exact (i.e. not stochastic), this will imply a unique program execution sequence and the Bag of Tasks code will execute deterministically. In this simple case, performance may be evaluated by simply calculating the critical path. A small number of the previous performance modelling techniques that were described in Chapter 2 are capable of estimating the performance of irregular programs using this deterministic assumption.

Unfortunately, due to the variations in processing and communication times that occur in reality (caused mainly by contention), the Bag of Tasks code will execute nondeterministically. Only a subset of the previous performance modelling techniques mentioned above account for contention, and even then only some types of contention. Surprisingly, none of those adequately account for the main type of contention that is encountered when processing the Bag of Tasks code: gather contention. This very important type of contention occurs when multiple slaves all try to communicate with the master simultaneously, thus flooding its communication link(s) and hence limiting program scalability. Although gather contention is conceptually quite simple, it can be difficult to model well in practice, which perhaps explains the absence of the machinery required to account for it in previous performance prediction tools. Unfortunately, neglecting to properly account for gather contention in is almost tantamount to ignoring precedence relationships and this will have dire consequences on the accuracy of performance predictions.

Accurate bottom-up performance prediction of the (irregular and nondeterministic) Bag of Tasks code can only be achieved by accounting for the effects of all types of contention (both direct and indirect) on a microscopic scale – which is exactly what the PEVPM was designed to do. Therefore, in the same way as described in the last section, the execution structure of the Bag of Tasks code listed in Figure 93 was simulated using the PEVPM for various configurations of Perseus, Orion and the APAC NF. These performance predictions, along with actual measured results are shown with dashed and solid lines in Figures 94-99. Once again, the accuracy of the PEVPM is clearly demonstrated, especially in its ability to predict when saturation will occur on each machine.

In contrast to these exemplary results are the results shown in black dots in the same figures, which were obtained by predicting the Bag of Tasks' performance with a restricted PEVPM evaluation engine that did not account for gather contention. While these restricted-PEVPM predictions closely match the normal PEVPM predictions for a small number of processors, they completely fail to foresee the performance saturation that ensues when larger numbers of processors are used. This is because, in the restricted-PEVPM model, there is no penalty associated with many processes trying to communicate simultaneously with the master. Given this unrealistic property, every process is able to communicate with the master just as fast as it would if it were the only slave. Hence, as the number of slaves is increased, not only do the computation costs scale by $1/n$ but the communication costs do also. Thus, constant efficiency is maintained (or it would be if the master did not count in the speedup calculations – because it does, however, efficiency starts at 0% for 1 processor and steadily rises until it reaches that defined by the $computation/(computation + communication)$ ratio). This leads to the close to perfect speedups shown in Figures 95, 97, and 99. In actual fact, the speedups are not as close to perfect as they seem, but merely look that way because of the different scales (i.e. linear versus logarithmic) along the different axes in each case. On Perseus, for example, the speedup predicted by the restricted-PEVPM had dropped to 102 using 128 processors; and efficiency had flattened out at almost exactly 80%. For Orion and the APAC NF the restricted-PEVPM predicted closer to linear speedups, however, because of the higher bandwidths of the networks in those machines.

Finally, it is worth reiterating that predicting a program's speedup when given a large

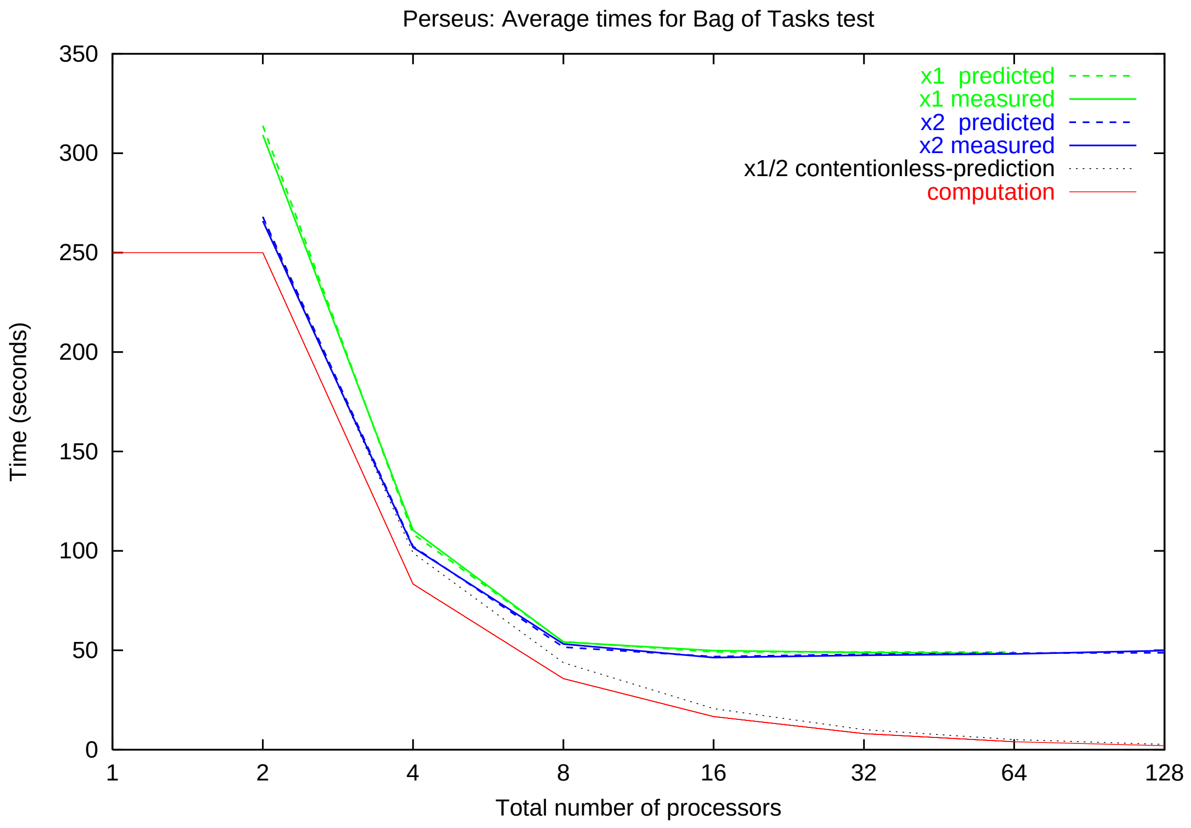


Figure 94: PEVPM-predicted average times and measured average times for the Bag of Tasks test test using 2-64x1-2 processes on Perseus.

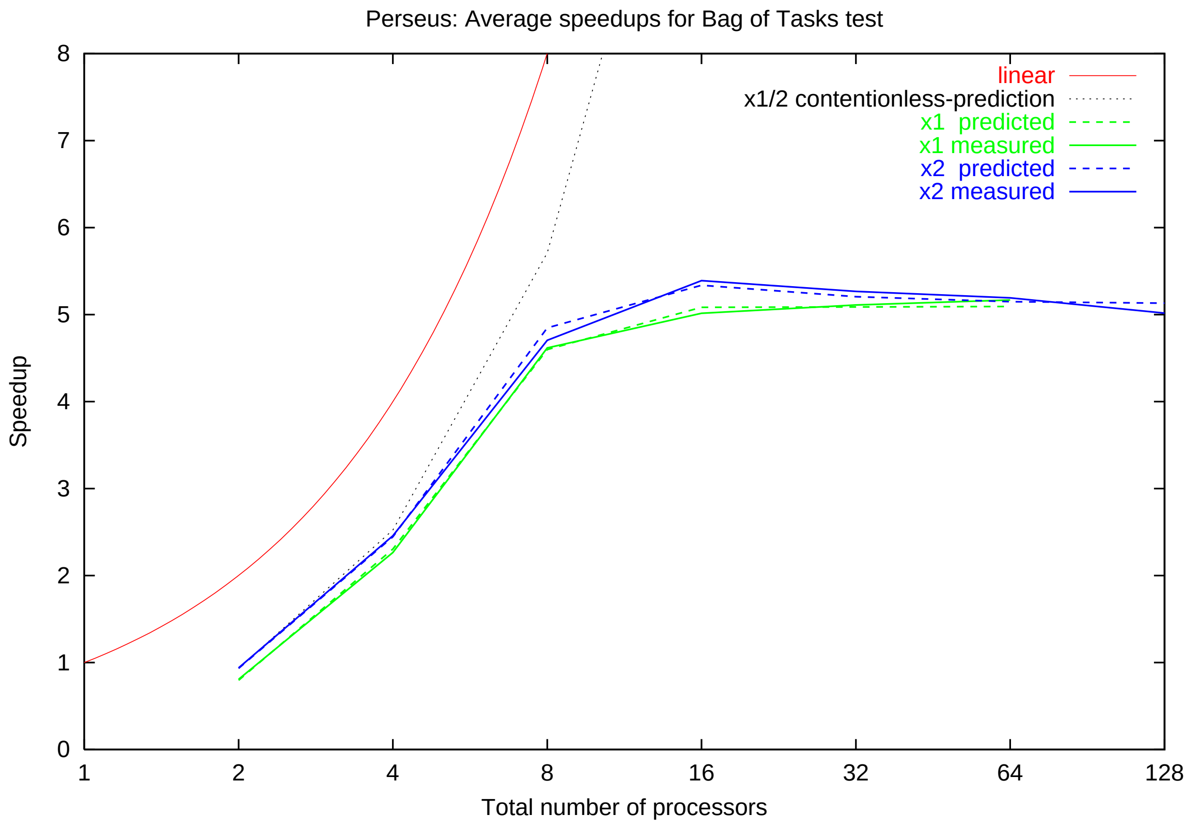


Figure 95: PEVPM-predicted average speedups and measured average speedups for the Bag of Tasks test test using 2-64x1-2 processes on Perseus.

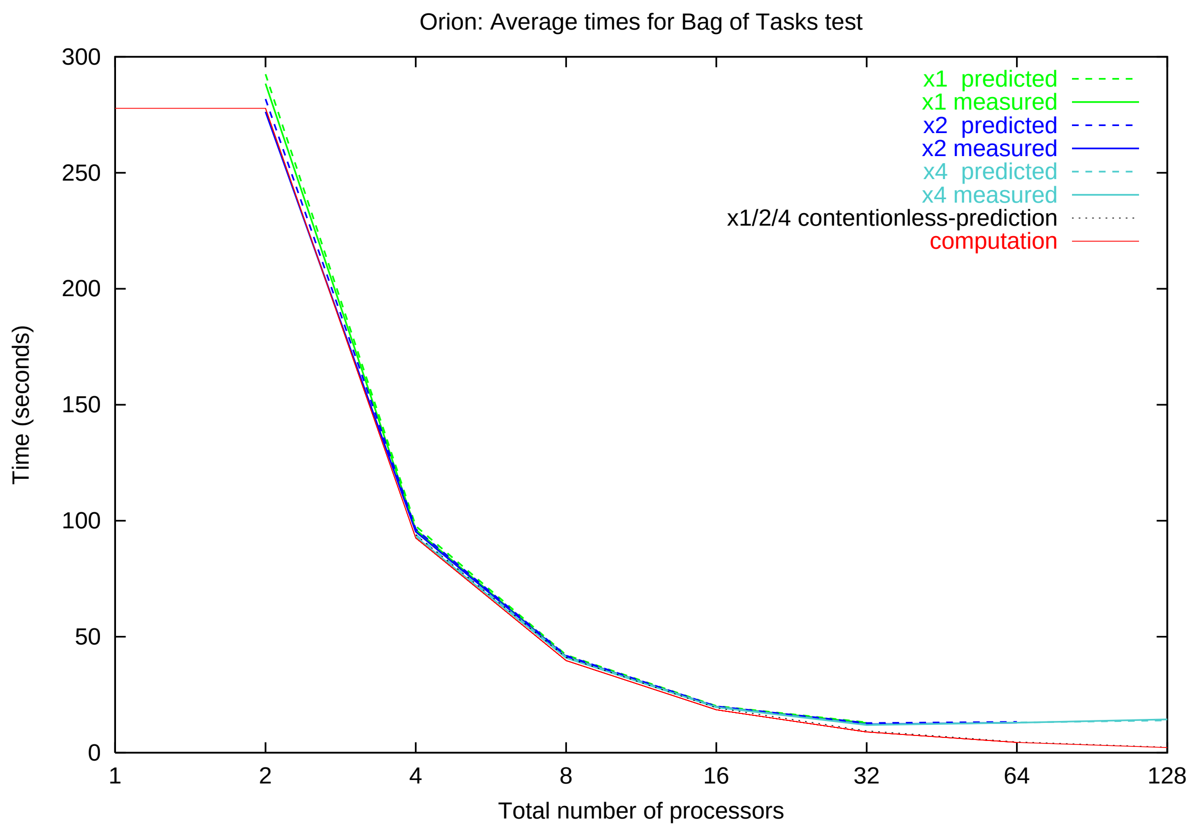


Figure 96: PEVPM-predicted average times and measured average times for the Bag of Tasks test test using 2-32x1-4 processes on Orion.

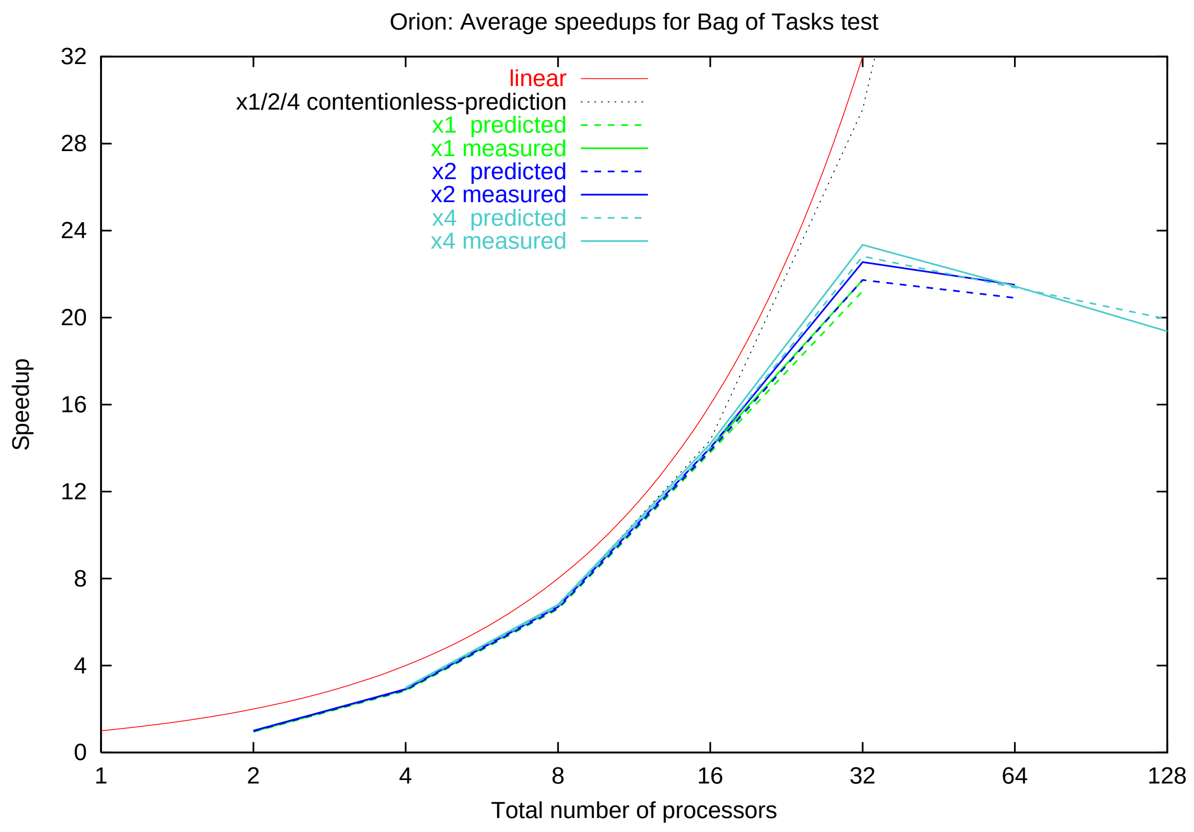


Figure 97: PEVPM-predicted average speedups and measured average speedups for the Bag of Tasks test test using 2-32x1-4 processes on Orion.

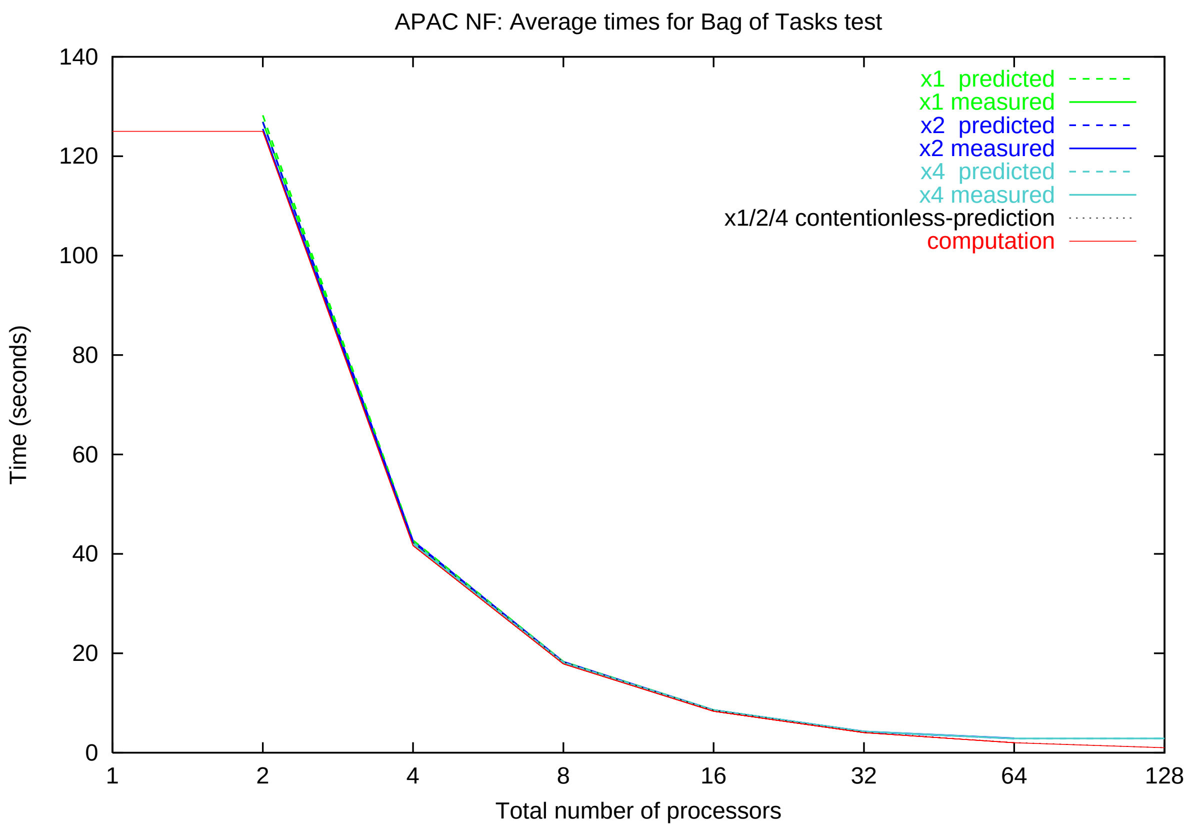


Figure 98: PEVPM-predicted average times and measured average times for the Bag of Tasks test test using 2-32x1-4 processes on the APAC NF.

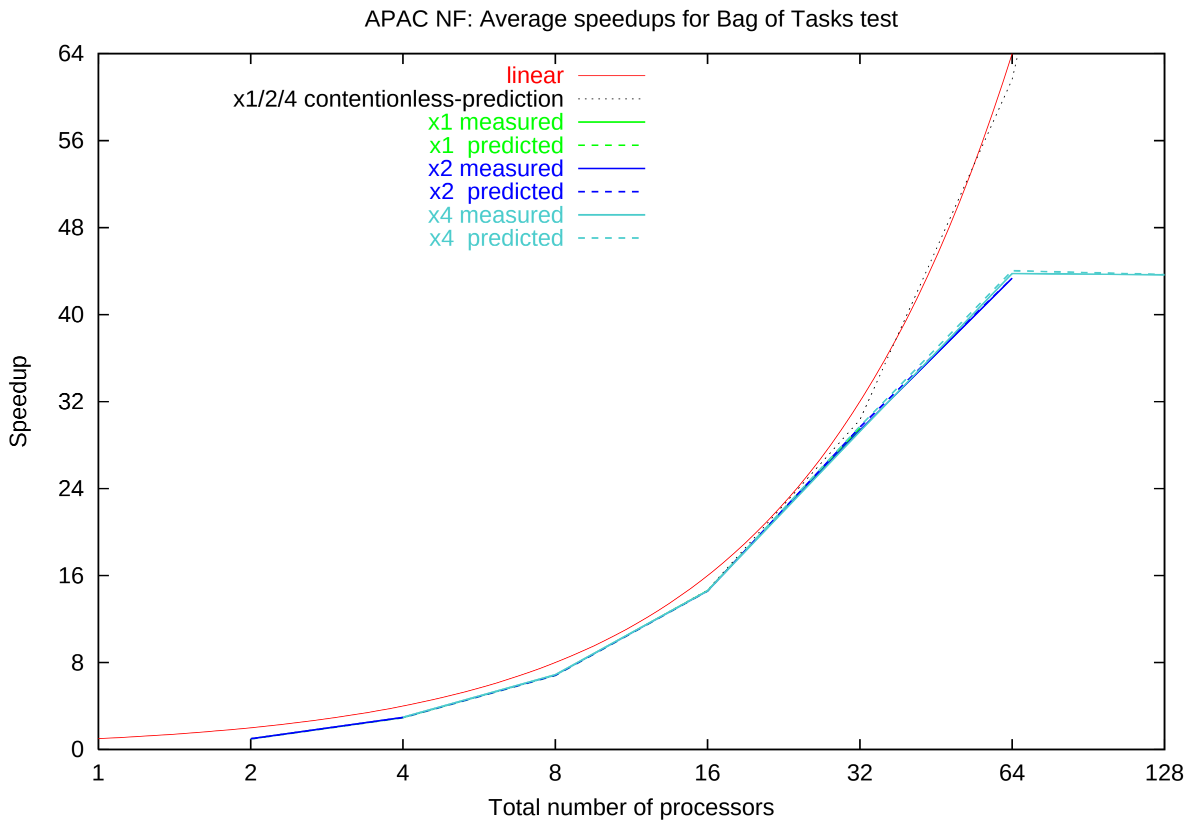


Figure 99: PEVPM-predicted average speedups and measured average speedups for the Bag of Tasks test test using 2-32x1-4 processes on the APAC NF.

numbers of processors is of crucial importance when sizing a parallel machine to a particular problem (or vice versa) in order to achieve maximum performance, speedup, efficiency, or cost-effectiveness. For example, the pseudorandom Bag of Tasks examined here could only be sped-up by at most 5.4 times on Perseus (using 8x2 processors), 23.3 times on Orion (using 8x4 processors) and 43.8 times on the APAC NF (using 64 processors or more in any configuration) – despite the 128 processors available in each case. The onset of performance saturation occurred with quite a small number of cooperating processors on Perseus, because of the relatively low bandwidth provided by its Fast Ethernet network; very little benefit could be gained by using more than 8 processors. In contrast, the superior bandwidth afforded by the Myrinet network in Orion allowed the problem to scale efficiency to about 32 processors. Even with the supreme (among the networks examined in this thesis) bandwidth provided by the QsNet network in the APAC NF, this Bag of Tasks could not be efficiently solved using all of the 128 available processors. Notably in all three cases, the performance improves almost perfectly with every added slave processor until saturation is reached. Thus, predicting when performance saturation will occur would be of crucial importance when determining the best (e.g. fastest or cheapest, etc) machine to use for solving this particular Bag of Tasks; such information can not be as accurately predicted using previous performance modelling techniques as it can with the PEVPM. Alternatively, the PEVPM simulation could be easily run with different Bags of Tasks, to determine which job mixes could be efficiently solved on any of the three machines examined here; or it could be used to predict this Bag of Tasks' performance on a hypothetical incarnation of Perseus, where the master's node was upgraded with a Gigabit Ethernet card (using, for example, extrapolated versions of the Fast Ethernet benchmark data for Perseus). Given the enormous cost of actually implementing any of these hypothetical situations and measuring program performance, the benefits of performance prediction are obvious provided that those predictions accurately reflect reality; hence the usefulness of the PEVPM modelling system.

6.4 Fast Fourier Transform

Fourier Transforms, which decompose time-varying signals into their frequency components, or spectrum, are at the heart of an astounding range of signal processing applications, from audio processing or image analysis to distinguishing natural seismic events from nuclear test explosions [46]. In practice, real-world (i.e. analogue) input signals are usually sampled so that they can be processed by digital computer hardware, using the closely related Discrete Fourier Transform. Without getting into algorithmic details (which are not important here), a (one-dimensional) Discrete Fourier Transform requires $O(n^2)$ operations to compute, where n is the total number of data samples. This can take quite a long time if there are a large number of samples. In 1965, Cooley and Tukey developed the Fast Fourier Transform [79, 285], which, given some restrictions, is able to compute the same result as an equivalent Discrete Fourier Transform in only $O(n\log_2 n)$ operations. This section examines the PEVPM's performance predictions for a parallel code that computes the Fast Fourier Transform of a two-dimensional grid of data.

The execution structure of the two-dimensional Fast Fourier Transform implementation that is used in this section is quite simple: it is comprised of an `MPI_Alltoall` operation sandwiched between two serial segments of computation. Because of this simplicity, and because of the highly structured and completely connected communication pattern of the `MPI_Alltoall` operation (see Section 5.5), two-dimensional Fast Fourier Transforms are often used as the canonical example of a regular-global code. While the performance properties of regular-global codes are slightly more complex than the regular-local codes examined in Section 6.2, they are significantly less complex than the irregular codes examined in Section 6.3. Unsurprisingly, therefore, most of the previous performance estimation techniques described in Chapter 2 are general enough to be sensibly applied to the modelling of regular-global parallel programs. However, because most regular-global codes are extremely demanding on communication networks, their performance characteristics are quite difficult to model accurately in practice. Unlike the PEVPM, none of the previous performance modelling techniques adequately account for the microscopic (but very significant when summed-up) precedence relationships (due to contention) which pervade regular-local codes; this reduces their predictive accuracy. Furthermore, the performance provided by many communication networks can degenerate almost completely under extreme load, with catastrophic consequences for overall program performance; simple network models are a completely inadequate basis for performance predictions in such cases. Later in this section, the accuracy of PEVPM predictions for a regular-global two-dimensional Fast Fourier Transform will be compared with performance predictions for the same code based on simpler network models.

The skeleton code listed in Figure 100 makes use of the typical parallelisation strategy for computing a multidimensional Fast Fourier Transform [69,134,156,351], which involves performing a succession of one-dimensional Fast Fourier Transforms on each line of data that is parallel to each axis in the grid of samples. For example, the two-dimensional Fast Fourier Transform of an m by n grid requires n one-dimensional transforms of length m along each row of the array followed by m one-dimensional transforms of length n down each column of the array. While these computations are quite easy to arrange for serial execution, they are much more difficult to organise for parallel execution, because the data grid must be redistributed amongst participating processes between each set of computations along any particular axis. In particular, this redistribution involves the complete transposition of the data grid. Because the data grid (`a`) is block-distributed

```
int i, j, k, cell, procnum, numprocs;
int iterations = 10000;
int size = 256;
int slice = size/numprocs;
complex a[size][size];
complex a_slice[slice][size];
complex a_chunks[numprocs][slice][slice];
complex b_slice[size][slice];

// a simple 1D FFT kernel snarfed from George Guscoria's tutorial at
// http://www.mhpcc.edu/training/workshop/mpi/samples/C/mpi_2dfft.c
void fft(complex *data, int size){
  ...
}

MPI_Comm_rank(MPI_COMM_WORLD, &procnum);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

for (i = 0; i < iterations; i++){

  // perform 1D row FFTs
  for (j = 0; j < slice; j++){
    fft(&a_slice[j][0], size);
  }

  // transpose 2D array
  for(cell = 0; cell < numprocs; cell++){
    for(j = 0; j < slice; j++){
      for(k = 0; k < slice; k++){
        a_chunks[cell][j][k].r = a_slice[j][k + (slice * cell)].r;
        a_chunks[cell][j][k].i = a_slice[j][k + (slice * cell)].i;
      }
    }
  }
  MPI_Alltoall(a_chunks, slice*slice, ... , b_slice, slice*slice, ... );
  for(j = 0; j < slice; j++){
    for(k = 0; k < size; k++){
      a_slice[j][k].r = b_slice[k][j].r;
      a_slice[j][k].i = b_slice[k][j].i;
    }
  }

  // perform 1D column FFTs
  for (j = 0; j < slice; j++){
    fft(&a_slice[j][0], size);
  }

}
```

Figure 100: Skeleton code for the 2D Fast Fourier Transform.

between available processors (into rows of `a_slice`), this is achieved by having each processor partially transpose its local slice of the data grid (into `a_chunks`), transposing those chunks between processors using the `MPI_Alltoall` operation and then having each processor complete the local transpositions (into columns of `a_slice`). Note that the skeleton code listed in Figure 100 does not include instructions to load a matrix of samples, perform the block decomposition of that grid (which would be achieved using an `MPI_Scatter` operation) or collate the results at one processor (which would be achieved using an `MPI_Gather` operation). In other words, this example code does not consider the I/O issues of supplying and retrieving data to and from processors; it implicitly assumes that the two-dimensional Fast Fourier Transform is only part of some longer pipeline of parallel computation. Also note that the algorithmic details of a one-dimensional Fast Fourier Transform are not listed in Figure 100. Those details are not important because, while complicated in its own right, the particular one-dimensional Fast Fourier Transform implementation [156] used for the case studies in this section is only a serial computation and is therefore modelled as a single serial segment by the PEVPM.

As was done for the Jacobi Iteration code in Section 6.2 and the Bag of Tasks code in section 6.3, the execution structure of the two-dimensional Fast Fourier Transform code listed in Figure 100 was simulated using the PEVPM for various configurations of Perseus, Orion and the APAC NF. These performance predictions, along with actual measured results are shown with dashed and solid lines in Figures 101-106.

In order to explain the trends in these results, it is first helpful to describe how the processing and communication requirements of the code listed in Figure 100 scale as the number of processors used is increased. In the first phase of the overall computation, each processor must carry out $slice = size/numprocs$ (row-based) one-dimensional Fast Fourier Transforms, each on $size$ samples. Because each one-dimensional Fast Fourier Transform is $O(n\log_2 n)$, each processor must carry out $O(\frac{size^2}{numprocs}\log_2 size)$ operations in this phase. Likewise, in the third phase of the overall computation each processor must carry out $O(\frac{size^2}{numprocs}\log_2 size)$ operations to compute the (column-based) one-dimensional Fast Fourier Transforms. For the 256x256 grid in this example, these phases together require $2 * 1048576/numprocs$ floating point computations per processor (where the leading 2 is present because each grid point is a complex number). For the `MPI_Alltoall` operation in the middle phase of the overall computation, each processor must send/receive messages of $O(size^2/numprocs^2)$ to/from each of the other $numprocs - 1$ processors. Thus, for a fixed $size$ problem, the total amount of data (in bytes) that each processor must send and receive (almost) halves as the number of processors is doubled (i.e. scales according to $1/numprocs$, like the computation requirements for phases 1 and 3), while the number of individual messages required to do so (almost) doubles (i.e. scales according to $numprocs$). In comparison, the 256x256 point Jacobi Iteration described in

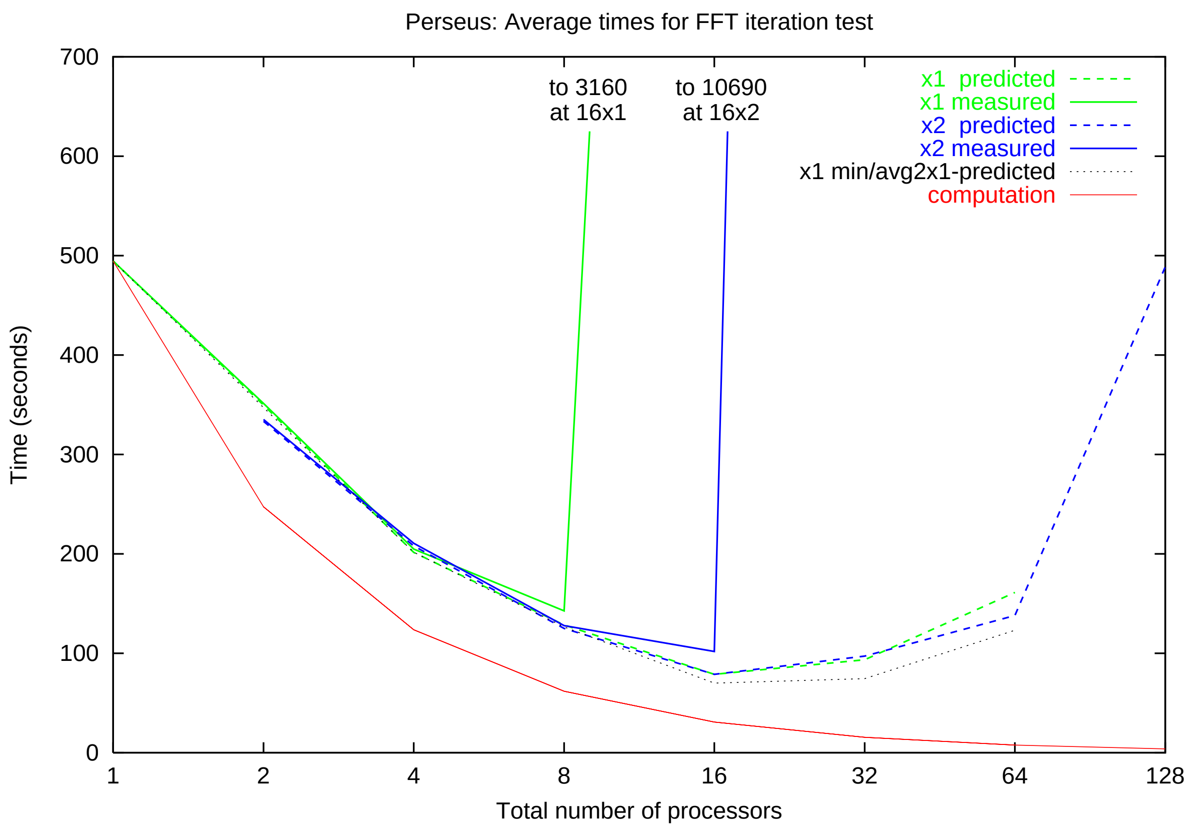


Figure 101: PEVPM-predicted average times and measured average times for 10,000 2D FFT operations on 256x256 points using 2-64x1-2 processes on Perseus.

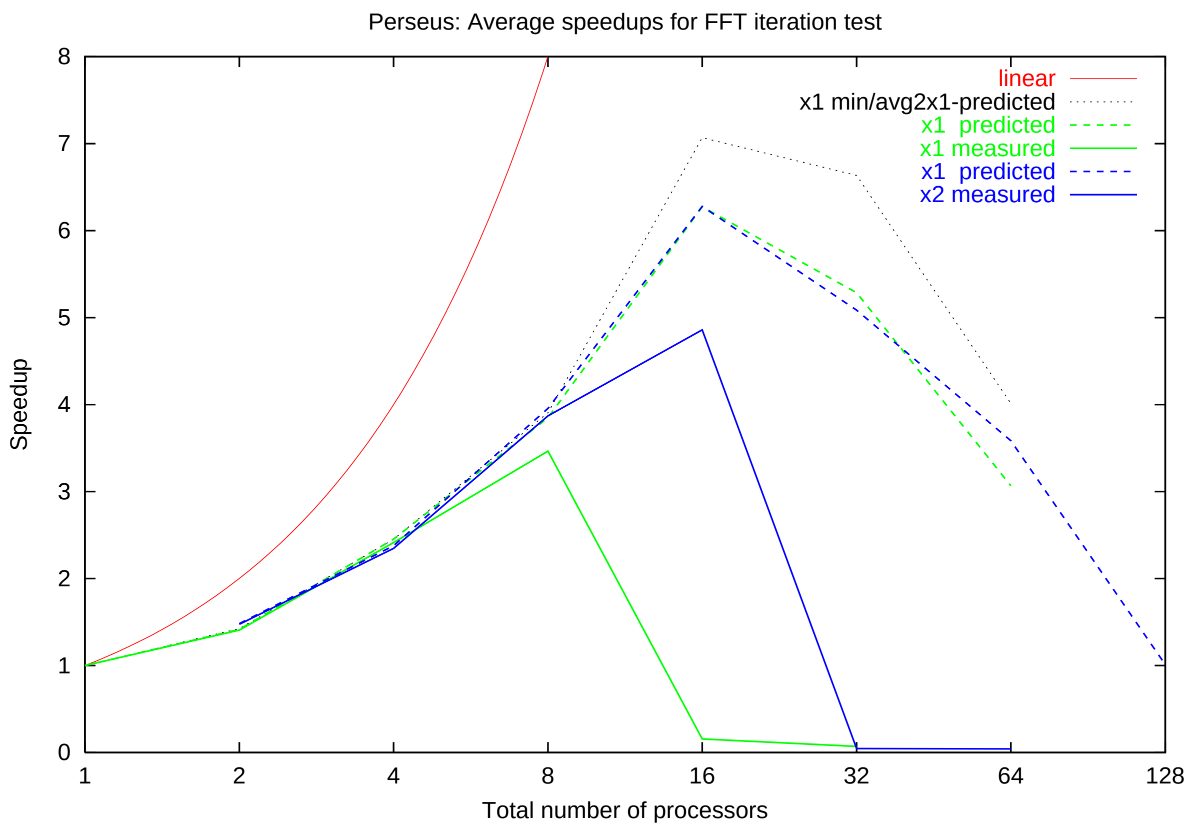


Figure 102: PEVPM-predicted average speedups and measured average speedups for 10,000 2D FFT operations on 256x256 points using 2-64x1-2 processes on Perseus.

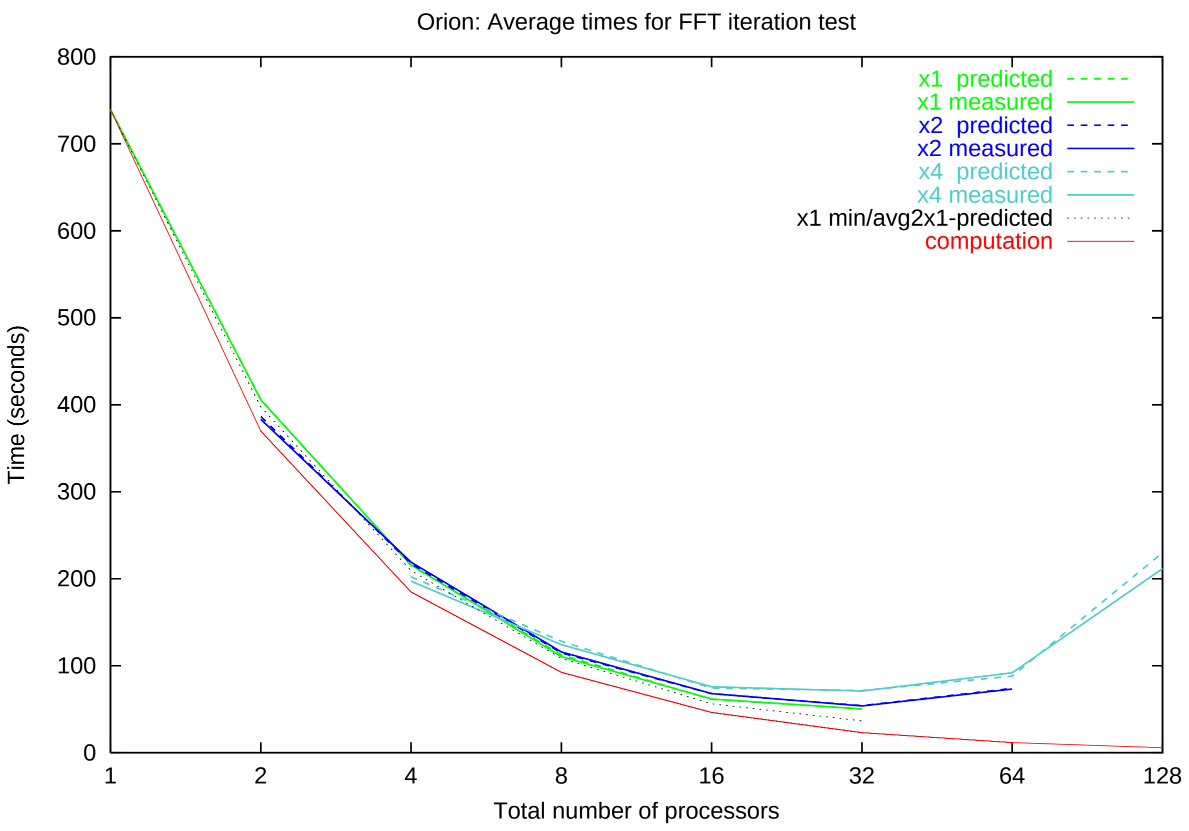


Figure 103: PEVPM-predicted average times and measured average times for 10,000 2D FFT operations on 256x256 points using 2-32x1-4 processes on Orion.

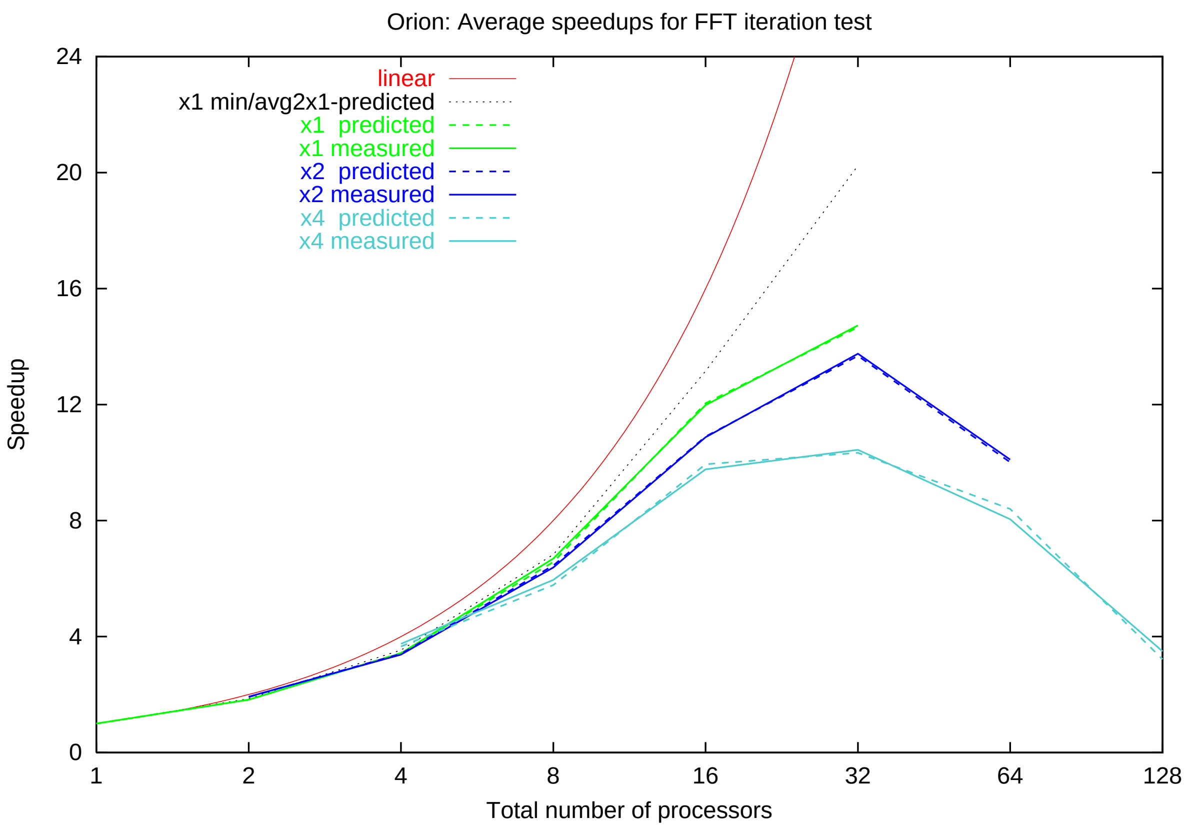


Figure 104: PEVPM-predicted average speedups and measured average speedups for 10,000 2D FFT operations on 256x256 points using 2-32x1-4 processes on Orion.

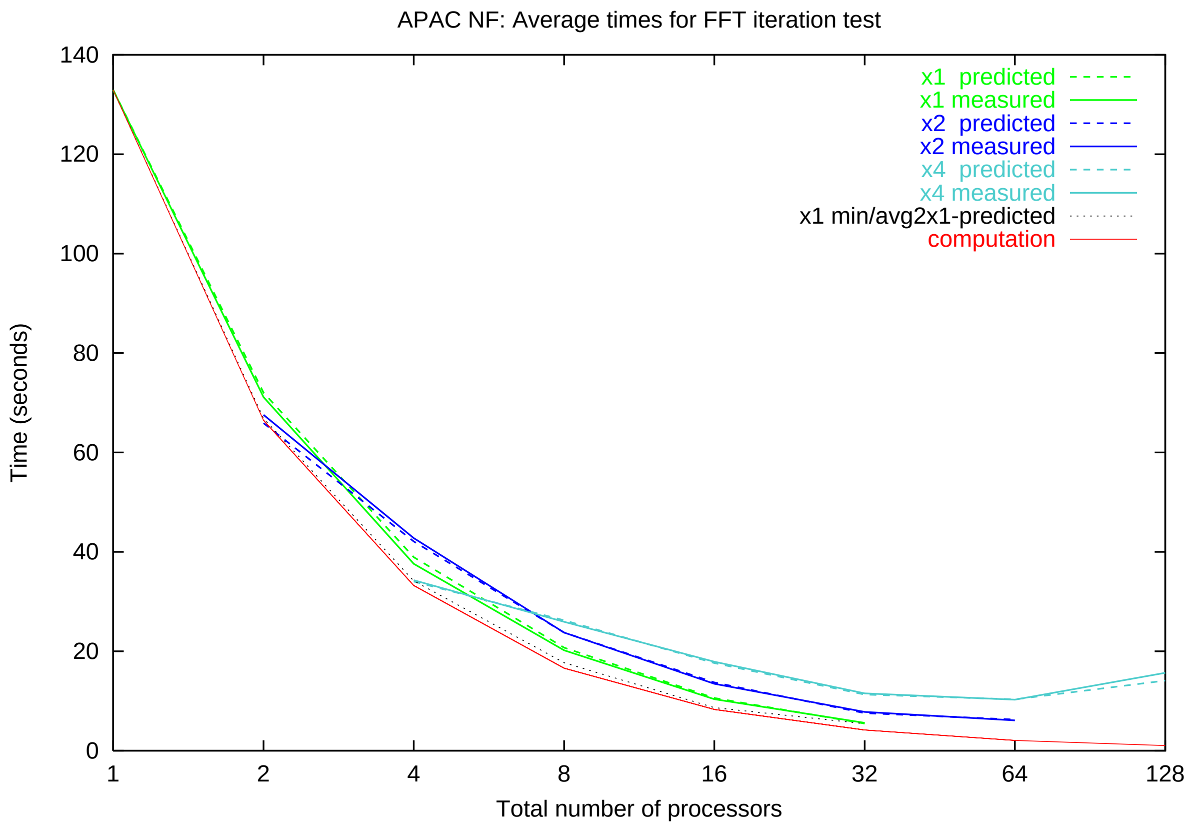


Figure 105: PEVPM-predicted average times and measured average times for 10,000 2D FFT operations on 256x256 points using 2-32x1-4 processes on the APAC NF.

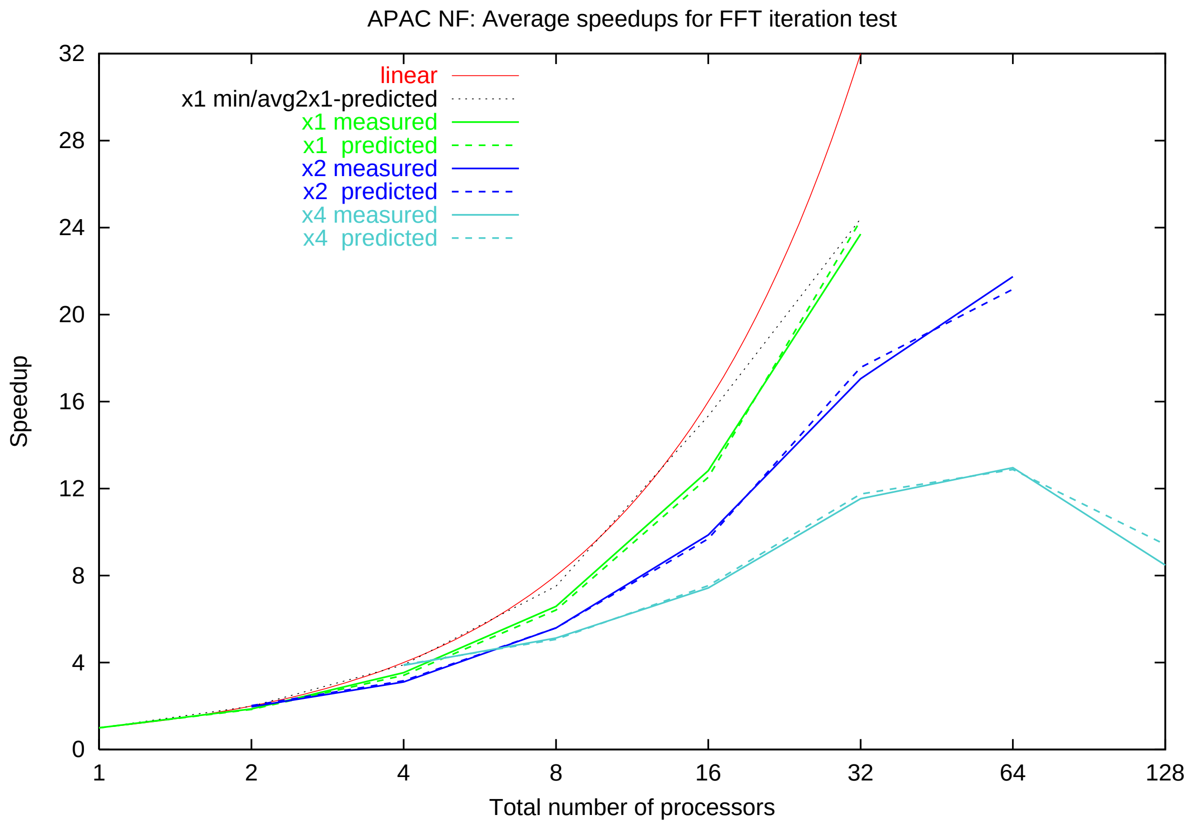


Figure 106: PEVPM-predicted average speedups and measured average speedups for 10,000 2D FFT operations on 256x256 points using 2-32x1-4 processes on the APAC NF.

Section 6.2 only required $O(4\frac{size^2}{numprocs}) \Rightarrow 262144/numprocs$ floating point computations per processor, and for each processor to transmit 2 kbytes of (bidirectional) data (in each iteration); note that the computation requirements for the Jacobi Iteration scaled in proportion to $1/numprocs$ while the communication requirements remained constant as the number of processors used was increased.

For both the two-dimensional Fast Fourier Transform and the Jacobi Iteration, the computation time is (to a very good approximation) directly proportional to the amount of computation that must be carried out, whereas the communication time is only linearly related to the amount of communication (in bytes) that must be transferred – there is a constant "y-intercept" that signifies minimum latency. Hence, because of the big-oh functions of the `MPI_Alltoall` operation, the two-dimensional Fast Fourier Transform exhibits an accentuated performance trade-off, which is based heavily on the latency and bandwidth characteristics of the underlying communication network and the number of processors involved; it is accentuated (compared to the Jacobi Iteration) because the effects on performance of communication latency become dominant very quickly as *numprocs* is increased (as explained in Section 5.5). Thus, the performance scalability of the two-dimensional Fast Fourier Transform listed in Figure 100 depends far more on low latency communication between processes than on high bandwidth. The accentuated scalability characteristics of the two-dimensional Fast Fourier Transform are manifest in the peakedness of the speedup plots for Perseus, Orion and the APAC NF shown in Figures 102, 104 and 106 (compared with the speedup plots for the Jacobi Iteration on the same machines shown in Figures 88, 90 and 92). It is also worth noting, however, that the detrimental effects of communication latency on `MPI_Alltoall` performance can be ameliorated by combining the messages that each process must send, and transmitting them to their destinations through intermediate processes using a tree-like algorithm (see Section 5.5). This strategy is only used by the MPI implementation on the APAC NF, not those on Perseus or Orion, which explains why the two-dimensional Fast Fourier Transform code scaled particularly well (especially for a moderate-large number of processors) on that machine.

As explained in Section 5.5, the `MPI_Alltoall` operation is the most demanding of all MPI routines. The extreme stress that it placed on Perseus' communication network, for example, produced a large number of very expensive communication protocol timeouts, hence leading to extremely poor performance (see Figures 77-80). This poor `MPI_Alltoall` performance is reflected in the poor performance of the two-dimensional Fast Fourier Transform on Perseus, shown in Figures 101 and 102. The measured performances (shown by the solid lines) in those figures matched the PEVPM-predicted performances (shown by dashed lines) very well up to 4xp processes, quite well for 8xp processes, and then very poorly for larger numbers of processes. The accuracy of these predictions correlates closely with the percentage chance of a process involved in the `MPI_Alltoall` operation being delayed by a retransmit timeout. In this case, it is clear that the performance model for message-passing time under normal circumstances (i.e. in the absence of message loss) is totally inadequate for predicting overall application performance. A possible counter-argument to the importance of this failing is that the `MPI_Alltoall` operation on Perseus should not behave like this (which it does because it was designed very badly – see Section 5.5), but the fact is that such bizarre performance problems are (unfortunately) not unheard of in practice, and being able to model them accurately is sometimes very important. At this point it must be made clear that the current implementations of MPIBench and the PEVPM do not completely support the proper modelling of such unusually long delays in message-passing time. The reason for this is that retransmit timeout delays do not usually occur with independent and identical distributions for all participating processors. For the way that the `MPI_Alltoall` operation is implemented on Perseus, for example, retransmit timeouts are more likely to occur at high-numbered processes. In that situation, it would be (strictly speaking) incorrect of the PEVPM to (Monte Carlo) sample from the performance distribution of the `MPI_Alltoall` operation recorded by MPIBench – although doing so would probably produce better predictions than ignoring the problem of outliers completely. In any case, the current PEVPM does not implement this heuristic, although it would be a useful feature to provide in the future, and there are no theoretical barriers to doing so.

Despite this small (implementational rather than inherent) inadequacy, the PEVPM still provides much better performance estimates than the simpler contention-ignoring predictions provided by previous performance modelling techniques. Like the dotted black lines in the figures from the last two sets of case studies, the dotted black lines in Figures 101-106 show the performance predictions the PEVPM made when supplied with the minimum and average message-passing times measured between 2x1 communicating processes rather than with complete performance distributions measured between *nxp* communicating processes. For Perseus, which has quite a high message-passing latency, the difference between the normal PEVPM predictions (shown by the dashed lines) and the restricted-PEVPM predictions (shown by the dotted lines) are certainly noticeable, but not enormous. For Orion and the APAC NF, however, the differences are much more significant. This is because the cumulative effects of the performance variation in the point-to-point messages that constitute the `MPI_Alltoall` operation (due to contention) become (comparatively) more significant in relation to the normal message-passing performance of the (nominally low latency) Myrinet and QsNet networks. On the APAC NF in particular, it is clear that neglecting to account for message-passing contention will lead to a considerable overestimation of the speedup that will be achieved by the two-dimensional Fast Fourier Transform code, at least up to 32x1 processors (see Figure 106).

For the 32x1 case, however, the restricted-PEVPM predictions converge on the normal PEVPM predictions. This is because the MPI implementation on the APAC NF uses the tree-based `MPI_Alltoall` algorithm rather than the circular neighbour approach (see Section 5.5) when this many processes are involved; and the larger (combined) message sizes used by the tree-based approach are (relatively) less susceptible to performance variation due to contention.

There are two final points to be made about the performance of the two-dimensional Fast Fourier Transforms studied in this section. Firstly, on Orion and the APAC NF, the performance degrades from single processor runs to dual processor or quad processor runs, (for the same total number of processors). This is caused in part by memory contention during the one-dimensional Fast Fourier Transforms and local transpositions, but mostly by memory and link contention during the `MPI_Alltoall` phase. The degradation of performance from single processor to dual processor runs does not occur on Perseus (and in fact, performance improves) because of interference from retransmit timeout delays. If this were not the case, however, the same degradation could be expected on Perseus, but to a lesser extent because the poor efficiency of Perseus' Fast Ethernet would provide some headroom for the dual processors to participate in `MPI_Alltoall` operations before running into significant link contention. Thus, in general (but not withstanding oddities like the outlier interference on Perseus), single processor nodes will usually perform two-dimensional Fast Fourier Transforms better than SMP nodes – at least, provided that each SMP processor does not have its own memory bus and while `MPI_Alltoall` operations remain unoptimised for SMP nodes (see Section 5.5). Secondly, it is worth calling attention to the far superior single processor performance of the APAC NF on the two-dimensional Fast Fourier Transform, compared with Perseus or Orion: one processor of the APAC NF can solve the two-dimensional Fast Fourier Transform code listed in Figure 100 in just 133 seconds, compared to the 494 seconds required by Perseus or the 739 seconds required by Orion. This demonstrates the supremacy of the Alpha EV68 processors, Compaq ES45's memory subsystem and compiler combination on the APAC NF for this application. Coupled with the fantastic performance of QsNet and the optimised performance of Compaq MPI's `MPI_Alltoall` operation, the APAC NF is clearly the best choice (of the parallel machines examined here) for performing *fast* Fast Fourier Transforms. As was the case with the Jacobi Iteration or the Bag of Tasks codes, however, other (e.g. economic) considerations may favour using Perseus or Orion; and the PEVPM approach provides the means with which to accurately weigh the merits of any particular solution.

6.5 Summary

This chapter has presented compelling evidence in support of the usefulness of the PEVPM approach to the prediction of parallel program performance. Unlike previous performance prediction techniques, the PEVPM approach has been shown to be completely general, arbitrarily flexible, very cost-effective and extremely accurate.

The generality of the PEVPM modelling system was demonstrated by its applicability to programs drawn from each of the three possible classes of parallel code – those with regular-local communication (via the Jacobi Iteration examples in Section 6.2), those with irregular communication (via the Bag of Tasks code in Section 6.3) and those with regular-global communication (via the two-dimensional Fast Fourier Transform code in Section 6.4) – all of which were executed on a wide range of parallel machines, in particular the low-end Perseus cluster, middle-of-the-range Orion TCF and high-end APAC NF.

The use of symbolic quantities in PEVPM directives (demonstrated for the Jacobi Iteration example in Section 6.2) highlighted the flexibility of the PEVPM approach. Once the complete PEVPM model of a code has been built up from fundamental computation and communication instructions, it can be easily evaluated and re-evaluated for different input data or machine characteristics. This gives the PEVPM the ability to support parametric performance studies, for instance to determine the best (i.e. fastest, or most efficient, or most economical, etc) parallel machine with which to solve a given code.

The PEVPM modelling technique is also very cost-effective. In addition to allowing parametric performance studies of any particular code, the low-cost of model creation makes it possible to build and simulate the performance of many alternative algorithmic solutions to any given problem. As shown for the Jacobi Iteration code in Section 6.2, PEVPM models are easy to create (for either real or hypothetical codes), through simply applying the rules for PEVPM directives detailed in Sections 3.4.4 and 3.4.5. Importantly, this process could be carried out by an automated compiler with little or no human intervention. While an automated compiler was not actually developed during this thesis (because doing so would require substantial software engineering effort and provide little research value), creating such a compiler would be a useful future endeavour. The second facet of the PEVPM's cost-effectiveness is the relative cheapness with which PEVPM models can be solved. The PEVPM simulation of the Jacobi Iteration code, for example, was carried out at 67.5 times the speed of the code's actual execution. While the speedup that will be achieved using any given PEVPM simulation will depend heavily on model granularity, most PEVPM models should be quite inexpensive to evaluate.

Finally, the results presented in this chapter have clearly demonstrated the superior accuracy of PEVPM predictions compared to the predictions made by previous performance estimation techniques. In particular, all of the case studies presented highlight how inaccurate performance predictions will be if contention effects are ignored. This was especially clear in the performance predictions for parallel codes running on clusters of SMP nodes, where contention was seen to be far more extensive. Finally, it was also shown that overall parallel application performance suffers horribly in the face of extensive communication protocol timeouts (such as occurred during the `MPI_Alltoall` operation on Perseus). No previous performance modelling systems take such communication delays into account. The current PEVPM implementation does not take these delays into account either, but the discussion in Section 4.8 paves the way for that to be implemented in the future.

Chapter 7

Conclusions and Further Work

High performance parallel computing is essential for solving very large and complex scientific and engineering problems in a reasonable amount of time. The two main tasks that must be carried out in order to deliver a good parallel computing solution to a given problem are choosing an appropriate parallel machine and writing a well-optimised parallel program. These tasks are often carried out in concert, by cycling through manifold candidate solutions, all the while conducting time-consuming empirical benchmarking until a satisfactory solution is found. An alternative approach is to use performance modelling techniques to more quickly and effectively choose from among a number of possible implementations. Unfortunately there is a scarcity of useful performance modelling methods, mostly due to the notoriously complex behaviour of parallel programs. The main contributor to this complexity is contention, which causes non-deterministic delays and therefore non-deterministic program execution. No previous performance modelling techniques for parallel programs (which were surveyed in Chapter 2) are able to adequately take these effects into account.

Given the limited capabilities of these previous techniques, a general, flexible, cost-effective and accurate performance modelling system for parallel programs called the Performance Evaluating Virtual Parallel Machine (PEVPM) was developed in Chapter 3. The PEVPM employs novel techniques which accurately yet inexpensively account for all of the performance effects of contention and thus performance variability that are observed in parallel programs. The first step of the PEVPM modelling process (described in Section 3.4) is to annotate existing source code or write pseudo-code with a performance directive language which defines a program's computation and communication structure. Once this is done those performance directives can be executed by the PEVPM, which simulates the program's execution structure, and thereby predicts its performance. The PEVPM uses Monte Carlo sampling techniques to dynamically construct submodels of individual computation and communication events. These submodels are based on: 1) data-dependencies; 2) current contention levels; and 3) probability distributions that describe the performance of a machine's low-level computation and communication operations. Because a PEVPM simulation evolves in virtual time, it automatically accounts for the effects of overlapping communication with computation, load imbalance and insufficient parallelism. It also explicitly models communication losses, synchronisation losses and the associated resource contention issues of each of these. Thus the PEVPM methodology accounts for all the sources of both performance and performance loss in message-passing parallel programs. As all of these events can be annotated, the PEVPM is able to automatically determine and highlight the location and extent of performance loss due to any source, which is of crucial importance for designing well-optimised parallel programs.

The work detailed at the beginning of Chapter 4 was carried out to provide for the PEVPM's need for very accurate performance characterisations of message-passing operations. Because of the inability of existing MPI benchmarking techniques to provide such data, a new benchmarking tool called MPIBench was developed (see Section 4.3). MPIBench provides the standard functionality available with existing MPI benchmarks (i.e. the ability to test the performance of many operations using different message sizes and in some cases using different communication patterns), but MPIBench is superior in three ways. Firstly, it was specifically designed to produce meaningful results when run on clusters of SMP nodes. Secondly, it uses an accurate global clock to make timing measurements at all of the processes in an MPI program, rather than simply making measurements at a single process. Thirdly, the fine resolution of the global clock in MPIBench allows timing data on individual MPI operations to be obtained. This gives MPIBench the unique ability to accurately quantify the performance variability of MPI operations due to contention, using probability distributions instead of average times.

Once developed, MPIBench was used to benchmark the communication performance of a low-end, a middle-of-the-range and a high-end parallel computer. The performance results obtained for point-to-point communication on each machine were detailed in Sections 4.5 and 4.6, while the results for collective communication were recorded in Sections 5.2 – 5.5. For point-to-point message-passing operations, performance variability due to contention was found to be very significant. This was especially true when large numbers of processes were trying to communicate simultaneously, particularly if large messages and/or bidirectional communication routines were being used. Probability distributions describing the performance variation of point-to-point message-passing operations were studied in Section 4.7, which concluded that the Pearson 5 distribution provides the best statistical explanation of point-to-point message-passing performance. While plots of the Pearson 5 location parameter were found to closely resemble existing (minimum) latency graphs, the accompanying shape and scale parameters presented important yet easily digestible information about message-passing performance in the face of contention. Furthermore, a study of the systematic outliers that were observed in performance measurements was carried out in Section 4.8. The main cause of these very slow message-passing times was identified as timeouts associated with the retransmission of lost messages. Such delays severely hamper the speedup that can be achieved by parallel programs running on a moderate to large number of processors.

As for collective communication operations, the results in Chapter 5 show how they can be modelled based upon their constituent point-to-point messages. Significantly, vast but repeatable differences were observed in the times that individual processes took to complete their part in collective operations. For example, software-based `MPI_Bcast`s were found to have Pascal's Triangle-shaped performance profiles, where the completion time of a given process is based on its position in the broadcast tree. While this sort of behaviour has been alluded to in other studies, none have been able to quantify it until now. Very complicated and even chaotic performance profiles were observed for other collective communication operations. Such strange results would have been almost impossible to understand using only the standard MPI performance benchmarking technique of measuring average completion time for collective operations at one process. However, the accurate performance profiles provided by MPIBench made their interpretation possible. For example, using MPIBench it could be seen that severe contention caused packet loss and hence network timeouts during `MPI_Alltoall` operations on a low-end Beowulf cluster. Quite clearly, the power of MPIBench opens up lots of possibilities for analysing the communication performance of various parallel machines, communication protocols and MPI implementations.

The theory behind the PEVPM modelling system (from Chapter 3) and the benchmark results (from Chapters 4 and 5) were applied in a series of case studies (see Chapter 6). These studies presented compelling evidence in support of the generality, flexibility, cost-effectiveness and accuracy of the PEVPM approach to parallel program performance prediction. The generality of the PEVPM modelling system was demonstrated by its applicability to programs drawn from all possible types of parallel code, running on a wide range of parallel machines. The use of symbolic quantities in PEVPM directives highlighted the flexibility of the approach: once the complete PEVPM model of a code has been built up from fundamental computation and communication instructions, it can be easily evaluated and re-evaluated for different input data or machine characteristics. Hence the PEVPM can perform parametric performance studies, for instance to determine the best (i.e. fastest, or most efficient, or most economical, etc) parallel machine on which to run a given code. Overall, the PEVPM modelling technique was also shown to be very cost-effective. The simplicity of PEVPM model creation, which only requires the mechanical insertion of performance annotations, makes it very cheap. In addition, these models can be solved quite inexpensively. While the speedup that will be achieved using any given

PEVPM simulation will depend heavily on model granularity, most PEVPM models could be expected to run about two orders of magnitude faster than a program's actual execution. Finally, the superior accuracy of PEVPM predictions was clearly demonstrated. In particular, all of the case studies showed how inaccurate performance predictions will be if contention effects are ignored. Because the PEVPM does not ignore contention effects, it should be able to provide accurate performance predictions of scalability for parallel programs running on very large numbers (e.g. thousands) of processors. The PEVPM is the first performance modelling system for parallel programs to concurrently provide generality, flexibility, cost-effectiveness and accuracy. Thus it will be useful for anyone who wants to break free of the empirical measure-modify cycle of parallel program development.

A number of enhancements that could be made to the current implementations of MPIBench and the PEVPM. These include the following, in ascending order of difficulty:

1. The current MPIBench/PEVPM implementations do not properly account for the (usually) rare message-passing delays caused by, for example, lost messages and subsequent retransmit timeouts. The discussion in Section 4.8, however, provides the base upon which this facility could be built.

2. MPIBench only records raw empirical and simple statistical performance information. Based on the knowledge that point-to-point message-passing operations are best described by Pearson 5 distributions, it would be valuable to extend MPIBench so that it can automatically fit measured data to such distributions and plot the resultant fit parameters across a range of message sizes and contention levels.

3. Because of the substantial software-engineering effort that would have been required, an automated compiler able to generate PEVPM models of input code was not developed during this thesis. The construction of such a tool would a very valuable exercise, and would greatly increase the availability of the PEVPM approach to ordinary programmers. Creating an automated compiler for annotating MPI/C source code with PEVPM instructions (according to the rules described in Section 3.4.4) using standard compiler construction tools such as `lex` and `yacc` should be fairly straightforward. Following that, the code-generation stage for that compiler could be extended to produce an execution-structure simulator for evaluating the PEVPM model produced by the preprocessor (according to the PEVPM evaluator described in Section 3.5).

4. The restrictions to the applicability of the PEVPM modelling system listed in Section 3.3 could be relaxed if appropriate techniques were developed to deal with the problems they cause. For example, the inability to model non-dedicated parallel

platforms could be rectified by combining submodels of individual parallel programs into a meta-program describing the entire workload.

5. It would be interesting to apply the PEVPM approach to emerging parallel programming techniques such as Grid programming. In theory, the PEVPM approach should be well suited to modelling programs on the Grid because of the highly probabilistic nature of communication performance between sites connected via the Internet.
6. It would be extremely useful if Pearson 5 performance parameters could be calculated from a machine's hardware description, as discussed at the end of Section 4.7. Then, PEVPM communication submodels could be simply obtained through computation, rather than via extensive benchmarking.
7. The PEVPM could possibly be enhanced to work with entirely symbolic performance quantities (instead of the current mixture of symbolic and empirical quantities). This would reduce the already low evaluation cost of PEVPM models, which would make it even more attractive for very wide-ranging parametric-based performance optimisation.

Appendix A

PEVPM Definitions

Machine Dependencies

Source code prototype

```
N/A
```

Performance schemas

```
N/A
```

PEVPM directives

```
/* To specify a processor model use:  */
// PEVPM Processor description = <processor identifier>
// PEVPM & timing basis = <processor speed>

/* To specify a network model use:  */
// PEVPM Network description = <network identifier>
// PEVPM & link <id> = <from> <to> <performance profile>
// PEVPM & ...  = ...

/* To specify default processor and network models use:  */
// PEVPM Default processor = <processor identifier>
// PEVPM & numprocs = <number of processors>
// PEVPM & speed = <processor speed>
// PEVPM & speed_i = <processor speed>
// PEVPM & network = <network descriptors>
```

Notes

These directives must appear at the top of a complete PEVPM model.

Serial Processing and Message Passing

Source code prototype

```
z = a + b;                              // simple statement 1
z = z + c;                              // ...
...                                     // ...
z = z + n;                              // simple statement n
MPI_Send(&z, ..., from, to, ...);      // MPI operation
y = y - m;                              // new compund statement
y = y - l;
...
y = y - d;
```

Performance schemas

$$\begin{aligned}
T(\textit{simple statement}) &= t_{\textit{simple statement}} \\
T(\textit{compound statement}) &= t_{\textit{simple statement}_1} + \ldots + t_{\textit{simple statement}_n} \\
T(\textit{MPI\_operation}) &= t_{\textit{MPI\_operation}}(type, when, size, from, to) \\
T(\textit{overall segment}) &= T_{\textit{serial segment}_1} + T_{\textit{MPI\_Send}} + T_{\textit{serial segment}_2}
\end{aligned}$$

PEVPM directives

```
/* To specify a simple or compound statement model use: */
// PEVPM Serial [on <processor identifier>] time = <basis time>

/* To specify an MPI_operation model use: */
// PEVPM Message type = <MPI operation>
// PEVPM & size = <message size>
// PEVPM & from = <processor number(s)>
// PEVPM & to = <processor number(s)>
// PEVPM & req/stat = <request/status identifier>
```

Notes

These performance directives must be used to separate overall models into segments of serial computation (with the Serial directive) and MPI operations (with the Message type directive).

Functions and Subroutines

Source code prototype

```
[return type] function|subroutine(arguments){
 // function|subroutine body
}
```

Performance schemas

$$T(subroutine) \;=\; t_{subroutine\ body}$$

PEVPM directives

```
N/A
```

Notes

Functions and subroutines should be expanded using preprocessor inlining.

Loop Constructs

Source code prototype

```
for (i = 0; i < n; i++) {
 // loop body
}
while (c) {
 // loop body
}
```

Performance schemas

$$T(loop) \;=\; t_{loop\ body_1} \;+\; ... \;+\; t_{loop\ body_n}$$
$$T(loop) \;=\; t_{loop\ body} \;*\; n$$

PEVPM directives

```
/* To specify a loop model use:  */
// PEVPM Loop iterations = <number of iterations>
// PEVPM {
// PEVPM   /* The model for the loop body must be inserted here */
// PEVPM }
```

Notes

The number of iterations may be a symbolic quantity.

Conditional Constructs

Source code prototype

```
if (condition1){
 // code to execute if condition1 is true
}
else if (condition2){
 // code to execute if condition2 is true
}
...
else{
 // code to execute otherwise
}
```

Performance schemas

$$T(conditional) = c_1 : t_{segment_1} \mid ... \mid c_n : t_{segment_n}$$

PEVPM directives

```
// PEVPM Condition c1 = <weighting of condition 1>
// PEVPM {
// PEVPM   /* The model for condition1 must be inserted here */
// PEVPM }
// PEVPM & c2 = <weighting of condition 2>
// PEVPM {
// PEVPM   /* The model for condition1 must be inserted here */
// PEVPM }
...
// PEVPM & cn = <weighting of condition n>
// PEVPM {
// PEVPM   /* The model for condition n must be inserted here */
// PEVPM }
```

Notes

The branch taken during PEVPM model execution should be made using Monte Carlo sampling, based on the weightings of the condition parameters.

Appendix B

Using MPIBench

B.1 Running MPIBench

NAME

```
    mpibench - a program to benchmark MPI performance
```

SYNOPSIS

```
    mpibench
          [ -test type
            -size initial increment maximum ] |
          [ -application type ]
            -reps repetitions
            -output filename
            -title string
            -ppn processes per node
```

OPTIONS

```
    -test type
        Benchmark a low-level MPI operation of type:
            allgathereach  for MPI_Allgather (size bytes per process)
            allgathertotal for MPI_Allgather (size bytes in total)
            alltoalleach   for MPI_Alltoall  (size bytes per process)
            alltoalltotal  for MPI_Alltoall  (size bytes in total)
            barrier        for MPI_Barrier
            bcast          for MPI_Bcast
            gathereach     for MPI_Gather    (size bytes per process)
            gathertotal    for MPI_Gather    (size bytes in total)
            isend          for MPI_Isend
            isendlocal     for MPI_Isend     (local completion time)
            sendrecv       for MPI_Sendrecv
            scattereach    for MPI_Scatter   (size bytes per process)
            scattertotal   for MPI_Scatter   (size bytes in total)

    -size initial increment maximum
        Messages sizes (in bytes) to test in a low-level benchmark

    -application type
        Benchmark an application kernel of type:
            bots           10,000 tasks using a master-slave pattern
            fft            100,000 256x256 point Fast Fourier Transforms
            jacobi         100,000 Jacobi iterations on a 256x256 grid

    -reps repetitions
        The number of the times the repeat a test

    -output filename
        The root name for output files ending with:
            .gnu           a gnuplot file to plot the recorded data
            .histograms    histograms recorded for each size
            .outliers      a list of outliers recorded for each size
            .subsamples    an unprocessed list of subsampled data
            .summary       minimum and average times for each size

    -title string
        The title string for gnuplot output

    -ppn processes per node
        The number of MPI processes per node (for process placement)
```

ENVIRONMENT

```
    MPIBENCH_HIST_BIN (default: 0.05ms)
        Bin size for histogram processing

    MPIBENCH_OUTLIER_PERCENTILE (default: 0.99)
    MPIBENCH_OUTLIER_PERCENTILE_MULTIPLIER (default: 5)
        Separate histogram and outlier data using the
        OUTLIER_PERCENTILE'th-ranked value multiplied by
        OUTLIER_PERCENTILE_MULTIPLIER

    MPIBENCH_RESEND_LIMIT (default: 1000)
        Continue synchronising until this many messages have been
        sent without an improvement in clock resolution

    MPIBENCH_SETUP_REPS (default: 10)
        Warm up connections and caches using this many repetitions

    MPIBENCH_SUBSAMPLES (default: 1)
        The number of individual points per processor to sub-sample

    numprocs
        The number of MPI processes invoked must be a multiple of 2
```

DIAGNOSTICS

```
    Logging data is printed to standard output.  The amount of
    feedback can be increased by setting MPIBENCH_VERBOSE
```

B.2 Customising MPIBench

This section describes how to port and extend MPIBench v1.x, which can be obtained from `http://dhpc.adelaide.edu.au/projects/MPIBench/`.

Porting MPIBench

The low-level timing code in MPIBench relies upon the ability to access a CPUs cycle counter. In particular, MPIBench requires a `get_ticks()` function to be supplied, which should return the value of the cycle counter in an unsigned 64-bit integer value; implementations are supplied out-of-the box for modern `x86`-based processors and sparcv9 processors. A portable `MPI_Wtime` implementation is also provided, but this should only be used on high-end parallel machines that provide a very accurate `MPI_Wtime` operation. Writing a `get_ticks()` function for a new architecture is relatively simple, but requires writing a very small amount of assembly language; refer to the hardware and compiler manuals for your machine for guidance.

In most SMP-based machines, component CPUs are not run in lock-step, and cycle counters are synchronised in software by the operating system rather than the underlying hardware. Although the clocks remain in relatively good synchronisation after this (usually to within several microseconds) the offset between the clocks can be quite large. Now, because most operating systems dynamically schedule CPU allocation in multiprocessor machines, it is quite possible (in these situations) that MPIBench will read the cycle counter to start timing an operation on one processor and read the cycle counter to start timing the same operation on another – leading to erroneous results. This needs to be overcome by binding an MPI process to a single physical processor, to ensure that timing at any process is conducted with the same physical clock. In particular, therefore, MPIBench requires a `bind_to_processor` function that takes `int procs_per_node` as an argument and should bind the i^{th} MPI process to physical processor `procnum%procs_per_node` in each SMP node. MPIBench currently supports Linux using the pset patch [179] and Solaris's using its `processor_bind` function. Note that processor binding is not required if `MPI_Wtime`-based timing is used. Adding a `bind_to_processor` function to MPIBench for a new operating system is usually relatively simple; refer to the programming manuals for your operating system for instructions.

Lastly, it is advisable to tweak the amount of time required for global synchronisation for different types of underlying networks by varying the MPIBENCH_RESEND_LIMIT. This controls the number of synchronisation messages that are sent during global synchronisation: synchronisation is deemed sufficient when this number of messages have been sent without an improvement in global clock resolution. Networks with small performance variation (such as Myrinet or QsNet) can use quite low values for this parameter (for example, 100-500), while other networks (such as Fast Ethernet) must use higher values (for example, 1000) to achieve reasonable results. The best way to determine a reasonable value this parameter is simply by playing around a little bit to get a feeling for the performance characteristics of a machine's communication network.

Extending MPIBench

The easiest way to add a new low-level operation or application kernel benchmark to MPIBench is to copy-paste-and-modify an existing case. The three types of operation that can be easily added are point-to-point operations (for example, imitate the `test_type==isend` case); collective operations (for example, imitate the `test_type==bcast` case); and application kernels (for example, imitate the `test_type==jacobi` case). For each of these cases, only a few sections of code need to be augmented: the `test_type struct`, the command-line processing code, the memory-initialisation code, the timing cradle and instructions to extract individual times from the array of recorded results. All of these modifications are quite trivial, and should only require a few lines of code.

Bibliography

[1] V.S. Adve. *Analyzing the Behavior and Performance of Parallel Programs.* PhD thesis, University of Wisconsin, Computer Sciences Department, December 1993.

[2] V.S. Adve and M.K. Vernon. The influence of random delays on parallel execution times. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 61–73, May 1993.

[3] T. Agerwala. An analysis of controlling agents for asynchronous processes. Technical Report 35, Johns Hopkins Computer Science Program, August 1974.

[4] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On communication latency in PRAM computations. In *Proceedings of the DAGS/PC Symposium*, pages 76–86, 1993.

[5] A. Aho, J. Hopcroft, and J. Ulman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[6] A. Aho, J. Hopcroft, and J. Ulman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[7] A. Aho, R. Sethi, and J. Ulman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[8] Khalid Al-Tawil and Csaba Andras Moritz. Performance modeling and evaluation of MPI. *Journal of Parallel and Distributed Computing*, 61(2):202–223, 2001.

[9] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 1984.

[10] Mark Allman. On the generation and use of TCP acknowledgements. *ACM Computer Communication Review*, 28(5):4–21, October 1998.

[11] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP congestion control. Technical Report RFC 2581, The Internet Society, April 1999.

[12] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for probabilistic real-time systems. In *Proceedings of the 18th International Conference on Automata, Languages and Programming (LNCS 510)*, 1991.

[13] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–236, April 1994.

[14] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. *Proceedings of the American Federation of Information Processing Societies*, 30:483–485, 1967.

[15] Yair Amir, Baruch Awerbuch, Amnon Barak, Ryan S. Borgstrom, and Arie Keren. An opportunity cost approach for job assignment and reassignment in a scalable computing cluster. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing Systems*, 1998.

[16] H.H. Ammar, S.M.R. Islam, M. Ammar, and S. Deng. Performance modeling of parallel algorithms. In *Proceedings of the International Conference on Parallel Processing*, volume 3, pages 68–71, 1990.

[17] T.E. Anderson, D.E. Culler, D.A. Patterson, and The NOW team. A case for NOW (Networks Of Workstations). *IEEE Micro*, pages 54–64, February 1995.

[18] T.W. Anderson and D.A. Darling. Asymptotic theory of certain goodness-of-fit criteria based on stochastic processes. *Annals of Mathematical Statistics*, 23:193–212, 1954.

[19] Cosimo Anglano. Predicting parallel applications performance on non-dedicated cluster platforms. In *Proceedings of Supercomputing*, pages 172–179, 1998.

[20] Cosimo Anglano. Cluster benchmarks web page. `http://www.di.unito.it/~mino/cluster/benchmarks/`, May 2001.

[21] Argonne National Laboratory and Mississippi State University. MPICH 1.2.0. Available from `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[22] Leon David Aronson. *A Theory of Routing in Parallel Computers*. PhD thesis, Delft University of Technology, October 1999.

[23] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1995.

[24] Remzi H. Arpaci, Amin M. Vahdat, Tom Anderson, and Dave Patterson. Combining parallel and sequential workloads on a NOW. Technical Report CSD-94-838, University of California Berkeley, Computer Science Department, 1994.

[25] D. Atapattu and D. Gannon. Building analytical models into an interactive prediction tool. In *Proceedings of Supercomputing*, pages 521–530, 1989.

[26] Australian Partnership for Advanced Computing (APAC). National facility home page. Available from `http://nf.apac.edu.au/`.

[27] Rajive Bagrodia, Ewa Deelman, Steven Docy, and Thomas Phan. Performance prediction of large parallel applications using parallel simulations. In *Proceedings of the ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, May 1999.

[28] D. Bailey, E. Barszcz, J. Barton, D. Browning, et al. The NAS parallel benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[29] Mark Baker and Geoffrey Fox. MPI on NT: The current status and performance of the available environments. In *Proceedings of the 5th European PVM/MPI Users' Group Meeting (LNCS 1497)*, pages 63–75, September 1998.

[30] V. Balasundaram, G. Fox, K Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the 3rd ACM SIGPLAN Symposium on PPoPP*, April 1991.

[31] Deb Banerjee, Thomas Tysinger, and Wayne Smith. A scalable high-performance environment for fluid flow analysis on unstructured grids. In *Proceedings of Supercomputing*, pages 8–17, 1994.

[32] Jeffrey C. Becker, Bill Nitzberg, and Rob F. Van der Wijngaart. Predicting cost/performance trade-offs for Whitney: A commodity computing cluster. In *Proceedings of the 31st Hawaii International Conference on System Sciences*, volume 7, pages 504–513, January 1998.

[33] B. Beizer. *Micro Analysis of Computer System Performance.* Van Nostrand Reinhold, New York, 1978.

[34] G Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs LogP. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32, June 1996.

[35] L.S. Blackford, A. Cleary, J. Choi, Dongarra, et al. LAPACK working note 93 installation guide for ScaLAPACK, May 1997.

[36] R. Blasko. Automatic modeling and performance analysis of parallel processes by PEPSY. In *Proceedings of the ASIM Symposium*, pages 241–246, October 1994.

[37] R. Blasko. Process graph and tool for performance analysis of parallel processes. In *Proceedings of the IMACS Symposium on Mathematical Modeling*, pages 60–64, February 1994.

[38] R. Blasko. A systematic strategy for performance prediction by improvement of parallel programs. In *Proceedings of the 4th International Workshop on Computer Aided Systems Technology*, May 1994.

[39] R. Blasko. Hierarchical performance prediction for parallel programs. In *Proceedings of the IEEE International Symposium on Systems Engineering of Computer Based Systems*, pages 398–405, March 1995.

[40] R. Blasko. Simulation based performance prediction by PEPSY. In *Proceedings of the 28th Annual IEEE Simulation Symposium*, pages 341–349, April 1995.

[41] R. Blasko. Performance analysis of parallel programs based on simulation. In *Proceedings of the 20th ASU Conference*, pages 70–79, September 1999.

[42] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Si. Myrinet – a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–38, February 1995.

[43] S.H. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Transactions on Software Engineering*, 7(6):583–589, 1981.

[44] Shahid H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, 30(3):207–214, 1981.

[45] Shahid H. Bokhari. Partitioning problems in parallel, pipelined and distributed computing. *IEEE Transactions on Computers*, 37(1):48–57, 1988.

[46] R. Bracewell. *The Fourier Transform and Its Applications.* McGraw-Hill, New York, 1999.

[47] B. Braden (ed.). Requirements for internet hosts – communication layers. Technical Report RFC 1122, University of Southern California, Information Sciences Institute, October 1989.

[48] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the 1994 SIGCOMM Symposium*, pages 24–35, August 1994.

[49] Thomas Brëanl. *Parallel Programming - An Introduction*, chapter 3. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[50] T.H. Bredt and E.J. McCluskey. A model for parallel computer systems. Technical Report STAN-CS-70-160, Stanford University, Digital Systems Laboratory, April 1970.

[51] J. Brehm, L. Dowdy, M. Madhukar, and E. Smirni. PrePreT - a performance prediction tool. In *Quantitative Evaluation of Computing and Communication Systems (LNCS 977)*. Springer-Verlag, 1995.

[52] E. Brockmeyer, H.L. Halstrøm, and A. Jensen. The life and works of A.K. Erlang. *Transactions of the Danish Academy of Technology and Science*, 2, 1948.

[53] Stephen D. Brookes. On the relationship of CCS and CSP. In *Advanced NATO Study Institute on Logics and Models for Verification and Specification of Concurrent Systems.* Institut National de Recherche en Informatique et Automatique, 1984.

[54] Shirley Browne, Jack Dongarra, and Kevin London. Review of performance analysis tools for MPI parallel programs. Technical report, University of Tenessee, Department of Computer Science, December 1997.

[55] H. Burkhart, C. Falcó Korn, S. Gutzwiller, P. Ohnacker, and S. Waser. BACS: Basel Algorithm Classification Scheme. Technical Report 93-3, Institut für Informatik der Universität Basel, March 1993.

[56] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[57] P.H. Canrns, W.B. Lignon III, S.P. McMillan, and R.B. Ross. An evaluation of message passing implementations on Beowulf workstations. In *Proceedings of the IEEE Aerospace Conference*, March 1999.

[58] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *Proceedings of Supercomputing*, November 2000.

[59] Franck Cappello and Olivier Richard. Performance characteristics of a network of commodity multiprocessors for the NAS benchmarks using a hybrid memory model. In *Proceedings of PACT'99*, 1999.

[60] Franck Cappello, Olivier Richard, and Daniel Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, 1998.

[61] Franck Cappello, Olivier Richard, and Daniel Etiemble. Understanding performance of SMP clusters running MPI programs. *Future Generation Computer Systems*, 17(6):711–720, 2001.

[62] Dryan Carpenter, Guansong Zhang, Geoffrey Fox, Xinying Li, and Yuhong Wen. HPJava: Data parallel extensions to Java. *Concurrency: Practice and Experience*, 10(11-13):873–877, 1998.

[63] Nick Carriero, Eric Freeman, David Gelernter, and David Kaminsky. Adaptive parallelism and piranha. *IEEE Computer*, 28(1):40–49, January 1995.

[64] K.M. Chandy and I. Foster. A deterministic notation for cooperating processes. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):863–871, 1995.

[65] B.M. Chapman, P. Mehrotra, and H.P. Zima. Extending HPF for advanced data parallel applications. In *IEEE Magazine on Parallel and Distributed Technology*, pages 59–70, 1994.

[66] J. Choi, J.J. Dongarra, R. Pozo, and D.W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, 1992.

[67] Yuan-Chieh Chow and Walter H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers*, 28(5):354–361, 1979.

[68] F. Christian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[69] E. Chu and A. George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms.* CRC Press, Boca Raton, Florida, 2000.

[70] Mark J. Clement and Michael J. Quinn. Multivariate statistical techniques for parallel performance prediction. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, volume 2, pages 446–455, January 1995.

[71] Mark J. Clement and Michael J. Quinn. Automated performance prediction for scalable parallel computing. *Parallel Computing*, 10(23):1405–1420, 1997.

[72] Mark J. Clement, Michael R. Steed, and Phyllis E. Crandall. Network performance modeling for PVM clusters. In *Proceedings of Supercomputing*, November 1996.

[73] M. Cole. Algorithmic skeletons: Structured management of parallel computation. In *Research Monographs in Parallel and Distributed Computing*. The MIT Press, Cambridge, Massachusetts, 1989.

[74] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony in the PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, June 1989.

[75] Compaq Computer Corporation. The Compaq AlphaServer SC supercomputer. Available from `http://www.compaq.com/hpc/systems/sys_sc.html`.

[76] Compaq Computer Corporation. Alpha architecture handbook: Version 4, February 1998.

[77] Compaq Computer Corporation. Compaq AlphaServer SC System Software, version 2.4A. Technical Report SPD 80.10.03, Compaq Computer Corporation, May 2002.

[78] Compaq Computer Corporation, Intel Corporation, and Microsoft Corporation. Virtual Interface Architecture specification, December 1997.

[79] J. W. Cooley and O. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, pages 297–301, April 1965.

[80] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with 'readers' and 'writers'. *Communications of the ACM*, 14(10):667–668, 1971.

[81] Jason A. Crawford and Clark Mobarry. Hrunting: A distributed shared memory system for the Beowulf parallel workstation. In *Proceedings of the IEEE Aerospace Conference*, 1998.

[82] Mark E. Crovella and Thomas J. LeBlanc. Parallel performance prediction using lost cycles analysis. In *Proceedings of Supercomputing*, pages 600–609, 1994.

[83] M.E. Crovella. *Performance Prediction and Tuning of Parallel Programs.* PhD thesis, University of Rochester, 1994.

[84] Lawrence A. Crowl. How to measure, present and compare parallel performance. *Parallel and Distributed Technology*, 2(1):9–25, 1994.

[85] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subruamonian, and T. Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 5th ACM SIGPLAN Symposium on the Principles and Practices of Parallel Programming*, pages 1–12, May 1993.

[86] Zarka Cvetanovic. The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems. *IEEE Transactions on Computers*, 36(4):421–432, 1987.

[87] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. Technical Report 965, University of Illinois Center for Supercomputing R&D, March 1990.

[88] W.J. Dally and D.S. Willis. Universal mechanisms for concurrency. In *Proceedings of Parallel Architectures and Languages Europe (LNCS 365)*, pages 19–33, 1989.

[89] M. Danelutto, R. Do Meglio, S. Pelagatti, and M. Vanneschi. High level language constructs for massively parallel computing. In *Proceedings of the 6th International Symposium on Computer and Information Sciences*, pages 777–788, October 1991.

[90] Sajal K. Das, Daniel J. Harvey, and Rupak Biswas. Dynamic load balancing for adaptive meshes using symmetric broadcast networks. In *Proceedings of the 12th ACM International Conference on Supercomputing*, pages 417–424, 1998.

[91] S. Dasgupta. A hierarchical taxonomic system for computer architectures. *IEEE Computer*, pages 64–74, March 1990.

[92] J. Davies and S. Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 138(10):243–271, 1995.

[93] J.W.M Davies. *Specification and Proof in Real-Time Systems.* Cambridge University Press, Cambridge, 1993.

[94] Bronis R. de Supinski and Nicholas T. Karonis. Accurately measuring MPI broadcasts in a computational grid. In *Proceedings of the 8th IEEE Symposium on High Performance Distributed Computing*, pages 29–37, August 1999.

[95] Thomas Decker, Reinhard Luling, and Stefan Tschoke. A distributed load balancing algorithm for heterogeneous parallel computing systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 933–940, 1998.

[96] Viktor K. Decyk, Dean E. Dauger, and Pieter R. Kokelaar. How to build an AppleSeed: A parallel Macintosh cluster for numerically intensive computing. In *Proceedings of the 6th International School for Space Simulation*, September 2001.

[97] Hank Dietz. Linux parallel processing HOWTO, January 1998.

[98] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Journal of the ACM*, 8:569, 1965.

[99] E.W. Dijkstra. Cooperating sequential processes. *Programming Languages*, pages 43–112, 1968.

[100] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[101] Bryan Dodson. *Weibull Analysis with Software*. ASQ Quality Press, Milwaukee, 1995.

[102] Jack Dongarra, Hans Meuer, and Erich Strohmaier. Top 500 supercomputer sites. Available from `http://www.top500.org/`.

[103] Jack Dongarra, Hans Meuer, and Erich Strohmaier. TOP500 supercomputer sites, 15th edition. In *Proceedings of Supercomputing*, June 2000.

[104] Jack Dongarra, Hans Meuer, and Erich Strohmaier. TOP500 supercomputer sites, 16th edition. In *Proceedings of SC2000*, November 2000.

[105] J.J. Dongarra, J.R. Bunch, C.B. Moler, and Stewart G.W. *LINPACK User's Guide*. SIAM, 1979.

[106] M. Dubois and M. Briggs. Performance of synchronized iterative processes in multiprocessor systems. *IEEE Transactions on Software Engineering*, 8:419–431, July 1982.

[107] Alistair Dunlop and Tony Hey. PERFORM - a fast simulator for estimating program execution time. *On-line Journal of Performance Evaluation and Modelling for Computer Systems*, November 1997. `http://www.netlib.org/utk/papers/PEMCS/`.

[108] A.N. Dunlop and D.J. Pritchard. Parallel performance estimator. Technical Report D5.3b, ESPRIT project, Department of Electronics and Computer Science, University of Southampton, 1995.

[109] Peter J. Dunning. The working set model of program behaviour. *Communications of the ACM*, 11(5):323–333, May 1968.

[110] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38:408–423, March 1989.

[111] A.K. Erlang. Solution of some problems in the theory of probabilities of significance in automatic telephone exchanges. *Elektroteknikeren*, 13, 1917.

[112] T. Fahringer. *Automatic Performance Prediction of Parallel Programs on Massively Parallel Programs.* PhD thesis, University of Vienna, 1993.

[113] T. Fahringer. Automatically estimating network contention of parallel programs. In *Proceedings of the 7th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, May 1994.

[114] T. Fahringer. *Automatic Performance Prediction of Parallel Programs.* Kluwer Academic, Boston, Massachusetts, 1996.

[115] T. Fahringer, R. Blasko, and H.P. Zima. Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 347–356, 1992.

[116] T. Fahringer and H.P. Zima. A static parameter-based performance prediction tool for parallel programs. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 207–219, July 1993.

[117] Kevin Fall and Sally Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *ACM Computer Communication Review*, 26(3):5–21, 1997.

[118] Rod Fatoohi and Sisira Weeratunga. Performance evaluation of three distributed computing environments for scientific applications. In *Proceedings of Supercomputing*, pages 400–409, 1994.

[119] Anja Feldman. Impact of non-Poisson arrival sequences for call admission algorithms with and without delay. In *Proceedings of Globecom*, 1996.

[120] Anja Feldman. Characteristics of TCP connection arrivals. Technical report, AT&T Labs-Research, December 1998.

[121] Anja Feldman and Ward Whitt. Fitting mixtures of exponentials to long-tail distributions to analyze network performance models. *Performance Evaluations*, 31, 1998.

[122] C. Figueira and Hernández. Benchmarks specification and generation for performance estimation on MIMD machines. *IFIP Transactions on Computer Science and Technology*, 44:215–223, 1994.

[123] Silvia M. Figueira and Francine Berman. Modeling the effects of contention on the performance of heterogeneous applications. In *Proceedings of the 5th International Symposium on High Performance Distributed Computing*, August 1996.

[124] Silvia M. Figueira and Francine Berman. Predicting slowdown for networked workstations. In *Proceedings of the 6th International Symposium on High Performance Distributed Computing*, August 1997.

[125] R. Figueiredo, B. Lin, V. Misra, and D. Towsley. On the autocorrelation structure of TCP traffic. Technical Report CMPSCI TR 00-55, University of Massachusetts at Amherst, Department of Computer Science, 2000.

[126] George S. Fishman. *Monte Carlo: Concepts, Algorithms and Applications.* Springer-Verlag, New York, 1996.

[127] G. Fleischmann. Performance evaluation of parallel program based on model calculations. *Parallel Computing*, 20(10-11), November 1994.

[128] G. Fleischmann and M. Gente. Modeling and evaluation of parallel programs using GIANT. In *Proceedings of the 6th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, September 1992.

[129] S. Floyd. Congestion control principles. Technical Report RFC 2914, The Internet Society, September 2000.

[130] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An extension to the Selective Acknowlegement (SACK) option for TCP. Technical Report RFC 2883, The Internet Society, July 2000.

[131] M Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:94, 1972.

[132] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[133] I. Foster and C. Kesselman (eds.). *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, San Francisco, 1999.

[134] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, 1998. Also see MPI FFTW available from `http://www.fftw.org/doc/fftw_4.html#SEC55`.

[135] K. Gallivan, W. Jalby, A. Malony, and H. Wijshoff. Performance prediction for parallel numerical algorithms. *International Journal of High-Speed Computing*, 3(1):31–62, 1991.

[136] Hasyim Gautama. A probabilistic approach to the analysis of program execution time. Master's thesis, Delft University of Technology, Information Technology and Systems, 1998.

[137] 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, 1994.

[138] E. Gelenbe and Z. Liu. Performance analysis approximations for parallel processing in multiprocessor systems. In *Proceedings of the IFIP Working Conference on Parallel Processing*, pages 363–375, April 1988.

[139] E. Gelenbe, E. Montagne, R. Suros, and C.M. Woodside. Performance of block-structured parallel programs. In M. Cosnard et al., editors, *Parallel Algorithms and Architectures*, pages 127–138. North-Holland, Amsterdam, 1986.

[140] V. Georgitsis and J. Sobolewski. Performance of MPL and MPICH on the SP2 system. In *Proceedings of the MPI Developer's Conference*, June 1995.

[141] Vasilios Georgitsis. *Message Passing Performance on SP Systems.* PhD thesis, University of New Mexico, 1996.

[142] Vladimir Getov, Hernández, and Tony Hey. Message-passing performance of parallel computers. *Lecture Notes in Computer Science*, 1300, 1997.

[143] V.S. Getov, R.W. Hockney, and A.J.G. Hey. Performance analysis of distributed applications by suitability functions. In *Proceedings of Programming Models for Massively Parallel Computers*, pages 191–197, September 1993.

[144] Wolfgang K. Giloi. Parallel supercomputer architectures and their programming models. *Parallel Computing*, 20(10-11):1443–1470, 1994.

[145] S. Girona and J. Labarta. Sensitivity of performance prediction of message passing programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 933–940, June 1999.

[146] S. Girona, J. Labarta, and Rosa M. Badia. Validation of Dimemas communication model for MPI collective operations. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting*, September 2000.

[147] L.M. Goldschalger. A unified approach to models of synchronous parallel machines. *Journal of the ACM*, 24(4):1073–1086, 1982.

[148] A.Y. Grama, Gupta. A, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August 1993.

[149] Greer Mountain Software. Stat::Fit software, version 1.1. Available from `http://www.geerms.com/`.

[150] William Gropp and Ewing Lusk. Reproducible measurements of MPI performance characteristics. In *Proceedings of the PVM/MPI Users' Group Meeting (LNCS 1697)*, pages 11–18, 1999.

[151] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[152] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, Cambridge, Massachusetts, 1994.

[153] Duncan A. Grove. Precise MPI performance measurement using MPIBench. In *Proceedings of HPC Asia*, September 2001.

[154] Bu Guanying and Xu Zhiwei. Grid system theoretical model. In *Proceedings of HPC Asia*, September 2001.

[155] M. Gupta and P. Banerjee. Compile-time estimation of communication costs of programs. In *Proceedings of the 2nd Workshop on Automatic Data Layout and Performance Prediction*, April 1995.

[156] George Gusciora. SP parallel programming workshop – 2D FFT example. Available from `http://www.mhpcc.edu/training/workshop/mpi/exercise.html`.

[157] John L. Gustafson, Don Heller, Rajat Todi, and Jenwei Hsieh. Cluster performance: SMP versus uniprocessor nodes. In *Proceedings of Supercomputing*, November 1999.

[158] B. van Halderen. A tool for application performance prediction. Master's thesis, University of Amsterdam, Department of Mathematics and Computer Science, September 1995.

[159] Tim J. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys*, 26(2):187–200, June 1994.

[160] J. Hartmanis and R.E. Stearns. On the computational complexity of algorithms. *Transactions AMS*, 117:285–306, 1965.

[161] K.A. Hawick, D.A. Grove, P.D. Coddington, and M.A. Buntine. Commodity cluster computing for computational chemistry. *Internet Journal of Chemistry*, 3:article 4, 2000.

[162] M.T. Heath. Performance visualization with ParaGraph. In *Proceedings of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing*, pages 221–230, 1994.

[163] Philip Heidelberger and Stephen S. Lavenberg. Computer performance evaluation methodology. *IEEE Transactions on Computers*, 33(12):1195–1220, December 1994.

[164] R. Henderson and D. Tweten. Portable Batch System: External reference specification. Technical report, NASA Ames Research Center, 1996.

[165] J.L. Hennessey and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 1996.

[166] H. Hermanns, Herzog U., U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. *Lecture Notes in Computer Science*, 1469:51–62, 1998.

[167] E. Hernandez and A.J.G. Hey. White-box benchmarking. In *Proceedings of the 4th International Euro-Par Conference (LNCS 1470)*, pages 220–223, June 1998.

[168] E. Hernández and T. Hey. Variations on low-level communication benchmarks. *Supercomputing*, 12(4):16–27, December 1996.

[169] A.J.G. Hey and D Lancaster. The development of Parkbench and performance prediction. *International Journal of High-Performance Computing*, 14(3):205–215, August 2000.

[170] Anthony J.G. Hey, Alistair N. Dunlop, and E. Hernández. Realistic parallel performance estimation. *Parallel Computing*, 23(1-2):5–21, April 1997.

[171] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation. *Journal of Parallel and Distributed Computing*, 16(3):233–249, November 1992.

[172] Todd Heywood and Claudia Leopold. Models of parallelism. In J.R. Davy and P.M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*. Oxford University Press, Oxford, 1995.

[173] T. Hickey and J. Choen. Automating program analysis. *Journal of the ACM*, 35:185–219, January 1988.

[174] High Performance Fortran Forum (HPFF). High Performance Fortran language specification. Available from `http://www.crpc.rice.edu/HPFF/`.

[175] Jim Hill, Michael Warren, and Patrick Goda. I'm not going to pay a lot for this supercomputer. *Linux Journal*, 45, January 1998.

[176] Jonathan M.D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[177] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[178] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[179] Tim Hockin. Pset - processor sets for Linux/SMP. `http://isunix.it.ilstu.edu/~thockin/pset/`.

[180] R.W. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 17(10), December 1991.

[181] R.W. Hockney and C.R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger, Bristol, 1988.

[182] Jenwei Hsieh. Design choices for a cost-effective, high-performance Beowulf-cluster. *Dell Power Solutions*, 3, 2000.

[183] Christopher J. Hughes, Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. Rsim: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, February 2002.

[184] Lars Paul Huse. Collective communication on dedicated clusters of workstations. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting*, pages 469–476, September 1999.

[185] R.N. Ibbet, T. Heywood, M.I. Cole, R.J. Pooley, et al. Algorithms, architectures and models of computation. Technical Report ECS-CSG-22-96, University of Edinburgh, Division of Informatics, 1996.

[186] Intel Corporation. Using the RDTSC instruction for performance monitoring, 1997. Available from `http://cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf`.

[187] SPARC International. *The SPARC architecture manual: Version 9.* Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[188] Nayeem Islam. Characterizing parallel and distributed applications. In *Distributed Objects*. IEEE Computer Society Press, 1996.

[189] Van Jacobson. Congestion avoidance and control. *ACM Computer Communication Review*, 18(4):316–329, 1988.

[190] R. Jain. *The Art of Computer Systems Performance Analysis.* Wiley, New York, 1991.

[191] N.I. Johnson, S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions.* Wiley, New York, 1995.

[192] H. Jonkers. *Performance Analysis of Parallel Systems: A Hybrid Approach.* PhD thesis, Delft University of Technology, Information Technology and Systems, October 1995.

[193] Ben H.H. Juurlink and Harry A.G. Wijshoff. Experiences with a model for parallel computation. In *Proceedings of the 12th ACM Symposium on the Principles of Distributed Computing*, pages 87–96, August 1993.

[194] Ben H.H. Juurlink and Harry A.G. Wijshoff. The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. In *Proceedings of the 2nd International Euro-Par Conference (LNCS 1124)*, 1996.

[195] Ben H.H. Juurlink and Harry A.G. Wijshoff. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems*, 16(3):271–318, August 1998.

[196] A. Kapelnikov, R.R. Muntz, and M.D. Ercegovac. A modeling methodology for the analysis of concurrent systems and computations. *Journal of Parallel and Distributed Computing*, 6(3):568–597, June 1989.

[197] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocol. *ACM Transactions on Computer Systems*, 9:365–373, 1991.

[198] Nicholas T. Karonis, Bronis R. De Supinski, Ian Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 377–384, May 2000.

[199] Richard Karp and Raymond Miller. Parallel program schema: A mathematical model for parallel computation. In *Proceedings of the 8th Annual Symposium on Switching Automata Theory*, pages 55–61, October 1967.

[200] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14:1390–1411, November 1966.

[201] D.G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the embedded markov chain. *Annals of Mathematical Statistics*, 24:338–354, 1953.

[202] Darren Kerbyson, Hank Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey Wasserman, and Mike Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of Supercomputing*, November 2001.

[203] D.E. Knuth. *The Art of Computer Programming Vol. I: Fundamental Algorithms.* Addison-Wesley, Reading, Massachusetts, 1968.

[204] D.E. Knuth. Big Omicron and Big Omega and Big Theta. *ACM SIGACT News*, 8(2):18–23, 1976.

[205] Charles Howard Koelbel. *Compiling Programs for Distributed Memory Machines.* PhD thesis, Purdue University, Department of Computer Science, 1990.

[206] A.N. Kolmogorov. On a logarithmic normal distribution law of the dimensions of particles under pulverization. *Dokl. Akad Nauk*, 31(2):99–101, 1941.

[207] D. Kranzlmüller and J. Volkert. NOPE: A nondeterministic program evaluator. In *Proceedings of the 4th International ACPC Conference (LNCS 1557)*, pages 490–499, February 1992.

[208] C. Kruskal and M. Snir. Cost-bandwidth tradeoffs for communication networks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 32–41, 1989.

[209] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, June 1991.

[210] J. Labarta, S. Girona, Pillet, Cortes abd T. V., and L. Gregoris. DiP: A parallel program development environment. In *Proceedings of the 2nd International Euro-Par Conference*, volume II, pages 665–674, August 1996.

[211] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 27(7):558–565, 1978.

[212] D. Lancaster. Parkbench and the new low-level Genesis communication tests. In *Proceedings of the 14th Real Applications on Parallel Systems Workshop*, November 1998.

[213] Averill M. Law and W. David Kelton. *Simulation Modeling & Analysis.* McGraw-Hill, New York, 1991.

[214] W.E. Leland, M.S. Taqqu, W. Willinger, and D.V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2:1–15, 1994.

[215] B.P Lester. A system for computing the speedup of parallel programs. In *Proceedings of the International Conference on Parallel Processing*, pages 145–152, August 1986.

[216] C. Lin and L. Snyder. The Kheystone benchmark for parallel performance prediction. Technical Report 92-06-01, University of Washington, Department of Computer Science and Engineering, 1992.

[217] R. Lipton, L. Snyder, and Y. Zalcstein. A comparative study of models of parallel computation. In *Proceedings of the 15th Annual IEEE Symposium on Switching and Automata Theory*, pages 145–155, October 1974.

[218] Jun S. Liu. *Monte Carlo Strategies in Scientific Computing.* Springer-Verlag, New York, 2001.

[219] Michael Lo and Sivarama P. Dandamudi. Performance of hierarchical load sharing in heterogeneous distributed systems. In *Proceedings of the 8th IASTED International Conference on Parallel and Distributed Computing Systems*, pages 370–377, 1996.

[220] Joseph Loncaric. Linux 2.0.36 [2.2.12] TCP performance fix for short messages. Available from `http://www.icase.edu/coral/LinuxTCP[2].html`.

[221] Joseph Loncaric. RFC: Linux networking tweaks. Posted to the Beowulf mailing list, available from `http://www.beowulf.org/listarchives/beowulf/1999/03/0338.html`.

[222] G. Lowe. Probabilistic and prioritized models of timed CSP. *Theoretical Computer Science*, 138:315–352, 1995.

[223] Gavin Lowe. *Probabilities and Priorities in Timed CSP.* PhD thesis, St. Hugh's College, Oxford University, 1993.

[224] R. Ludwig and K. Sklower. The Eifel retransmission timer. *ACM Computer Communication Review*, 30(3), July 2000.

[225] Reiner Ludwig and Randy H. Katz. The Eifel algorithm: Making TCP robust against spurious retransmissions. *ACM Computer Communication Review*, 30(1), January 2000.

[226] Margaret Mackisack and Ronald Stillman. A cautionary tale about Weibull analysis. *IEEE Transactions on Reliability*, 1996.

[227] B.M. Maggs, L.R. Matheson, and R.E. Tarjan. Models of parallel computation: A survey and synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 61–70, January 1995.

[228] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hllberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, February 2002.

[229] Eric Maillet and Cecile Tron. On efficiently implementing global time for performance evaluation on multiprocessor systems. *Journal of Parallel and Distributed Computing*, 28(1):84–93, July 1995.

[230] V.W. Mak. *Performance Prediction of Concurrent Systems.* PhD thesis, Stanford University, Computer Science Department, 1987.

[231] V.W. Mak and S.F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–270, July 1990.

[232] W.T. Marshall and S.P. Morgan. Statistics of mixed data traffic on a local area network. In *Proceedings of Computer Networks and ISDN Systems*, pages 185–195, 1985.

[233] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 85–97, 1997.

[234] W.F. McColl. Foundations of time-critical scalable computing. In *Proceedings of the 15th IFIP World Computer Congress*, pages 93–107, 1998.

[235] C.E. McDowell and D.P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[236] P. Mehra, M. Gower, and M. Bass. Automated modeling of message-passing systems. In *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 187–192, January 1994.

[237] P. Mehra, C.H. Schulback, and J.C. Yan. A comparison of two model-based performance prediction techniques for message-passing parallel programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 181–189, May 1994.

[238] C.L. Mendes, J-C Wang, and D.A. Reed. Automatic performance prediction and scalability analysis for data parallel programs. In *Proceedings of the 2nd Workshop on Automatic Data Layout and Performance Prediction*, April 1995.

[239] Message-Passing Interface Forum (MPIF). MPI-2: Extensions to the Message Passing Interface. Available from `http://www.mpi-forum.org`.

[240] Message-Passing Interface Forum (MPIF). MPI: A Message Passing Interface standard. Available from `http://www.mpi-forum.org`.

[241] Sun Microsystems. Sun HPC ClusterTools software. `http://www.sun.com/software/hpc/`.

[242] Sun Microsystems. Sun Technical Compute Farm. `http://www.sun.com/desktop/suntcf/`.

[243] Robin Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science (92)*. Springer-Verlag, New York, 1980.

[244] Jeonghoon Mo, Richard J. La, Venkat Anantharam, and Jean C. Walrand. Analysis and comparison of TCP Reno and Vegas. In *Proceedings of IEEE INFOCOM*, pages 1556–1563, March 1999.

[245] A.G. Mohamed, G.C. Fox, G. von Laszewski, M. Parashar, T. Haupt, K. Mills, Y.H. Lu, N.T. Lin, and N.K. Yeh. Application benchmark set for Fortran-D and High Performance Fortran. Technical Report SCCS-327, Northeast Parallel Architectures Center, Syracuse University, June 1992.

[246] J. Mohan. *Performance of Parallel Programs: Model and Analyses.* PhD thesis, Carnegie Mellon University, School of Computer Science, July 1984.

[247] Csaba Andras Moritz and Matthew I. Frank. LoGPC: Modeling network contention in message-passing programs. *ACM SIGMETRICS Performance Evaluation Review Special Issue*, 26(1), 1998.

[248] Ronald Mraz. Reducing the variance of point-to-point transfers for parallel real-time programs. *Parallel and Distributed Technology*, 2(4):20–31, 1994.

[249] Philip J. Mucci, Kevin London, and John Thurman. The MPBench report. Technical Report UT-CS-98-394, University of Tenessee, Department of Computer Science, November 1998.

[250] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541–580, April 1989.

[251] Myricom Incorporated. Myrinet Protocol Module – Implementation of Sun HPC ClusterTools MPI over GM. Available from `http://www.myricom.com/scs/READMES/README-clustertools`.

[252] Myricom Incorporated. The GM API. Available from `http://www.myri.com/scs/GM/doc/gm_toc.html`.

[253] R Nelson. A performance evaluation of a general parallel processing model. *ACM SIGMETRICS Performance Evaluation Review*, 18(1):14–26, 1990.

[254] David Nicol and James Townsend. Accurate modeling of parallel scientific computations. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 165–179, 1989.

[255] Michael G. Norman and Peter Thanisch. Models of machines and modules for mapping to minimise makespan in multicomputers. Technical Report 9114, University of Edinburgh, Edinburgh Parallel Computing Centre, 1996.

[256] G.R. Nudd, D.J. Kerbyson, Papaefstathiou E., S.C. Perry, J.S. Harper, and D.V. Wilcox. PACE - A toolset for the performance prediction of parallel and distributed systems. *The International Journal of High Performance Computing Applications*, 14(3):228–251, Fall 2000.

[257] G.R. Nudd, E. Papaefstathiou, T. Papay, T.J. Atherton, C.T. Clarke, D.J. Kerbyson, A.F. Stratton, M.J. Zemerly, and R. Ziani. A layered approach to the characterisation of parallel systems for performance prediction. In *Proceedings of Performance Evaluation of Parallel Systems*, pages 26–34, November 1993.

[258] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface. Available from `http://www.openmp.org`.

[259] Pallas GmbH. Pallas MPI benchmarks home page. `http://www.pallas.com/e/produces/pmb/`.

[260] Pallas GmbH. Vampir home page. `http://www.pallas.com/e/products/vampir/`.

[261] C. Papadimitriou and J. Ullman. A communication-time tradeoff. *SIAM Journal of Computation*, 19:322–328, 1990.

[262] Efstathios Papaefstathiou. *A Framework for Characterising Parallel Systems for Performance Evaluation*. PhD thesis, University of Warwick, Computer Sciences Department, September 1995.

[263] Efstathios Papaefstathiou and D.J. Kerbyson. Predicting communication delays of detailed application workloads. In *Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems*, August 2000.

[264] Efstathios Papaefstathiou, D.J. Kerbyson, G.R. Nudd, and T.J. Atherton. An overview of the CHIP$_3$S performance prediction toolset for parallel systems. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems*, pages 527–533, 1995.

[265] Paradyn Project. Paradyn project home page. `http://www.cs.wisc.edu/~paradyn/`.

[266] Manish Parashar. *Interpretive Performance Prediction for High Performance Parallel Computing*. PhD thesis, Syracuse University, Department of Electrical and Computer Engineering, July 1994.

[267] Manish Parashar and Salim Hariri. Interpretive performance prediction for parallel application development. *Journal of Parallel and Distributed Computing*, 60:17–47, 2000.

[268] K. Park and W. Willinger (eds.). *Self-Similar Network Traffic and Performance Evaluation*. Wiley, New York, 2000.

[269] Jeff Parker and George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, October 1989.

[270] V. Paxon and S. Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3:226–244, 1995.

[271] V. Paxson and M. Allman. Computing TCP's retransmission timer. Technical Report RFC 2988, The Internet Society, November 2000.

[272] Vern Paxson. Empirically derived analytic models of wide-area TCP connections. *IEEE/ACM Transactions on Networking*, 2(4):316–336, 1994.

[273] Vern Paxson. Automated packet trace analysis of TCP implementations. *ACM Computer Communication Review*, 27:14–18, September 1997.

[274] J. Peterson and T. Bredt. A comparison of models of parallel computation. In *Proceedings of the IFIP Congress*, pages 466–470, August 1974.

[275] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[276] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware- and software-based collective communication on the Quadrics network. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, October 2001.

[277] Fabrizio Petrini, Salvador Coll Coll, Eitan Frachtenberg, and Adolfy Hoisie. Performance evaluation of the Quadrics interconnection network. *Journal of Cluster Computing*, 2002.

[278] Fabrizio Petrini, Wu-Chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, February 2002.

[279] B.L. Peuto and L.J. Shustek. An instruction timing model of CPU performance. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 165–178, March 1977.

[280] H. Pfneiszl. Synthetic workload generation for parallel processing systems. Master's thesis, University of Vienna, Institute of Applied Computer Science, January 1996.

[281] Jon Postel (ed.). Internet protocol. Technical Report RFC 791, University of Southern California, Information Sciences Institute, September 1981.

[282] S. Prakash. *Performance Prediction of Parallel Pgorams.* PhD thesis, University of California Los Angeles, Computer Science Department, 1996.

[283] S. Prakash and S. Bagrodia. Using parallel simulation to evaluate MPI programs. In *Proceedings of the Winter Simulation Conference*, December 1998.

[284] R.L. Prentice. A log-gamma model and its maximum likelihood estimation. *Biometrika*, 61:539–544, 1974.

[285] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1986.

[286] V. Puente, J.A. Gregorio, and R. Beivide. SICOSYS: An integrated framework for studying interconnection networks in multiprocessor systems. In *Proceedings of the 10th Euromicro Workshop on Parallel and Distributed Processing*, pages 15–22, 2002.

[287] V. Puente, J.M. Prellezo, C. Izu, J.A. Gregorio, and R. Beivide. A case study of trace-driven simulation for analyzing interconnection networks: cc-NUMAs with ILP processors. In *Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing*, January 2000.

[288] B. Qin, H.A. Sholl, and R.A. Ammar. Micro time cost analysis of parallel computations. *IEEE Transactions on Computers*, 40:613–628, May 1991.

[289] Xiaohan Qin and Jean-Loup Baer. A performance evaluation of cluster architectures. *ACM SIGMETRICS Performance Evaluation Review*, 25(1):237–247, June 1997.

[290] Jacek Radajewski. Beowulf supercomputer HOWTO draft, January 1998.

[291] G.M. Reed and A.W. Roscoe. A timed model for CSP. In *Proccedings of ICALP (LNCS 226)*, pages 314–323, 1996.

[292] Jeff S. Reeve. A performance estimator for parallel programs. In *Proceedings of the International Euro-Par Conference*, pages 193–202, 1999.

[293] S.K. Reinhardt, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, and D.A. Wood. The wisconsin wind tunnel: Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[294] M. Reiser and S.S. Lavenberg. Mean value analysis of closed multichain queueing networks. *Journal of the ACM*, 27:313–322, April 1980.

[295] Chance Reschke, Thomas Sterling, Daniel Ridge, Daniel Saverese, Donald Becker, and Phillip Merkey. A design study of alternative network topologies for the Beowulf parallel workstation. In *Proceedings of the 5th International Symposium on High Performance Distributed Computing*, 1996.

[296] Ralf Reussner, Peter Sanders, and Jesper Larsson Träff. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Computing*, 10, 2001.

[297] Ralf H. Reussner, Peter Sanders, Lutz Prechelt, and Matthias Müller. SKaMPI: A detailed, accurate MPI benchmark. In *Proceedings of the 5th European PVM/MPI Users' Group Meeting (LNCS 1497)*, pages 52–59, September 1998.

[298] Daniel Ridge, Donald Becker, Phillip Merkey, and Tomas Sterling. Beowulf: Harnessing the power of parallelism in a Pile-of-PCs. In *Proceedings of IEEE Aerospace*, 1997.

[299] Graham D. Riley. Techniques for improving the performance of parallel computations. Master's thesis, University of Manchester, 1996.

[300] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods*. Springer-Verlag, New York, 2000.

[301] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, Englewood Cliffs, New Jersey, 1997.

[302] Rsim Research Group. Rsim home page. Available from `http://rsim.cs.uiuc.edu/rsim/`.

[303] R. Saavedra and A. Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical Report USC-CS-92-524, University of Southern California Los Angeles, Computer Science Division, September 1992.

[304] R. Saavedra and A. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, November 1996.

[305] S. Salza. Approximating response time distributions in closed queueing network models of computer performance. In *Proceedings of Performance*, pages 133–145, 1981.

[306] V. Sarkar. Determining average program execution times and their variance. In *Proceedings of the SIGPLAN Notices Conference on Programming Language Design and Implementation*, pages 298–312, 1989.

[307] Pasi Sarolahti and Alexey Kuznetsov. Congestion control in Linux TCP. In *Proceedings of the USENIX Annual Technical Conference*, pages 49–62, June 2002.

[308] Bryan Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, Trinity College, Oxford University, 1998.

[309] Christian Schaubschläger. Automatic testing of nondeterministic programs in message passing systems. Master's thesis, Johannes Kepler University Linz, Department for Computer Graphics and Parallel Processing, 2000.

[310] S. Schneider. An operational semantics for timed CSP. *Information and Computation*, 116(2):193–213, February 1995.

[311] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach.* Wiley, New York, 2000.

[312] J.M. Schopf. *Performance Prediction and Scheduling for Parallel Applications on Multi-user Clusters.* PhD thesis, University of California San Diego, Department of Computer Science, December 1998.

[313] J.M. Schopf and F. Berman. Performance prediction in production environments. In *Proceedings of the 12th IEEE International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, March 1998.

[314] H.T. Schwartz. Ultracomputers. *ACM Transactions on Programming Language Systems*, 2(4):484–521, 1980.

[315] A.C. Shaw. Deterministic timing schema for parallel programs. In *Proceedings of the 5th IEEE International Parallel Processing Symposium*, pages 56–63, April 1991.

[316] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, 1992.

[317] Jaswinder Pal Singh, Edward Rothberg, and Anoop Gupta. Modeling communication in parallel algorithms: A fruitful interaction between theory and systems? In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 189–199, 1993.

[318] Steve Sistare, Rolf vande Vaart, and Eugene Log. Optimization of MPI collectives on clusters of large-scale SMP's. In *Proceedings of Supercomputing*, 1999.

[319] David Skillicorn. *Foundations of Parallel Programming*, chapter 8, pages 123–169. Cambridge University Press, Cambridge, 1994.

[320] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2), June 1998.

[321] D.B. Skillicorn. A taxonomy for computer architectures. *IEEE Computer*, 21(11):46–57, November 1988.

[322] D.B. Skillicorn. Parallelism and the Bird-Meertens Formalism. Technical report, Queen's University, Computing and Information Science, April 1992.

[323] D.B. Skillicorn. Questions and answers about categorical data types. Technical report, Queen's University, Computing and Information Science, May 1994.

[324] D.B. Skillicorn. Abstract machine models for parallel and distributed computing. In M. Kara, J.R. Davy, D. Goodeve, and J. Nash, editors, *Communication Skeletons.* IOS Press, Amsterdam, 1996.

[325] D.B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28(1):65–83, July 1995.

[326] D.B. Skillicorn, Jonathon M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 199.

[327] J.E. Smith and W.R. Taylor. Accurate modeling of interconnection networks in vector supercomputers. In *Proceedings of the 5th ACM International Conference on Supercomputing*, pages 264–272, 1991.

[328] Quinn Snell, Glenn Judd, and Mark Clement. Load balancing in a heterogeneous supercomputing environment. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 951–957, 1998.

[329] Quinn Snell, Armin Mikler, and John Gustafson. Netpipe: A network protocol independent performace evaluator. In *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

[330] L. Snyder. Type architectures, shared memory, and the corollary of modest potential. *Annual Review of Computer Science*, pages 289–317, 1986.

[331] F. Sötz. A method for performance prediction of parallel programs. In *Proceedings of CON-PAR 90-VAPP IV (LNCS 457)*, pages 98–107, 1990.

[332] Standard Performance Evaluation Corporation. The SPEC benchmark suite. *SPEC Newsletter*, 2(1), 1990.

[333] M. Steed and M. Clement. Performance prediction of PVM programs. In *Proceedings of the 10th IEEE International Parallel Processing Symposium*, pages 803–807, April 1996.

[334] Thomas Sterling, Donald J. Becker, Michael R. Berry, Daniel Savarese, and Chance Reschke. Achieving a balanced low-cost architecture for mass storage management through multiple fast Ethernet channels on the Beowulf parallel workstation. In *Proceedings of the 10th IEEE International Parallel Processing Symposium*, 1996.

[335] Thomas Sterling, Donald J. Becker, and Daniel Savarese. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the International Conference on Parallel Processing*, 1995.

[336] Thomas Sterling, Donald J. Becker, Daniel Savarese, Bruce Fryxell, and Kevin Olson. Communication overhead for space science applications on the Beowulf parallel workstation. In *Proceedings of the 4th International Symposium on High Performance Distributed Computing*, 1995.

[337] W. Richard Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley, Reading, Massachusetts, 1994.

[338] W. Richard Stevens (ed.). TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Technical Report RFC 2001, National Optical Astronomy Observatory, USA, January 1997.

[339] B. Stramm and F. Berman. Predicting the performance of large programs on scalable multicomputers. In *Proceedings of the Scalable High Performance Computing Conference*, pages 22–29, April 1992.

[340] Theodore B. Tabe, Janis Hardwick, and Quentin F. Stout. Statistical analysis of communication time on the IBM SP2. *Computing Science and Statistics*, 27:347–351, 1995.

[341] Anthony Tam Tat Chun. *Performance Studies of High-Speed Communication on Commodity Cluster.* PhD thesis, University of Hong Kong, December 2001.

[342] Anthony Tam Tat Chun and Cho-Li Wang. Realistic communication model for parallel computing on cluster. In *Proceedings of the International Workshop on Cluster Computing*, pages 92–101, 1999.

[343] Wi Bing Tan and Peter Strazdins. The analysis and optimization of collective communications on a Beowulf cluster. In *Proceedings of the International Conference on Parallel and Distributed Systems*, December 2002.

[344] Athar B. Tayyab and Jon G. Kuhl. Stochastic performance models of parallel task systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 284–285, 1994.

[345] The LAM Team. LAM MPI 6.1. Available from `http://www.lam-mpi.org/`.

[346] D.R Thoman, L.J. Bain, and C.E. Antle. Inferences on the parameters of the Weibull distribution. *Technometrics*, 11(5):445–460, 1969.

[347] P. de la Torre and C.P. Kruskal. Submachine locality in the bulk synchronous setting. In *Proceedings of the 2nd International Euro-Par Conference (LNCS 1124)*, 1996.

[348] D. Towsley, G. Rommel, and A. Stankovic. Analysis of fork-join program response times. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):286–303, July 1990.

[349] T. Tsuei and M.K. Vernon. Diagnosing parallel program speedup limitations using resource contention models. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 185–189, 1990.

[350] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society (series 2)*, 42:230–265, 1936-7.

[351] Keith Underwood, Ron R. Sass, and Walter B. Lignon III. Acceleration of a 2D-FFT on an adaptable computing cluster. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.

[352] United States Department of Defense and AdaTEC and SIGPLAN Technical Committee on Ada. Reference manual for the Ada programming language, 1982.

[353] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Proceedings of Supercomputing*, pages 46–57, 2000.

[354] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[355] A.J.C. van Gemund. *Performance Modeling of Parallel Systems.* PhD thesis, Delft University of Technology, Information Technology and Systems, April 1996.

[356] Kees van Reeuwijk, Arjan J.C. van Gemund, and Henk J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, 1997.

[357] F.A. Vaughan, D.A. Grove, and P.D. Coddington. Network performance issues in two high performance cluster computers. In *Proceedings of the Australian Computer Science Conference*, February 2003.

[358] B. Veltman, B.J. Lageweg, and J.K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16:173–182, 1990.

[359] Virtutech. Simics home page. Available from `http://www.simics.com/`.

[360] D.F. Vrsalovic, D.P. Siewiorek, Z.Z. Segall, and E.F. Gehringer. Performance prediction and calibration for a class of multiprocessors. *IEEE Transactions on Computers*, 37(11):1353–1365, November 1988.

[361] H. Wabnig and G. Haring. PAPS - the parallel program performance prediction toolset. In *Proceedings of Computer Performance Evaluation: Modelling Techniques and Tools (LNCS 794)*, May 1994.

[362] Michael S. Warren, Donald J. Becker, M. Patrick Goda, John K. Salmon, and Thomas Sterling. Parallel supercomputing with commodity components. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1372–1381, 1997.

[363] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18:528–539, September 1975.

[364] Wallodi Weibull. A statistical distribution function of wide applicability. *Journal of Applied Mechanics*, 18:293–297, 1951.

[365] Waloddi Weibull. A statistical representation of fatigue failure in solids. *Transactions on the Royal Institute of Technology*, 27, 1949.

[366] P.J. Weinberger. Cheap dynamic instruction counting. *Bell Systems Technical Journal*, 63:1815–1826, October 1984.

[367] Yuhon Wen and Geoffrey C. Fox. A performance estimator for parallel hierarchical memory systems - PetaSIM. In *Proceedings of Parallel and Distributed Systems*, pages 205–210, August 1999.

[368] R. Clint Whaley. Installing and testing the BLACSv1.1, May 1997.

[369] Shirley A. Williams. *Programming Models for Parallel Systems.* Wiley, New York, 1990.

[370] Joel Williamson. Convex Computer Corporation.

[371] W. Willinger, M.S. Taqqu, W.E. Leland, and D.V. Wilson. Self-similarity in high-speed packet traffic: Analysis and modelling of Ethernet traffic measurements. *Statistical Science*, 10:67–85, 1995.

[372] W. Willinger, M.S. Taqqu, R. Sherman, and D.V. Wilson. Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5:71–85, 1997.

[373] Frederick Wong, Richard Martin, Rmezi Arpaci-Dusseau, and David Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Proceedings of the Conference on High Performance Networking and Computing*, November 1999.

[374] J. Worlton. Towards a taxonomy of performance metrics. *Parallel Computing*, 17:1073–1092, 1991.

[375] Y. Yang, X. Zhang, and Y. Song. An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous NOW. *Journal of Parallel and Distributed Computing*, 38(1):63–80, October 1996.

[376] N. Yazici-Pekergin and J.M. Vincent. Stochastic bounds on execution times of parallel programs. *IEEE Transactions on Software Engineering*, 17:1005–1012, October 1991.

[377] H. Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[378] Albertus P. Zwart. *Queueing Systems with Heavy Tails.* PhD thesis, Eindhoven University of Technology, September 2001.