University of Cape Town
Department of Computer Science

An Efficient Parallelization of a Real Scientific Application

by
Elizabeth Post

A thesis
prepared under the supervision of
Assoc. Prof. H.A. Goosen
in fulfilment of the requirements for the
degree of Master of Science in Computer Science

Cape Town
February, 1995

University of Cape Town
Department of Computer Science

An Efficient Parallelization of a Real Scientific Application

by
Elizabeth Post

A thesis
prepared under the supervision of
Assoc. Prof. H.A. Goosen
in fulfilment of the requirements for the
degree of Master of Science in Computer Science

Cape Town
February, 1995

Acknowledgements

I would like to thank:

my supervisor, Associate Professor Henk Goosen, for his constructive guidance and insight into the problems of parallel processing, and in particular for his thorough and invaluable guidance in the art of technical writing,
my colleagues, especially Philip Machanick, for their valuable assistance, suggestions, and cooperation,
the systems administrator, Sandi Donno, for her ever-willing support and helpfulness,
Dr. Tom Ackerman of Pennsylvania State University, USA, and his assistants Steve Nagle and Eugene Clothiaux, for providing the original serial application, and for their willing assistance in clarifying the meteorology,
Dr. Bruce Hewitson of the Department of Environmental and Geographical Science at the University of Cape Town for introducing me to Dr. Ackerman and for his assistance and the use of his Sun workstation,
The University of Cape Town and the FRD, who supported my studies financially,
my husband for his assistance and support throughout this project,
and, last but not least, my long-suffering children for putting up with "Mommy's Masters" for so long!

Abstract

In the past decade the cost of computing has come down considerably making high-powered computing more easily affordable. As a result many institutions and organisations now have networks of high-powered workstations. Such networks provide a large, generally untapped, source of computing power which can be used for running large scientific applications which previously could only be run on supercomputers.

This dissertation shows that a substantial improvement in performance can be achieved by the parallelization of a real scientific application for a heterogeneous network of Sun and Silicon Graphics workstations connected by an Ethernet network, but that this is affected by a number of factors. These factors include communication delays, load balancing, and the number of slaves used. This dissertation shows that performance can be improved by sending more, shorter messages, and by overlapping communication with computation.

Part of this thesis concerns the difficulties involved in the evaluation of parallel performance on a heterogeneous network. This dissertation shows that conventional methods such as speedup and efficiency are not appropriate for evaluating the performance of a heterogeneous system, and that linear speed gives a much more representative indication of the actual performance achieved.

We also proposed new concepts of **perfect linear speed** and **linear efficiency**, which help to evaluate the improvement in parallel performance on a heterogeneous system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel processing is becoming increasingly important as the practical limits of serial processing are reached, and more practicable as increasingly powerful processors become cheaper, and high-speed networks such as FDDI and ATM are more generally available. In the past decade, the development of portable and efficient parallel libraries has greatly contributed to the parallelization of more real applications.

These advances are particularly important in such scientific fields as climatology, meteorology and aerospace research, where the problems often have memory or computation requirements which are too large for the program to be run serially in a reasonable time. Previously such large programs could only be run on supercomputers, which were not generally available. Nowadays many institutions already have networks of workstations, and since efficient and portable parallel libraries have been developed it is becoming easier to parallelize applications for such networks.

Networks of powerful workstations represent an important resource for the potential delivery of a supercomputer level of performance. A significant obstacle to realising this performance is that not much is known about the performance of real applications on a network of workstations, and particularly for networks of heterogeneous workstations.

The typical network in most institutions is heterogeneous, as most networks consist of workstations of a number of different makes and models, so not all machines will have the same CPU, or the same amount of memory or cache. In addition, many such networks are multi-user networks, and dynamically varying workloads will cause even machines which are identical to have different performance capabilities.

If the computing power available in such networks is to be successfully exploited for parallel processing, then it is essential to acquire a better understanding of how real parallel applications perform on a heterogeneous network. In the past researchers have often concentrated on the performance of scaled down versions, or kernels of real applications. Now that many of the basic problems have been solved researchers have realised that it is important to study the performance of complete, full-size applications. This is particularly important in a heterogeneous environment, where the disparate performance capabilities of the workstations make it difficult to achieve efficient synchronisation and load balancing. These difficulties are compounded in a shared environment where the dynamically varying workloads on the processors constantly affect the performance capability of the workstations.

In the past it has also been difficult to use machines of different architectures together for parallel processing. However, with the development of modern portable parallel libraries it is now possible to run a parallel application using a heterogeneous group of machines.

This dissertation shows that a significant improvement in performance can be achieved by parallelizing a real scientific application for a distributed group of heterogeneous workstations, connected by Ethernet. Several crucial factors that affect parallel performance of a real application are identified. This dissertation shows that:

- the performance of an application on a network of workstations is sensitive to over-utilization of the network, and that a heavily loaded network will cause a severe degradation in performance;
- that these effects can be alleviated by breaking long messages into shorter messages, and by avoiding bursts of network traffic interspersed with periods of low usage, by ensuring that all machines do not try and send at the same time;
- network latency can be reduced by overlapping communication with computation;
- in a widely heterogeneous system little or no advantage may be gained by using slow processors together with very much faster processors, and adding slow processors may even cause a reduction in performance.

The measurements shown in this dissertation confirm the misleading nature of commonly used means of measuring performance, such as speedup and efficiency, and show that linear speed, as proposed by Crowl [Crowl94], more closely reflects the actual performance achieved, especially for heterogeneous networks.

We also propose some extensions to Crowl's work which will assist in understanding the performance of heterogeneous systems. These are the concepts of "perfect linear speed", and "linear efficiency", which will be described in section 5.4.4.

The work done for this thesis in studying the factors affecting the performance of a real parallel application will contribute to both the use, and the development of, parallel programming environments. Such parallel programming environments are currently under development as part of the move to make parallel programming more practicable for the real user (see section 1.4).

The application studied in this thesis, Cloud, is a cloud radiation model which was obtained from, and parallelized on behalf of, the Department of Meteorology at Pennsylvania State University (PennState) in the USA. The passage of photons through a stratocumulus cloud deck is simulated, using the Monte Carlo method, which is a computation-intensive technique used widely in scientific computing. The transmissivity, reflectivity and absorptivity of the cloud are then calculated from the results obtained. A more detailed description of Cloud is given in Appendix A.

The *p4* parallel library, from Argonne National Laboratories[1], was used to parallelize the application, to run on both homogeneous and heterogeneous groups of Sun and Silicon Graphics workstations, connected by an Ethernet network. Since all workstations used were uniprocessors, the message-passing paradigm was used.

The remainder of this chapter looks at some of the factors that are contributing to parallel programming becoming a practical reality for the real user.

1.1 Untapped computer resources

Existing networks of workstations can now be used as multicomputers, to solve large scientific problems, so that parallel processing can exist as a by-product of a normal, highly distributed workstation environment, without the need for specialised multicomputers or supercomputers [Bell94]. An additional benefit

[1] The *p4* parallel library is in the public domain and can be obtained via anonymous `ftp` from **info.mcs.anl.gov** at Argonne National Laboratory. The distribution contains all the source code, a meta-makefile to build *p4* on a number of different machines, a set of examples, and a User's Guide.

is that a user may have exclusive use of a network of workstations, which could result in a shorter elapsed time for a parallel run than the elapsed time on a supercomputer, where the resources have to be shared with other users [Minn93].

Nowadays many organisations and institutions can afford, and already have available, networks of uniprocessor workstations, which are usually under-utilized, and which represent a large, generally untapped, source of computing power. Researchers at the University of Zurich observe that the typical workstation in a LAN lies idle for long periods [Cap93]. Statistics of their LAN demonstrate an average idle percentage of at least 90%. They also postulate that, for a large percentage of their lifetime, these high performance machines are used merely for small tasks such as editing files, and reading email.

Similarly, researchers at the Lawrence Livermore National Laboratory observed that users spend about three times as much time on workstations as on supercomputers, and that these workstations are only about 15% utilized [Bell94]. And for the 15% of time when these workstations are being utilized, they deliver about five times the power of the supercomputers at the Laboratories! In 1993 workstations and PCs comprised 9% and 45% respectively of computer expenditures in the USA, thus providing a great untapped resource for parallel processing. Bell claims that, as LAN-based workstations evolve to be connected by high-speed networks, they will have the capability of modern multicomputers [Bell94].

An advantage of a distributed system is that each machine has its own memory, and usually a cache, and this adds up to a considerable amount of memory, which is usually more than the memory and cache available for a serial processor. The available memory for each machine is reduced by such factors as having the operating system, and separate copies of the executable and global variables, in the memory of each processor, but often more data can be in memory, or cache, at any one time on a distributed system, than on a serial processor. This reduces the paging necessary, and can contribute towards superlinear speedup being achieved.

1.2 The development of standard parallel libraries

The development of portable and efficient parallel libraries has contributed to making parallel processing possible for the real user. Previously heterogeneous machines could not easily be used together for parallel processing. Now, with modern portable parallel libraries, such as *p4*, the computing power available in a heterogeneous network can be more easily exploited.

Suitable parallel processing hardware has been commercially available for approximately a decade, yet relatively few real applications have been successfully parallelized. This was principally due to the lack of parallel languages, libraries and compilers, the lack of skills and the difficulties of implementing parallelism by low-level, hardware-dependent, programming. The difficulties of parallel programming were also immense, as each different platform had its own way of implementing such things as communication, so that a programmer had to relearn the techniques for every different environment. Few real users were willing to invest a large amount of time and money in parallelizing a serial application, when it might have to be redone two or three years later for a different hardware platform. There were few people with the necessary skills available commercially, as most of those who knew how to do parallel programming were researchers. Many users were also disillusioned because few parallel environments lived up to the performance claims made by vendors. Users found that parallel programming was just too hard, and too expensive, and the performance gains were just not good enough. Kuck claims that, even now, users have not been presented with usable practical parallel processing facilities, and that this matter should be addressed urgently [Kuck94].

In the past decade, a number of these problems have been solved with the development of portable libraries such as *p4* [Butl94], *PVM* [Sund94], *SpaceLib* [Mach92], *Linda* [Carr89][Carr94] and *PARMACS* [Calk94]. For a survey of some of these, and others, see [McBr94]. These libraries generally consist of extensions to C or Fortran, and they have made it possible for relatively unskilled programmers to parallelize real applications easily, and efficiently, in such a way that they are easily portable to other hardware platforms, or can be run in heterogeneous environments. This effort will be considerably enhanced by the adoption of the new *MPI* standard [Walk94], which is based on those features of the most common libraries such as *p4*, *PARMACS* and *PVM*, which have been found in practice to be the best way of implementing message-passing parallelism. This standardisation was not possible until sufficient real applications had been parallelized, so that researchers could see what was required, and could start to develop suitable libraries. Now that more experiments have been conducted, the best methods are emerging as standards.

1.3 Performance of real parallel applications

Good performance for parallel programs on a network of workstations will only be achieved if the factors affecting the performance of applications on such a network are properly understood. And these factors can only really be identified by studying the performance of complete, real applications, such as Cloud.

Mainly as a result of increased standardisation, and the development of portable parallel libraries, there has been increasing emphasis in research on the performance of real parallel applications in the past decade. In serial computing, the speed of the processor is of prime importance, but in parallel computing factors such as memory requirements and communication costs also have a significant impact on performance, particularly in a distributed system.

Initially, parallel processing research has focused on improving the performance of parallel architectures, by such methods as reducing memory and communication latency, in experiments with small test programs or kernels. However, the good performance of a small test program does not necessarily mean that comparably good performance can be achieved when the program is run with a larger data set, requiring considerably more memory, and perhaps involving extensive input and output, The results of such research can therefore be misleading regarding the performance of real applications, and frequently performance claims for certain architectures cannot be achieved with real programs, which behave differently from the kernels used in the performance tests.

For users it is important that performance claims relate to complete real applications, and not just kernels. As a result researchers are now establishing suites of benchmark programs, which contain real applications with a broad range of different characteristics, so that the performance of parallel systems can be assessed more accurately. These benchmark programs are also portable between platforms, so that different parallel environments can now be compared (SPLASH [Sing92], NAS benchmarks [Bail93], Perfect benchmarks [Cybe90], SPEC benchmarks [Dixi91], Genesis benchmarks [Hey91], and a survey of parallel benchmarks in [Weic91]).

The performance of a parallel program can be affected by the parallel algorithm used. If a program is merely a parallelized version of a serial program, this may not necessarily be the most efficient parallel program. A totally new algorithm may be a better and more efficient way of solving a problem in parallel. Such a parallel algorithm may, for example, take advantage of the extra caches and memory available in a distributed system, by handling the data differently, or by using the different capabilities of heterogeneous processors to handle certain tasks for which they are particularly suited, such as for the rendering of graphics images.

There is still much room for research in the area of developing parallel algorithms which take the most advantage of a parallel environment, as opposed to parallelized serial algorithms.

1.4 Parallel problem solving environments

For parallel processing to become a successful alternative to supercomputers it must become easier to implement parallel programs. A major contribution in this area is current research on the development of parallel problem solving environments. These environments will assist a programmer to develop an efficient parallel program quickly and easily. The results of this study on the performance of Cloud will assist in both the development and the use of such parallel programming environments.

Parallel processing is now at the stage where the major architectural problems have been solved, and some parallel languages, libraries and compilers have been developed. Because the costs of parallel programming are so high, it is essential that parallel programming must become simpler and faster. Generally, an automatic parallelizing compiler can do low-level parallelism and communication very efficiently. However, it is extremely difficult for parallelizing compilers to recognise high-level parallelism, and dependencies in the application being parallelized. This can be done much better by a programmer who has some familiarity with the application to be parallelized, and an understanding of parallel processing.

The necessity to make parallel programming simpler and faster has led to the development of dynamic parallel programming environments, which combine the strengths of a programmer in identifying high-level parallelism, together with the strengths of an automatic compiler for implementing efficient low-level parallelism [Kuck94]. These environments assist a programmer in developing applications for both shared-memory and distributed systems. Some of the environments that are currently under development are Enterprise, Mentat, Presto and Schooner. Enterprise produces code for a network of workstations [Scha93]. Mentat enables the parallelization of data-parallel applications for a wide variety of MIMD platforms, from loosely coupled networks of workstations to tightly coupled multicomputers [Grim93a] [Grim93b][Grim94]. Presto is intended for developing parallel applications for shared-memory multiprocessors [Bers88]. Schooner is specifically intended to exploit the varied resources available in a heterogeneous system [Home94]. These environments, particularly Enterprise, allow the user to specify high-level parallelism but automatically implement the low-level parallelism efficiently.

It will still be some time before such environments are generally commercially available, as there is still much research to be done on the behaviour of real applications, so that the performance characteristics of such programs can be properly understood. As more users are becoming involved with parallel processing, and more real applications, such as Cloud, are parallelized, it will assist researchers in the development of such environments. Also programmers will gain expertise which will enable them to parallelize applications efficiently by using such an environment.

1.5 Structure of this dissertation

The next chapter reports on other work in this area. This is followed by a description of the parallelization of the application. The subsequent chapter is an account of the experiments conducted in studying the performance of this parallel application. Following this, there is a summary of results obtained, together with a discussion of their relevance. This chapter also includes discussion on the problems of evaluating the parallel performance of a heterogeneous system. Finally, conclusions are drawn concerning the factors affecting the performance of this parallel application, and some recommendations are made on how to achieve the best performance for a parallel application, in a similar environment.

Chapter 2

Related Work

There is increasing research in the programming and performance of real applications, as researchers attempt to understand the behaviour of programs in actual use. This chapter describes some of the work in this area.

Cloud has many similarities to the research described in this chapter. These include making use of existing underutilized networks of workstations, parallelizing a common scientific application by using a freely available parallel library, running the program on both homogeneous and heterogeneous machines, heterogeneous partitioning of work, and dynamic load balancing.

The experiments using Cloud were run on both homogeneous and heterogeneous groups of Silicon Graphics Indigo, Sun ELC[1], Sun SPARCclassic and SUN SPARCstation 1+ workstations, all connected by Ethernet. Some of the projects described in this chapter were run on similar groups of Sun workstations, also connected by Ethernet. Other projects were run with groups IBM RISC System/6000 workstations, some connected by Ethernet, and others with a Token Ring, or fiber-optic implementation. The remainder of the work described in this chapter was conducted on different types of parallel systems, such as transputer networks, but these projects were similar to Cloud in the programming techniques used.

Section 2.1 describes a number of other implementations of the Monte Carlo technique. After that, some work on the more efficient utilization of networks of heterogeneous workstations is described in the next section. Section 2.3 contains a summary of a number of real scientific applications which have been parallelized using the *PVM* library. After that there is a short description of the implementation of a data-parallel application for a network

[1]SPARCstation, SPARCstation 1+, SPARCstation 10, SPARCclassic and SPARCstation ELC are trademarks of Sun Microsystems, Inc.

of heterogeneous workstations, together with some discussion of the load balancing techniques used. The last section lists some other projects where real applications have been parallelized in the fields of meteorology and climatology.

2.1 Other parallel projects using Monte Carlo simulation

Monte Carlo simulation is a common scientific technique which has been used in Cloud, and in many other applications. Frequently this type of simulation is the only way to solve scientific problems which would either take too long to be solved, or cannot be solved at all. Typically, Monte Carlo simulations have minimal memory requirements, require almost no communication, and are very time-consuming. This section describes some other applications which use Monte Carlo techniques.

2.1.1 Monte Carlo benchmarks

In many ways the minimal communication requirements of Monte Carlo simulation make it an ideal way to estimate the upper limits of a system's floating point performance. This is evident by the fact that Monte Carlo routines are part of at least six sets of parallel benchmarks: the QCD program in the Perfect Benchmarks [Cybe90], the Embarrassingly Parallel program of the NAS Parallel Benchmarks [Bail92][Bail93], the DODUC program in the SPEC benchmarks [Dixi91][Weic91], kernel 16 in the Livermore Fortran kernels [Berr91], INTMC, GAMTEB, VGAM and SCALGAM in the Los Alamos benchmarks [Berr91] and QCD1 in the Genesis distributed memory benchmarks [Hey91].

The Embarrassingly Parallel program is in some ways very similar to Cloud, and typical of many other Monte Carlo applications, in that two-dimensional statistics are calculated from a large number of pseudorandom numbers [Bail93].

2.1.2 Efficiency evaluation of some parallelization tools on a workstation cluster using the NAS parallel benchmarks [Suku94]

At Vienna University of Technology, Sukup has conducted an efficiency evaluation of *PVM 2*, *PVM 3*, *p4*, *Express* and *Linda* by implementing and running four of the NAS Parallel Benchmarks [Suku94]. The NAS Parallel benchmarks used were the Embarrassingly Parallel benchmark (Monte Carlo simulation), the Simple 3D Multigrid Benchmark, Solving an Unstructured Sparse Linear System by the Conjugate Gradient Method and the Parallel Sort Over Small Integers. The tests were conducted on a cluster of nine IBM RS6000-320H workstations, each with a 25 MHz clock rate and 16 Mb memory per workstation.

These benchmarks were implemented using each of the five libraries, and the results were ranked according to the performance obtained with each library. For the Embarrassingly Parallel benchmark, all libraries gave almost exactly the same performance results. However, there was considerable variation in which library gave the best performance for the other benchmarks, but the results for most libraries were reasonably similar. Finally the libraries were ranked according to their overall performance with each benchmark, and also other factors, including the ease of learning and programming with that tool, the costs of startup and closedown time, the configuration of the tool, and the ease of debugging. The final results ranked *Express* first, followed by *PVM 2.x*, *Linda*, *p4* and *PVM 3.x* in that order. This was in contrast to the results of Cap and Strumpen (see section 2.2), where *Linda* gave the worst results. This dissertation shows in Chapter 5 how there are many factors which affect parallel performance. Since the NAS benchmarks are "paper and pencil" benchmarks, and their implementation differs for different experiments, Sukup's performance results may not necessarily be conclusive. If these factors which affect performance are considered, it may be that re-implementations of Sukup's version of the benchmarks may give different results.

In the network performance results, described in section 5.2.2, the data rates that were achieved for a small *p4* program, that was used to test the performance of the Ethernet network, were similar to the data rates achieved for *PVM*, by Sunderam et al in [Sund94].

Sukup reported a number of problems with *p4*, which included lost or hanging processes. This was consistent with the experiments with Cloud, where similar problems occurred. However, Sukup was favourably impressed with the debugging tools in *p4*, which were also useful in the development of Cloud.

2.1.3 Monte Carlo simulation in radiation dosimetry [Ma93]

Chang-ming Ma implemented a parallel version of the EGS4 (Electron Gamma Shower version 4) Monte Carlo code system on the Edinburgh Concurrent Supercomputer (ECS), which is a multiple-transputer system consisting of 423 T800 20-MHz transputers. This program was used to calculate the absorbed dose of radiation by using Monte Carlo simulation. The program, DOSIMETER, is a task-farm[1] procedure written in Meiko Fortran, and which uses CSTools for the underlying communication. The interface routines were based on those provided in the Meiko Task Farm and the Wheatfarm.

The program was similar to Cloud as it consisted of a control process to generate tasks, a number of simulation processes to consume tasks and generate results, and an analysis process to collect and analyze the results. Currently the number of particle histories traced by Ma is 10^5, if the incident particles are electrons and 10^6, if the incident particles are photons, and these numbers are similar to the number of photon histories ($1.2x10^6$) traced in the experiments with Cloud.

Ma found that, in the case of the ECS transputer domain, communication costs become negligible when running large simulation tasks. Similarly, the initialization time, although it increases with the domain size, becomes negligible when compared to the computing time for large calculations.

However, the optimal speed on the T800 transputers can only be achieved when using the transputer's native language, Occam. When running a program written in Fortran, less than half the stated performance could be achieved. Ma's results showed that a linear increase in computer speed was

[1]A **task-farm** is a technique for implementing self-scheduling calculations. In a task farm, a "source" process generates a pool of jobs, while a "sink" process consumes results. In between, one or more "worker" processes repeatedly claim jobs from the source, turn them into results, despatch those results to the sink, and claim their next jobs. If the number of jobs is much greater than the number of workers, task farming can be an effective way to load balance a computation [Wils93].

achieved, as the number of transputers used on the ECS was increased. Ma also includes results obtained from running the code on a number of other machines including 5 different VAXs, an HX i860, a Division i860, a Motorola 88000 and an HP 9000/720. [Ma93]

2.1.4 Monte Carlo simulation of radiative heat transfer on a SPARCStation farm [Minn93]

Minnich and Pryor, at the Supercomputing Research Center in the USA, are developing a distributed shared-memory model, Mether. To test the performance of their model, they implemented a parallelized Monte Carlo simulation of radiative heat transfer, to run on a computing farm consisting of 16 SPARCStation ELCs (33MHz SPARC 1 processor) connected by an Ethernet network, which is fairly similar hardware to that used for most of the work with Cloud. The application simulates the radiative heat transfer among surfaces of arbitrary 2-D enclosures, and is used for the modelling of the geometry of a laser-isotope separation (LIS) unit, for which the accurate determination of radiant exchange factors is an important component in the larger simulation of the isotope separation process.

The application involves the tracing of photons which are emitted from the surfaces of the enclosure, reflect from one or more intermediate surfaces, and are then absorbed into, or transmitted through, terminating surfaces. The LIS has 37 sides, and 1 million photons are emitted uniformly from each side. The path of each photon may include several reflections, which may be specular or diffuse, before it is transmitted through, or absorbed by a surface. The counts of the photons transmitted and absorbed are recorded in two 2-dimensional matrices. This problem is very similar to Cloud in the amount of computation, and the total elapsed time on the farm.

The same C code was run on a 16-processor ELC farm, and on one processor of a Cray 2. The total CPU time on the Cray was 262 min (10h 20m elapsed time), and for the ELC-farm the elapsed time was 28 minutes. This time of 28 minutes for the ELC-farm is similar to the time needed for Cloud on 11 Suns. The long time for the Cray is primarily because there were other users on the Cray. However, the program was run on only one processor on the Cray and the code did not automatically vectorize easily. The researchers claim that if they optimized the program to take full advantage of the Cray, the same task would run on the Cray in approximately 10-13 minutes. Similarly, if certain optimizations were made for the program running on the ELC-farm, the time could be improved, but would probably still be in the order of 20-30 minutes.

Nevertheless, it is claimed that for this example, a radiative heat-transfer simulation, it was possible to achieve supercomputer-level performance on a network of cheap workstations. For this problem, a 3-processor Cray 2 would run about 3 times as fast as a 16-station farm, and a 16-processor C90 Cray supercomputer would run approximately 40-50 times as fast as the farm. However, the cost of a C90 supercomputer is about 500 times the cost of the farm, so that performance/price is correspondingly higher [Minn93].

Results also showed that the startup overhead was relatively small for large amounts of computation, but for smaller runs this became increasingly significant. It was estimated that the farm would scale easily to about 32 processors, but for more than 32 processors another method of starting the processors should be found, as the startup costs would limit further scalability.

2.2 Efficient parallel computing in distributed workstation environments [Cap93]

This thesis is to illustrate how an existing network of heterogeneous workstations can be used for efficient parallel computing. This section describes similar work at the University of Zurich, where Cap and Strumpen are conducting research into more efficient utilization of existing networks of heterogeneous workstations [Cap93].

Research has shown that, for the workstations in the network at the University of Zurich, the average idle percentage is at least 90%, and that for much of their lifetime these high-power workstations are used for small tasks, such as editing files, and reading email. Although the research at Zurich is primarily concerned with data-parallel processing, the principles are discussed here, as many of them are valid for all distributed parallel processing.

Cap and Strumpen have developed THE PARFORM to use idle workstations for parallel computing. Even on dedicated parallel machines it can be difficult to obtain good speedup and efficiency. However, the heterogeneity, and dynamically varying load situation of a non-dedicated workstation network, considerably increase the difficulties of obtaining good parallel performance. In a parallel program with communicating processes one slow workstation can cause the whole program to proceed at the rate of the slowest processor, which may result in a decrease in efficiency. In a non-dedicated network where the workload of each processor is changing constantly, it is not possible to determine beforehand which processor will do the most work in a certain time. Even if the situation is known at the beginning of a run, it will probably change several times during the run.

THE PARFORM uses heterogeneous partitioning and dynamic load balancing to improve performance. It uses a number of sensor processes to determine the load situation of each processor, and whether the processor is busy running a time-consuming program, or it is currently idle, except for very short or interactive jobs. Another process determines approximately how much computation can be expected from the processor. THE PARFORM thus partitions the work, so that the fastest processors get the most work, and the slowest get the least, according to the comparative rates, and computation potential, of the processors, as determined by the sensor processes. These sensor processes run throughout the time the parallel program is running, so as the load situation and computation potential of the workstations changes, THE PARFORM dynamically adjusts the load for each processor, to obtain the greatest efficiency possible.

The increase in efficiency achieved by heterogeneous partitioning and dynamic load balancing more than compensated for the overhead of running the sensor processes. THE PARFORM is in many ways similar to *PVM*, but heterogeneous partitioning and dynamic load-balancing are not supported by *PVM* so this has to be done entirely by the programmer, which makes it difficult to achieve high efficiency for a network running under a normal daily load. *p4* also does not support these techniques implemented by THE PARFORM, and like *PVM* these will also have to be implemented by the programmer.

A parallel program may slow down the response times of interactive processes as a result of CPU sharing, and it is important to use a sensible scheduling mechanism, together with the other processes described above, to ensure that a parallel program does not interfere with interactive users, but uses idle machines instead. So far researchers at Zurich have found that, even when their parallel program was running at normal priority, there was minimal impact on other users, and frequently it was not noticed at all. However, they intend to introduce a mechanism whereby their parallel program runs at low priority to reduce the impact on other users.

The performance obtained by using THE PARFORM was evaluated by using the same program implemented with each of THE PARFORM, *Linda*

(*POSBYL*), *SCALinda* and *PVM*. This program was run on a heterogeneous network of 40 SPARCstations and SPARCservers, of five different types. In all cases the application was a 2-D grid-based iterative solver for heat conduction, and the network was dedicated. The performance was also compared to the same program running on a tightly-coupled Transputer Multicluster MC-2/32-2. Results showed that the speedup was essentially constant for all versions, except for the *Linda* (*POSBYL*) implementation which performed badly. This can be explained by the overhead of tuple space management, as *Linda* is more suited to shared-memory processing. However, in overall speed THE PARFORM was marginally better than the *SCALinda* and *PVM* versions, and a factor of approximately 3 times as fast as the Transputer Multicluster, which was primarily due to the lower performance of the T800 processors. In these experiments, for this application at Zurich, the workstation network scaled almost exactly like the tightly-coupled Transputer Multicluster. Cap and Strumpen also observed that, for a parallel version on a network, there is more memory available than for a serial version, and this can result in super-linear speedup for a program requiring a large amount of memory, as the serial version will require excessive paging [Cap93].

Excellent, near-linear speedup was obtained for up to 20 workstations, but there was a breakdown in performance for more than 30 processors. Cap et al found that network saturation and congestion were serious inhibiting factors which limited scalability. Congestion occurred when too many stations attempted to transmit at the same time, and saturation happened when the work was split into so many tasks that the network became saturated with communication. Similar problems were experienced with Cloud. Cap et al suggest that the problem of saturation may be solved by using a different algorithm, as is shown in the experiments with Cloud, and that modern high-speed networks may partially solve the problem of congestion, although increases in the performance capabilities of workstations make this problem more prominent.

2.3 Parallelization of scientific applications using the *PVM* parallel library

PVM is a parallel library similar in many aspects to *p4*. There have been a number of projects using *PVM* to parallelize real scientific applications for a network of workstations. Some of these projects are described briefly below.

All four of the projects listed here run on networks of IBM RISC System/6000 workstations. The numbers of workstations used are similar to the numbers of machines used in the experiments with Cloud. Three of the projects described in this section were implemented on groups of homogeneous machines, but in one case the master was a different processor. In the fourth project the experiment was conducted on both homogeneous and heterogeneous groups of machines.

The results of these experiments indicate some of the problems that arise when parallelizing real applications for a network of workstations, particularly when the network is heterogeneous, and how these factors affect the optimal performance that can be achieved.

2.3.1 Distributed computation of wave propagation models using *PVM* [Ewin94]

Some of the developers of *PVM* used it to parallelize an application which simulates the propagation of seismic waves. The primary purpose of this project was to demonstrate that many organizations have considerable computing power available, in the form of existing networks of workstations, and that for no extra cost, by using a parallel library such as *PVM*, they can use this power for running parallel scientific applications, which are generally too large, or too slow, to run serially. [Ewin94]

The program was run on a network of six homogeneous processors connected by Ethernet. Reasonable speedups were obtained, but were limited by communication overheads. It was observed that the network could become a significant bottleneck, and that for many applications speedup will be less significant as processors are added, and network communication becomes saturated. Similar problems were also observed for Cloud, which may be limited in its scalability because of network congestion, and an overloaded master.

Another factor which may affect load balancing is that the *PVM* tests were performed on an isolated homogeneous network. Performance is expected to degrade in a heterogeneous environment, or a network with varying load conditions which cause additional load balancing problems.

2.3.2 Parallelization of the two-dimensional Ising Model on a cluster of IBM RISC System/6000 workstations [Alte93]

The Ising model is another commonly used scientific technique for statistically obtaining a solution, when an exact solution cannot be found easily, if at all. In such cases computer simulations become an important tool to confirm and understand experimental data. This method has much in common with the Monte Carlo method, and is also well-suited for parallelization, as it has only modest communication between processors. Here this technique is used in an application to derive statistically the thermodynamic properties of macroscopic bodies [Alte93].

For this example, an automatic parallelizer was not the best solution, since such parallelizers seem to be limited to fine-grain parallelizations, and these tend to have a high latency on workstation clusters. For this program, a new algorithm, which was more coarse-grained, and therefore had less communication, was thus implemented manually.

This application was run, at the IBM Research Center in Germany, on a platform of one RTS/6000-560 and four RS/6000-550 workstations, connected by both a Token Ring network, and with Serial Optical Channels. The speedups and efficiency achieved were satisfactory. However, it was noted that there was a "critical lattice size", and that when the lattice was below this size, the amount of communication was so much that performance was degraded, while for large lattice sizes the communication costs were negligible. This is because only the edges of the lattice are communicated to other processes, and as the lattice size increases, the edges form a smaller percentage of the data.

Again it was observed that static decomposition of the domain into a number of subdomains, with each subdomain to be associated with a particular processor, was only appropriate in a homogeneous, dedicated system. For such an application, that is limited by inter-process communication, dynamic load-balancing would be required on a heterogeneous, non-dedicated network.

2.3.3 Lattice Boltzmann method on a cluster of IBM System/6000 workstations [Bete93]

The lattice Boltzmann method is yet another scientific procedure in common use, particularly for the simulation of complex hydrodynamic phenomena. It was implemented, using *PVM*, by Betello et al [Bete93]. Since the application was to be run on a homogeneous network, *PVM* was optimized, to exploit the high speed available on a fiber channel, by removing some of the handshaking, and data conversions, that would be required for a heterogeneous system

This application was run on a homogeneous network of IBM RS/6000-550 workstations connected by a fiber-optical channel. A good speedup of 6.5 to 6.8 for 8 processors was obtained. The scalability of this application was analyzed to determine the critical number of processors beyond which adding another processor would not increase the speedup.

For this application it was determined that, for a grid of size 512, the application would scale with almost linear speedup to 18 processors. If the problem size was increased to a grid of size 2048 then the application would scale well to 35 processors, and with 20 processors the speedup would be about 15. These speedups compare well with those obtained for Cloud on similar numbers of machines.

2.3.4 Performance of IBM RISC System/6000 workstation clusters in a quantum chemical application [Nana93]

The *ab initio* determination of the electronic structure of molecules demands considerable computing power, especially when electron correlation effects are taken into account. The **concurrent computation Many-Body Perturbation Theory** (ccMBPT) is a method in which such a problem can be divided into a set of completely independent sub-problems, which can each be handled by a different processor. Using this theory and *PVM*, an application to solve the many-electron correlation problem, using a network of workstations, has been modified from the original program, which ran on an Cray Y-MP C90 [Nana93].

Nanyakkara et al first studied the performance of this quantum chemical application on the two homogeneous sub-groups, and then measured the performance of the combined heterogeneous group of IBM RISC

System/6000 workstations, connected by Ethernet. The two homogeneous groups were up to eight RS6000-320H workstations, and up to eight RS6000-340 workstations. The heterogeneous group was a combination of these two groups. The master workstation was an RS6000-550. This approach of first studying the performance of the homogeneous sub-groups, and then the performance of the combined heterogeneous group, is similar to the approach used in the experiments with Cloud.

It was observed that although the theoretical performance of Ethernet on a dedicated strand is 10 Mbits/second, in practice only about 5-7 Mbits/second could be achieved for a dedicated strand.

Results showed that although a significant decrease in the total elapsed time could be achieved for clusters of four homogeneous workstations (speedup greater than 3.1 and efficiency 0.78 to 0.87), comparable improvements were not achieved for clusters of six or eight homogeneous workstations, with speedup factors in the range of 4 to 4.5 and efficiency dropping to 0.5 to 0.56. With the larger group of up to 16 heterogeneous workstations there was only a small reduction in elapsed time for 6 (3 of each model) and 8 processors (4 of each model), and an increase in the elapsed time for 16 processors (8 of each model). The speedup for the heterogeneous group ranged from 3.86 to 4.55 for 8 processors (4 of each model), with corresponding efficiencies of 0.48 to 0.57, and from 2.82 to 4.02 for 16 processors (8 of each model), with corresponding efficiencies of 0.18 to 0.25.

This lack of improvement for larger number of processors is attributed mainly to data migration, and to the low bandwidth of Ethernet. Again, these values for speedup and efficiency are similar to those obtained for Cloud, with similar numbers of processors.

Thus, although groups of 300 to 400 RISC workstations could theoretically match the computing power of a Cray Y-MP C-90, in practice similar results could not be achieved for this application, by using the cluster of workstations. As with Cloud, the low bandwidth communication links are the major inhibiting factor preventing good performance with a cluster of workstations. However, restructuring of the algorithm to minimize data migration could reduce the communication overheads.

2.4 Data-parallel programming on a network of heterogeneous workstations [Nede93]

This project illustrates the use of a new parallel language, *DataParallel C*. Like Cloud, and like THE PARFORM, this system also performs dynamic load-balancing which adjusts to the varying load situation in a multi-user, heterogeneous environment. In *DataParallel C* the load balancing is achieved by the non-uniform redistribution of virtual processors between workstations, after monitoring the load situation on each workstation. The network used in these experiments was similar to that used in the experiments with Cloud.

Four typical scientific applications were run to test the performance of the system, using *DataParallel C*, on a network of heterogeneous SPARCstations connected by Ethernet. The performance achieved was reasonable, with near-linear speedup obtained for the computation-intensive application, and a lower speedup for those applications that required more communication.

The advantage of this system is that it is dynamic, and will automatically adjust to changing conditions without programmer intervention. This language is to be ported to run on a network of IBM RISC System/6000 workstations where even better performance is expected [Nede93].

2.5 Parallel programming in meteorology and climatology

This chapter is concluded by a look at some other parallel processing projects in the fields of meteorology and climatology. Most of the applications described in this section are quite different from Cloud, and run on completely different hardware configurations. However, the projects described in this section illustrate how parallel processing is being increasingly used in these fields.

Scientists in meteorology and climatology have long been hampered by the lack of computing power, and memory, to run their very large programs. For scientists without access to supercomputers, this lack of available computer power has meant that often problems have either been partitioned into smaller independent serial programs, or run as serial simulations with a coarse resolution. Such coarse simulations frequently give false results, since it is impossible to model every aspect accurately, and a small deviation may result in a large propagated error, such as in the Lorenz effect [Glei87]. With recent advances in parallel processing several independent, but related problems, may be run in parallel, to get a better overall picture [Kuck94]. Also, with increased computer power and memory, simulations may be run with a finer resolution to obtain more realistic results.

Now that parallel processing is becoming a practical reality meteorologists and climatologists have been among the first to make use of parallel processing for real applications. This section describes some of the projects that have been undertaken.

2.5.1 Weather prediction [Gärt93][Gärt93a]

The European Centre for Medium-range Weather Forecasts (ECMWF) has cooperated with scientists in Germany, in an effort to parallelize the ECMWF's weather forecast program by using the *PARMACS* library. *PARMACS* was chosen so that the program would be portable to a number of different environments. However, once the *MPI* standard has been defined, this weather forecast program will probably be re-implemented in *MPI*, in accordance with several other European projects. Previously, such parallelization was not feasible, as it was just too expensive to parallelize a large program such as this weather forecast program, when it would probably only have a lifetime of about 10 years, and it would not have been portable to other platforms. However, with the development of such portable libraries as *PARMACS*, *p4* and *MPI* such exercises are now possible in a realistic time, and for a reasonable cost.

In this project sections of the program were parallelized separately, and then integrated. This program uses a spectral transform technique, which involves a large number of 3-dimensional data structures, and a considerable amount of communication. Nevertheless, the spectral transform method is a considerable improvement on straight 3-dimensional iterative calculation. Initially, parallelization of the 2D-case has been completed [Gärt93], and work is now continuing on the parallelization of the 3D-case [Gärt93a].

The model was implemented on a number of different parallel systems, including Intel iPSC/860, Meiko i860 CS, CM5 and a network of IBM RS6000 workstations. Satisfactory performance was obtained in all cases, except on the workstations, where the interconnect network was a simple

Ethernet, and the communication capacities were too low. This poor performance with Ethernet is consistent with the other work described in this chapter, and with the results of the experiments with Cloud.

2.5.2 Climate modelling [Sela94]

The example in section 2.5.1 incorporates real weather data, and uses this data to forecast the weather. In climate modelling researchers study long-term climate change by simulating weather data. Programs such as general circulation models (GCMs) were among the first real scientific applications that were parallelized. GCMs have such large computing and memory requirements that, until recently, full-size GCMs have only been run on supercomputers such as Crays, and even then they run for months.

A description of the parallelization of the USA National Meteorological Center's spectral model for a Connection Machine (CM-200) is given in [Sela94]. Results showed that the time for the program to run on a 256-node CM-200 are similar to those of a Cray Y-MP/1. It was also shown that the program would scale efficiently from a 256-node machine up to a 2048-node machine.

The main problems with GCMs is that it may take months to get a solution. Consequently, to obtain a solution in a shorter time, GCMs will usually be run with a resolution which is too coarse to simulate climate well. For instance, in GCMs it is very difficult to model cloud cover well. Thunderstorms, which are the clouds with the greatest energy, and that affect the weather the most, cover a very small area of only a few square kilometres. These clouds cannot be modelled realistically on a GCM where one gridpoint is used to represent an area of hundreds of square kilometres. Thus, researchers are interested in studying such programs as cloud radiation models, like Cloud, to establish the best way to simulate clouds of different types in such programs as a GCM. As computing power increases, it will become feasible to combine such simulation programs to obtain a more realistic GCM. Monte Carlo simulation is one of the methods that researchers use to simulate the behaviour of weather, in preference to more time and computation intensive techniques, such as 3D-grid iterative methods.

Chapter 3

Implementation

This chapter describes the original serial program, Cloud, and how it was parallelized, using *p4*, for a distributed system of heterogeneous workstations. Certain features were implemented particularly to investigate the answers to specific questions, and these are explained both in this chapter, and the next.

Cloud uses the Monte Carlo method to simulate the passage of photons through a stratocumulus cloud deck. Cloud is described more fully in Appendix A. The original serial program, which was written in Fortran by Lin as part of an MSc thesis in Meteorology at PennState [Lin93], was converted to a more extensive program, written in C, by Steve Nagle of PennState. The C version of Cloud is the serial version that was parallelized for this thesis.

The Department of Meteorology at Pennsylvania State University required that Cloud should be parallelized to run on a heterogeneous network of uniprocessor Sun and SGI machines. The message-passing paradigm was chosen, as this was well-suited to this type of environment.

To run a parallel program in a heterogeneous environment it is necessary that there is some means by which machines of different architectures may communicate with one another. In the past this has been difficult, if not impossible. Now, with the advent of modern portable parallel libraries such as *p4* [Butl94], *PVM* [Sund94], and *PARMACS* [Calk94], this has become feasible. *p4* was chosen because it was easy to implement, and it would run on a wide variety of machines, including all five architectures of workstation used in the experiments with Cloud, and it would be portable to the processors at PennState.

3.1 Brief description of the *p4* parallel library

The *p4* portable parallel processing library has been developed by Argonne National Laboratories, as the successor to the popular *parmacs* or *monmacs* macros, which first appeared in about 1984. These macros were available for both Fortran and C, and were implemented by using the *m4* macro preprocessor.

p4 has now been implemented as libraries of routines for portable, efficient, and simple parallel programming. These libraries are available for both C and Fortran [Butl94]. The libraries are portable to a number of different architectures. Both message-passing and shared-memory parallel processing are supported.

p4 has both advantages and disadvantages when compared to other libraries, but will usually give much the same results. Its main advantage is that it can be used for both message-passing and shared-memory parallel processing, which is not true of most libraries, and that it is available for a wide variety of machines. It also has more functionality and tools for debugging than most other libraries.

p4 is particularly well-suited to being used in a heterogeneous environment because of its use of a process group file which contains a list of all machines to be used as slaves, and for each machine the number of slave processes to run on that machine, and the full path name for the executable to be used by that machine. This means that each machine may use a different executable, as is necessary for slaves of different architectures, or machines of a similar architecture may share the same executable. Also different slaves may even perform different tasks, so, for example, a slow slave may do I/O, and a faster slave may do computation.

At Vienna University of Technology, Sukup has conducted an evaluation of the efficiency of two versions of *PVM*, *p4*, *Express* and *Linda* in implementations of four of the NAS Parallel Benchmarks [Suku94]. In comparison with the other libraries the performance of *p4*, for this implementation of these benchmarks, ranged from very good to very bad. Since the NAS Benchmarks are "paper and pencil", in that specification of the problem is only algorithmic, and the user must implement his own code, it is possible that this variation in performance may be attributed to some of the factors which are discussed in Section 5.6.

3.2 Description of the original serial program

The serial program reads three input files, describing atmospheric and thermodynamic data, and a set of user-configurable input parameters, such as the heights of the top and bottom of the cloud, the number of photons to be fired, the number of wavelengths and the angle of the incident light. For this research, exactly the same input data was used for all experiments that were run, and none of the data in these input files was changed.

During each run a number of different cases are simulated, each with a different set of input data, with one case for each interval of each wavelength. The number of photons leaving the cloud through the top or bottom of the column of the cloud, or being absorbed in the cloud, are counted in a number of collection arrays (See Appendix A for more details about the collection arrays). In the serial program each interval of each wavelength is processed sequentially, the contents of the collection arrays are written to file, and the same collection arrays are re-used for each case.

All through the program various results are output to a file of intermediate results, and a file containing the contents of the collection arrays for each case. When all processing is complete, a further file of final results is saved.

3.3 Parallelization of Cloud

This section describes how the serial program was converted to run in parallel on a distributed system of heterogeneous workstations. The original C program was well written and modular, so that the "core" of scientific processing in the program could be used unchanged in the parallel version.

To parallelize this program it was only necessary to make some changes to the main program, together with some slight modifications in the allocation of memory, and to write some additional routines. Some changes were necessary purely because a parallel program requires a different approach from that of a serial program. Others features were implemented particularly to improve the efficiency of the parallel program, or to explore the factors which affect the efficiency of a parallel program.

Cloud is, in many ways, an ideal program to parallelize. In Monte Carlo simulation each photon history is independent of every other. Thus it is possible to break down a task into a number of sub-tasks, so that each slave does some of the photon histories for a particular wavelength. Then the results can be added together, before the final results are calculated. This means that all slaves can operate totally independently of each other, and synchronization is unnecessary. This parallel program is therefore very suitable for a heterogeneous environment, where fast slaves can do more work than slower machines, and the whole program is not therefore slowed down by a slow machine.

3.3.1 Basic structure of the parallel program

The operation of the parallel program is described as follows. The master reads the input data from files, creates a list of tasks, sends them to the slaves and receives results. When all results have been received, it then closes down the slaves, and computes the final results, and writes these to file.

To reduce communication costs, and also to provide for possible future extension, some global variables, such as the heights of the top and the bottom of the cloud, the mean pathlength and the cosine of the incident zenith are sent, as part of the task message, to the slave. The program was designed this way to make it more general and extensible.

Whenever the master is not otherwise engaged with the administration of the job, it also executes Monte Carlo tasks. Between Monte Carlo tasks it checks if any results messages have been received. If there are such messages it processes them, and sends a new task to the slave that sent the result. It continues with collection and collation of results until there are no more messages waiting for attention, when it will again execute a Monte Carlo task.

When the last tasks have been sent out to the slaves, the master remains idle, as it waits until all outstanding results have been received from the slaves. At first consideration it seems that this could be improved, so that the master will instead execute tasks for which results have not yet been received from slaves. However, as at this stage nearly all results have been received, this could result in a delay in the completion of the job, since all the slaves may finish, while the master is still busy on a duplicate task. For this reason this feature has not been implemented.

An additional benefit that arises because the master also does Monte Carlo (slave) work, is that the same executable version can run either as a serial program on one processor, or as a master with any number of slaves, without need of separate compilation. This is a result of the way in which *p4* starts up slaves. To use *p4*, a process group file is created. This file consists of one line for the master, and one line for each slave, specifying the name of the slave, the number of processes to be run on that slave, and the full path of the executable. If this file contains no slaves, then the program will execute on the master only. If the program is written as described above then the master will do all the work, as there will never be any tasks sent to slaves or results received from slaves. Thus, the same executable can be used for both serial and parallel runs, which is very useful in comparing performance.

The *p4* process group file also makes it easy to run the program on a heterogeneous system consisting of machines of different architectures. In this case, a separate executable is specified for each machine architecture, though all machines with the same architecture use the same executable.

Another benefit of the *p4* process group file is that it is easy to change the group of slaves to be used for a run, by changing the slave entries in this file, and it is not necessary to recompile the program. In this way, the number and identity of the slaves to be used will be established at run time, by the contents of the process group file.

3.3.2 Input and output files used

The parallel version reads exactly the same input files, and produces exactly the same output files as the serial program, although minor changes were made to ensure that only the master process would read from, and write to, disk. This was to prevent clashes if the slaves tried to write to disk at the same time as another processor. For practical programming purposes it was also easier to have all file access centralised in one process. All files are NFS-mounted, and all the files used in this research are on the disk of one machine, so that all file access is via the network.

In the serial program, each interval of each wavelength was processed sequentially, and the collection arrays for each wavelength were written to file before the next wavelength was processed. For the parallel program the simulations for each wavelength could be completed in any order, which made it necessary to keep the collection arrays for all wavelengths until the end of the run. At the end of the run the collection arrays, containing the collated data for all wavelengths, were saved to file in the correct order.

3.3.3 Memory requirements of serial and parallel programs

In the serial program the same collection arrays were re-used for each task, which was the calculation required for each interval of each wavelength. As each task was completed, the data in the collection arrays was written to file. In the parallel version these tasks may be split up into smaller sub-tasks, and it is impossible to determine the order in which results would be returned. The final results, for any interval of a wavelength, cannot be determined until all results for that wavelength interval have been received. Therefore the master program allocates one set of collection arrays for each interval of each wavelength, so that results could be collated for the correct wavelength interval, as they were returned, in any order, by the slaves. Thus, for the data set of fifty wavelength intervals used in the experiments, the parallel version required fifty times as much memory than was needed for the serial program. Since the collection arrays for each task required just under 210 kb, the parallel version required approximately 10.5 Mb memory to store the collection arrays for all the wavelengths.

The remaining memory requirements for the master to store global variables were relatively trivial, less than 1 Mb in addition to the collection arrays. Since the master also does Monte Carlo work, an additional 210 kb is required for reusable collection arrays for doing this "slave work".

In addition, both the master, and each slave, require a variable amount of memory for message-passing — approximately 210 kb for each set of messages from a slave. This memory is reusable, and the amount required by the master depends on how many messages are being processed at any one time. For instance, if there are many slaves, and many messages, then more memory is required to store the messages in the master's buffers, until they can be processed. If there is just one slave the master has more time to process the previous message from that slave, before the next one is received, so the memory is freed in time to be reused for the next set of messages.

The slaves required considerably less memory than the master. Each slave needed only about 1 Mb of memory for data storage. This relatively small amount of memory for each slave made the program suitable to be run in an environment with other users, as it is quite possible that the program and data would not have to be swapped out of memory for another process to run.

3.3.4 Overlapping communication with computation

Message-passing is slow, and the time spent in communication by a parallel program is often a significant part of the overall elapsed time. The best performance will be achieved by ensuring that the communication time for each computation task is minimized, and that no processor should spend time idle, while waiting for a message.

There is considerable overhead in message-passing. In *p4* this consists of the cost of sending a message, including building an address and the allocation of memory, the cost of communication via the network, and the cost of the other process allocating memory and receiving the message. If one process sends a message to another process, and then has to wait for a reply message from that process before it can continue, then it will be delayed for the time needed for the round trip. If the other process is busy, or otherwise delayed, this may significantly increase the time that the original process must wait for a reply. Also, if the network is congested the collision rate will increase, and drastically reduce the data transmission rate, thus increasing the waiting time of the original process even more.

In a typical message-passing program the master sends a task to a slave, which executes this task, and then returns result messages to the master. When the master receives these results, it sends the next task to the slave that sent the results. If a program is implemented this way, the slave will be idle between sending the results and receiving the next task.

In *p4*, the overhead of sending or receiving a message cannot be avoided, as the communicating process blocks until the message has been sent or received. This is different from *PVM*, which starts daemons to perform the communication in the background, thus allowing the main communicating process to continue processing once a **send** has been initiated, even if the **send** has not completed, or to process until a message has been received. Although, in *p4* these overheads cannot be avoided, the communication time via the network can be hidden by overlapping this communication time with computation time. This overlapping technique is implemented automatically by some parallel compilers such as Fortran D [Hira94].

Cloud was written to hide communication latency by overlapping communication with computation. To do this the master initially sends two task messages to each slave. Thus, when the slave has sent the result messages for a task to the master, it can immediately begin working on the next task, without having to wait for the round-trip communication time for the master to receive the results, and send the next task. By the time the slave has completed the task it has on "standby", the following task should have been received from the master, and be waiting in a buffer to be received. In this way no processes should have to wait for work.

To prove that this queue of available work improved efficiency, the length of the task queue was implemented as a run-time parameter, so that the user could specify how many tasks were to be sent initially to each slave. The default value if no queue length is specified is two — one task to work with, and one immediately available when the previous task has been completed.

3.3.5 Reducing communication overheads

There is a basic component of the overhead in sending and receiving messages, which is independent of the size of the message and cannot be reduced. Thus, if fewer messages are sent there will be less overhead. There is therefore a potential advantage in combining several smaller messages to make one larger message. Some modern parallel compilers perform this optimization automatically [Hira94]. The remaining overhead in sending a message is proportional to the length of a message. As a message increases in size it takes longer to get it from one processor to another, which can result in delays as a processor waits for a message [Dennis94].

There are also additional overheads to be taken into account when choosing an optimal message length. These include the Ethernet overhead of 18-26 bytes per packet, with a maximum of 1500 bytes of data per packet, and the *p4* overhead of 40 bytes per message.

To see whether it was more efficient to use a number of shorter messages, or one longer message, two slightly different versions of Cloud were implemented. They were identical except for the number and size of the messages sent. In the first version of Cloud, which was used for most of the experiments, each slave returned, for each task, the contents of the five collection arrays in five messages of sizes 4064, 34608, 34608, 69208 and 69208 bytes. In the second version, all five collection arrays were returned in one long message of size 211688 bytes.

Overheads to allocate memory for sending and receiving messages can also be reduced by using the *p4* feature which allocates re-usable buffers, to be used for sending and receiving messages, at the beginning of the program. The sizes of these buffers can be specified to match the sizes of the messages, so as not to waste time by sending unnecessarily large messages which are partly empty. For this program the sizes of the first four buffers were left as the default sizes, and the last four buffers were set to sizes just larger than the sizes of the messages used in the program. Sizes were chosen that were a multiple of 64 bytes which meant they were also multiples of 4, 8 and 16 bytes. This was because it is often more efficient to send messages in blocks of these sizes

The overhead of copying data from a data structure into a message buffer can also be avoided by using the *p4* memory allocation routines to allocate data structures as message buffers (with headers), which are ready to be sent, and do not need to be copied first. Cloud was not implemented in this way, as this would have necessitated considerable rewriting of serial code. However, when a program is written as a parallel program, and not by parallelizing an existent serial program, it may be advantageous to allocate some data structures as message buffers to avoid copying overheads when message-passing.

Communication overheads can sometimes be reduced by using an alternative algorithm needing less communication. However, this should be undertaken with care as an algorithm with less communication may require more iterations to converge, and thus require more overall run time. It is pointless to use a numerically inefficient algorithm merely to exhibit artificially high performance rates on a particular parallel architecture. [Bell94]

3.3.6 Exploiting redundancy and fault tolerance

A distributed network of workstations used as a multicomputer has both advantages and disadvantages. Any part of the network may fail at any time. However if the program is designed to degrade gracefully, so that if one processor fails the others will continue with the program, then as long as one processor is working the program can continue. If all processors are functional then peak performance can be achieved [Yen93].

Cloud has been parallelized so that, if a slave finishes its last task, and there is no more work available, the master will look in a table to see if there are still results outstanding. If so, it will choose one of the tasks for which results have not yet been received, and send this duplicate task to the slave that has no work. Although this does mean duplication of effort, it could result in the job being completed quicker if this faster slave completes the duplicate task before the slower slave which was allocated the original task. This feature is idempotent in that only the first set of results received for a task will be accepted by the master. All subsequent results for the same task will be discarded.

This feature also provides fault tolerance in that, if a slave fails, then it will not return the results of the tasks which were sent to it. However, eventually other slaves will finish their work, and then the master will send these undone tasks to other slaves, which have completed their work. The total job will take longer than if all slaves had been functioning, but at least the job will be completed, so long as the master, and at least one slave, remain functional. The program could easily be extended to ensure that even if all slaves failed, but the master remained functional, the job could be finished, even if it took a long time.

In performance testing, it is important that all slaves are functional, as the best times can only be achieved in this case. Also, it is necessary that all runs perform a known number of tasks, which should be the same for each repeatable run, so that results can be compared. Therefore, the feature to exploit redundancy, and provide a fault-tolerant program, has been disabled for the performance tests. For these tests, all runs of the same set executed the same number of tasks, and if a slave failed then the run was aborted, and repeated later. However, this feature is present in the program and easily enabled.

Due to frequent failures of the parallel program it was necessary to implement a "timeout" feature in the program, so that if the master heard nothing from a slave, for a period longer than the longest time expected for the slave to finish the sub-task, then the run was aborted. The results of these aborted runs were not included in the performance studies.

3.3.7 Dynamic load balancing

The best performance for a parallel program will usually be achieved if all processors are busy for the entire duration of a run, and if all processors finish as nearly as possible at the same time. To achieve this in a heterogeneous system requires careful load balancing. If the work is equally divided among the slaves, so that each slave does exactly the same number of tasks as the others, then the faster machines will finish first, and will then have to wait for the slower machines to complete their tasks. The idle time of the faster machines is wasted time, as it is not used for processing, and as a result the total time for the whole run will not be optimal. It is therefore better to allocate more work to the faster machines, and less to the slower.

A group of workstations may be heterogeneous either because the processors have different performance capabilities, or because on a non-dedicated network the load on each machine, and on the network, changes constantly. For a heterogeneous network it is therefore impossible to determine beforehand how much work should be allocated to each machine, and this must be done dynamically while the program is running.

More work can be sent to faster processors either by sending larger tasks with a dynamically varying task size, or by keeping the task size static and sending more tasks. The computation/communication ratio will be better if larger task sizes are used, but the administration of dynamically varying task sizes is more difficult. Also, if the load situation changes, so that a processor which was previously classed as a fast processor becomes heavily loaded, and takes longer to complete a task, then if large task sizes are being used it will take longer for the master to adjust to sending smaller tasks to this processor. In Cloud, the task size is kept static throughout a run, and dynamic load balancing is achieved by sending more tasks to faster processors, and fewer tasks to slower processors.

Dynamic load-balancing was relatively easy to implement, as all tasks were independent, and no machines had to wait for any others as in a data-parallel program. Load-balancing in Cloud was designed so that the master sends a new task to each slave as soon as a slave returns the result of the previous task. At the same time, whenever the master is not engaged in communication and collation of results, it itself does work. In this way more tasks will be sent to the faster processors, and fewer to the slower ones, and all processors will finish at more or less the same time. Also, if the workload of a machine on a non-dedicated network increases, so that the processor takes longer to complete a task, this will be compensated for automatically as less work will then be sent to this slave, and more to the others.

At the end of the run, some of the slaves may finish before others. This wasted time can be minimized if small sub-tasks are sent to each slave. It was simple to change the sub-task size for a run, since each main task could be divided into a number of equal-sized sub-tasks. The total size of the task for the run is read in from a file of input data. The size of the sub-task, in photons, can be adjusted for each run by specifying it as a command-line parameter, although there is a default sub-task size of 10000 photons if no sub-task size is supplied.

The size of the sub-task is not adjusted dynamically while the program is running because, for performance testing, it is important to achieve repeatable results. Also, one of the factors being studied in this thesis was to determine the optimal sub-task size, so it was important that this should not change during the run. However, a proposal of a load balancing method where the task size is changed dynamically is suggested in section 6.7.2

The method of load balancing used in these experiments also takes into account failed slaves, and apportions the extra work to the other processors. So if one or more slaves should fail the work will still be done, even though the best performance will not be achieved.

Efficient dynamic load balancing is considerably more difficult to achieve for programs that require communication and synchronization between processes, such as data-parallel programs. For these programs some method has to be developed to adjust the workloads of each processor dynamically, so as to minimize the delays due to synchronization [Cap93].

3.3.8 Scalability

Amdahl's law states that the speedup of a program is limited by its serial overheads [Wils93]. The ideal program will scale linearly with an increase in the number of processors. By scaling a problem to a sufficiently large size to improve the computation to communication ratio, overhead can be reduced to increase processing rates [Sing93][Gram93][Bell94]. Workstations provide size scalability and evolvability to some degree, although LAN communication rates have remained constant [Bell94]

In practice, scalability may be limited by such factors as communications overheads, synchronization bottlenecks, granularity and load balancing problems. For instance, adding further processors does not always produce an increase in performance, and may even result in a decrease in performance as a result of increased overheads. Bell claims that there is a lack of understanding about application scalability for scalable machine characteristics, and this guarantees a negligible application market using existing third-party vendors [Bell94].

It is particularly difficult to assess the scalability of a heterogeneous network, particularly if there is a large difference in the performance capabilities of the machines used. Sometimes little or no gain will result if a slow processor is added to a network of much faster machines. For this reason, experiments with different numbers of slaves were also run on the smaller homogeneous sub-groups, to establish whether Cloud scaled well for homogeneous groups.

This program has been implemented so that it is easy to investigate its scalability. It is simple to change the problem size in the file of input parameters, and easy to adjust the sub-task size which is an input parameter. The number of slaves for a run can be changed simply by altering the number of entries in the process group file. This program was run with various numbers of slaves, for all task sizes, on from 1 to 18 processors. The results of these experiments can be seen in Chapter 5.

3.3.9 Generation of random numbers

Monte Carlo simulation requires the generation of millions of random numbers. Numbers generated by computers are not truly random, but rather pseudo-random, in that a sequence of machine-generated random numbers has a finite length and will ultimately repeat itself. This is important for Monte Carlo simulation, because there is no point in producing photon histories which have already been generated. Ma discusses the subject of generation of random numbers for Monte Carlo simulation at some length in [Ma93]. Another point made by Ma is that some random number generators are faster than others, which is significant in Monte Carlo simulation when millions of random numbers must be generated. Usually the better random number generators, which have a longer cycle before repeating themselves, are slower. It is therefore important, when choosing a random number generator for Monte Carlo simulation, that these points be taken into account, so that the results are valid, and at the same time the program runs as quickly as possible.

The original serial program has a specially written random generator, which is seeded from the clock of the processor. For the parallel program this same random number generator was used, but it is seeded separately for each slave from the slave's clock. It is therefore extremely unlikely that any two processors will begin with the same seed, and thus produce the same photon histories. However, the random number generator used by this program should be checked by a statistician, to ensure that valid results are produced.

During this study, the problem of choosing a good random number generator has been largely ignored. This thesis concerns primarily the factors affecting the performance of a parallel program, and for this it is does not matter whether photon histories are repeated.

Chapter 4

Design of Experiment

This chapter describes the experiments that were conducted to investigate the factors which affect the performance of a parallel program on a network of heterogeneous workstations. Since the performance of a heterogeneous network is a complex issue, some experiments were conducted on homogeneous sub-groups of the heterogeneous group. The results of these experiments were useful for understanding the behaviour of Cloud on the heterogeneous system.

4.1 Experimental environment

The network used consisted of 18 heterogeneous uniprocessor workstations, connected by an ordinary Ethernet. These were 1 Silicon Graphics Indigo2 Extreme, 2 Silicon Graphics Indigo, 2 Sun ELC, 9 Sun SPARCclassic and 4 Sun SPARCstation 1+ workstations. Technical specifications of these machines, and Ethernet, are given in Appendix B. All these machines are in several different rooms on the same floor of one building. The experiments were run on various groups of these workstations with the groups ranging in size from 1 to 18 machines.

Since most of this work was done during term-time in an academic teaching environment, it was not possible to have dedicated use of either the workstations, or the network. Approximately fifty people had access to the workstations involved in the experiments, and about three hundred people used the network via other terminals and computers. In addition, all the machines are Network File Servers (NFS) and some were also Network Information Servers (NIS) which had some impact on the performance.

All files involved were stored on the hard disk in one of the Silicon Graphics Indigos. The minimal I/O in Cloud was performed only by the master, and had little impact on the performance.

The primary purpose of this thesis was to study performance, so most runs were at night and at weekends, usually between midnight and 08h00, when there was the lightest load on the network, and the best results could be achieved. These results, which are described in Chapter 5, were as close to being repeatable as possible on a non-dedicated network, and gave a good indication of the best performance that would be possible on a dedicated network. Few real users have exclusive use of a network of workstations, so these results are generally more realistic than those that would be obtained on a dedicated network.

4.2 Compiler options and executables

The parallel library, *p4*, supports running a parallel program on a network of either homogeneous or heterogeneous processors.

For this study there were two primary architectures - Sun and Silicon Graphics. The executables for both architectures were compiled from exactly the same source code. The executable for the three Silicon Graphics machines was compiled on the Silicon Graphics Indigo2 Extreme, using the native **cc** compiler with the compiler optimization option of **O3**. (The higher **O4** optimization produces R4000 specific code which will not run on the two R3000 Indigos) The Sun ELC, SPARCclassic and SPARCstation 1+ workstations all used a second executable, which was compiled on a SPARCstation 1+ workstation, using the native **cc** compiler with compiler optimization option of **O4**.

The same executable was used for both the master and the slave computers. In Cloud, the master itself does Monte Carlo simulation whenever it is not involved in communication with the slaves. The Monte Carlo code comprises the bulk of the program, so little would be gained in separating the tasks of the master and slaves into two different executables.

The number of slaves used in any run does not affect the executable, as this is dependent only on the machines listed in the *p4* process group file, which is external to the program. Thus, for a serial run there will be no slave machines listed in the *p4* process group file, and the master will do the entire run by itself. This means that exactly the same executable was used for both serial and parallel runs, which made it simpler to conduct the experiments, and was particularly beneficial in the calculation of speedup.

4.3 Problem size

It has been noted that small benchmarks lose their predictive value with the advent of on-chip caches, and sophisticated optimizing compilers [Weic91]. Frequently, such small benchmarks do not reflect the impact of input and output on the performance of a program, or the proportion of computation to input/output is different to that of a real application. Also, some small benchmarks may run with only a subset of data, which may fit entirely in cache, thus giving good performance results. However, when the full data set is used in a real application, it may be too large to fit in memory, thus resulting in increased swapping, which causes a deterioration in performance. Also, if the actual elapsed time of a program is very short, the startup and closedown times of the parallel run may constitute such a great proportion of the total time that it is difficult to separate this from the actual parallel run time, thus giving misleading results. For these reasons, many current benchmark efforts, such as SPEC [Dixi91], now concentrate on larger, complete applications, which have a real use.

In this study a realistic problem size has been chosen, even though in its serial version it runs for nearly an hour on the fastest workstation used, and almost six hours on the slowest machines, and even the parallel version on a group of 18 workstations takes from 16-18 minutes. The task size chosen was for 120000 photons to be fired through the cloud. This task size is typical of the actual task size that would be used by climatologists, and is large enough to give meaningful results. Thus the performance results obtained in this study are typical of those that would be obtained for this application in actual use.

As a result of this approach, it took a great deal of time to collect the results of these experiments: over a period of several months, groups of between 1 and 18 workstations were running this program for anything up to 8 hours per night. In all, over 800 runs, comprising more than 3000 hours of elapsed computer time, were executed.

4.4 Number of runs for each experiment

For any experiment such as this, it is necessary that the results should be repeatable. This is particularly difficult in a non-dedicated network, where the environment is constantly changing. Therefore, each experiment should be repeated a number of times to ensure that the results are consistent. Although the program took so long to run for each test, most runs were repeated at least three times. Where there were more than three runs the best (fastest) three results were taken. The variations in the results for such long runs was found to be so small that it was felt justified to repeat the run only three times. The means of the three best runs were used for the graphs, and in the calculation of speedup.

There was one exception to the repeating of each run three times. The experiments using very long messages caused such serious network congestion that performance was seriously degraded, and sometimes the run even failed entirely due to lost messages. In these experiments it was too time-consuming, and too difficult, to repeat the runs as often as three times, particularly as the network congestion seriously interfered with other users of the workstations, and the network. However, sufficient results were obtained to show the effects of such an experiment.

In some of the other experiments, such as those testing the performance of the Ethernet, the tests were repeated more than three times to obtain more accurate results. These tests are described more fully later in this chapter.

4.5 Timing of runs

There is considerable discussion concerning the correct way to time parallel programs [Crow94]. The generally accepted view is that the total elapsed time is of primary importance. However the CPU time and system time are also of interest, as it is useful to know the time each process spends on computation, and how much time is spent on overheads such as communication. Thus the experiments conducted in this study were timed in a number of ways.

First, all runs were timed using the Unix[1] **time** command. This provided, for the master, the CPU time used by the application, the system time used by the application, and the overall elapsed time for the run.

In addition, the *p4* timing functions were used to measure the time taken by various parts of the program. The times were wallclock times, and included time when the application was swapped out. These times were recorded in milliseconds. In addition to these timing statistics, the number of

[1] Unix is a trademark of AT&T Bell

sub-tasks executed by each workstation was recorded. The items measured were as follows:

For the master:

- the number of sub-tasks executed by the master,
- the overall execution time from start to closedown,
- the time the master spent in starting up and initialization before any tasks were sent to the slaves, or Monte Carlo work executed by the master,
- the time spent by the master in the parallel section of the program, (i.e. not including initialization and closedown),
- the time spent by the master in executing Monte Carlo simulation,
- the time spent by the master in waiting for messages from slaves,
- the time spent by the master in computing the final results after all work is completed.

For each slave:

- the number of sub-tasks executed by the slave,
- the overall execution time from start to closedown,
- the time spent by the slave in executing Monte Carlo simulation,
- the time spent by the slave in waiting for messages from the master.

The remaining time that was not spent in computation, or waiting for messages, was primarily that needed for the sending and receiving of messages, and this could be calculated from the above measurements.

4.6 Calibration of the workstations and network

It is particularly difficult to evaluate the performance of a heterogeneous network, as all the machines have different performance capabilities. Conventional methods of evaluating performance, such as speedup, are not immediately suitable for assessing the performance of a heterogeneous system. Thus, before evaluating the performance of a heterogeneous system it is first necessary to understand the performance of the individual components of the system.

The following experiments were conducted to measure the serial performance of each individual machine, and the performance of the network. These serial measurements were then used to group the workstations into sub-groups of like machines, so that the behaviour of Cloud on homogeneous machines could be studied. It is easier to study the performance of such homogeneous groups, and to establish whether the program is performing efficiently in such aspects as load balancing, and whether there is a constant improvement in performance as like processors are added. These results can then be used to understand and assess the performance of the parallel program on a network of heterogeneous workstations. This method of first studying the performance of homogeneous sub-groups, and then the performance of the combined heterogeneous group was also used by Altevogt et al, as described in section 2.3.2 [Alte93].

4.6.1 Serial runs on every workstation

At least three serial runs of Cloud were run for each workstation involved in the study. The executable was the same as that for the parallel experiments but there were no slaves in the process group file. In some cases more than three serial runs were executed, but in all cases the three runs with the shortest overall elapsed time were used for the calculation of means, standard deviations and speedups.

4.6.2 Performance of the Ethernet

Part of this study concerns the size and number of messages used in communication between the slaves and the master. The impact of message size on performance was established by measuring the optimal data rate that could be achieved on Ethernet, by using the *p4* library, and comparing this with the performance achieved for different message sizes when running the parallel program.

4.6.2.1 Bandwidth of Ethernet

Nanayakkura et al observe that, although the theoretical performance of Ethernet on a dedicated strand is 10 Mbits/second, this is seldom achieved in practice. They found the rate to be approximately 5-7 Mbits/second on a dedicated strand [Nana93]. Halsall confirms this, and points out that even under light load conditions, and with a cable bandwidth of 10 Mbits/second, the actual performance of Ethernet is more likely to be nearer to tens of kilobits rather than megabits/second. This is a result of processing delays associated with the higher protocol layers. [Hals85]

4.6.2.2 Packet size of Ethernet

The message length may have considerable impact on performance. In Ethernet, there is an overhead of 18 or 26 bytes per packet, and each packet may transfer from 1 to 1500 bytes of data. If the amount of data is less than 46 bytes, then this is padded to 46 bytes. If the messages are longer than 1500 bytes, they must be broken down, sent, and reassembled. There is also an additional overhead of 40 bytes per message in *p4*. This reduces the maximum amount of data that can be sent in the first packet of a message to 1460 bytes, but all subsequent packets, of the same message, may carry 1500 bytes of data. The optimal message length should be such that the most data is sent for the least message overhead, which will occur when the maximum amount of data is sent in each packet. Ponnusamy et al, in their experiments with the CM-5 (non-Ethernet network), showed that the best data rate was obtained with full packets [Ponn93].

For Ethernet, the required interpacket spacing (i.e. the minimum time between consecutive transmissions from a single station) is 9.6 μs. This means that a station transmitting a message must pause between each packet, and another station may seize the line at this time. This will delay the first station, which will have to wait for a break in transmission before it can continue transmission.

When a station detects a collision, it backs off an increasing integer number of slot times for each collision detected. Thus for very long messages needing many packets, on a heavily loaded network, a workstation can be considerably delayed in waiting for a chance to transmit, and this can have a serious impact on overall performance. If there are a number of stations, all trying to transmit at the same time, this can result in considerable delays for all stations. This problem is alleviated to some extent on a heterogeneous network when the slaves take a different amount of time to finish a task, so the communication is more evenly spaced as the slaves will send at different times.

4.6.2.3 Experiment to test Ethernet performance

A separate test program was written, using *p4*, to establish the data rate of the Ethernet network used, whether the message size affected this performance, and whether performance could be improved by sending longer messages so that the proportion of data, as compared to overhead, was increased. This program was a modification of part of the program **systest**, which is released with the *p4* library as an example program.

The largest messages used Cloud were approximately 210 kb long, so in the test program messages, of lengths ranging from 1 byte to 210 kb, were sent between the master and one slave, with messages of each length being sent 15 times each. Each message was sent from the master to the slave and then back to the master. The time was measured using the *p4* timing functions. The data rate was then calculated, using the size of the message in 8-bit bytes, and the time of the round trip in milliseconds, to give a rate in Megabits/second, so that this could be compared with the theoretical performance of Ethernet of 10 Mbits/second. These results were also used to calculate the time required for sending and receiving a NULL message. The results of this test program are given in section 5.2.2.

4.7 Grouping of machines into homogeneous and heterogeneous groups

The results of the serial runs described in section 4.6.1 were used to define approximately homogeneous and heterogeneous groups of machines, which were used for the experiments. It is useful to group machines into homogeneous groups, so that any improvement in performance of the program that is gained by adding extra machines can be better evaluated, when compared to the expected performance for that number of machines. These results can then be used to understand the performance of the heterogeneous system.

Homogeneity is a complex issue for a group of workstations connected by a network. Workstations are generally only considered homogeneous if they have the same CPU chip, and the same amount of memory and cache. Similar machines with differing amounts of memory and cache are not truly homogeneous, as a different amount of paging or cache misses may result in different performance. Also, if the workstations are in a non-dedicated environment where either the workstations, or the network, or both, are shared by other users and processes, then these external processes may affect the performance, so that the workstations can no longer be considered truly homogeneous [Cap93]. This will be most evident in the longer overall elapsed time, and the uneven distribution of work, as the machines will no longer perform the same amount of work as they would if they were dedicated.

The machines used in the experiments were grouped into approximately homogeneous groups, according to their serial performance, as described in section 5.2.1. These groups were not truly homogeneous because they were in a shared-environment, and some of the machines had different amounts of memory, and even different CPUs. However, the serial performances of the machines in each group were sufficiently similar, that for the purposes of these experiments the groups could be considered homogeneous.

For the purpose of these experiments the "homogeneous" groups defined on the basis of their serial performance were:

- The **ELC/Classic group**, consisting of 2 Sun ELCs (33 MHz) and 9 Sun SPARCclassics (50 MHz), which all had approximately the same performance (see section 5.2.1). The ELC with the most memory (32 Mb), and the best serial performance, was used as a master for this group. The other ELC (16 Mb memory) was used only as the tenth slave, and was excluded from all runs with 1 to 9 slaves, so as to obtain the best homogeneity possible. Three of the Classics had 32 Mb of memory, and the remaining six had 16 Mb.

- The **SPARCstation 1+ group**, consisting of 4 Sun SPARCstation 1+ machines (25 MHz). For this group, the workstation with the largest amount of memory (28 Mb) was used as the master, and the other machines, with 12 Mb of memory each, were used as slaves.

- The **SGI group**, consisting of 1 Silicon Graphics Indigo2 Extreme (100 MHz, 128 Mb memory), and 2 Silicon Graphics Indigos (33 MHz, 24 Mb & 16 Mb memory), where the Indigo2 was considerably faster than the other 2 Indigos. The Indigo2 was used as a master in this group so that the 2 slaves had similar capabilities.

The heterogeneous groups were:

- The **18-machine group**, consisting of all 18 machines that were available, with the Indigo2 as the master, since this was the fastest machine and had the most memory, with the other 2 SGIs, the 2 Sun ELCs, the 9 Sun SPARCclassics and the 4 Sun SPARCstation 1+ workstations as slaves.

- The **14-machine group**, consisting of same machines as the **18-machine group**, but excluding the 4 Sun SPARCstation 1+ workstations, which were the slowest machines.

Most of the initial experiments were conducted on the **ELC/Classic** group because this is the largest group of nearly homogeneous slaves.

4.8 Slave startup times

The time taken to start up slaves can be significant in a parallel program [Minn93]. To establish whether the startup time for Cloud was a significant proportion of the overall elapsed time, Cloud was modified so that the master terminated immediately after starting up the slaves listed in the process group file, and similarly the slaves finished immediately they had been started. The resulting executables were exactly the same size as those used for the other tests. The results of this test therefore reflect the actual times for the starting up of the Cloud program.

The modified version was run 20 times for each of 10 groups of slaves ranging from 1 to 10 in number for the **ELC/Classic** group, for each of 2 groups of from 1 to 2 slaves for the **SGI** group, and for each of 3 groups from 1 to 3 slaves for the **SPARCstation 1+** group. These runs were timed using the Unix **time** command.

4.9 Varying the number of slaves

For each "homogeneous" group, experiments were conducted to measure the performance of the program, with the number of slaves varying from 1 to the maximum number of slaves in that group. These experiments with the homogeneous groups were useful in assessing whether there was a constant improvement in performance as additional slaves were added. These results could then be used to understand the performance of the heterogeneous group.

The heterogeneous groups of 14 and 18 machines were combinations of the machines in the "homogeneous" groups, so performance tests for the heterogeneous groups were run only for the maximum number of 14 and 18 slaves.

4.10 Load balancing and granularity

The efficiency of the load balancing was investigated by dividing the work into a number of smaller sub-tasks, and running the program with different sub-task sizes for each run. For each run, the size of all the sub-tasks was static.

As described in section 3.3.8, the number of photons to be processed in each sub-task was a parameter. The overall number of photons to be fired for each wavelength was chosen to be 120000, since this was a realistic problem size, and it also factorized easily into sub-task sizes of 5000, 10000, 20000, 30000, 40000, 60000 and 120000 photons.

For each experiment the program was run three times for each sub-task size, but in all cases the total number of photons fired for each wavelength was 120000. For these experiments there were 50 wavelengths.

Thus the experiments were as follows:

50 sub-tasks of 120000 photons, (1 for each wavelength)
100 sub-tasks of 60000 photons, (2 for each wavelength)
150 sub-tasks of 40000 photons, (3 for each wavelength)
200 sub-tasks of 30000 photons, (4 for each wavelength)
300 sub-tasks of 20000 photons, (6 for each wavelength)
600 sub-tasks of 10000 photons, (12 for each wavelength)
1200 sub-tasks of 5000 photons. (24 for each wavelength)

4.11 Overlapping communication with computation

As described in section 3.3.4, the Monte Carlo program has been written so that, at all times, the slave process has its next task waiting in a queue.

An experiment was conducted to see if having the next task immediately available for the slave was significant in reducing the overall elapsed time. Additional experiments were conducted to see if having one task available was sufficient, or whether it was advisable to have more than one task in the queue.

The **ELC/Classic** group with 10 slaves was used for these experiments, as this was the largest homogeneous group available, and it is most likely that there will be communication delays when the master has more slaves. This is also the case which is most likely to have network congestion, as there are more machines sending messages in a shorter overall time. This homogeneous group was used for this experiment because it was easier to assess the impact on performance where all slaves were similar. If this experiment was run using the heterogeneous group the results may be confused by other factors arising from the disparate nature of the slaves.

Three runs were run for each task size for queues of 0, 1, 2 and 3 available tasks. The queue length was a command-line parameter so the same executable was used in all cases.

4.12 Changing the number and size of messages

As described in section 3.3.5, two versions of Cloud were implemented to see if there was an improvement in performance if communication overheads could be reduced by combining several messages in one longer message. In the first version, each slave returned the results of the simulation in five different messages of sizes 4064, 34608, 34608, 69208 and 69208 bytes, and in the second version, all the results were returned in a single message of size 211688 bytes. In both cases the master sent each new task to the slave in a single message of 8088 bytes.

The data for the version returning five result messages has been collected in the other experiments, since all experiments, unless otherwise specified, used this version of the program. So this experiment consisted of running the version returning one long result message, using the **ELC/Classic** group of machines, as this was the largest homogeneous group available. This version of Cloud was run several times for each task size, as described in section 3.3.7, on groups of 5, 6, 7, 8, 9 and 10 slaves. It was considered that running it for these groups of slaves would be sufficient to compare the results with those for the program returning five result messages, and using the same groups of machines, and it was not necessary to run it for groups with 1, 2, 3, and 4 slaves as well. The groups with larger number of slaves were chosen to increase the chances of network congestion.

Chapter 5

Results and Discussion

The usual methods of presenting parallel performance results include the measurement of elapsed time, speedup and efficiency. However, there is at present no firm consensus on how to measure the performance of a parallel program on a heterogeneous distributed system. This chapter investigates some of the issues involved, and discusses some alternatives proposed by others. These issues are illustrated by using various methods to present the measurements obtained from the experiments in Chapter 4. The technical specifications of the workstations and of Ethernet, can be found in Appendix B.

The first section of this chapter discusses how a parallel program should be timed. The next section considers what is meant by serial performance, and presents the serial performance results for Cloud. Speedup and efficiency are defined in the following section, and the difficulties of using these measures in a heterogeneous environment are discussed. The next section shows some alternative methods for evaluating the performance of a parallel program in a heterogeneous environment. Several different methods of presenting parallel performance results are compared by using these alternative methods to present the results of the experiments described in Chapter 4. After that some of the factors which affect parallel performance are discussed, and illustrated with the results of the experiments. The chapter concludes with some predictions of the expected performance of this program if the number of slaves was increased.

All results in this chapter, unless otherwise specified, are the mean of the 'best' three runs for that particular experiment, where 'best' means the shortest elapsed time for the run.

Each experiment, using Cloud, was run for all seven task sizes and for all numbers of slaves in the group being tested. For those graphs in which serial performance is compared with parallel performance, the serial performance for the program executing the task size of 120000 photons is used. This is because there is no point in breaking the work into smaller tasks when running the program on a single processor. Thus the value for serial performance for all task sizes was arbitrarily chosen to be that for the 120000 photon size. In some graphs this value for serial performance is repeated seven times, so as to correspond to the seven task sizes. The serial results shown in these graphs are therefore **not** seven different values for seven task sizes, but the same result for the 120000 photons repeated.

In all the line graphs presented in this chapter lines are used to connect sets of results. This is not strictly valid as these results are discrete rather than continuous. However, the lines serve as guidelines to indicate which sets of results belong together.

5.1 Elapsed time vs CPU time

It is well established that elapsed time should be used to measure the performance of a program, since this is most representative of the actual time a user must wait for a result [Henn90]. It is relatively simple to measure the elapsed time for experiments on a dedicated system, where it is easy to obtain repeatable results. Difficulties arise, however, when timing runs on a non-dedicated system, when the performance of an application may be seriously affected by other processes, whether these are other users or operating system daemons. Thus, some researchers have been tempted to use CPU time alone to measure the performance of their applications, since the elapsed times measured when repeating runs may vary widely [Crow94]. However, CPU time alone may be misleading, and may not reflect the true performance characteristics of the program.

This is well illustrated by the experiment in which two different versions of Cloud were run on the same group of machines to produce the same output. These experiments showed that while the CPU time was almost identical for both versions, there was a considerable difference in the elapsed times measured. In both versions the same output was returned, and the total length of the messages (207 kb) was the same. The only difference between the versions was that, in one version, five shorter messages (4064, 34608, 34608, 69208, 69208 bytes) were used, and in the other version, one long message (211688 bytes) was used to return the results.

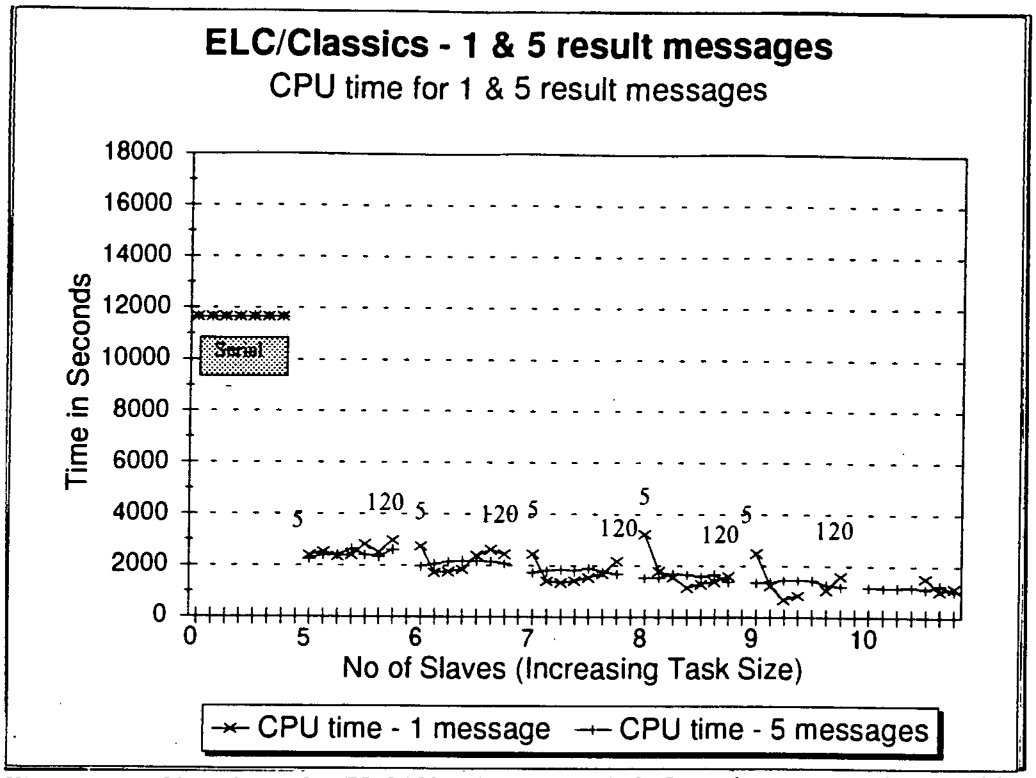

Figure 1: CPU times for **ELC/Classic** group - 1 & 5 result messages (grouped by number of slaves)

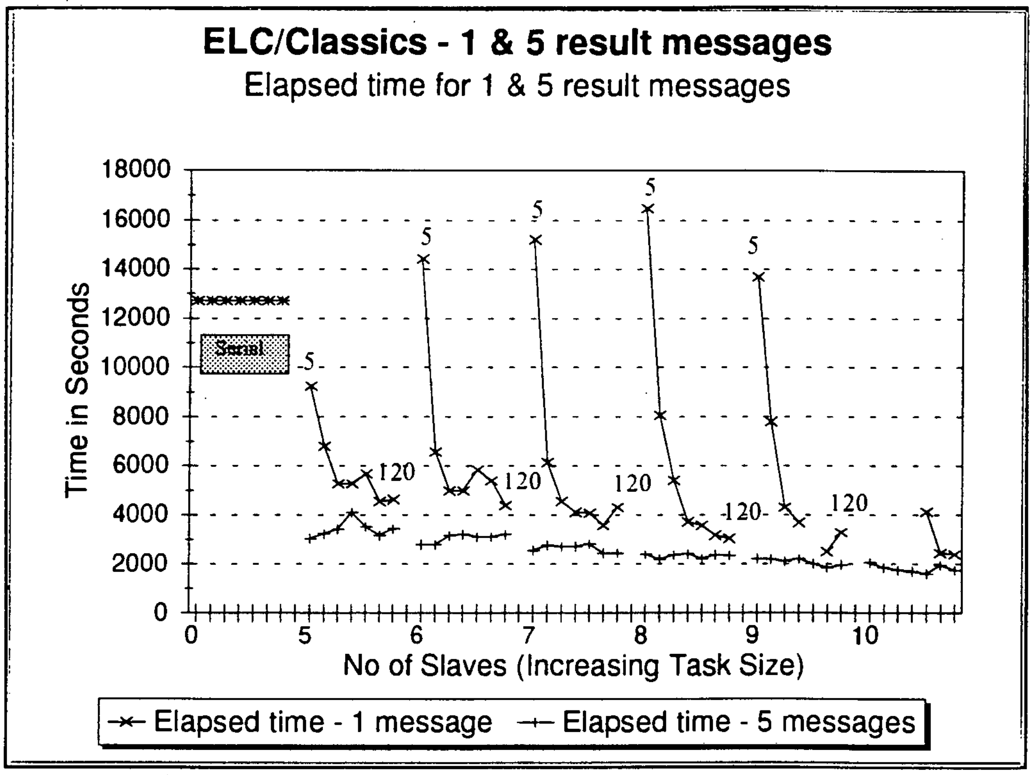

Figure 2: Elapsed times for **ELC/Classic** group - 1 & 5 result messages (grouped by number of slaves)

Figure 1 and Figure 2 compare the measurements for these two versions of Cloud. Figure 1 shows the CPU time, and Figure 2 shows the elapsed time. The results are grouped first by the number of slaves for the run, and within this group by the size of the sub-tasks for the run. The number of slaves in each group is shown on the x-axis. For each group of slaves the results are grouped with increasing task size from left to right, from 5000 up to 120000 photons. For each group of results the 5 on the left indicates the result for the 5000 photon task size, and the 120 on the right indicates the result for the 120000 photon task size. The intermediate points represent the results for the 10000, 20000, 30000, 40000 and 60000 photon task sizes in that order.

Figure 1 shows that there is relatively little difference in the CPU time for the two versions, but Figure 2 shows that there is considerable difference in the overall elapsed time, with the one-message version taking from 1.5 to 8 times as long as the five-message version. The reasons for this are discussed in section 5.6.4. In this experiment, the CPU time alone would give misleading information about the performance of the program. (The missing data in Figure 1 and Figure 2 is because the network congestion from some experiments was so serious that some of the experiments could not be completed, owing to lost messages, and inordinately long run times.)

These results show that the overall elapsed time is more representative than CPU time in assessing parallel performance, as the elapsed time includes not only the time spent in computation, but also the time spent paging and performing I/O. Even though some of the elapsed time may be spent while the program contends for resources with other unrelated programs, much of the waiting is integral to the program and must be considered when timing performance [Crow94].

However, it is also informative to measure both CPU and system time, as these can give some indication of the behaviour of the program, and may suggest ways in which performance can be improved. In the example illustrated by Figure 1 and Figure 2, the excessive system time measured for the version returning a single results version indicates that there is a problem.

Similarly, if the total elapsed time differs significantly from the sum of the CPU and system time, this may be due to time-sharing and sharing resources with other applications, but could also indicate that the program is blocking[1], such as when waiting for messages. If the waiting is inherent to the program, then redesign may reduce this waiting time, perhaps by overlapping communication with computation, and thus improve performance.

For all the experiments conducted for this thesis the elapsed times were used to compare performance, even though it was difficult and time-consuming to obtain these times. The elapsed time is what the user of the system will experience. The CPU time, as has been shown in this section, is a poor indicator of elapsed time, even though it is easier to obtain repeatable measurements of CPU time.

5.2 Basic performance measurements of the hardware used

This section presents measurements of the serial performance of the workstations, and measurements of the data rate that could be expected for the network.

Figure 3 is a schematic diagram of the network topology, and shows the positions of the workstations used. The technical specifications of the workstations, and of Ethernet, can be found in Appendix B.

The network consists of five sections, with each section connected to a multiport repeater. This is as a result of the limitations of Ethernet which limits the maximum length of each section to 186 metres. The multiport repeater repeats each signal received from any section to all the other sections, so for practical purposes the network used in these experiments can be considered as one single broadcast network.

5.2.1 Serial performance of the workstations

To evaluate the performance of a parallel system, the parallel performance is compared to the serial performance of the workstations used. Thus, in a study such as this, the basic serial performance of each workstation must be measured before parallel performance can be evaluated. However, the question then arises of what is meant by serial performance. Should it be the elapsed time of the best serial version available, or the elapsed time of the parallel

[1] "blocking" is used in this dissertation to mean that the program is halted for some reason, such as when waiting for messages, or competing for resources.

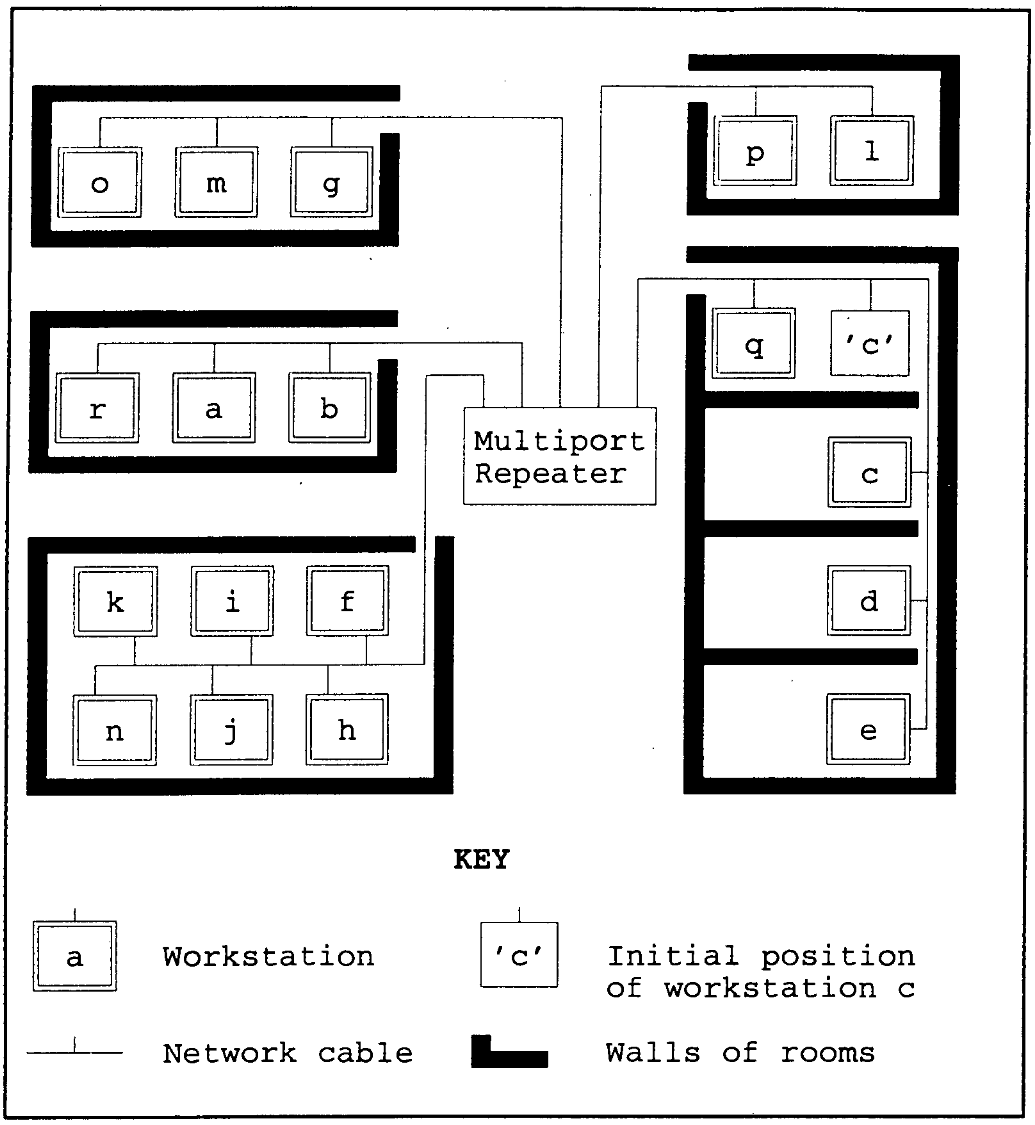

Figure 3: Schematic diagram of network topology

version running on a single machine?

A parallel program contains additional code to implement parallelism, and this creates additional overheads. If a parallel program is run on a single processor, there is no advantage to be gained by parallelism, and the parallel overheads will reduce the possible performance. Also, a parallel algorithm may perform badly on a serial processor, where a serial algorithm for the same problem may give much better results on a single machine.

Cloud has been parallelized so that if there are no slaves present, then the code for implementing parallelism is ignored, and Cloud behaves almost exactly the same as the original serial program.

Table I shows a comparison between the performance of the original serial program, and the serial performance of the parallel version of

Table I: Comparison between performance of original serial program and performance of parallel program on a single processor

Workstation	Elapsed time of the original serial program (Seconds)	Elapsed time of parallel program (1 processor) (Seconds)	Percentage difference
a - SGI2 [-O3]	3017	3111	1.0
b - SGI (16 Mb) [-O3]	5769	5565	-3.5
c - SGI (24 Mb) [-O3]	6220	6366	1.0
d - ELC (32 Mb) [-O4]	12858	12715	-1.1
e - ELC (16 Mb) [-O4]	12754	13226	3.7
f-n - Classics (Mean) [-O4]	13245	14105	6.5
o-r - SPARCstation 1+ (Mean) [-O4]	19873	20464	3.0

Cloud, but running on a single processor. The serial performance of the parallel version of Cloud was very similar to the performance of the original serial program, with differences in elapsed time ranging from 1% to 6.5%. The CPU times for both versions were nearly the same, and most of the difference was caused by an increase in system time for the parallel version, which uses considerably more memory. The compiler optimization used is shown in square brackets [] in the first column of Table I.

In the evaluation of the parallel performance of Cloud, the serial performance of the parallel program, running on a single processor, was used, rather than the performance of the original serial program. This was because it was easier to use exactly the same executable for both the serial and the parallel runs, and because the differences in performance of the original serial program and the parallel program running on one processor, as shown by Table I, were so small that they could be considered negligible.

Thus, the serial performance of all workstations used in this study was measured by running the parallel version of Cloud, compiled with the highest possible compiler optimisation, on each processor alone without any slaves. The program was run on each of the 18 workstations at least three times, and the shortest three elapsed times for each computer were used in this study. These measurements were then used as serial performance measurements for the calculation of speedup.

Table II: Serial performance of workstations

Name	Arch.	CPU Time Mean Seconds	System Time Mean Seconds	Percentg Utiliztn	Total Elapsed Time Mean (Std.Dev)[%] Seconds
a	SGI	3079.67	15.47	99.47	3111 (6) [0.2]
b	SGI	5338.67	85.63	97.50	5565 (109) [2.0]
c	SGI	5944.00	215.00	96.75	6366 (63) [1.0]
d	ELC	11655.17	151.60	92.86	12715 (116) [0.9]
e	ELC	12677.37	28.90	96.07	13226 (19) [0.1]
f	Cls	13760.00	8.67	98.02	14047 (53) [0.4]
g	Cls	13799.33	7.67	97.25	14198 (142) [1.0]
h	Cls	13777.00	8.67	98.04	14061 (63) [0.4]
i	Cls	13531.67	8.67	98.05	13810 (96) [0.7]
j	Cls	13789.00	8.67	96.83	14254 (269) [1.9]
k	Cls	13789.00	8.33	97.55	14145 (150) [1.1]
l	Cls	13773.00	5.00	97.25	14168 (278) [2.0]
m	Cls	13880.67	4.33	98.38	14114 (105) [0.7]
n	Cls	13786.33	9.33	97.53	14144 (57) [0.4]
o	SS 1+	19889.23	141.60	97.19	20610 (144) [0.7]
p	SS 1+	20033.53	72.73	98.28	20458 (189) [0.9]
q	SS 1+	19857.77	72.47	98.31	20272 (49) [0.2]
r	SS 1+	19975.97	90.07	97.80	20517 (121) [0.6]

Table II shows the mean serial performances of the workstations used. The CPU time, system time and elapsed time for these runs were measured using the Unix **time** command. The mean times for each processor are shown in seconds, with the standard deviation of the elapsed time given in curved brackets (), and the standard deviation as a percentage of the elapsed time in square brackets []. The percentage utilization is the sum of the CPU and system time, expressed as a percentage of the elapsed time.

The high utilization in Table II shows that all computers could be considered as dedicated for these serial runs. The percentage utilization ranges from 92.86% to 99.47%, and fourteen of the eighteen machines have a percentage utilization of more than 97%. Of the four machines with a value of less than 97%, three (**c**, **d** and **e**) are on the same network segment. One of these machines (**c**) gave better performance in earlier experiments. Then it was moved to a new location, one step along the network towards **d** and **e**, and there was a noticeable deterioration in its performance (refer to Figure 3). In the earlier experiments **c** gave approximately the same serial performance as a similar machine, **b**. After the move, **c** showed elapsed times approximately 10% longer than that of **b**. Also Table II shows that **c** and **d** use a considerable amount of system time, which is much more than that for similar machines on other links of the network. All these factors indicate that there is a network problem which affects the performance of both **c** and **d**, and to a lesser extent **e**. This problem is allegedly because the network cables connecting these machines is very old, and in a bad condition. However, the deterioration in performance was very small, and was considered negligible as it did not invalidate the performance results of these experiments.

Table II also shows that the standard deviation of the elapsed time was less than 1% of the total elapsed time for 12 of the 18 workstations, and for the remaining 6 machines it was between 1% and 2% of the total elapsed time. These standard deviations, together with the closeness of the elapsed times for machines of the same architecture, indicate that the performance values for these serial runs were reasonably repeatable, and may legitimately be used for the evaluation of parallel performance.

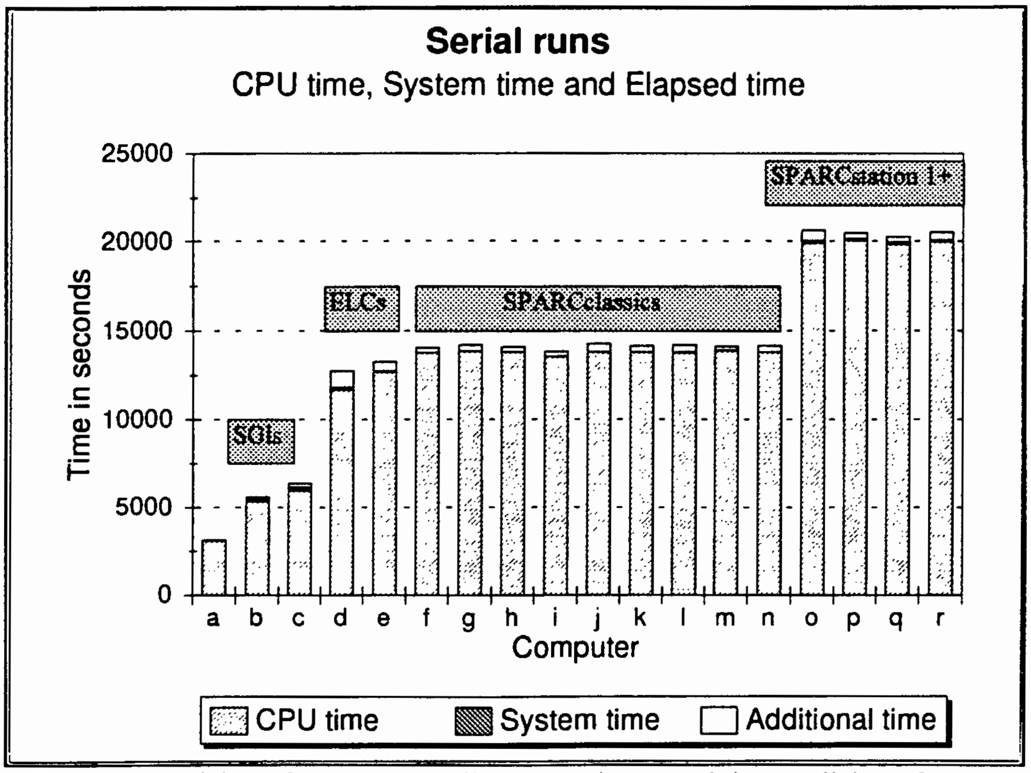

Figure 4: Serial performance of all workstations used in parallel performance experiments

The closeness of the elapsed times for machines with the same architecture can be seen more clearly in Figure 4, which is a stacked-bar graph showing the mean CPU time, the mean system time and the mean elapsed time for each processor. The bottom, and by far the largest section of the bar, shows the CPU time for each processor. For some processors (particularly **c** and **o**), a very small amount of system time is shown by the next section of the bar. For the remaining processors the amount of system time was too small to be seen on this graph. The top section of the bar shows the difference between the sum of the CPU and system time, and the total elapsed time, so the elapsed time for each processor is represented by the top of each bar. The small difference between the sum of the CPU and system time, and the total elapsed time, for most processors, illustrates the high utilization shown in Table II. This small difference is because all runs were run when the workstation was not otherwise being used.

The performance of each workstation is affected by three factors: the CPU performance, the memory access time, and other overheads, such as interference from other processes and communication overheads. The memory access time is a function of application behaviour, and of the memory hierarchy which includes TLB, cache and virtual memory. Figure 4 shows that machines with the same type of CPU all have similar performance. Even though not all machines in each group have the same amounts of memory, there is no discernible trend between the performance and the amount of memory.

Figure 4 shows that the 4 SPARCstation 1+ machines (**o**, **p**, **q**, **r**) gave virtually identical performance, regardless of the amount of memory, as did the 9 SPARCclassics (**f**, **g**, **h**, **i**, **j**, **k**, **l**, **m**, and **n**).

The two Sun ELCs (**d**, **e**) were similar in performance, but the one with 32 Mb memory was faster than the one with 16 Mb memory. It is not clear whether this difference in performance is due to the different amount of memory, but if the results on the Classics and SPARCstation 1+s are compared it seems that the amount of memory makes little difference to the performance for this program.

The Silicon Graphics Indigo2 (machine **a**) has by far the best performance, about 6 times as fast as the SPARCstation 1+s, 4 times as fast as the Classics, and 3.5 times as fast as the ELCs. Part of the reason for this good performance was the high processor speed. However, a contributing factor was the fact that this machine has a 1 Mb secondary cache, 8 kb instruction cache, and 8 kb data cache. The other two SGIs have no secondary cache, although they have a 32 kb instruction cache and a 32 kb data cache.

The Sun workstations appear to have 64 kb write-through caches. (See Appendix B for further hardware details.)

The other two SGIs (**b** and **c**) gave very similar performance, but the one with 16 Mb (**b**) performed better than the one with 24 Mb memory (**c**). This was probably due to bad network cables, since the SGI with 24 Mb (**c**) was moved along the network as discussed earlier. In the earlier tests these two SGIs gave almost identical performance, with the 24 Mb SGI (**c**) tending to have slightly better results.

The results in this section thus show that for serial performance the amount of memory makes little or no difference in the performance results obtained for machines of the same CPU, but with different amounts of memory. However, the results in Table I show that there was a difference of about 6.5% in the performance of the original serial program, and the parallel version of Cloud run on a single processor, on the Classics. Six of the nine Classics have 16 Mb of memory, and the remaining three have 32 Mb. The operating system running on these machines is Solaris, which takes up most of the 16 Mb memory available on most of the Classics. The parallel version of Cloud needs approximately fifty times the amount of memory needed by the original serial version. These different memory requirements account for the 6.5% difference in performance between the serial and parallel versions of Cloud, as shown in Table I, as the parallel version with its greater memory requirements will cause considerably more paging of the small amount of available memory.

In the parallel experiments with Cloud the slaves use about the same amount of memory as the original serial version, and the master needs about fifty times as much memory as the serial version. Thus, for each group of machines used, the master is the machine in that group with the largest amount of memory. Since the slaves need so much less memory it does not matter if some slaves have more memory than others. This is also confirmed by the results of the parallel experiments, where it was seen that for most runs all slaves did nearly exactly the same amount of work, regardless of the amount of memory.

5.2.1.1 Grouping the workstations into "homogeneous" groups.

This thesis is to study the factors affecting the performance of a parallel program on a heterogeneous network. However, it is difficult to evaluate the performance of a heterogeneous system, and to determine whether there is a linear improvement in performance as more processors are added. To understand a heterogeneous system one first needs to understand the performance of each individual component. It is much easier to evaluate the performance of a homogeneous system, and to identify any problems such as bad load balancing, and to establish whether linear speedup can be obtained. For this reason all the workstations in the heterogeneous group were first grouped into three approximately homogeneous sub-groups, by using the data summarised in Table II, so that the parallel performance results obtained for these sub-groups could be used to understand and calibrate the performance of the heterogeneous group.

The four SPARCstation 1+ machines (**o**, **p**, **q**, **r**) had nearly identical serial performance, and could be considered as homogeneous. The only difference was that one machine had more memory than the others. Therefore these four machines were grouped together to form the **SPARCstation 1+** group, with the machine with the most memory as the master.

Table II and Figure 4 show that the serial performance of the ELCs was fairly similar to that of the SPARCclassics (machines **g-n**), so that for the purposes of these experiments the ELCs and Classics could be considered to be the same type of machine. The ELC with most memory (**d**) took 90% of the mean serial time taken by the Classics, and the other ELC (**e**) took 94% of the mean serial time taken by the Classics. Thus these eleven machines were grouped into one so-called homogeneous group, the **ELC/Classic** group, with the faster ELC as the master, and the other ELC and the nine Classics as the slaves. To make the group as homogeneous as possible, the slave ELC was used only as the tenth slave. For all experiments with fewer than ten slaves only Classic slaves were used. Of these Classics, three had 32 Mb of memory, and the other six had 16 Mb. For these experiments this difference in memory did not matter, as the slaves did not need more than 16Mb of memory, and there was no noticeable difference in the performance of those slaves with less memory.

The three Silicon Graphics machines (**a**, **b**, **c**), were grouped together into the **SGI** group merely because all three were SGIs, and there were too few to make a truly homogeneous group. Only two of the three SGI machines had similar performance, so the fastest machine was used as the master, and the two similar machines were slaves.

Table III: Mean serial performance of homogeneous groups of workstations

Group of workstations	CPU time	System time	Elapsed time
9 SPARCclassics	13765 (141) [1.0]	8 (2) [24.9]	14105 (198) [1.4]
2 ELCs and 9 SPARCclassics	13474 (667) [4.9]	22 (42) [186.4]	13898 (486) [3.5]
SPARCstation 1+	19939 (104) [0.5]	94 (33) [35.5]	20464 (184) [0.9]

Table III shows the mean times for the **SPARCstation 1+** group, and **ELC/Classic** group, and also for the Classics alone. The results in this table are to illustrate the homogeneity of the **SPARCstation 1+** and **ELC/Classic** groups. The **SGI** group is not included in this table because this group was not homogeneous. These mean times were calculated by taking the best three runs for each machine in each group, and using these to calculate the mean and standard deviation of the group. The first value in each column is the mean time for the group, in seconds. The next value, in (), is the standard deviation for the group, in seconds. The last value in each column, in [], is this standard deviation, as a percentage of the mean time.

The data shown in Table III shows that the **SPARCstation 1+** group can be considered as homogeneous. The standard deviation in elapsed time for the four machines was only about 184 seconds, which was about 0.9% of the mean serial time for these machines.

The means of the 9 Classics and the 11-machine **ELC/Classic** group, given in Table III, are used to illustrate that the **ELC/Classic** group can be considered as virtually homogeneous. The mean of elapsed time of the Classics (14105s) differs from the mean of elapsed time of the 9 Classic and the 2 ELCs (13898s) by just over 200 seconds, which is less than 1.5% difference. The standard deviation in elapsed time of the Classics only, was also just less than 200 seconds, which was about 1.4% of the mean serial elapsed time for the Classics. When the mean of elapsed time of the 2 ELCs together with the 9 Classics was calculated, the standard deviation was just under 500 seconds, which was about 3.5% of the mean serial elapsed time for the group of 11 machines. These differences are small enough that, for the purposes of these experiments in a shared environment, the **ELC/Classic** group can be considered as homogeneous.

These means shown in Table III were the values used for the calculations of speedup in section 5.5. The SGI values used for the calculation of speedup in section 5.5 were taken from Table II.

5.2.2 Network performance

Although Ethernet bandwidth is specified to be 10 Megabits/second (see Appendix B), reported performance falls short of this, typically between 5 and 7 Megabits/second [Nana93][Gärt93][Nede93].

The data rate that could be expected for this experimental environment was established with a small *p4* program, which measured the round-trip time for a master to send a message to a slave, and for the slave to receive this message, and send it back to the master.

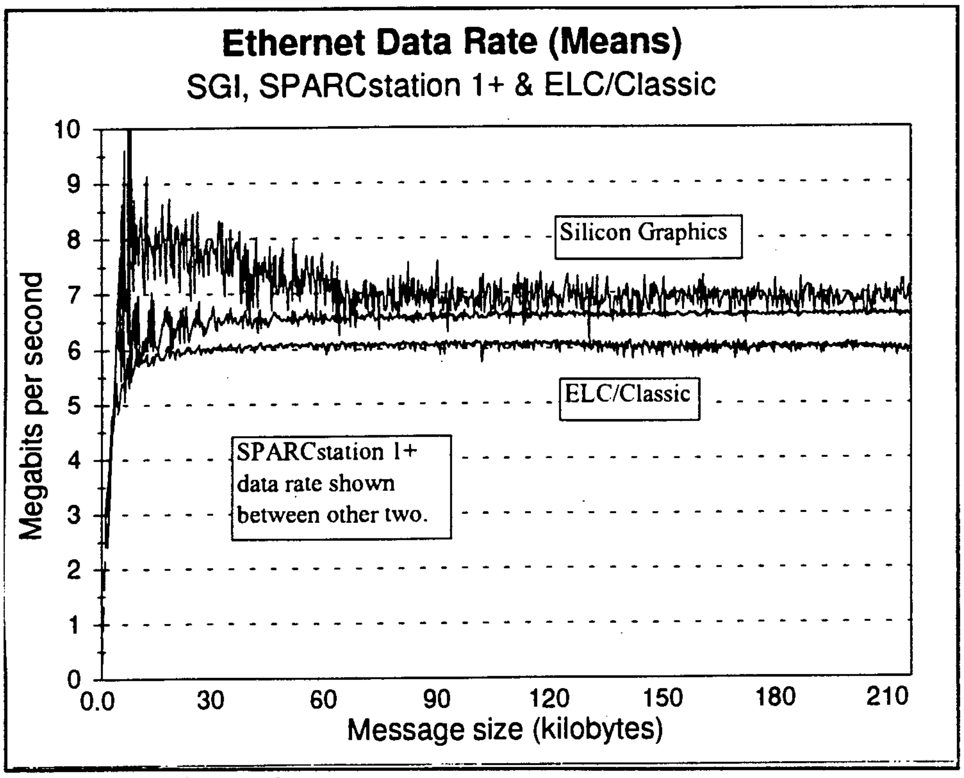

Figure 5: Ethernet data rate

As the largest message used in Cloud was just under 210 kb long this program sent messages ranging in size from 1 byte to 210 kb. There is a *p4* header of 40 bytes per message, which will be included in the first packet of a message. Since the maximum amount of data that can be sent in an Ethernet packet is 1500 bytes, it is not necessary to test messages of every length, as the results would vary minimally for messages of similar sizes. So the message size was incremented in steps of 300 bytes, making 5 increments per packet.

All experiments were run in the early hours of the morning when there were generally no other users of the network. The loop sending messages from 1 byte to 210 kb in steps of 300 bytes was repeated 15 times. The best 10 times, for each size message, were used to calculate the mean round-trip time for each message size. This value, together with the message size, were used to calculate the data rate for an uncongested network. These experiments were run using the master and one slave for each of the **SGI**, **ELC/Classic** and **SPARCstation 1+** groups.

Figure 5 shows the data rate achieved on a lightly loaded network for the SGI2 Extreme master and 1 SGI slave, for a SPARCstation 1+ master and 1 SPARCstation 1+ slave, and for an ELC master with 1 Classic slave. The size of the message in kilobytes is shown along the x-axis, and the data rate in Megabits/second on the y-axis.

The best data rate was achieved with the SGI2/SGI machines. Apart from the very high data rate of about 8.5 to 9 Mbits/second, for messages between 5 kb and 10 kb, the best data rate for the SGIs was nearly 8 Mbits/second for messages between 10 kb and 40 kb. This dropped to about 7.5 Mbits/second for messages between 40 kb and 60 kb, and then to 7 Mbits/second for messages longer than about 60 kb, and then remained more or less constant. The unexpectedly high data rates for messages less than 5 kb long may well be due to an inability of the timing routine to measure too short a time accurately.

For the **ELC/Classic** group the data rate was the slowest. For messages larger than about 10 kb the data rate is fairly constant at about 6 Mbits/second.

The data rate for the **SPARCstation 1+** group is slightly better than that for the **ELC/Classic** group, being just over 6.5 Mbits/second for messages larger than about 10 kb, and then remaining more or less constant at 6.5 Mbits/second for messages of 30 kb onwards.

In all three cases Figure 5 shows that the data rate for messages less than about 5 kb (about 3 packets) is poor, and that the best data rates are

Table IV: Comparison of *p4* and *PVM* data rates

Msg Size (kb)	Test - SGI	Test - ELC +Classic	Test - SPARC 1+	**ANL - p4 SPARC 10**	**PVM -**
4	6.5	5.1	6.2	2.4	4.4
16	8.6	5.8	6.4	2.7	5.2
20	8.2	6.0	6.4	2.7	-
64	7.3	6.1	6.5	-	6.2

achieved for messages bigger than about 10 kb (about 6 packets). Thereafter the data rate remains fairly constant regardless of message size, although a slight drop in the rate is observed for the **SGI** machines for messages longer than 60 kb (about 40 packets) .

The data rates achieved with these three groups of machines are compared, in Table IV, with those obtained by the developers of *p4* at ANL, using 2 SPARCstation 10 machines connected by Ethernet [Butl94], and those obtained by the developers of *PVM*, using 2 workstations connected by Ethernet [Sund94]. It is not known which workstations were used in the *PVM* experiments.

PVM is reputed to handle communication more efficiently, in that its message-passing is not blocking, and processing can continue while a message is being sent, as communication is handled by daemons working in the background. However, if one examines the data presented in the last column of Table IV it appears that, depending on the workstations used, *p4* is not inferior to *PVM*, although the data rate for *PVM* does appear to improve as the messages get longer. This is probably because in *PVM* the process is not delayed by message-passing.

The data rates achieved at ANL [Butl94] (5th column of Table IV) are surprisingly low, especially as these were probably measured using the program **systest**, which is supplied as a sample program with the *p4* library, and which was the basis of the program I used for measuring the data rates for this thesis.

This data rate measured at ANL seems more likely to be that obtained when other users were using the network, or there may be a mistake in the calculations made at ANL, with the data rates shown in [Butl94] being calculated from the full time for the round trip instead of half the time. If this mistake was made, then the rates would be 4.8 and 5.4 Mbits/sec, which is comparable with the results obtained in my experiments.

5.3 Speedup and efficiency

The overall elapsed time is of fundamental importance in measuring the performance of a parallel program, as users are primarily interested in how long it takes to achieve a solution. The problem with a conventional elapsed time graph is that it is difficult to see how well the system is performing, as the values for higher numbers of processors receive little visual space. It is important to understand whether the performance achieved is optimal, and whether or not there is room for improvement.

Speedup and efficiency are commonly used to evaluate parallel performance in this way, by comparing the parallel performance to the serial capabilities of the processors used. Poor speedup and efficiency may indicate such problems as inefficient load balancing or synchronization, and that redesign may improve performance. Good values for speedup and efficiency show that the performance is close to optimal, and there is not much that can be improved. For a parallel program to be worthwhile, substantial benefit should be obtained by using more processors, and speedup and efficiency are means of measuring this benefit.

However, there are a number of difficulties inherent in the use of speedup and efficiency to evaluate parallel performance, and some of these will be discussed in this section. In particular, this section shows that the conventional means of calculating speedup and efficiency are not directly appropriate for evaluating the performance of a heterogeneous system, and that it is actually very difficult to evaluate parallel performance on a heterogeneous system.

Speedup for homogeneous parallel machines is generally defined as the ratio of the sequential execution time of a program on a single processor, divided by the execution time of the program on a number of processors. Yet, even this simple definition raises the question of what serial program should be measured on the single machine. Should it be the best serial version available, or the parallel version running on a single machine? As described in section 5.2.1, the serial performance of Cloud, used to determine speedup in this dissertation, was the elapsed time of the parallel program running on the single processor, because this was sufficiently close to the elapsed time of the original serial program, and it was easier to use the same program for all experiments.

A complementary measure of performance to speedup is efficiency, which is a means of measuring how efficiently the processors are utilised. This is defined, for a homogeneous system, to be the ratio of the speedup divided by the number of processors. Ideally efficiency should be 1, as the perfect speedup should be equal to the number of processors. However, as processors are added, it is likely that the efficiency will decrease as a result of increased overheads [Eage89].

Ideally, performance should improve linearly, as processors are added. However, there is even confusion in what exactly is meant by "linear speedup". Is speedup linear only when the efficiency remains at 1 as the number of processors increases (perfect speedup), or can speedup be considered linear when speedup is directly proportional to the number of processors, but the efficiency is less than 1 [Eage89]? Since true linear (perfect) speedup can rarely be achieved, most researchers consider speedup to be linear if there is a constant rate of improvement in speedup as processors are added.

Then it is commonly known that slow machines often exhibit better speedup than faster machines [Sun91]. Since communication costs may depend more on the communication medium, such as an Ethernet network, than on the communicating machines, a slow machine may have a better computation/communication ratio than a fast machine, thus exhibiting better speedup.

Amdahl's Law states that speedup is limited by the serial component of the application. This serial component may be composed of many factors, one of which is likely to be the time to access disk, or slow memory. However, as more processors are used there is generally more memory available, and some of this may be caches. So it may be that increasing resources, such as memory, could lead to superlinear speedup as more data can be in cache and memory at the same time, thus reducing the time needed for accessing disk [Gust88][Fisc91][Dona94][Sing94]. On the other hand, increasing the number of processors increases the communication overhead, thus reducing the efficiency, and the potential speedup.

All these points make it difficult to judge whether speedup and efficiency are portraying good performance or not. These difficulties are compounded when it comes to measuring the speedup and efficiency of a heterogeneous system, such as that used in the experiments described in this dissertation.

There is considerable discussion on just how to measure the speedup of a heterogeneous system, but this has not yet been consistently defined [Dona94]. In a heterogeneous system the processors have different performance capabilities. In addition, some processors may have special capabilities, such as graphics or floating point chips, which enable them to perform certain tasks much faster than processors without these capabilities. It is also possible that some applications cannot even be run on some serial processors, which do not have the required capabilities. Another factor to be considered is that if the system is not dedicated, then the performance of processors will be impacted by other processes, and users external to the application.

It is easy enough to measure the elapsed time of a parallel program on a heterogeneous system, but with what should it be compared to determine its speedup? If the execution time for the program on the slowest processor is used, then artificially high speedups will be obtained, and these may even be superlinear. On the other hand, if the time of the fastest processor is used, then the speedup will appear low.

Donaldson et al suggest that the serial elapsed time of the fastest processor should be used for calculating the speedup of a heterogeneous system [Dona94]. On the other hand, Kumar et al use the sum of the time spent by all the processors on useful computation, as the single processor time in the speedup calculation [Kuma94]. Schnekenburger goes further in considering the problem of determining the efficiency of parallel programs in heterogeneous, non-dedicated, multi-tasking environments, with dynamically changing service requests of external tasks, and arbitrary scheduling strategies [Schn93]. Schnekenburger develops the concept of *dynamic efficiency,* which takes into account the effect of the service requests of external tasks, which affect the service rate of resources. He thus calculates for a resource, the time spent in serving the application being studied, the time spent servicing other applications, and the time spent idle, and uses this as a term in his equation. Since there exist an immense number of external loads it is impossible to compare different results. He therefore takes this concept further by using a stochastic process to determine the *stochastic efficiency.*

Donaldson's proposal to use the time of the fastest processor would show very poor speedup in the system studied for this thesis, as the fastest machine is up to 6 times as fast as the slowest machine. Kumar's method does not include the cost of overheads in the calculation, and, as has been shown in section 5.1, these could be highly significant. It is quite difficult to determine some of the terms in Schnekenburger's equation, and this may involve the use of sampling techniques such as found in several existing performance analysing tools [Schn93].

In calculating heterogeneous speedup, one must consider exactly what one is trying to show. In essence, speedup is a measure to show how performance is improved by using more processors. If the parallel performance is compared with the single processor time of the fastest processor, the speedup shows the number of machines, with the same capabilities as the fastest processor, that would give the result achieved. But if the system consists of widely disparate processors, as in the system studied in this thesis, is this relevant?

Correspondingly, the efficiency of a homogeneous system can be easily understood, as the ratio of the speedup divided by the number of processors. However, this calculation of efficiency is not necessarily relevant for a heterogeneous system. If the speedup of a heterogeneous system is calculated by comparing parallel performance to the serial performance of a particular machine, such as the fastest or slowest processor, then a value for efficiency that is derived by dividing this speedup by the number of processors is meaningless.

For example, when the parallel performance for 14 processors is compared to the serial performance of the fastest SGI2 processor, giving a speedup of 3.3, the corresponding efficiency, calculated as this speedup divided by the number of processors, is 0.3. However, this is not a true reflection of parallel performance, since for Cloud all processors are in fact working to a high efficiency. This suggests that using this formula for the calculation of efficiency of a heterogenous system is not valid.

On the other hand, if the speedup of the heterogeneous system is calculated by comparing the parallel performance with the mean of the serial performances of all the machines used, this gives a speedup of 11, for 14 processors. Dividing this speedup by the number of processors, will give a corresponding efficiency of about 0.8. This speedup and efficiency give a better indication of the improvement in performance gained by adding more processors.

Thus, if the parallel performance of a heterogeneous system is to be compared with the performance of a particular machine, then the serial performance of that machine should be used to calculate speedup. On the other hand, if, as in this study, the parallel performance is to be compared with the actual performance capabilities of the machines used, then the mean of the serial performances of all the machines used, should be used for the calculation of speedup, and thus efficiency, as this gives a more accurate measure of the improvement in performance through parallelization. These points are illustrated by the results presented later in this chapter.

5.4 Alternative ways of evaluating parallel performance

Graphs of elapsed time, speedup and efficiency are common ways of presenting parallel performance. However, as described in section 5.3, and illustrated in this section, and section 5.5, there are many problems associated with this, and particularly when evaluating the performance of a heterogeneous system.

Crowl has critically evaluated several methods of portraying parallel performance, and suggests some alternatives to the conventional elapsed time and speedup graphs [Crow94]. Most of the discussion in this section can be related to Crowl's paper.

Two sets of results are used to illustrate Crowl's arguments. These are the results for the 40000 photons task size, run on the **ELC/Classic** group, and the results for the 20000 photons task size, run on the group of 18 heterogeneous machines. These examples were chosen because these task size gave the best performance for these two groups, and these two groups were the largest groups of near homogeneous and heterogeneous machines. These sets of results are used for all the graphs in this section.

The results for a homogeneous group were used because it is easier to understand Crowl's points when examining the results for a homogeneous group, as it is easy to see if there is a constant improvement in performance as more processors are added. The results for the heterogeneous group then illustrate how Crowl's proposals are especially valid for evaluating the performance of a heterogeneous system. Crowl's proposal of linear speed is used in section 5.5 to present the performance results of the experiments with Cloud.

5.4.1 Conventional elapsed time graph

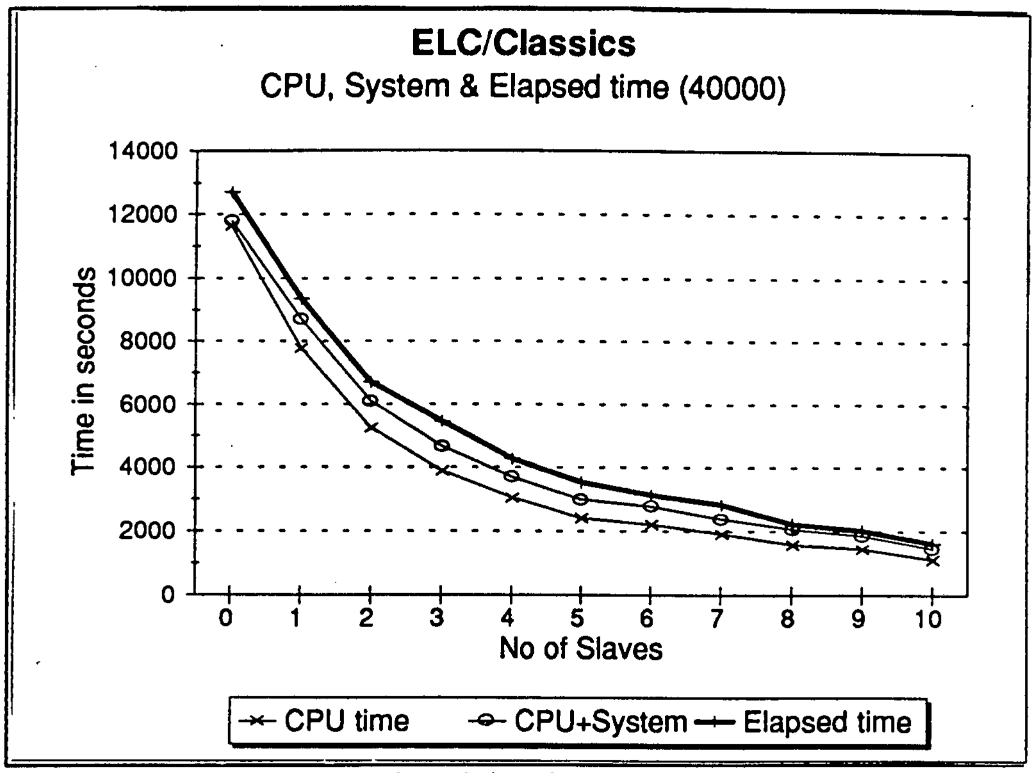

Figure 6: CPU, system and elapsed time for **ELC/Classic** group - 40000 photon task size

Figure 6 shows a conventional elapsed time graph for the 40000 photons task size, for the **ELC/Classic** group. CPU time and system time are also shown for interest. The system time is the difference between the line showing the sum of the CPU and system time, and the line showing the CPU time. The elapsed time is very close to the sum of the CPU and system time, thus indicating that, for these runs, the system could be considered as dedicated. The number of slaves is shown along the x-axis.

This graph shows a consistent decrease in elapsed time, as the number of slaves is increased. However, on a graph in which a linear decrease in elapsed time is represented by a curve rather than a straight line, the results for higher numbers of processors have little visual space. This makes it extremely difficult to see whether such a graph is showing improving or deteriorating performance as processors are added.

5.4.2 Conventional speedup graph

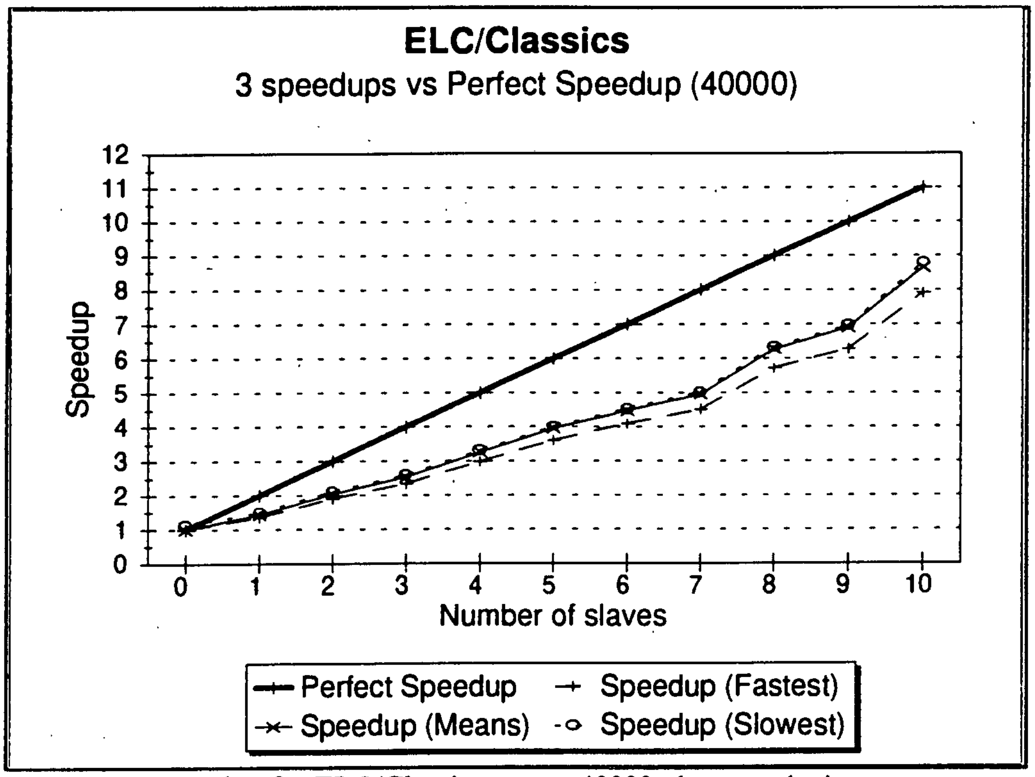

Figure 7: Speedup for **ELC/Classic** group - 40000 photon task size

Speedup can be used to illustrate whether the improvement in performance is linear. The speedup, for the same data as in Figure 6, can be seen in Figure 7.

Three different methods of calculating speedup are shown, to illustrate one difficulty of using speedup as a measure of performance. These are the speedups showing the parallel performance compared to the serial performance of the fastest processor (the ELC master), the parallel performance compared to the serial performance of the slowest processor (mean of the Classics), and the parallel performance compared to the mean serial performance of all the processors used (9 Classics and 2 ELCs).

Figure 7 shows that the speedup where the parallel performance is compared to the mean of the serial performance of the Classics, and the speedup compared to the mean of all the processors used, are virtually identical, thus confirming that the performance of the ELCs and of the Classics are close enough for this group to be considered homogeneous. As is expected, the speedup, when the parallel performance is compared to the fastest machine (ELC master), is not as good as the other speedups.

All three speedups show that there is a near linear increase in speed up to 9 slaves (10 processors), with a slight decrease in performance for 7 slaves. After that there is an improvement in the speedup for 10 slaves. This improvement in speedup for the tenth slave is because this slave is an ELC, which is slightly faster than the other 9 Classic slaves. The reduction in performance for 7 slaves is probably because of uneven distribution of work among this number of processors, leading to increased idle time for some slaves, and a corresponding reduction in efficiency.

5.4.3 Conventional efficiency graph

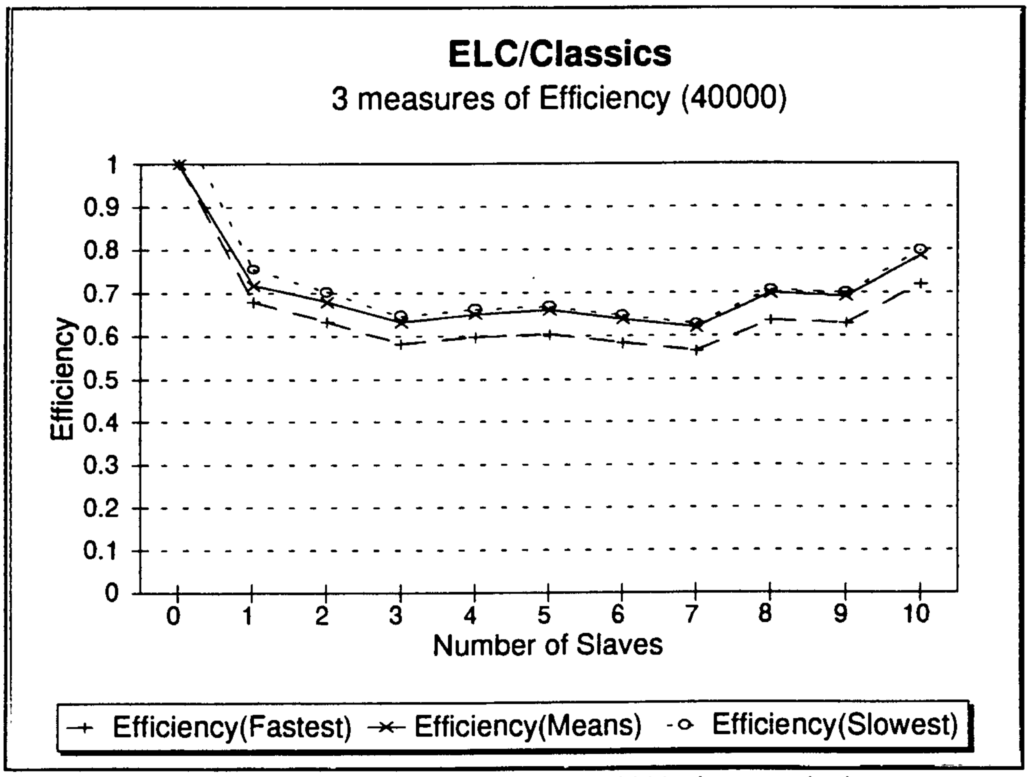

Figure 8: Efficiency for **ELC/Classic** group - 40000 photon task size

The efficiency for the same data as in Figure 6 and Figure 7 is shown in Figure 8, with the three measures of efficiency shown corresponding to the three calculations of speedup shown in Figure 7.

Figure 8 shows that there is a noticeable difference in the efficiency as compared to the fastest machine, and as compared to the slowest machine. This indicates how easily calculations of efficiency may be misleading, and that it is important to state clearly how efficiency is calculated, and what it means.

The graph of efficiency shows variation in performance more clearly than the speedup graph in Figure 7. The higher efficiencies for 1 and 2 slaves are because, for these numbers of slaves, the master does up to half the Monte Carlo work, and since the master has no communication overhead the overall efficiency is better. From 3 to 7 slaves the efficiency remains fairly constant. For 8 and 9 slaves there is an improvement in the efficiency. This may possibly be because it is more efficient for the master to receive 8 or 9 sets of results before changing back to Monte Carlo work, than when for fewer numbers of slaves it receives 4 or 5 sets of results, and then changes to Monte Carlo work. Thus, for higher numbers of slaves, the efficiency seems to improve, and this may be due to the master being either consistently busy on communication, or on Monte Carlo work, and changing between these less often, thus resulting in less paging. The higher efficiency for the tenth slave is because this is the ELC slave, which is faster than the other 9 slaves.

5.4.4 Linear speed graph

Because of the problems associated with speedup and efficiency, especially in the evaluation of the performance of a heterogeneous system, Crowl suggests a graph of **linear speed** as an alternative to speedup.

A linear speed graph shows the inverse of elapsed time, plotted against the number of slaves. This graph is visually similar to speedup, but it is independent of hardware and other considerations, and is therefore not a derivative measure, unlike conventional speedup which is related to the serial performance of individual processors. This makes linear speed particularly useful for the evaluation of the performance of a heterogeneous system. This section first illustrates the use of a linear speed graph for a homogeneous system, and then for a heterogeneous system.

Linear speed is a way of showing how much work is completed in unit time. In the experiments in this study the total amount of work for all runs was the same, so linear speed could be computed simply as the inverse of elapsed time, without calculating how long it took for each separate solution. However, linear speed is a good way of comparing runs where different amounts of work are done, by comparing how many solutions are obtained in unit time.

For linear speed graphs an increase in the gradient of the graph indicates an improvement in performance as processors are added, and a

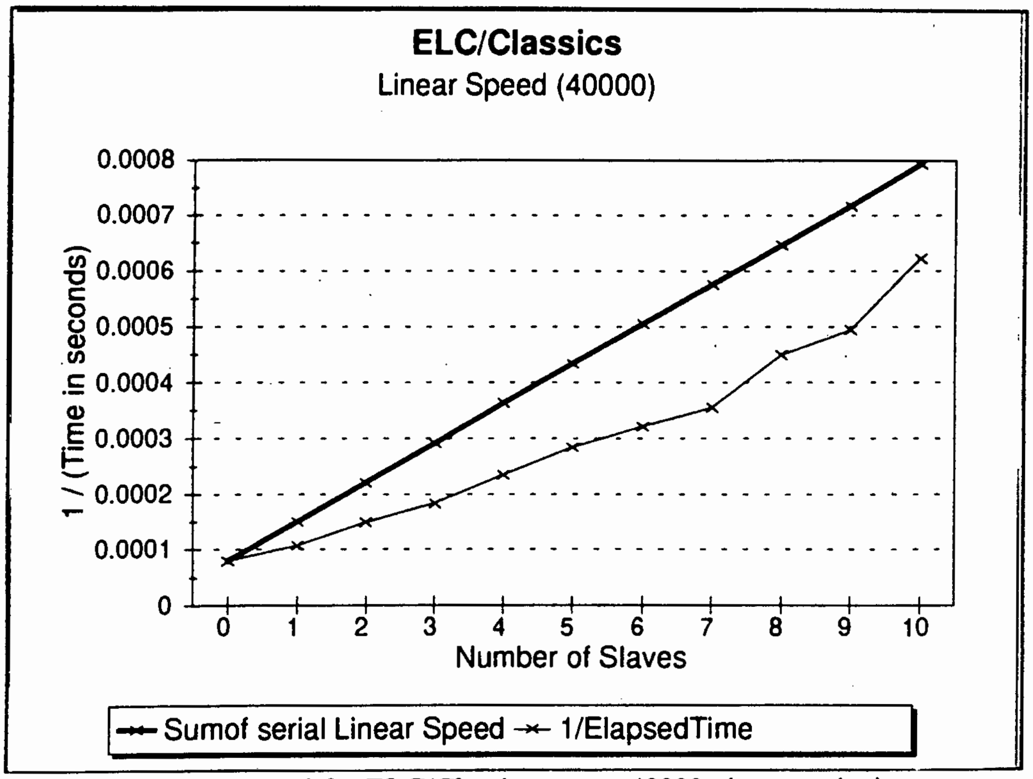

Figure 9: Linear speed for **ELC/Classic** group - 40000 photon task size

decrease in the gradient shows a deterioration in performance.

A graph of linear speed is a good way of showing how performance changes as further processors are added. However, it would be useful to have some way of establishing whether the performance is the best that could be achieved, or whether there is room for improvement.

We propose that, as an extension of Crowl's work, a line showing the sum of the serial linear speeds should be used to indicate the best performance possible for that system, in the same way that the line depicting perfect speedup is used to indicate perfect performance on a conventional speedup graph. The serial linear speed is the amount of work that can be performed by a processor in unit time. Therefore, the total performance capability of a system, whether homogeneous or heterogeneous, can be expressed as the sum of the serial linear speeds of the processors comprising the system.

Figure 9 shows the linear speed graph for the same data as the graphs in Figure 6 and Figure 7. Also shown in Figure 9 is a line depicting the sum of the linear speeds of the serial runs for the processors used in the experiment. The changes in performance discussed in sections 5.4.2 and 5.4.3 can be seen clearly in Figure 9, where the changes in the gradient of the linear speed graph show the changes in performance as processors are added. The closeness of the linear speeds achieved to the line showing the sum of the linear speeds of the serial runs indicates that good performance was achieved. This is discussed further in section 5.4.5. Since this graph is for a homogeneous group the line depicting the "perfect linear speed" is a straight line, which will not be the case for a heterogeneous system, as will be shown in Figure 11.

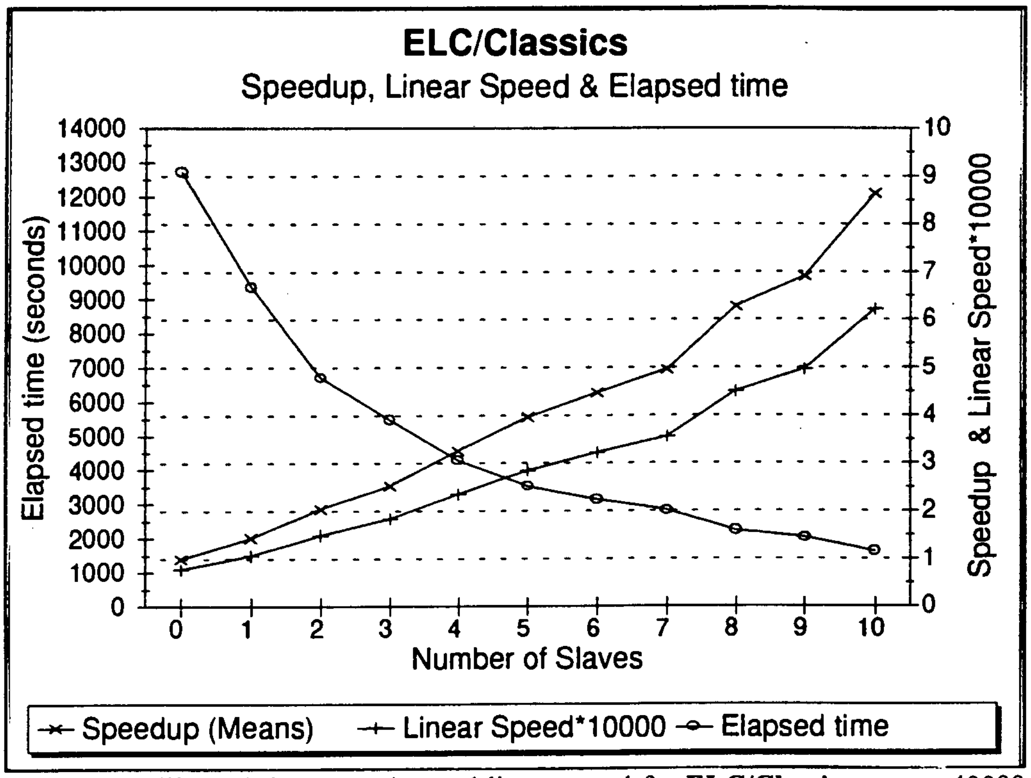

Figure 10: Elapsed time, speedup and linear speed for **ELC/Classic** group - 40000 photon task size

The three different ways of showing parallel performance with elapsed time, speedup (with speedup calculated according to the mean serial performance of the processors used), and linear speed graphs are compared in Figure 10, which combines the graphs for the same data shown in Figure 6, Figure 7 and Figure 9. The left-hand y-axis in Figure 10 depicts the elapsed time in seconds. The values of linear speed were multiplied by 10000 so that the graph of linear speed could be shown on the same axes (right-hand side y-axis) as the speedup graph.

Figure 10 shows that both the speedup and linear speed graphs show changes in performance more clearly than the elapsed time graph. This graph also shows that this linear speed graph has the same shape as a speedup graph, so that it is valid to use linear speed instead of speedup to illustrate performance. Note that it is the shape of the graph that is important, and that the overall slope is dependent on the scale used. The different scales used in Figure 9 and Figure 10 are the reason that the same linear speed graph has a different slope in the two figures.

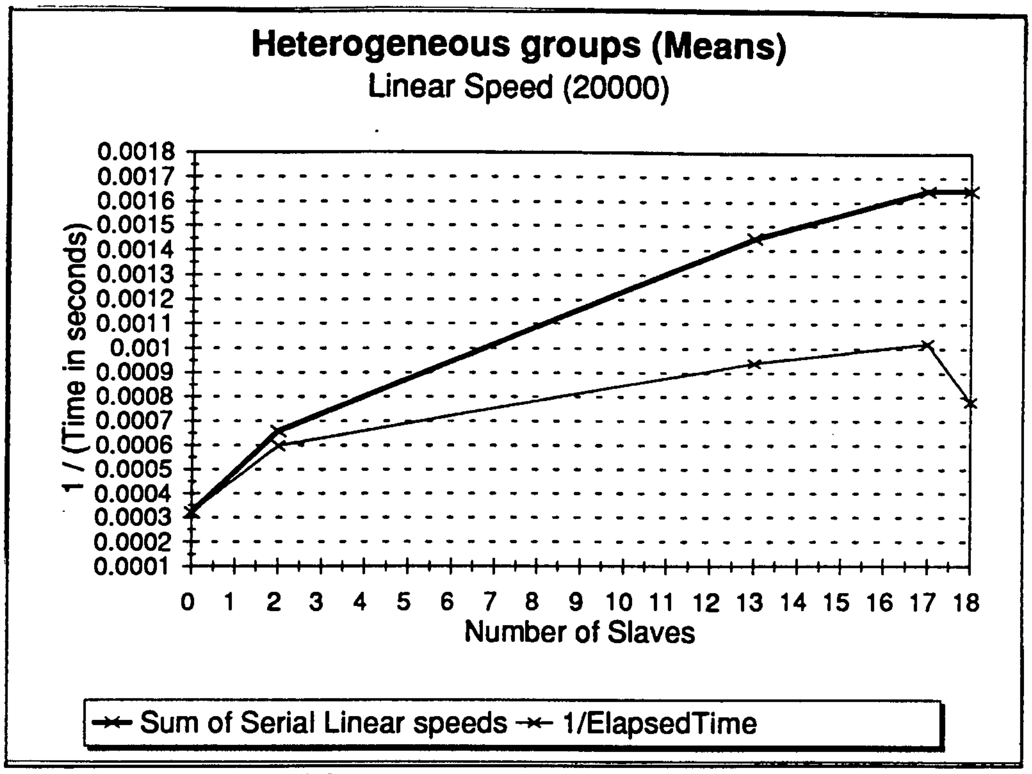

Figure 11: Linear speed for **heterogeneous** groups - 20000 photon task size

Linear speed is particularly useful for portraying the performance of a heterogeneous system as shown, in Figure 11. This graph illustrates the linear speed for a system of 18 processors, where values are shown on the graph for 2 slaves (2 Indigo slaves), 13 slaves (2 Indigo + 2 ELC + 9 Classic slaves), 17 slaves (2 Indigo + 2 ELC + 9 Classic + 4 SPARC 1+ slaves), and 18 slaves (2 Indigo + 2 ELC + 9 Classic + 4 SPARC 1+ slaves, plus an additional slave on the same processor as the master). The results are shown for the 20000 photon task size, which is the task size that gave the best performance results.

In Figure 11 the line showing the sum of the serial linear speeds of the processors used gives a good indication of the potential total performance in this heterogeneous system, according to the serial capabilities of the processors used. The steep slope of the line showing the sum of the serial linear speeds between 0 and 2 slaves indicates the good performance of

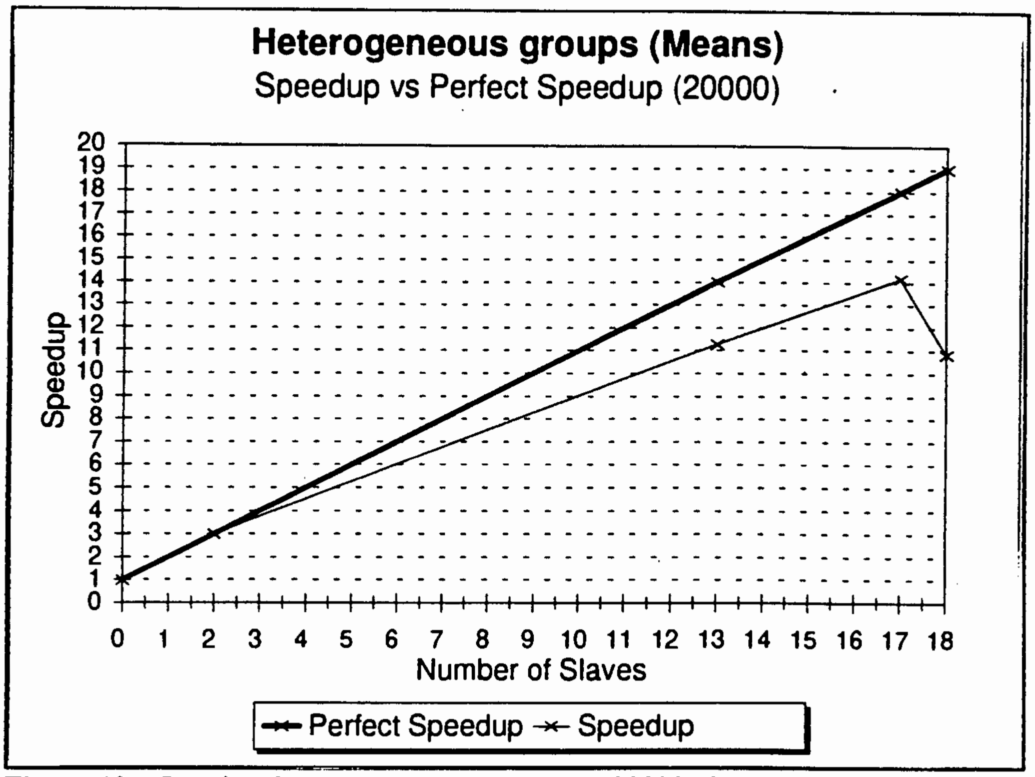

Figure 12: Speedup for **heterogeneous** groups - 20000 photon task size

the two Indigo slaves. The slope of this line then decreases between 2 and 13 processors, when the group of 11 ELC and Classic slaves are added. This shows that the ELC and Classic slaves have less performance capability per processor than the two Indigo slaves. The slope of the graph decreases again between 13 and 17 slaves, when the four slow SPARCstation 1+ slaves are added, thus showing the limited processing potential of the SPARCstation 1+ slaves. This line then remains level for 17 and 18 slaves, because the 18th slave is run on the same processor as the master, and eighteen processors are used for both 17 and 18 slaves.

The line showing linear speed (inverse of elapsed time) shows that, for this group of 18 processors, very good performance is achieved for the two Indigo slaves, as the slope of the line showing linear speed is very close to the line showing the sum of the serial linear speeds. When the 11 ELC/Classic slaves are added (slaves 3 to 13) the improvement in performance by adding more processors is not so good, and the line showing linear speed diverges from that showing the potential performance. And when the four slow SPARCstation 1+ machines are added (slaves 14 to 17) the gradient of the line showing linear speed decreases even more, indicating that the SPARCstation 1+ machines do not contribute much in processing potential. There is a distinct drop in the performance when the extra (18th) slave is run on the same processor as the master processor, since no extra processors are added, just that one is shared between two processes.

The deceptive nature of a graph showing speedup for a heterogeneous system can be seen if the speedup graph in Figure 12 is compared to the linear speed graph in Figure 11. Figure 12 shows the speedup for the same group of 18 processors, with the speedup calculated as compared to the mean serial performance of the processors used. Figure 12 shows the perfect speedup as a straight line. The graph showing the observed experimental speedup suggests that the improvement in performance was near linear, but this not a true representation of the actual performance, as has been shown by the linear speed graph in Figure 11, and will be clearly shown in section 5.5.4.

The graphs in Figure 11 and Figure 12 show clearly that speedup is not a valid means for evaluating the performance of a heterogeneous system, and that linear speed will give a better indication of the actual performance.

5.4.5 Linear efficiency

A linear speed graph gives a good indication of how well a system is performing. However, the actual percentage utilization of the processing power of a heterogeneous system is not immediately obvious from a graph of linear speed.

We therefore propose, as a further extension to Crowl's work, that the concept of **linear efficiency** should be used to show this percentage utilization, where the linear efficiency is calculated as the linear speed divided by the sum of the linear speeds of the processors used. This concept of **linear efficiency** is thus analogous to the conventional measure of efficiency for a homogeneous system.

Figure 13 shows the linear efficiency for the same group of processors as the linear speed graph in Figure 11. Here the deterioration in efficiency as compared to the total potential performance is clearly visible, with the highest efficiency of just over 0.9 for the two Indigo slaves, deteriorating to 0.65 when the ELC and Classic slaves are added, and again to 0.6 when the SPARCstation 1+ slaves are added.

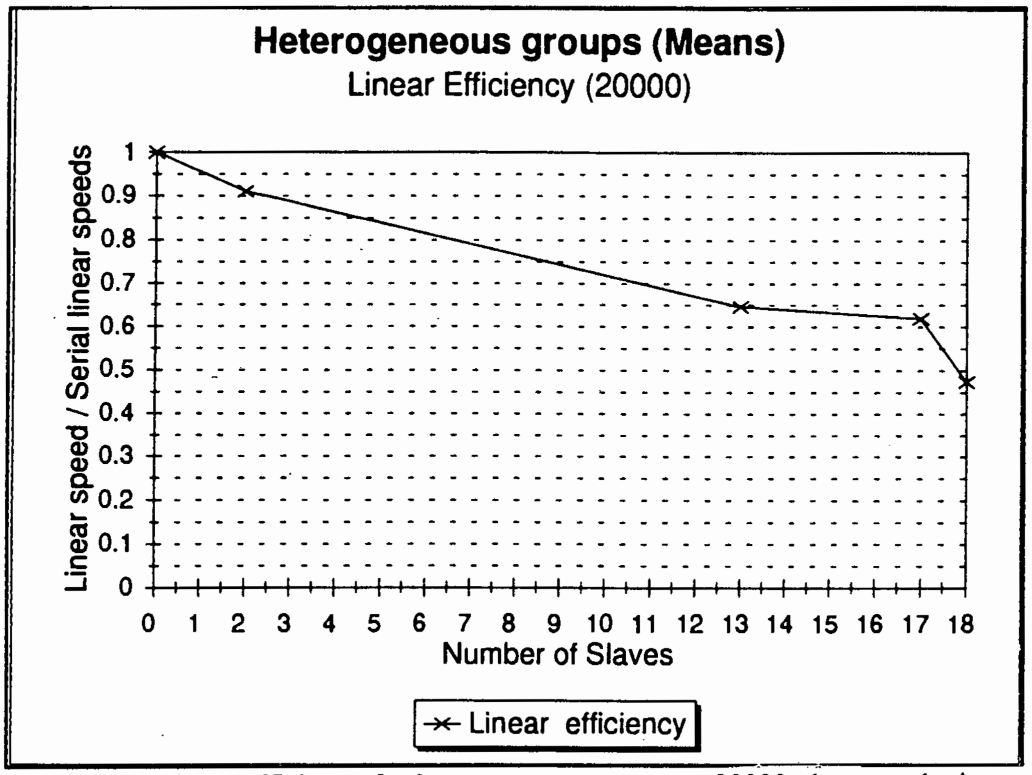

Figure 13: Linear efficiency for **heterogeneous** groups - 20000 photon task size

5.4.6 Log-time graphs and Log-speed graphs

Normal x-y plots, such as the elapsed time graph in Figure 6, do not show qualitative changes well, as performance for large numbers of processors receives very little visual space. Also a constant improvement in performance is seen more clearly as a straight line on paper, and conventional elapsed time graphs are curves, on which it is difficult to see changes in performance.

Logarithmic graphs can be beneficial in these circumstances, as they give more visual space to the results for the higher number of processors, and may even accentuate changes in performance. Crowl therefore suggests two further ways of showing performance. The first of these is a Log-log time plot, where the log (to the base 2) of the time is plotted against the log (also to the base 2) of the number of processors. A similar graph to the Log-log time graph is a Log-log speed graph, where the log (to the base 2) of the speed (inverse of time) is plotted against the log (also to the base 2) of the number of processors. The improvement in performance, as the number of processors increases, is shown by the changing slope of the graph as the number of processors increases. For further information on Log-log time graphs and Log-log speed graphs, refer to Crowl [Crow94].

For the experiments described in this dissertation these last two graphs show little that is not also apparent in the Linear Speed graph. However, all three graphs proposed by Crowl are good ways of showing qualitative change. For example, if the program began slowing down at a certain number of processors, then this would be difficult to see on a conventional elapsed time graph, but is easy to see on any of the graphs proposed by Crowl.

5.5 Performance of Cloud on groups of homogeneous and heterogeneous workstations

It is difficult to understand the performance of a heterogeneous system, because of the widely varying performance capabilities of the processors used. Thus, it is first necessary to evaluate the performance of the homogeneous sub-groups, since if good performance can be shown for all homogeneous sub-groups this indicates that the performance of the combined heterogeneous group is close to optimal. Also, most researchers typically report speedups, so if linear speedups can be shown for a homogeneous group, then in some sense the results are comparable with other researchers.

This section presents the performance results obtained from testing Cloud on various groups of workstations. This performance is shown using graphs of elapsed time, speedup, efficiency, linear speed and linear efficiency. Conventional graphs of elapsed time, speedup, and efficiency are used to present the results of the homogeneous sub-groups. However, as speedup and efficiency have been shown to be inappropriate for evaluating the results of a heterogeneous system, linear speed and linear efficiency are used to evaluate the results of the combined heterogeneous group. The results for the homogeneous sub-groups are shown first, and then the results of the heterogeneous group consisting of the combination of these sub-groups.

The data presented in section 5.4.2 has illustrated, that for this study, the speedup should be calculated by comparing the parallel performance to the mean of the serial performances of the actual machines used for that run: for example, the mean of the serial performance of the ELC master and 1 ELC slave and 9 Classic slaves for the 10-slave runs; the mean of the serial performance of the ELC master and 9 Classic slaves for the 9-slave runs; and so on. Therefore all the speedups presented in this section were calculated in this way. In all cases the means were calculated from the results of the best 3 runs for each machine.

For each group of slaves, the graphs show the results grouped first by the size of the sub-tasks (5000, 10000, 20000, 30000, 40000, 60000 and 120000 photons), and within these groups by the number of slaves for the run. Thus the graphs for all task sizes are shown on the same graph.

5.5.1 Performance of "homogeneous" Sun ELC/Classic group - 11 machines

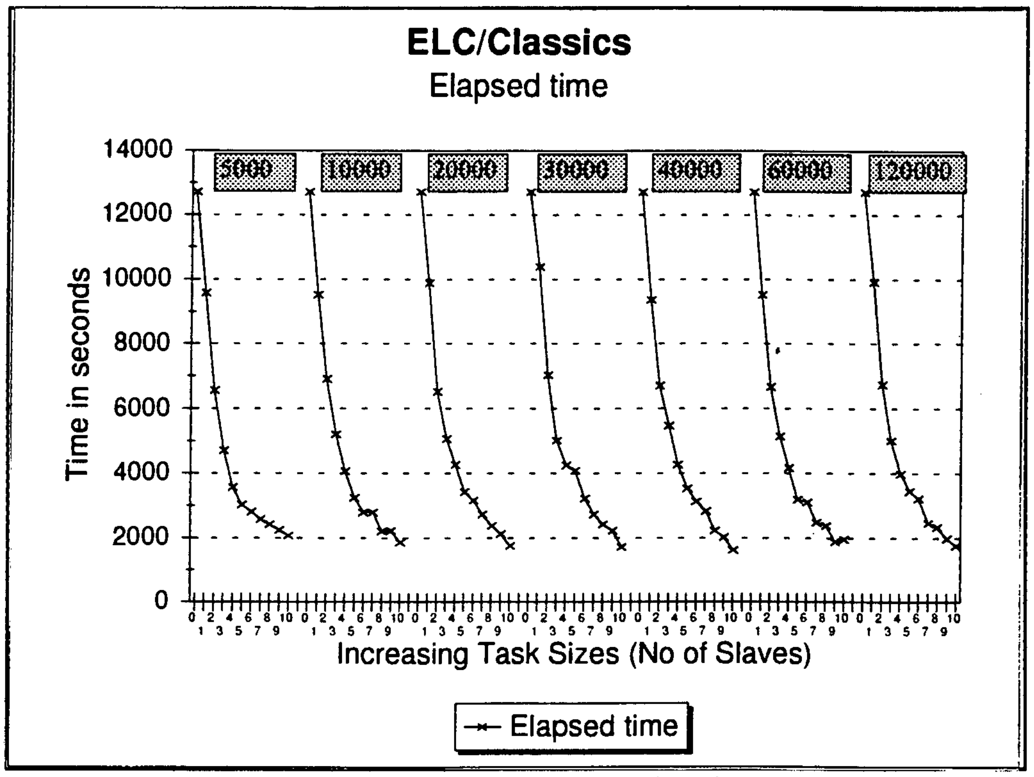

Figure 14: Elapsed times for "homogeneous" **ELC/Classic** group (grouped by task size)

Figure 14 shows the elapsed times for the experiments with an ELC master, with from 0 to 10 Classic and ELC slaves, with the results grouped within sub-task size. For each task size, the sub-graph shows the times for runs from 0 to 10 slaves.

The size of each sub-task is indicated in each header block, with the number of slaves increasing from left to right within each sub-task size group. Figure 14 shows that the times for all sub-task sizes are similar, with slightly better times for a sub-task size of 40000 photons, and with the overall shortest elapsed time for a task size of 40000 photons and 10 slaves (11 processors in all). In all cases, there is a regular decrease in total elapsed time as the number of slaves is increased.

The speedup for the **ELC/Classic** group, as compared to perfect speedup, is shown in Figure 15. Each curve refers to one task size as shown on the horizontal axis. Each tick on the horizontal axis indicates an increase in the number of slaves. For each task size the speedup is shown for runs from 0 slaves (serial run, speedup of 1) to 10 slaves, as compared to the perfect speedup.

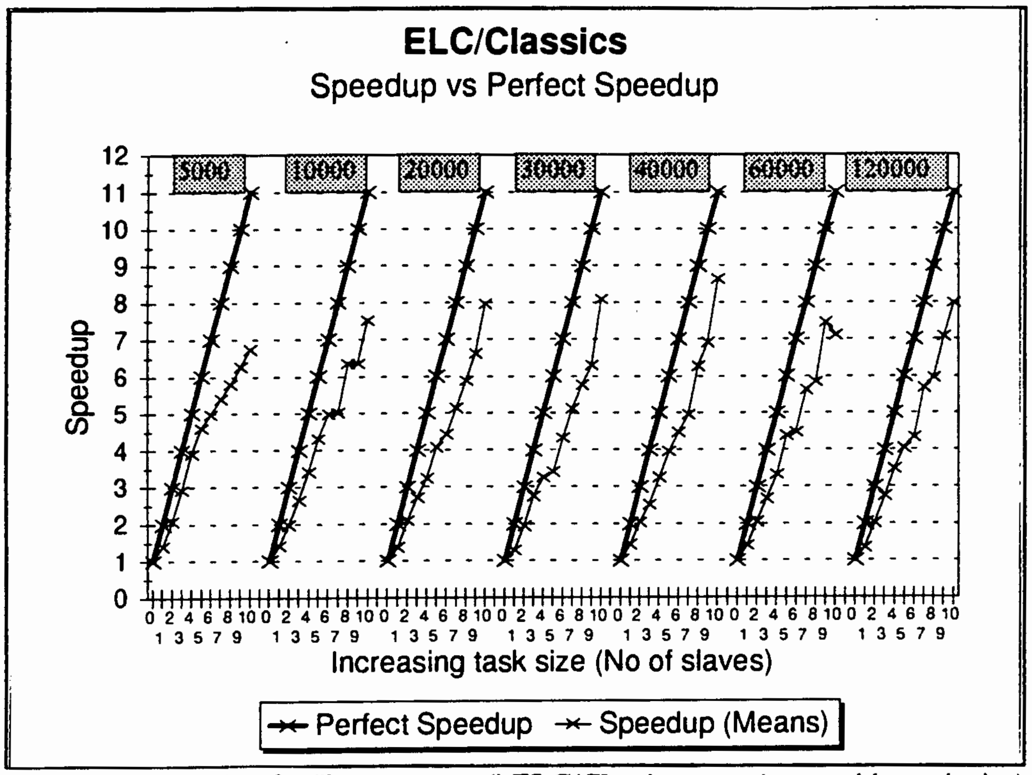

Figure 15: Speedup for "homogeneous" **ELC/Classic** group (grouped by task size)

Figure 15 shows that initially the 5000 task size gave the best speedup, with a speedup of nearly 5 for 6 processors. However, from 6 slaves onwards (7 processors) the speedup deteriorates. This is because, for few slaves, the master spends less time on communication, and therefore does far more Monte Carlo work, with no communication overhead, so overall efficiency is good. As the number of slaves increases, the master spends more time on communication, and less on computation, so proportionately more work is done by the slaves. For every task done by the slaves there is a communication overhead, so overall efficiency is reduced, and the speedup is not as good as for fewer slaves.

Figure 15 also shows that the best overall speedup is for the task size of 40000 photons, with 10 slaves, with a speedup of nearly 9 for 11 processors. This indicates that 40000 photons is the optimal task size, with the best tradeoff between an efficient computation/communication ratio, and efficient load balancing. For larger task sizes, the computation/communication ratio will be better, as there is more computation per task, for the same amount of communication. However, since the larger task sizes take longer to complete, and the work is divided into fewer tasks, there is an increased likelihood of processor idle time due to poor load balancing. For smaller task sizes the load balancing will be good, because small tasks only take a short time to complete. However, the computation/communication ratio will be bad, because for smaller tasks there is only a small amount of computation per task, but the amount of communication remains the same.

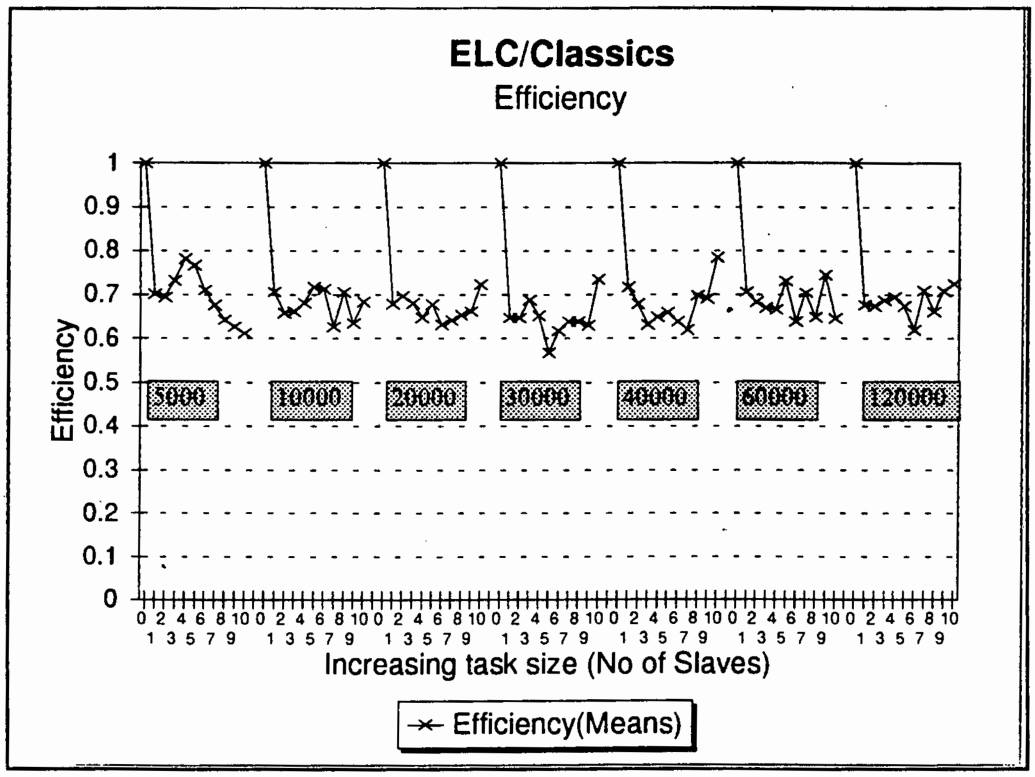

Figure 16: Efficiency of "homogeneous" **ELC/Classic** group (grouped by task size)

Figure 15 shows that, for the task sizes of 20000, 30000 and 40000 photons, the speedup is improving as the number of slaves increases. As described in section 5.4.3, this is because for higher numbers of slaves there is reduced swapping of Monte Carlo and communication code in and out of cache for the master.

The "steps" which occur on the graphs in Figure 15 for 10000, 60000 and 120000 photons are probably due to imperfect load balancing, where the work is not evenly divided between the processors, because the number of tasks does not divide well between this number of homogeneous processors.

Figure 16 shows the efficiency of Cloud, calculated from the speedup as compared to the mean serial performance. In each case the efficiency was obtained by dividing this speedup by the number of processors, as discussed in section 5.3.

Each curve displays the results for the task sizes (in photons) given in the shaded boxes. Within each task size group, tick marks on the horizontal axis indicate the number of slaves used in each part of the experiment. This graph shows the values for each task size (in photons), with the number of slaves varying from 0 to 10.

As is expected from the speedups shown in Figure 15, Figure 16 shows that the efficiency for the task size of 5000 photons increases with the increase in number of slaves up to 4 slaves. This is because for this small task size the load balancing is very good, and there is little processor idle time. From 5 slaves onwards the master has to process the results of at least 5 slaves. This takes longer than the time each slave takes to complete one task, and communication delays begin to occur, leading to a decrease in efficiency as the number of slaves increases. An efficiency of 0.8 is achieved for the 5000 photon task size with 4 slaves, but this is deceptive since from 5 slaves onward the efficiency deteriorates for this task size. This indicates how an initially promising efficiency can be seriously affected by increased communication overheads, as the number of slaves increases.

Figure 16 shows that the efficiency for the larger task sizes decreases initially, then remains fairly constant, before increasing again for larger numbers of slaves. The initial decrease is because for 1 and 2 slaves the master does from two-thirds to half of all the work, and for all tasks done by the master there is no communication overhead, and therefore a high efficiency. As the number of slaves increases and the master does fewer tasks, the efficiency decreases, due to increased communication overhead. Then, as the number of slaves increases, from 6 or 7 upwards, the master is more continuously busy with communication, and consequently spends less time swapping between Monte Carlo processing and communication. This results in increased efficiency due to reduced swapping.

Figure 16 also shows that the best efficiency is for the smaller and larger task sizes. For larger task sizes this is because there is more computation for the same amount of communication overhead per task, which gives a better computation/communication ratio, and thus higher efficiency. For the smaller tasks it is because the tasks take a shorter time to complete, so the load balancing is better, with less processor idle time, and consequently better efficiency.

In most cases the efficiency is approximately 0.65, and the best efficiency achieved is nearly 0.8, for the task size of 40000 photons, with 10 slaves. Thus 40000 photons is the task size where there is the best tradeoff between efficient load balancing and a good computation/communication ratio.

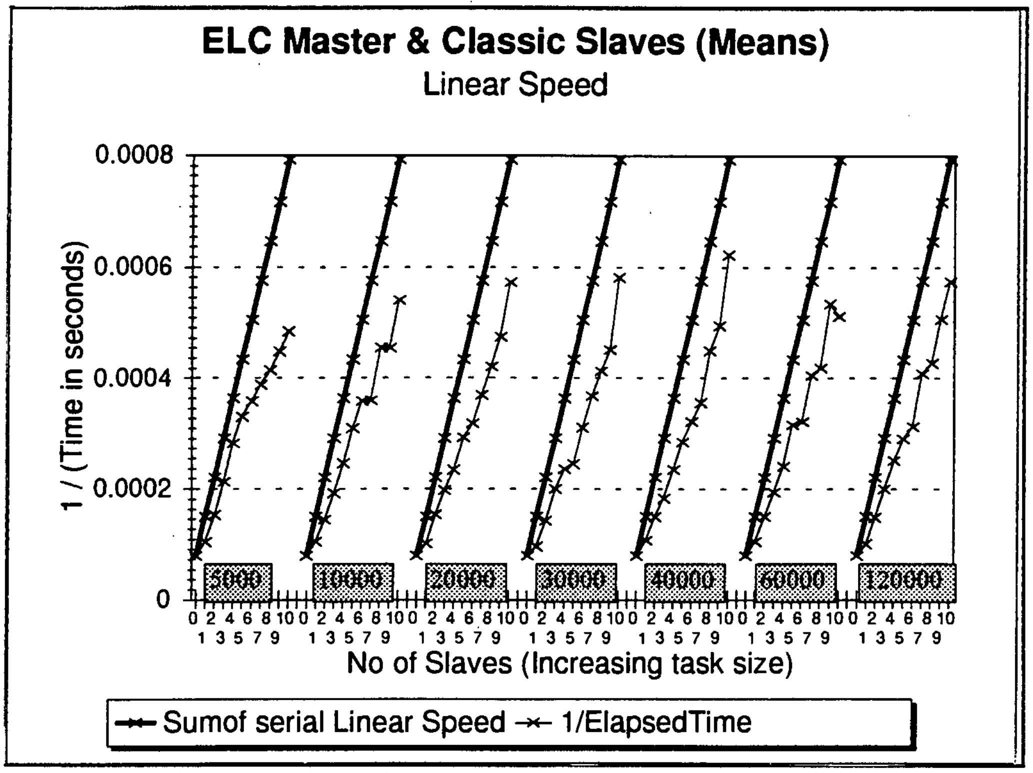

Figure 17: Linear speed for "homogeneous" **ELC/Classic** group (grouped by task size)

Figure 17 shows a Linear Speed graph for the **ELC/Classic** group. This graph is visually similar to the speedup graph in Figure 15 (**ELC/Classic** speedup) and gives much the same information, but is not dependent on hardware considerations.

The deterioration in performance, as the number of slaves increases beyond 4, can be seen clearly for the 5000 photon task size. Also, the improvement in performance when the faster 10th (ELC) slave is added, is clearly visible for the middle task sizes. This graph shows clearly that the best performance is achieved for the 40000 photon task size, with 10 slaves.

5.5.2 Performance of homogeneous Sun SPARCstation 1+ group - 4 machines

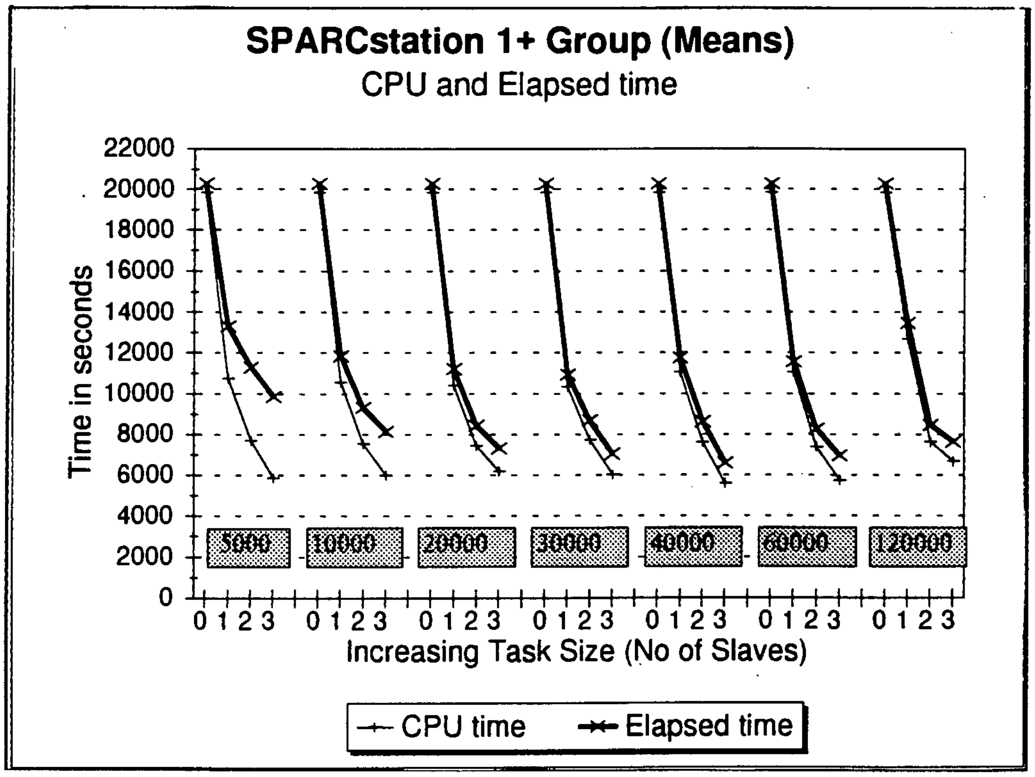

Figure 18: CPU and elapsed times for homogeneous **SPARCstation 1+** group (grouped by task size)

Figure 18 shows the CPU time and elapsed time of the master workstation for the **SPARCstation 1+** group. The elapsed time is very close to the sum of the CPU and system time, so this sum of CPU and system time is not shown on this graph.

The CPU times in Figure 18 are very similar in all cases, but it is very clear that there is a significant increase in system time for the smaller task sizes, particularly the 5000 photon size, when there are many more messages to be sent and received. This causes an increase in overall elapsed time. This graph also shows a steady decrease in the elapsed time, as the number of slaves increases, for all task sizes. Again the best time is achieved for a task size of 40000 photons, with the maximum number of slaves (3 slaves, 4 processors in all). This suggests that, for this group of machines, the best tradeoff between a good computation/communication ratio, and efficient load balancing, is also achieved for the 40000 photon task size.

The longer elapsed times for the 120000 photon task size can be attributed to increased processor idle time, due to inefficient load balancing for this larger task size.

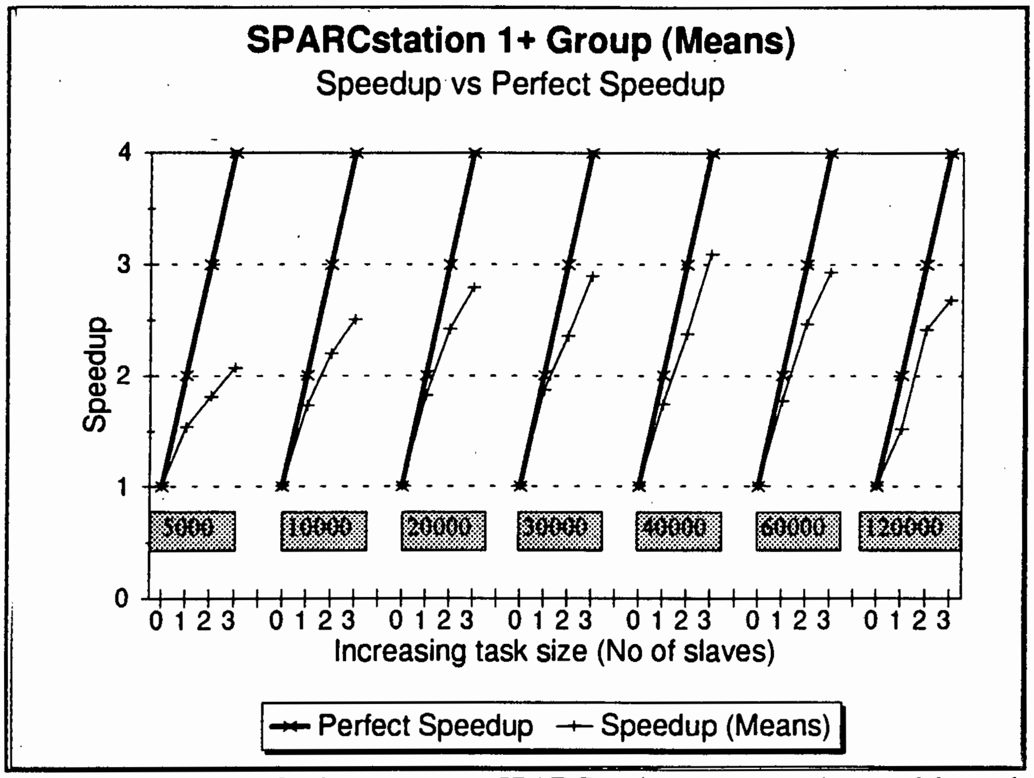

Figure 19: Speedup for homogeneous **SPARCstation 1+** group (grouped by task size)

Figure 19 shows the speedups calculated for the 4 processors of the **SPARCstation 1+** group. Only the speedup compared to the mean serial time for the group is shown, since all machines have the same architecture, and showed very similar serial performance, and can therefore be considered homogeneous.

This graph shows good speedup for most task sizes. The speedup for the task size of 40000 photons was nearly linear, with reasonable speedup also obtained for the 30000 photon task size. For all other task sizes the improvement in speedup deteriorates, as the number of slaves increases. For the smaller task sizes this is due to increased communication, and for the larger task sizes it is because of inefficient load balancing. The middle task sizes have the best tradeoff between efficient load balancing, and a good computation/communication ratio, and this is evident in the improved speedups for these task sizes.

The best speedup of 3.09 for 3 slaves (4 processors) was for the task size of 40000 photons, confirming that this was the best task size. The worst speedups were for the task size of 5000 photons. This was due to the large amount of message-passing for the 5000 photon task size.

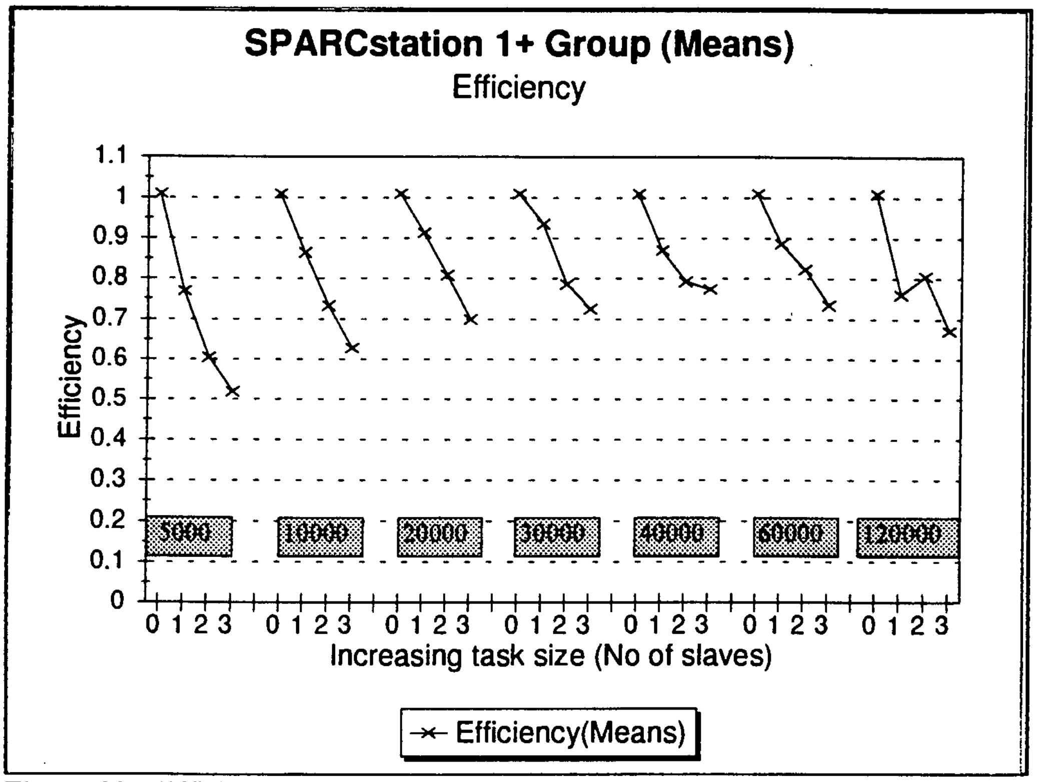

Figure 20: Efficiency of homogeneous **SPARCstation 1+** group (grouped by task size)

The next graph, Figure 20, shows the efficiency calculated for this group. These values are grouped together for each task size, with 0 to 3 slaves for each task size.

The efficiency for one processor is marginally better than 1, because the serial performance of the master is very slightly better than the mean serial performance of the 4 processors. Therefore, when the speedup is calculated for 1 processor, as the serial performance for the master compared to the mean serial performance for the group, this works out as just over 0.99 instead of 1, resulting in an efficiency marginally better than 1.

Figure 20 shows that the efficiency tends to decrease as the number of slaves increases, but still improves as the task size increases. As described for the **ELC/Classic** group in section 5.5.1, the best efficiency is achieved for

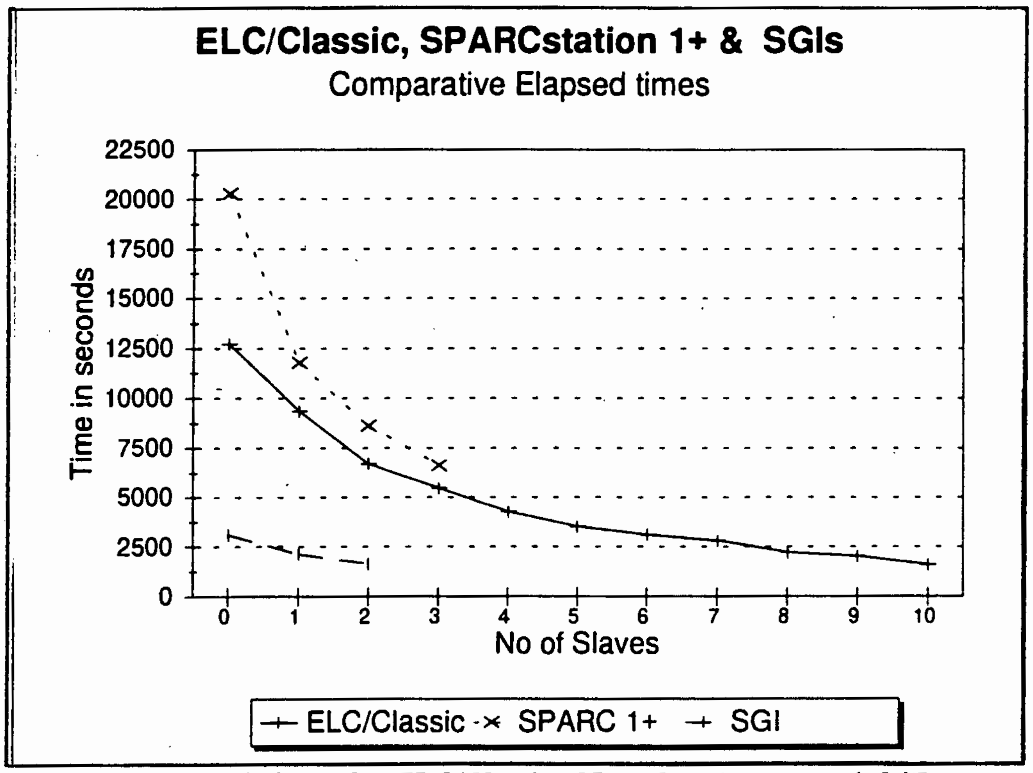

Figure 21: Elapsed times for **ELC/Classic**, **SPARCstation 1+** and **SGI** groups (40000 photon task size)

the middle task sizes, where the tradeoff between efficient load balancing and a good computation/communication ratio is best. There are not enough slaves available for this group to show whether the efficiency will improve again for higher numbers of slaves, as described in section 5.5.1, for the **ELC/Classic** group.

The efficiencies for this group are better than those for the **ELC/Classic** group, being mostly better than 0.8, as compared to 0.65 to 0.8 for the **ELC/Classic** group. However, the elapsed times for the **SPARCstation 1+** group are **longer** than those for the **ELC/Classic** group! To illustrate this, the elapsed times for the **ELC/Classic** group, for the **SPARCstation 1+** group, and the **SGI** group, are shown in Figure 21. More details of the performance of the **SGI** group will be given in section 5.5.3. Figure 21 shows the data for the 40000 photon task size, because, for all three groups, the best results were achieved for this task size. The number of slaves is shown on the x-axis.

Figure 21 shows that the elapsed time to run Cloud on 4 SPARCstation 1+ machines is almost the same as the time taken on 1 ELC and 2 Classics, and about four times as long as on 1 ELC master with 9 Classic and 1 ELC slaves. This shows that, although good speedup and efficiency can frequently be achieved for slow machines, as a result of a good computation/communication ratio, this does not mean that the overall performance is good, as the elapsed time may be long. It is therefore essential, when evaluating performance, to consider the elapsed time, as ultimately the shortest elapsed time indicates the best performance.

5.5.3 Performance of SGI group - 3 machines

Unlike the **ELC/Classic** and **SPARCstation 1+** groups, the **SGI** group was not homogeneous, as it consisted of a fast machine and two much slower machines, with the Indigo2 master being nearly twice as fast as the two slower, near homogeneous, Indigo slaves. These three machines were grouped together merely because they were all Silicon Graphics machines, and because their performance was so different from the two Sun groups.

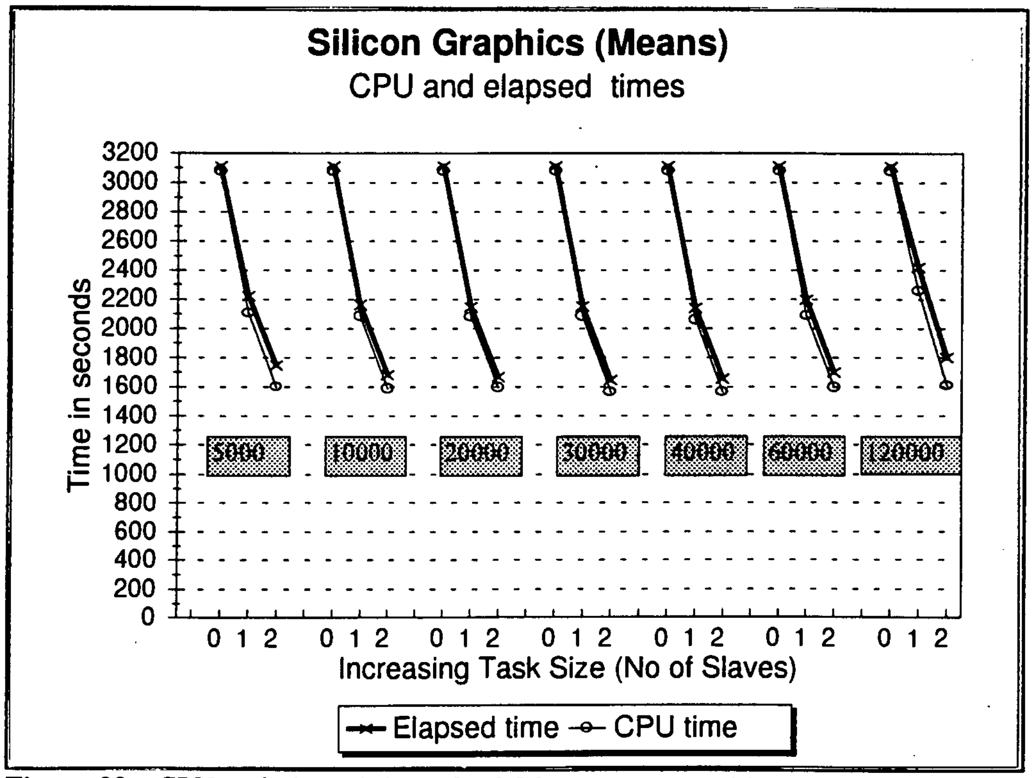

Figure 22: CPU and elapsed times for **SGI** group (grouped by task size)

Section 5.2.1 described how the performance of the Indigo with 24 Mb memory deteriorated after it was moved from one room to another, and that this performance was slightly worse than the other Indigo with 16 Mb, when the reverse would be expected. This deterioration is largely ignored in the graphs presented in this section, as the difference in the results was small, and, if the network fault was corrected, it is expected that the other Indigo slave would give nearly identical results. The graphs in this section therefore show the results for 0, 1 and 2 slaves, with the results for 1 slave being those of the Indigo slave that gave slightly better performance.

Figure 22 shows the CPU and elapsed times for 0, 1 and 2 slaves for each task size. The sum of the CPU and system times is not shown, as this is nearly identical to the overall elapsed times for this group (The difference is less than 1% of the elapsed time). Therefore the system time can be interpreted as the difference between the CPU and elapsed times.

Figure 22 shows increased system time, and thus a longer elapsed time, for both the 5000 photon task size when there are many more messages, and also for the task size of 120000 photons, when load balancing is less efficient for this larger task size. Again the shortest elapsed time is for the task size of 40000 photons, with the maximum number of slaves. These results are similar to those of the **ELC/Classic** and **SPARCstation 1+** groups.

The elapsed times of the **SGI** group are compared with those of the **ELC/Classic** and **SPARCstation 1+** groups in Figure 21. This graph shows that the shortest elapsed time, of approximately 28 minutes, for the 3 Silicon Graphics machines, is almost identical to the shortest elapsed time for the ELC master with 10 slaves. This shows how the same performance can be achieved for two different groups, containing 3 and 11 machines respectively. The best performance of the 4 **SPARCstation 1+** machines is approximately 4 times as long as this best performance for the **SGI** and **ELC/Classic** groups.

Some difficulties of calculating speedup for a heterogeneous system are illustrated in Figure 23. This graph shows the speedups for the group of 3 Silicon Graphics machines, and illustrates how widely varying speedups may be obtained, depending upon how the speedup is calculated. The speedups shown are the speedup compared to the serial performance of the fastest machine (the SGI2 master), the speedup compared to the mean of the serial performance of the actual machines involved (1, 2 or 3 machines), and the speedup compared to the serial performance of the slowest machine (SGI slave). There is no line showing perfect speedup in Figure 23, as this would hide the graph showing the speedup compared to the mean of the serial performance of the machines used.

Figure 23 shows a speedup, compared to the serial speed of the fastest machine (SGI2 master), of about 1.5 for 2 machines, and about 1.9 for 3 machines. This indicates how many Indigo2 Extreme machines would give

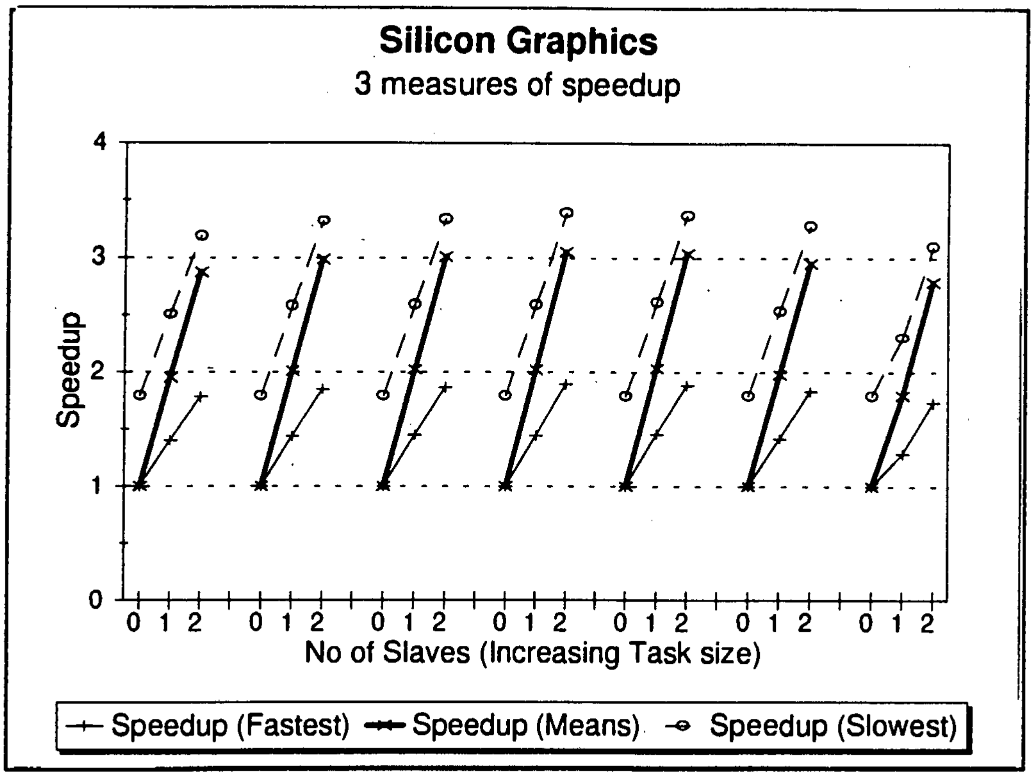

Figure 23: Speedup for **SGI** group (grouped by task size)

the same performance. These speedups are poor, as is to be expected, seeing that the master is approximately twice as fast as each slave. When the parallel performance is compared to the serial performance of the slowest machine (SGI slave), then there are superlinear speedups of about 1.8 for 1 processor, about 2.6 for 2 processors, and approximately 3.4 for 3 processors. These speedups show how many Indigos would give the same performance as this group of 1 Indigo2 and 2 Indigos. This superlinear performance is due to the difference in performance capability between the master and the slaves.

Yet, when the parallel performance is compared to the mean serial performance of all the processors used, the speedup achieved is near perfect, with a speedup of 1 for 1 processor, 2 for 2 processors, and 3 for 3 processors. This good performance is primarily due to the large amount of cache and memory in the master (8kb+8kb primary cache, 1 Mb secondary cache, and 128 Mb memory), so that there is no paging necessary, which means minimal system time is needed, and a high efficiency is achieved.

These three speedups show that comparing the parallel performance with the mean serial performance of the machines used gives a better indication of the efficiency of a heterogeneous system.

As for the **ELC/Classic** and **SPARCstation 1+** groups, Figure 23 also shows that the best speedups for the **SGI** group are for the middle task sizes, where there is the best tradeoff between efficient load balancing and a good computation/communication ratio. The poorer speedup for the smaller task sizes is due to increased communication overhead, because of a larger number of messages. For the larger task size, the lower speedup can be attributed to poor load balancing.

There is no graph showing the efficiency for the **SGI** group, since the speedup graphs in Figure 23 show clearly that, if speedup is calculated by using the mean of the serial performance of all processors used, then near perfect speedup is achieved. This is turn means a near perfect efficiency of 1, for all numbers of processors used, except for the task size of 120000 photons when the efficiency is marginally less than 1.

5.5.4 Performance of heterogeneous groups of 3, 14 and 18 Sun & SGI machines

Altogether there were 18 Sun and SGI workstations used in the experiments with Cloud. They were all used together to find the best overall performance that could be achieved. This section presents the results of the experiments on these heterogeneous groups.

The groups used, with the Silicon Graphics Indigo2 Extreme as master, were:

- the other 2 SGI machines as slaves (as shown in section 5.5.3),
- these 2 SGI machines, plus the 2 ELCs and 9 SPARCclassics, as slaves (13 slaves and 14 processors in all), and
- these 13 slaves, plus the 4 SPARCstation 1+ machines, as slaves (17 slaves, and 18 processors in all),
- these 17 slaves, plus an 18th slave process **running on the master**, in addition to the main process, so that the master workstation had **two** processes running. (18 slaves, 18 processors in all).

The Silicon Graphics Indigo2 Extreme was used as the master because this machine was different from all the others. This meant that for each subset of slaves there were at least two with similar performance capabilities, so the actual performance of like slaves in the heterogeneous environment could be compared.

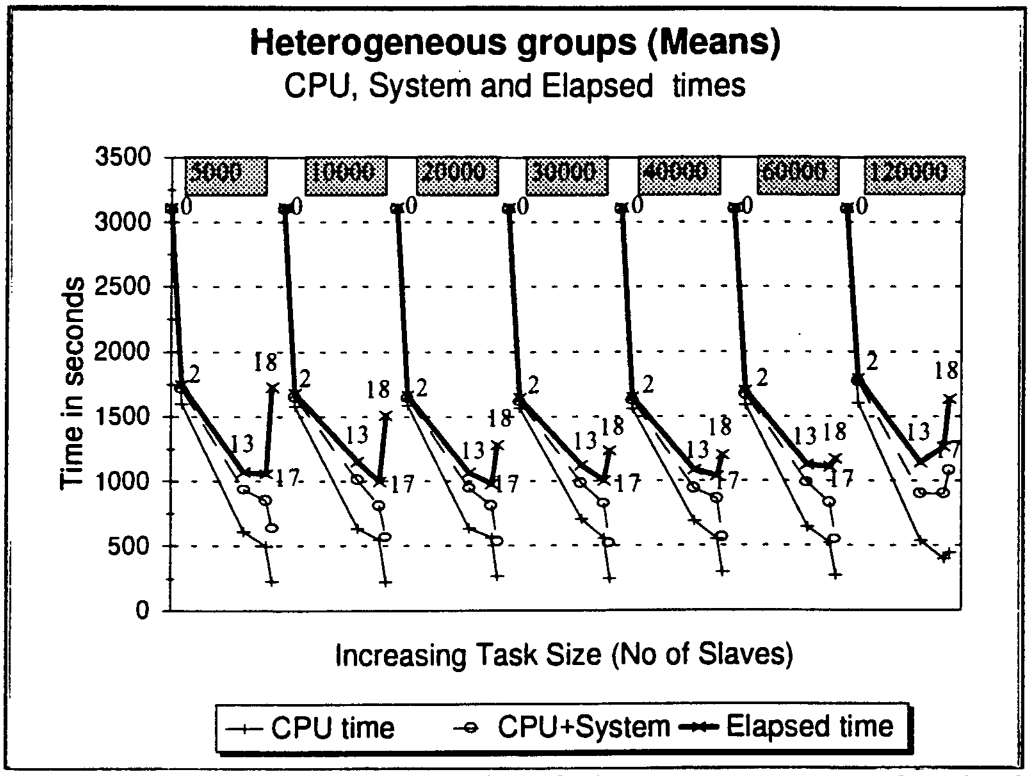

Figure 24: CPU, system and elapsed times for **heterogeneous** groups of 2, 13, 17 and 18 slaves (grouped by task size)

The CPU time, sum of the CPU and system times, and elapsed time for the master process, with these heterogeneous groups of slaves, are shown in Figure 24. The system time is thus the difference between the graph showing the sum of the CPU and system times, and the graph showing the CPU time. All these values are shown, so that the time when the master is swapped out can be seen as the difference between the elapsed time, and the sum of the CPU and system time. The results are grouped together by task size, with the numbers of slaves shown next to the markers.

Figure 24 shows that there is a decrease in elapsed time, as the number of slaves increases from 2, to 13 and 17 slaves. However, when an 18th slave process is run on the master workstation the performance deteriorates, and becomes worse than that for 17 slaves. The increase in the 'gap', between the overall elapsed time and the sum of the CPU and system time, shows the time when the master process was swapped out, in favour of the slave process. There is also an increase in the amount of system time for the master, when a slave process is running on the same machine.

These results show, as would be expected, that it is more efficient for a master to do "slave work" within the master process, than to run a

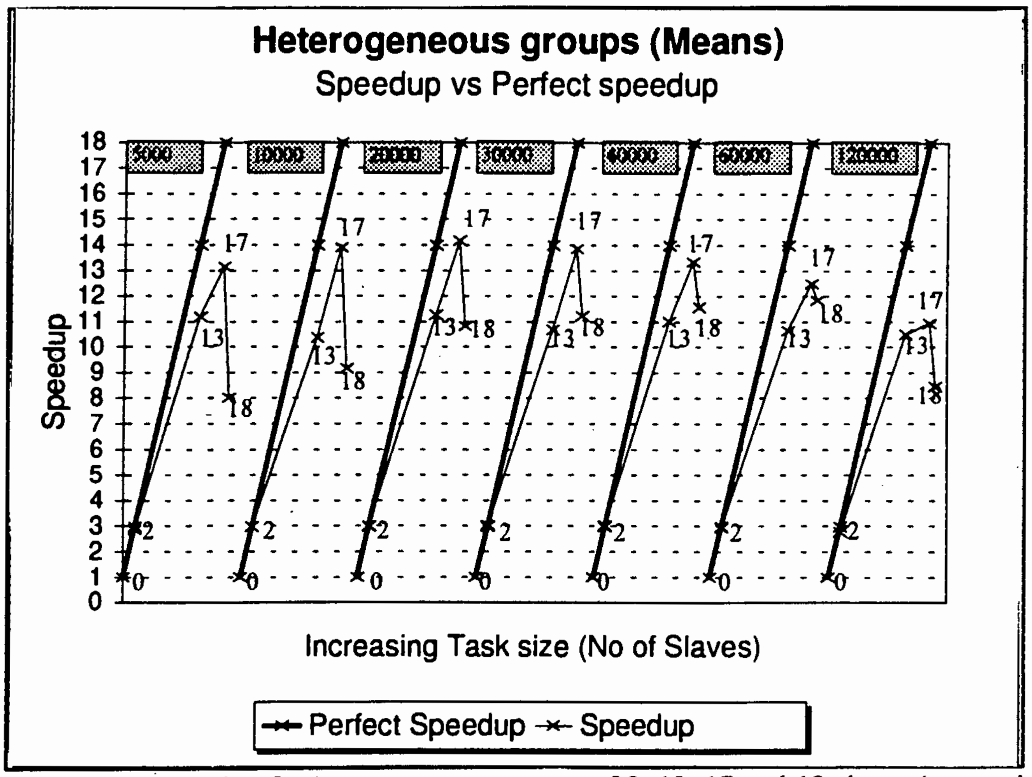

Figure 25: Speedup for **heterogeneous** groups of 2, 13, 17 and 18 slaves (grouped by task size)

separate slave process on the same machine. This can probably be attributed to increased overheads due to context switching between processes on the same processor, and also the increased overheads due to interprocess communication whenever the master communicates with the slave on the same machine. However, these results were obtained at the end of the cycle of experiments. This data is included here because it was collected, but further work is necessary to prove this.

Section 5.4 has shown that speedup is inappropriate for evaluating the results of a heterogeneous system. However, the speedups for this heterogeneous group are shown in Figure 25, so as to indicate some of the difficulties in using speedup as a measure of parallel performance. Figure 25 shows the speedups obtained for these four heterogeneous configurations. The task size for each set of curves is indicated in the shaded block at the top. The numbers of slaves are shown next to the markers.

Section 5.5.3 has shown that, when evaluating the efficiency of a heterogeneous system, the speedup should be calculated by comparing the parallel performance with the mean of the serial performances of the processors used, as this is most representative of the optimal performance that can be expected from a heterogeneous system. Thus, the speedups for Figure 25 were calculated this way. For example, the mean serial performance used for calculating the speedup of the runs with 13 slaves (14 processors) was calculated as the mean of the serial performance of the SG Indigo2 master, the other 2 SGIs, the 2 ELCs and the 9 Classics. For the runs with 17 slaves, the value was the mean of the serial performance of all 18 workstations.

Figure 25 suggests that the speedup up to 17 slaves was near linear. However, remaining results in this section will show that this is not a valid representation of the parallel performance.

As was shown in 5.5.3, Figure 25 shows that the speedup for 2 slaves (3 processors) is close to 3, which is near perfect. A good speedup of just over 11 for the group with 13 slaves (14 processors) is shown for most task sizes, with a speedup of just over 10 for the remaining task sizes.

The best speedup for this group of 13 slaves was for the task size of 20000 photons, whereas for the **ELC/Classic** group alone the best speedup was for the 40000 photon task size. This can be attributed to the disparate performance of the processors in the heterogeneous group. For the **ELC/Classic** group all the processors had very similar performance, so they would be expected to finish almost at the same time, even for larger task sizes, such as 40000 photons. However, there is a wide difference in the performance of the processors comprising the heterogenous group of 14 processors, with the master being about four times as fast as a Classic, and the other SGIs more than twice as fast as an ELC or Classic. This means that, if the load balancing is such that some processors finish before other slower processors, there may be considerable processor idle time while waiting for the slower processors. Therefore, for a heterogeneous group, where the performance of the processors varies widely, better load balancing, and therefore better performance, can be achieved with a smaller task size such as 20000 photons.

Figure 25 shows that a similar, but slightly lower, speedup of approximately 11, for 13 slaves, was also achieved for the 40000 and 5000 photon task sizes. For the 40000 photon task size, this was because of the better computation/communication ratio for the larger task size. The good speedup for the 5000 photon size, as compared to the poor speedups for this task size for the homogeneous **ELC/Classic** and **SPARCstation 1+** groups, illustrates one benefit of a heterogeneous system. In a homogeneous system, all processors will finish each task at approximately the same time, and this may lead to communication bottlenecks, both with clashes on the network, and also there may be delays caused by the master having to process results from all slaves at the same time. In a heterogeneous network the processors have different performance capabilities, and this will mean that the slaves will finish their tasks at different times, thus staggering the impact on the network and the master. This is shown in the good speedup for the 5000 task size for 13 heterogeneous slaves, where the benefit of good load balancing, for the small task size, outweighs the disadvantage of a poor computation/communication ratio, because the communication is staggered. Since part of this study was to establish the optimum task size, the task size remained constant throughout each run. However, some possible solutions that will reduce these communication bottlenecks are suggested in section 6.7.3 and section 6.7.4.

Figure 25 becomes misleading when showing the speedup for 17 heterogeneous slaves. Speedups of up to about 14 are shown for the group of 17 slaves, with the speedup calculated as compared to the mean serial performance of these 17 slaves and the master. However, this improvement in speedup is deceptive, as is seen when the elapsed times for these runs are studied. These mean elapsed times, in seconds, are shown in Table V.

Table V: Comparison of elapsed times for 13 and 17 heterogeneous slaves

Task size (photons)	Elapsed time 13 slaves [Std dev]	Elapsed time 17 slaves [Std dev]	Difference (seconds)
5000	1070 [23]	1056 [14]	14
10000	1155 [34]	997 [16]	158
20000	1065 [37]	979 [18]	86
30000	1122 [6]	1001 [7]	121
40000	1088 [27]	1038 [36]	50
60000	1126 [68]	1111 [23]	15
120000	1141 [7]	1267 [42]	-126

Table V shows that adding four slow machines to the group of slaves is of very little benefit, and that almost the same elapsed times can be achieved without these processors. This is because the four SPARCstation 1+ machines are so slow that they do almost none of the work, and almost all of the work is completed by the faster processors.

The values in Table V show that the runs with 13 slaves and 17 slaves have very similar elapsed times. Any improvement achieved by using 17 slaves is minimal, and in the case of the 120000 photon task size the elapsed time with 17 slaves is 126 seconds slower than that with 13 slaves. This amount is not trivial, and compared to the elapsed time of 1141 seconds, for 13 slaves, it is 11% slower. Yet the speedup shown in Figure 25, for this task size, still shows an improvement! This shows how speedup can be deceptive when evaluating performance. The differences in the elapsed times, for the two groups of 13 and 17 slaves, are very similar to the standard deviations, in seconds, of the means of 3 best elapsed times (as shown in [] in Table V). In some cases the difference in elapsed time is less than the standard deviation. This shows that there is no significant improvement to be gained by adding the 4 slow SPARCstation 1+ slaves. For the largest task size, it was actually detrimental to overall performance to use these four slow machines.

The poor performance of the largest task size illustrates how poor load balancing on a disparate heterogeneous system can seriously affect performance. For example, for the task size of 120000 photons there are only 50 tasks to be shared between 17 heterogeneous slaves and the master. Cloud initially sends two tasks to each slave, so as to ensure that, immediately after returning the results of the previous task, each slave has a spare task ready to process. So, for 50 tasks and 17 slaves, the master would initially send out 2 tasks per processor, 34 in all. This means that 8 tasks, 2 per processor, are committed to the 4 slow SPARCstation 1+ machines. In the time that each of the SPARCstation 1+ takes to complete 1 task, each of the SGI slaves can complete 4 tasks, and each of the ELCs and Classics about 1.5 tasks each. This means that the remaining 16 tasks (50 less the 34 tasks originally sent to the slaves) are completed by the faster slaves before the SPARCstation 1+ machines have finished their second task. And since the time for a SPARCstation 1+ to complete one task of 120000 photons is about 6 to 7 minutes, this could mean that the faster slaves may finish up to about 6 minutes before the SPARCstation 1+ slaves, and this is inefficient, and causes an unnecessarily long elapsed time. In this case the technique of sending a spare task to each processor is not beneficial, and actually results in a longer elapsed time.

Performance can be improved by reallocating the final tasks from slower to faster processors, which will complete these tasks in a shorter time. Cloud was written to do this, as described in section 3.3.6. However, to evaluate performance the same amount of work should be done in each run, so for these performance tests the feature to reallocate tasks to faster processors was not implemented, as this would have meant that some tasks were duplicated, and the total amount of work would have varied from run to run. This is discussed further in section 6.7.1.

The other important point, illustrated by Figure 25, is that running a separate slave process on the same processor as the master is detrimental to performance. The speedup, in this case, is calculated by comparing the parallel performance for 19 processes (1 master, 17 slaves on other machines and 1 slave on the same machine as the master), with the mean serial performance of the 18 processors used. Figure 24 shows that, in all cases, the elapsed times for the runs with the 18th slave process on the master's machine were longer than those for the 17 slaves.

Figure 25 shows clearly the deterioration in speedup for the runs with 18 slaves. This shows that the overheads of running two processes on the same processor are such that it is more efficient for the master to do slave work in the master process, than for the master process to run a separate slave process on the same processor.

Figure 25 has shown that speedup is not valid for evaluating the performance of a heterogeneous system. As an alternative, Figure 26 shows a linear speed graph as proposed by Crowl. Figure 26 illustrates the benefit of using a hardware-independent means to show parallel performance.

Figure 26 gives a much better idea of the actual parallel performance achieved for this group of heterogeneous workstations than the speedup graph in Figure 25. Figure 26 shows clearly the divergence between the actual performance and the best possible performance, as depicted by the line showing the sum of the serial linear speeds.

The lack of improvement in performance, for most task sizes, when the four SPARCstation 1+ slaves (slaves 13 to 17) are added is clearly visible. Even when there is some improvement in performance when these four slaves are added, for the task sizes of 10000, 20000 and 30000 photons, Figure 26 shows that this improvement is very small, and this is more representative of the actual parallel performance than the good speedup shown in Figure 25.

Also, Figure 26 correctly shows a deterioration in performance for the 120000 photon task size for 17 slaves, as opposed to the speedup graph in Figure 25, which shows an improvement in speedup for this case! The deterioration in performance for 18 slaves (with 1 slave process on the same

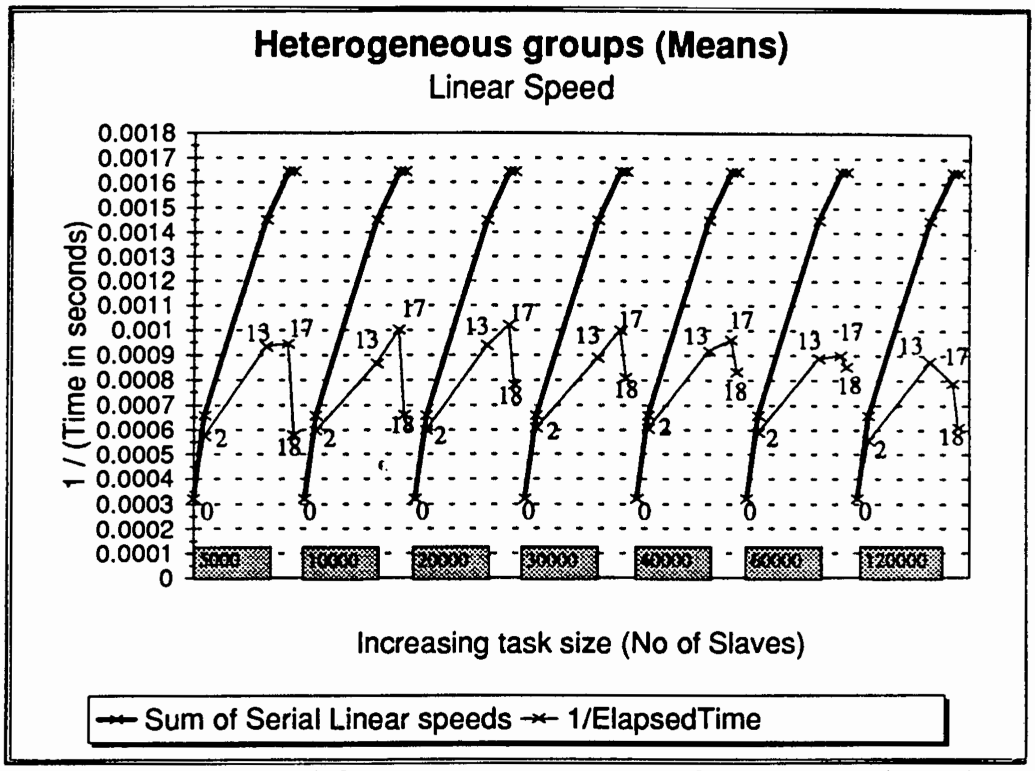

Figure 26: Linear speed for **heterogeneous** groups of 2, 13, 17 and 18 slaves (grouped by task size)

processor as the master) is clearly evident.

Linear speed graphs are independent of hardware, and the number of processors or processes used, and are therefore perhaps a more reliable method of portraying the parallel performance of a heterogeneous system, than a conventional speedup graph, which may be misleading.

The linear efficiency for this group of 18 heterogeneous processors is shown in Figure 27. For this graph the linear efficiency was calculated by dividing the linear speed achieved in the experiments, by the sum of the serial linear speeds of all processors used, where this sum represented the total potential performance capability of the system. Figure 27 shows that a high efficiency of approximately 0.9 was achieved for the two Indigo slaves. This dropped to a range of 6.0 to 6.5 when the ELC and Classic slaves were added (slaves 3-13). This graph also shows very clearly the deterioration in efficiency, to about 0.6, for most task sizes when the four SPARCstation 1+ slaves are added (slaves 14 to 17), and even more so the poor efficiency of 0.35 to 0.5 when a slave process is run on the same processor as the master.

The linear speed graph in Figure 26, and the linear efficiency graph in Figure 27 are therefore much more representative of the true parallel

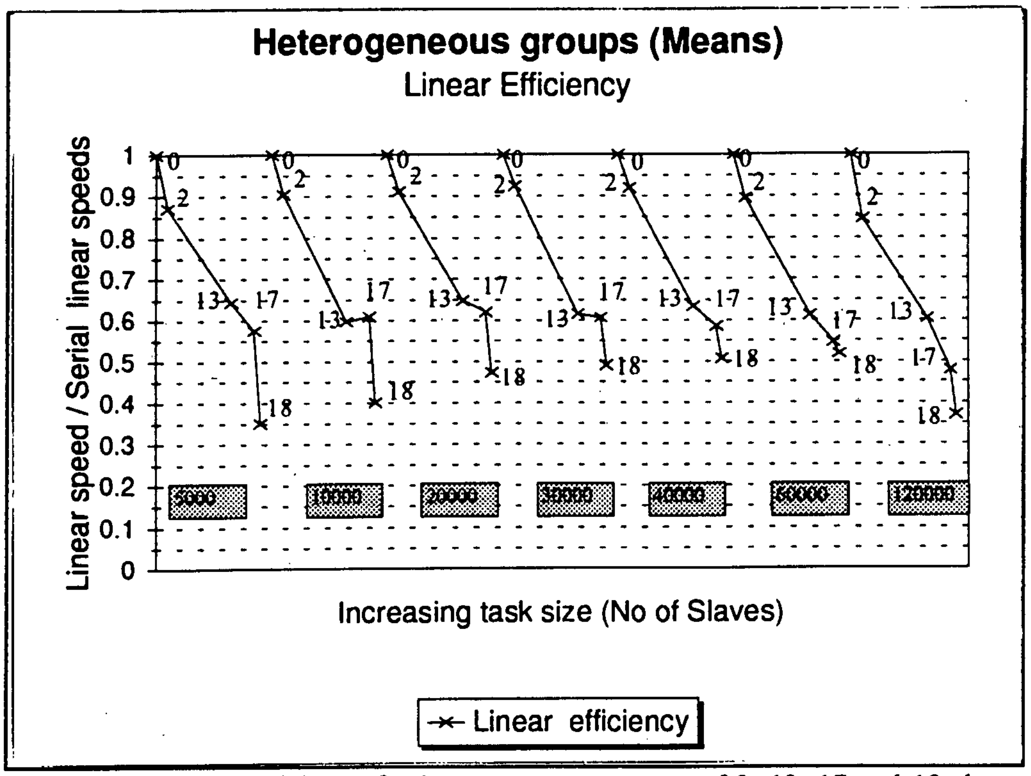

Figure 27: Linear efficiency for **heterogeneous** groups of 2, 13, 17 and 18 slaves (grouped by task size)

performance of this heterogeneous group, and suggest that linear speed, as proposed by Crowl, and linear efficiency, as suggested in this dissertation, should be used to evaluate the performance of a parallel system, in preference to the conventional methods of speedup and efficiency, which give misleading results for a heterogeneous system.

5.6 Factors affecting parallel performance

There are a number of factors which may affect parallel performance. These include the costs of starting up slave processes, the amount of memory used, load balancing and granularity, overlapping communication with computation, and the number and size of messages. This section describes the quantitative impact of these factors as shown by the results of experiments.

5.6.1 Startup costs

The time to start up slave processes can sometimes be a significant proportion of the overall elapsed time [Minn93]. If this time is minimized, then performance can be improved. This section shows that, for Cloud, the startup costs can be considered negligible for the number of slaves used in these experiments, although for more slaves it may be necessary to investigate whether the startup time could be reduced, perhaps by using the *p4* secure server.

To determine the overhead involved in starting up slaves for Cloud, the first executable statement of the program was changed to **exit()**, so that the program would terminate immediately after starting up. No other changes were made to the program, so the size of the executable remained the same, and the startup costs measured would be realistic.

Table VI: Startup costs for each group of slaves

Group → No slaves ↓	ELC/Classic (Seconds)	SPARCstation 1+ (Seconds)	SGI (Seconds)
1	4.80 (0.91)	2 (0)	1.5 (0.07)
2	8.87 (1.15)	5 (0)	2.9 (0.3)
3	12.87 (1.26)	7 (0)	-
4	17.20 (1.17)	-	-
5	22.00 (1.63)	-	-
6	25.00 (1.21)	-	-
7	30.93 (2.51)	-	-
8	36.27 (4.01)	-	-
9	43.00 (5.34)	-	-
10	44.40 (3.86)	-	-

This section shows the startup costs for each group of machines. For each experiment the master started up groups of slaves containing from 1 slave to the maximum number of slaves in the group and then terminated. Slaves terminated immediately they were started. These runs were repeated 20 times, for each number of slaves, for each group. The times shown in Table VI are thus the mean time needed to startup and closedown the processes. The values shown in brackets, (), are the standard deviations in seconds.

Table VI shows that to start up the slaves in the **ELC/Classic** group took approximately 4-6 seconds per slave, and those in the **SPARCstation 1+** group needed about 2 seconds per slave. The **SGI** machines showed that to start 1 machine as a single slave took less than 2 seconds, and to start both slaves was about 3 seconds.

These startup times are a very small fraction of the overall elapsed times of the experiments described in section 5.5. For the **ELC/Classic** group with 10 slaves, the startup time is less than 3% of the elapsed time. For all numbers of slaves less than 10, the fraction of startup time is an even smaller percentage. Even when 17 slaves are started up by the SGI2 master, the startup time is less than 5% of the elapsed time. In most cases the time for starting up slaves is less than the variation in the elapsed times when the same experiment, with the same number of slaves, and the same task size, is repeated. This suggests, that for these experiments with Cloud, that little would be gained by optimizing the starting of slaves, by such means as using the *p4* secure server.

5.6.2 Load balancing and granularity

Parallel performance can be seriously affected by the load balancing. When the load balancing is good, all processors finish at almost the same time. If it is poor, there may be increased processor idle time as some processors finish before others. Bad load balancing is most likely to happen on a heterogeneous system, where some processors are faster than others. As described in section 3.3.7 Cloud implements dynamic load balancing, to compensate for the different performance capabilities of the processors used.

As shown in section 5.5, the best performance for Cloud is when there is the best tradeoff between efficient load balancing, and a good computation/communication ratio. Load balancing is best for small task sizes, when there is least processor idle time, as all processors finish at almost the same time. However, the best computation/communication ratio is usually for larger granularity, when there is more computation for the same amount of communication.

The results in section 5.5 show that, for Cloud, the best performance is for a granularity of 40000 photons, for the homogeneous groups, and for a granularity of 20000 photons, for the heterogeneous groups. These were the granularities with the best balance between the optimal load balancing for task sizes of 5000 photons, and the most efficient computation/communication ratio for the largest task size of 120000 photons.

To show the efficiency of the load balancing for Cloud, the following extra measurements were recorded, as well as recording the CPU time, system time and elapsed time using the Unix **time** function. The *p4* timer was used to measure:

- the total elapsed time for each processor,
- the time spent on Monte Carlo work, and

- the time each processor spends waiting for messages which have not yet arrived.

In addition, the number of tasks executed by each processor are counted. The time spent on communication by each processor was calculated as the difference between the total execution time, and the sum of the computation time and the time spent waiting for messages.

This section uses these results to show the efficiency of the load balancing for the **ELC/Classic** group, which was the largest homogeneous group available. It is easier to show good load balancing with a homogeneous group, as all processors would be expected to do the same amount of work, and to finish at approximately the same time.

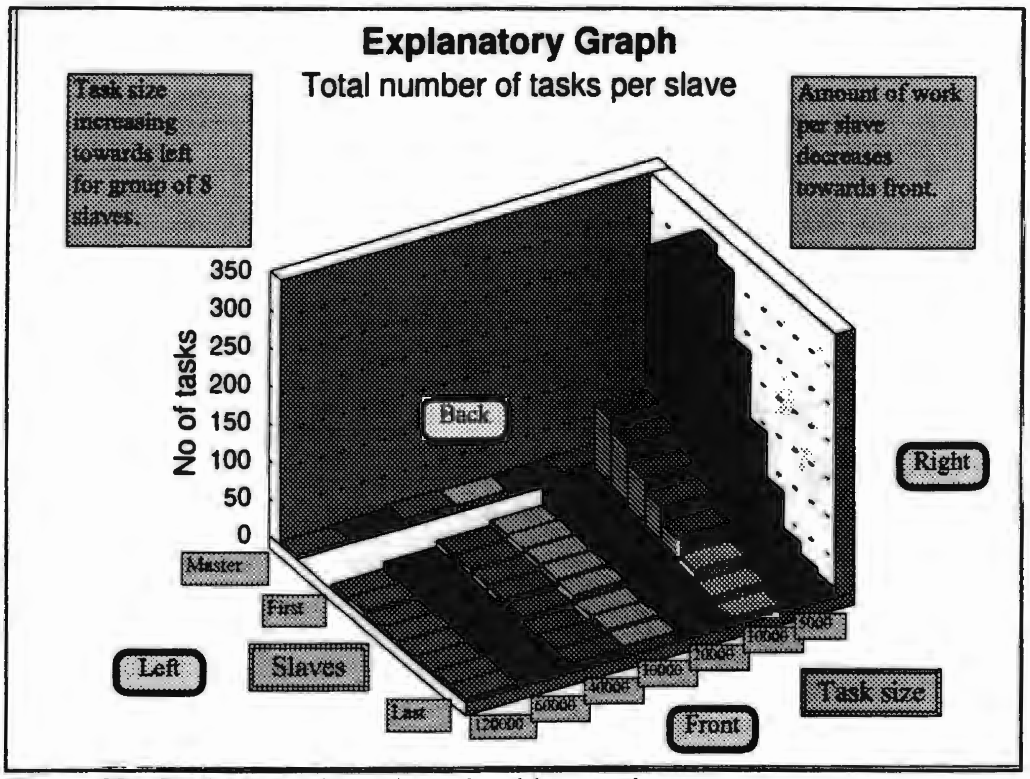

Figure 28: Explanatory three-dimensional bar graph.

In the description that follows in this chapter, some three dimensional bar graphs will be used to illustrate certain points. An annotated example of one of these is shown in Figure 28, and described in detail, to assist the reader in interpreting these graphs.

The directions left, right, front and back were arbitrarily chosen as indicated in Figure 28. All further discussion will follow this convention.

The vertical axis shows the number of tasks executed, as recorded at each event. The height of each bar is proportional to the number of tasks executed by each slave.

The axis on the left shows the master (against the back wall), and the individual slaves (towards the front). Slaves that received their tasks first are shown nearer to the back wall, and the slaves that received their tasks last are shown near the front. This also reflects the order in which the slaves were started up, with the slave closest to the back wall being the first slave started up, and the slave closest to the front being the last slave started up.

The front axis of the graph denotes task sizes ranging from 120000 photons on the left to 5000 photons on the right. In this graph only one sub-series is shown, but ***in all subsequent graphs many sub-series like this are shown on the same graph,*** *one for each number of slaves in the experiment.*

Figure 28 shows that the slaves that received their tasks first execute more tasks than those that receive their tasks later, and that this effect is decidedly more pronounced for smaller task sizes than for larger task sizes.

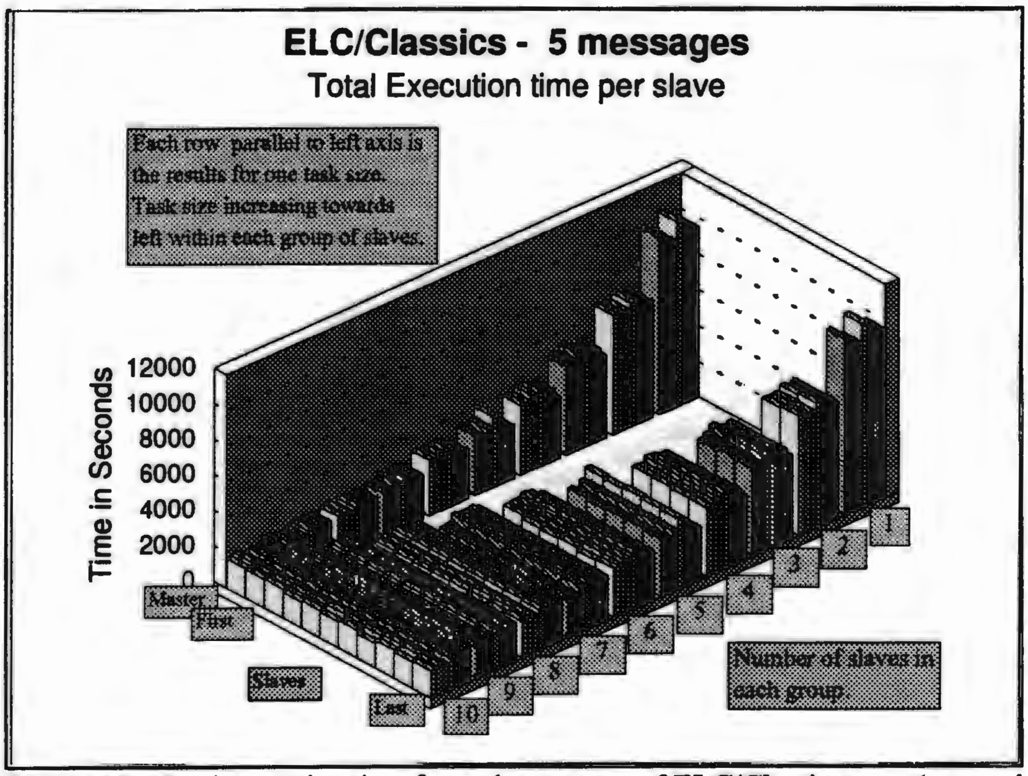

Figure 29: Total execution time for each processor of **ELC/Classic** group (grouped by number of slaves)

Figure 29 shows the total elapsed time for each of the processors in the **ELC/Classic** group, from startup to closedown, for each run. The times for the master are the bars along the back wall of the graph. For each group of slaves, the row of bars on the right is for the task size of 5000 photons, with one row for each task size, with the task size increasing towards the left.

In Figure 29 it is the overall impression that is important, rather than the detail. Each row of vertical bars shows the elapsed times for each processor in each run. This graph is illustrating that, for each run the heights of the bars are nearly identical in height, thus indicating that the load balancing was good, with almost exactly the same elapsed time for each of the processors in any particular run. As would be expected, the runs for the smaller numbers of slaves took longer than those with more slaves.

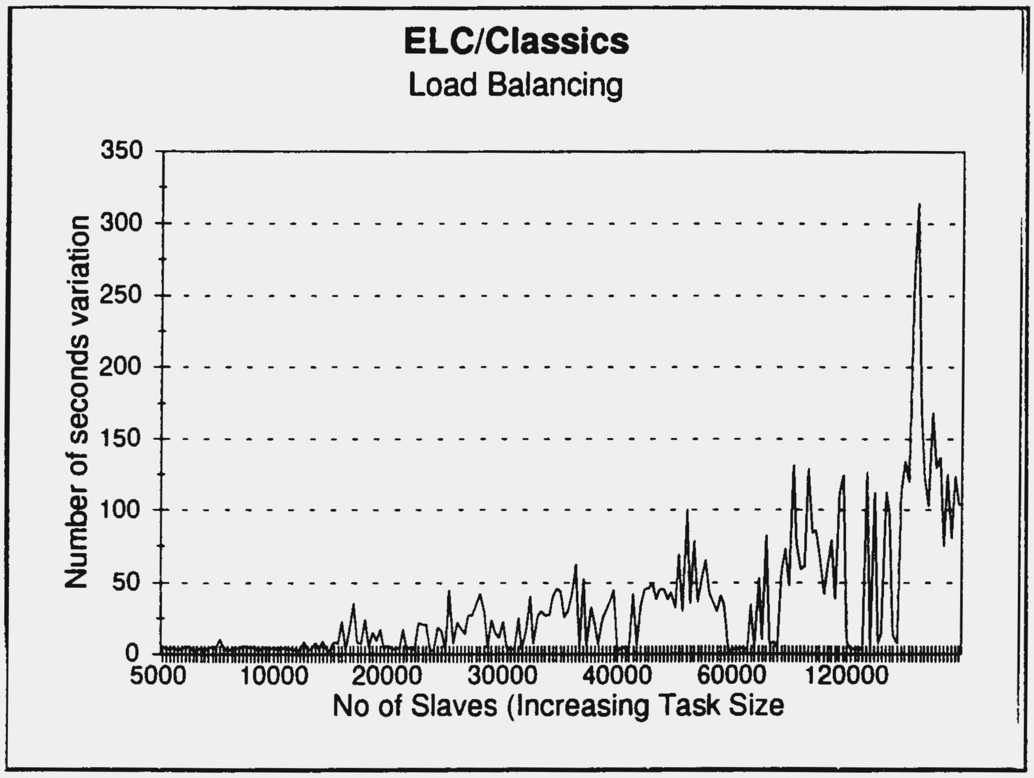

Figure 30: Standard deviation in total execution time, in seconds, for processors of the **ELC/Classic** group (grouped by task size)

The efficiency of the load balancing is illustrated in Figure 30. The mean and standard deviation of the actual execution times for all 11 processors used in each run was calculated. The standard deviation, in seconds, for these runs is shown in Figure 30, grouped according to task size.

Figure 30 shows that, as expected, the load balancing for the small task size of 5000 photons is very good, with almost no variation in total execution time for the processors. As the task size increases, there is more variation in the time taken by the processors to complete execution. However, in most cases (except for the largest task size) the variation is still less than 1 minute, which for practical purposes can be considered an acceptable waiting time. For the five smaller task sizes this variation, is less than 2% of the total execution time. For the 60000 task size it is about 4%, and for the 120000 task size it is about 6%.

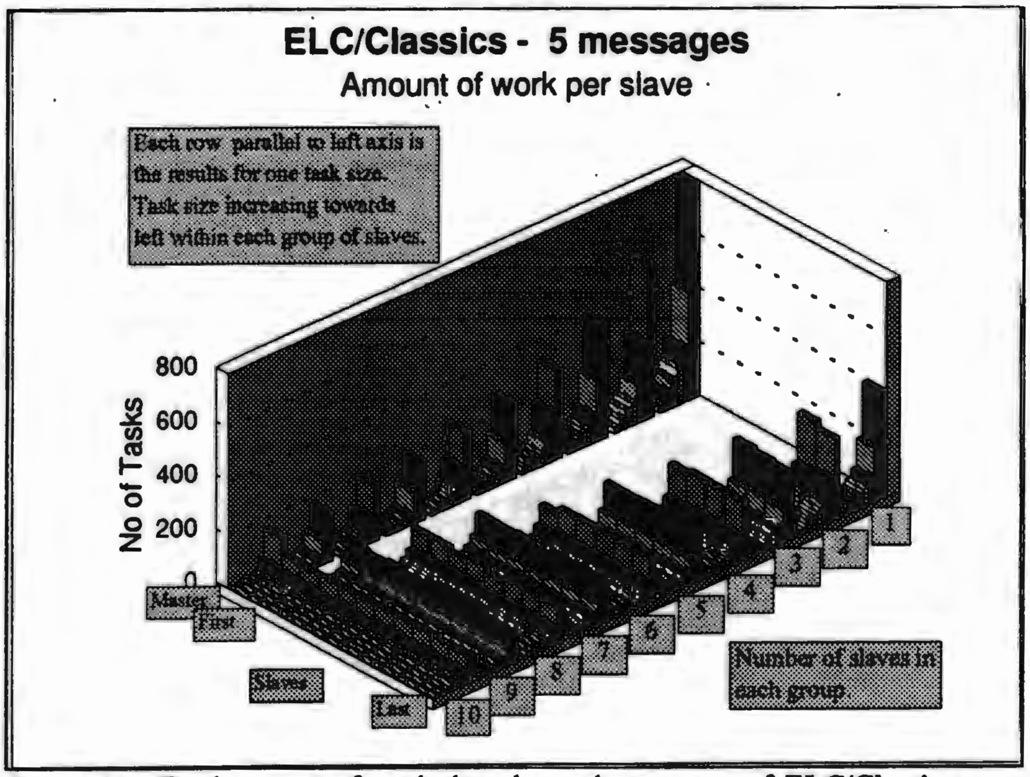

Figure 31: Total amount of work done by each processor of **ELC/Classic** group (grouped by number of slaves)

The next graph, Figure 31, shows the number of tasks executed by each processor. Again the values for the master are along the back wall of the graph, and the rightmost row, for each number of slaves, is for the task size of 5000 photons, when there are 1200 tasks in total to be executed. This graph shows the actual number of tasks executed, because this is clearer than showing percentages. In total there were 1200, 600, 300, 200, 150, 100 and 50 tasks, for runs with the task sizes 5000, 10000, 20000, 30000, 40000, 60000 and 120000 photons respectively. The same total amount of 120000 photons is processed for each run.

Figure 31 shows that all slaves do approximately the same amount of work per run, which shows that the load balancing was good. For smaller number of slaves the master does far more tasks than any of the slaves, even though its serial performance is very similar to that of the slaves. This is because for small numbers of slaves the master does not spend so much time on communication as it does for more slaves, and therefore has more time available for Monte Carlo work. And when the master executes a task it has no communication overhead, so it can execute more tasks than can be executed by a slave in the same time.

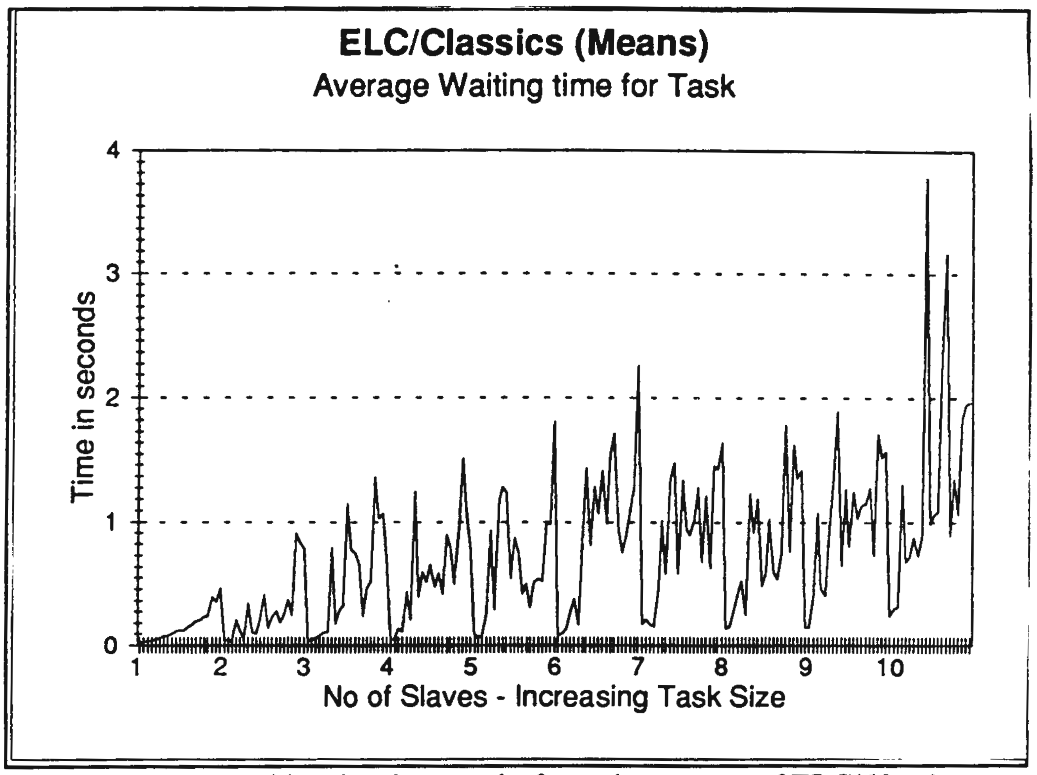

Figure 32: Mean waiting time in seconds, for each processor of **ELC/Classic** group (grouped by number of slaves)

The average waiting times per task, for each slave, are shown in Figure 32. Apart from 3 exceptions, no slave had to wait longer than 2 seconds for a message from the master, and in most cases for less than 1 second. The waiting time increased as the number of slaves was increased, and also as the size of the task was increased. This shows that, with more slaves, there may be a slight delay when the master is busy with communication. The increase in waiting time, as the size of the task increases, is probably due to the master working on tasks itself, in between handling communication. If a result arrives immediately after the master has commenced a task, this result will not be processed, and a new task will not be sent to the slave, until the master has finished its own Monte Carlo task. However, this graph shows that the policy of overlapping communication with computation, so that every slave has a spare task waiting, as described in sections 3.3.4 and 4.11, is generally sufficient, and there are no long delays.

Some of the waiting time could also be attributed to the cost of starting up all slaves at the beginning of the program, when some slaves are ready to receive tasks before the master has actually sent any. There is no easy way to avoid this, since *p4* starts up all slaves in the process group file in one operation, and this must be done before any messages can be sent to the slaves.

5.6.3 Overlapping communication with computation

It was proposed, in sections 3.3.4 and 4.11, that the efficiency of a program could be improved if no processor has to spend time waiting for a message, before it could continue working. To investigate this, experiments were conducted where the master initially sent each slave either 1, 2, 3 or 4 task messages. If 1 task message was sent, the slave would do this task, then send the results to the master, and then have to wait for the master to send it a new task. If 2 task messages were sent, then, whenever a slave had sent the results of the previous task to the master, it already had the next task waiting in its message-buffer, and could continue working immediately. Meanwhile, the master would process the results of the previous task, and then send the next task to the slave, and ideally this new task would arrive before the slave finished processing its current task.

Each of the two cases, where the slave had no spare messages, and 1 spare message, was tested. In addition, 3 or 4 task messages were sent to each slave, so that the slave would have 2 or 3 spare tasks, to see whether any further improvement could be gained, in case the master had not sent the next task in time. Since the number of messages to be sent to each slave was a run-time parameter, the same executable was used for all experiments.

For each of these experiments runs were conducted for all task sizes, but only for the **ELC/Classic** group of 10 slaves. This is because this was the largest number of homogeneous slaves possible, and would keep the master busiest, thus increasing the chances that the master would not send a task to a slave on time, and a slave would have to wait. This homogeneous group was used for this experiment because it was easier to assess the impact on performance where all slaves were similar. If this experiment was run using the heterogeneous group the results may be confused by other factors arising from the disparate nature of the slaves.

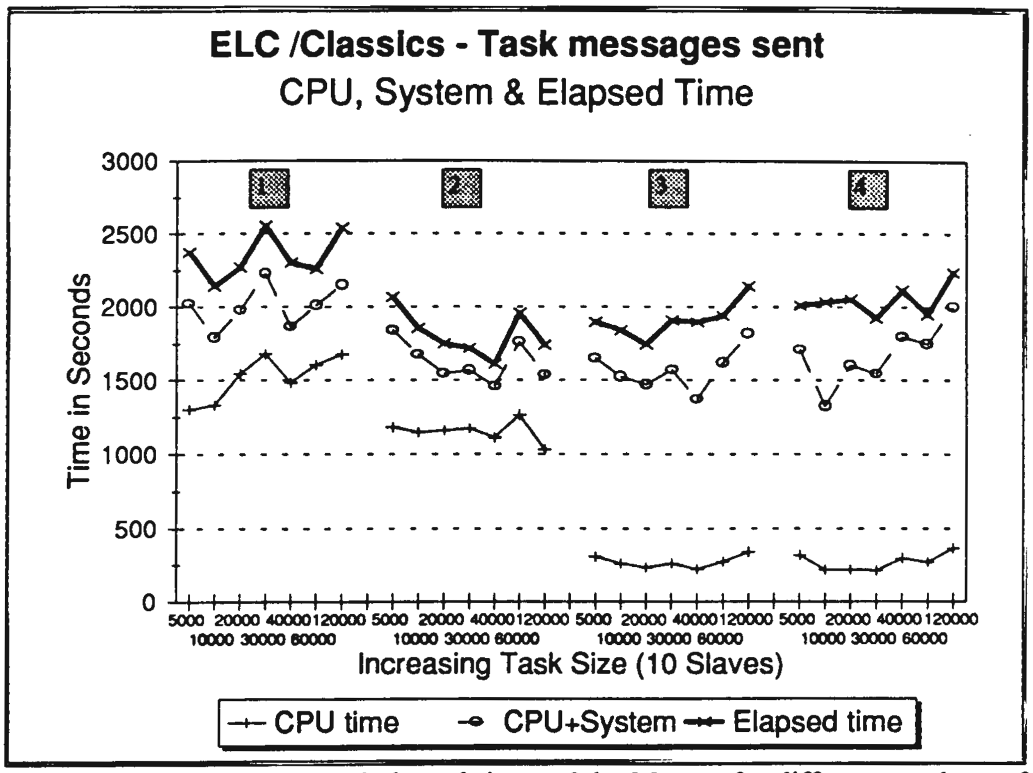

Figure 33: CPU, system and elapsed times of the Master, for different numbers of task messages sent, for **ELC/Classic** group (grouped by number of messages sent)

Figure 33 shows the CPU, system and elapsed times for runs with 1, 2, 3 and 4 task messages (0, 1, 2 and 3 spare messages) being sent to the slaves. The number of task messages sent is shown in the small grey boxes near the top of the graph. All data is for a group of 10 slaves, with the data for each task-message queue length shown for increasing task sizes.

The shortest elapsed times were achieved for a queue length of 2, where each slave had 1 task to work with immediately, and 1 spare task to work with as soon as it had sent the results of the previous task to the master. The times when there were no spare tasks, and the slave always had to wait for the next task, were by far the worst times. The times with no spare tasks were from 15% to 48% longer than the elapsed times with 1 spare task.

Also, the elapsed times when the slaves were sent 3 or 4 tasks were worse than those for queues of 2 tasks. This is partially because the master had to send out 3x10 tasks, and 4x10 tasks, respectively before it could even begin processing, and by that time the first result messages were arriving, so the master did not have time to do Monte Carlo work. And then, since each slave had a queue of 2 or 3 spare tasks, this meant that no other processor could execute these tasks, as the feature to reallocate tasks from slower to faster processors was not implemented (see section 3.3.6). So the master would possibly have to wait for 1 or more slow slaves to finish the assigned tasks, while other faster processors finished earlier, and the load balancing would be poor.

The results of the serial runs described in section 5.2.1 clearly show that there is so little system time for the serial runs that it is negligible. This suggests that for the parallel runs the system time is almost entirely due to parallel overheads, most of which are communication. Thus, in Figure 33, the increase in system time, and decrease in CPU time, for the master, for the cases where 3 or 4 messages are sent, clearly shows that the master spends more time on communication, than on Monte Carlo computation.

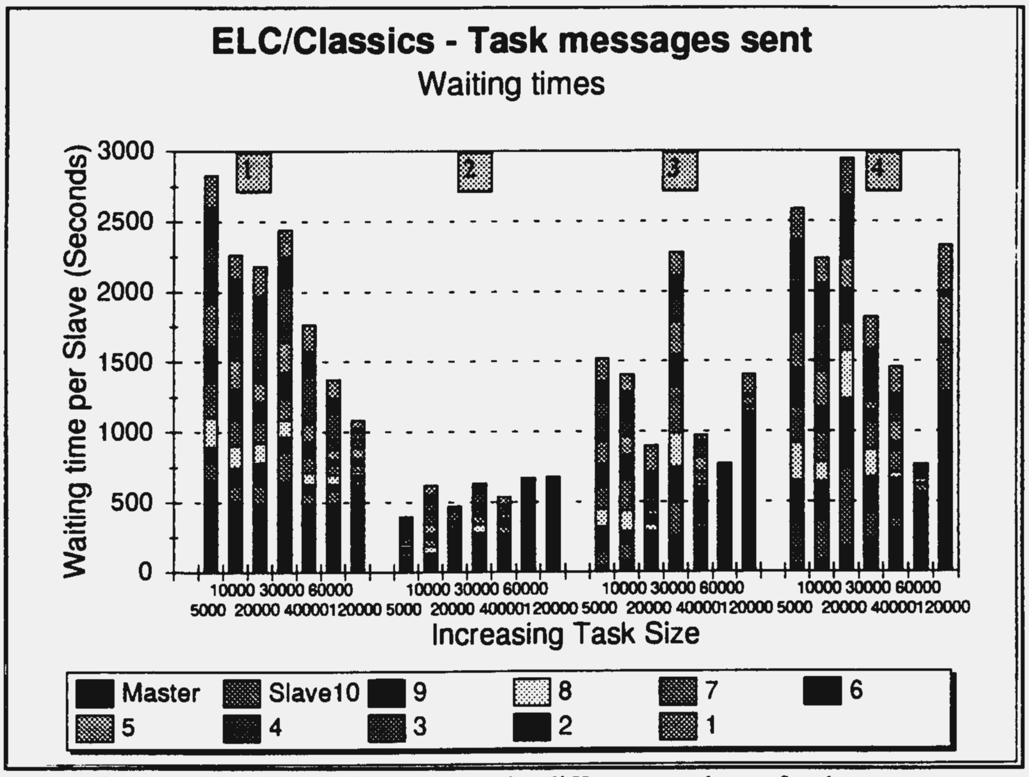

Figure 34: Waiting times of processors, for different numbers of task messages sent, for **ELC/Classic** group (grouped by number of messages sent)

The time each processor spends waiting for messages was measured. Figure 34 is a stacked bar graph showing the total waiting times for all processors for all runs. The numbers of tasks sent to each slave are shown in small grey boxes near the top of the graph. Each shaded section of each stacked bar represents the length of time spent waiting by a slave or master process. The waiting time for the master is the dark-shaded bottom section of the bar, with the waiting times of the 10 individual slaves above. The total height of each bar shows the total waiting time of all processes for a run.

It is evident that there is least waiting time when 2 tasks are sent to each slave, 1 to work on and 1 spare one. But, it is interesting to see that there is almost as much waiting time, and sometimes more, where the master sent 3 or 4 tasks to each slave, than when it only sent 1 task, and the slave had to wait for its next task. This can happen when the master is so busy sending out the initial 3 or 4 tasks per slave, that results return from the slaves before it has finished sending out the initial tasks. From then on, it is so busy processing results that, for the smaller tasks sizes, the slaves finish their work first, and eventually have to wait for the master. For the larger task sizes the slaves have to wait less. For the smaller task sizes the master is so busy that it has almost no waiting time.

The long waiting times for the master, for the large task sizes, in the cases where 3 and 4 tasks were sent to each slave, illustrate the point that the master has to wait a long time for the slaves to finish the last 3 or 4 large tasks in the queue, and this is wasted time for the master. It would be more efficient if the master could have shared this work, as described in section 6.7.1.

Figure 35 shows the amount of work done by each processor. Each shaded section of each stacked bar represents the number of tasks executed by a slave or master process. Each whole bar represents the total number of tasks for each run, that is, 1200 tasks for the 5000 photon task size, down to 50 tasks for the 120000 photon size which explains the difference in height of the bars. This graph shows that, as expected for a homogeneous group, the load balancing for all runs was fairly good, with each slave doing approximately the same amount of work.

The most important point to see from Figure 35 is that, when 1 or 2 tasks are sent to each slave, the master does at least as many, and sometimes more, tasks than the slaves. When 3 or 4 tasks are sent to the slaves, the master does no tasks at all. This is inefficient for two reasons. First, it is more efficient for the master to do a task, than for a slave, as the master has no communication overhead, so can execute more tasks in the same time. Second, if there are 2 or 3 spare tasks in the queue for each slave, once the master has despatched the last task to the slaves, it waits idle until all these tasks have been executed by the slaves. Figure 34 shows how the master spends more

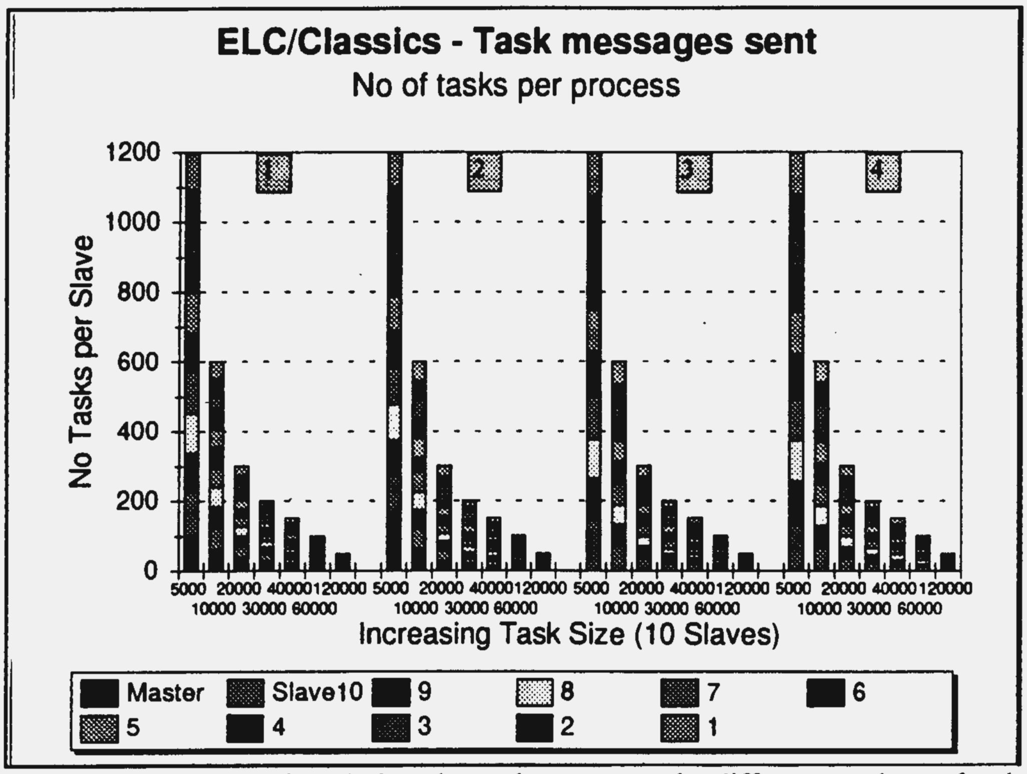

Figure 35: Amount of work done by each processor, for different numbers of task messages sent, for **ELC/Classic** group (grouped by number of messages sent)

time waiting, as the size of the task increases.

Thus, the best load-balancing is achieved when each slave is sent 2 task messages, and having too many tasks "committed" for execution by a specific slave is detrimental to efficient load balancing at the end of the run. Also, the smaller task sizes execute too quickly for the corresponding results to be processed in the same time. So the tasks should be large enough, that the master has enough time to process the results for all slaves in the time the slaves take to execute one task. Thus, having only one master to control a large number of slaves can be a bottleneck, and cause a reduction in efficiency. This could be solved by increasing the task sizes, but this would also cause poor load balancing, as a larger task takes longer to complete. Alternatively the program could be redesigned so that there are several "cluster masters", each administering a sub-group of slaves, and one overall master to collate the results from these "cluster masters". There would then be less chance of communication bottlenecks as each "cluster master" would only have to administer a few slaves. This is discussed further in section 6.7.2.

5.6.4 Changing the number and size of messages

As described in section 4.12, two versions of the program were tested to see if there was an improvement in performance if one longer message was sent, instead of several shorter ones. In one version, the results were returned in five different results messages, and in the other, in a single results message, with the same amount of data being returned in both cases. Apart from this difference, the two versions of the program were identical.

Most of the experiments with Cloud used the version returning five results messages, and the results for these experiments have been presented in section 5.6.2. The graphs that will be presented in this section should be compared with the corresponding graphs in sections 5.1 and 5.6.2, so that the difference in the performance of the two versions of Cloud can be seen clearly. This section shows the results for the version with the results returned in one message, as tested with the **ELC/Classic** group, for groups of 5, 6, 7, 8, 9 and 10 slaves.

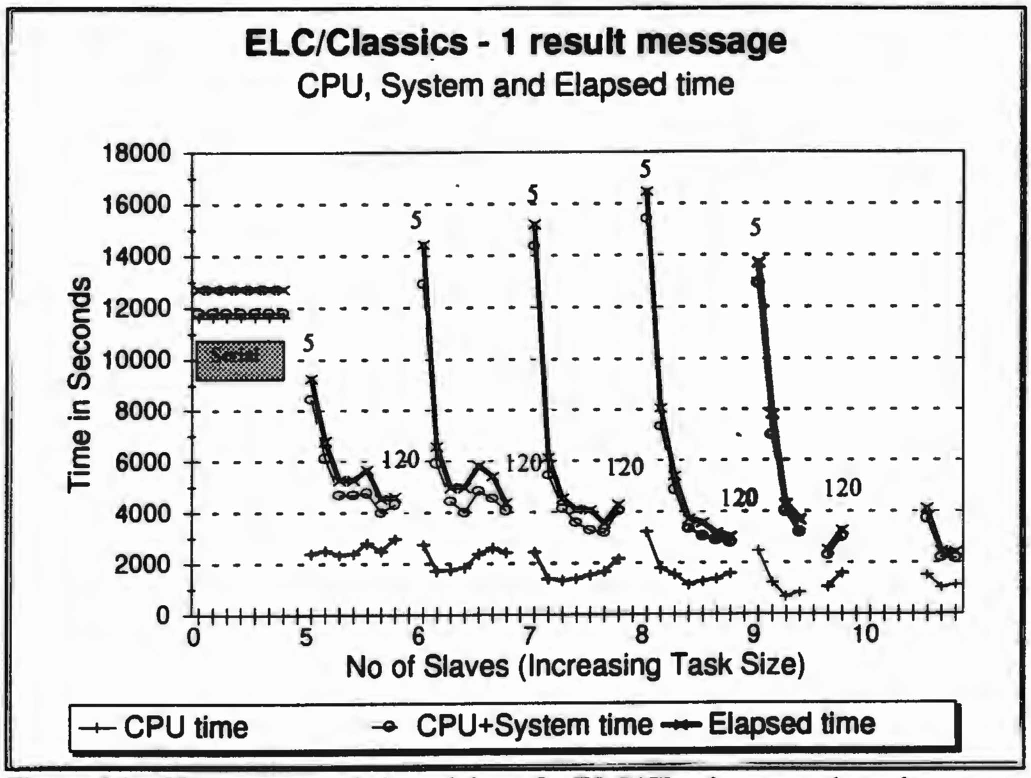

Figure 36: CPU, system and elapsed times for **ELC/Classic** group - 1 result message (grouped by number of slaves)

Problems with lost packets, or with packets taking too long to be received, made it necessary to implement a "time-out" feature in the single message version, so that a program did not hang if a message was lost. The results for these runs were discarded, since if one or more slaves had failed the best times could not be achieved, as the work of the failed slaves would have to be reallocated to the other slaves, which would then have increased the overall elapsed time.

The CPU time, system time, and total elapsed time of the master for these runs are shown in Figure 36. This graph shows that, although the CPU time remains fairly constant, there is a big increase in the amount of system time for the runs with the smaller task sizes, and this causes long elapsed times. In fact, for groups with 6 or more slaves, the elapsed time for the task size of 5000 photons was longer than the elapsed time for the serial run on the master alone!

As will be shown, the considerable increase in processing time for the smaller task sizes was caused by network congestion, caused by Cloud, which resulted from sending very long messages of about 210 kb. This network congestion was so serious, that it became almost impossible to collect these results for the groups of 9 and 10 slaves, owing to the excessive time taken for the runs, and because messages were lost, and some runs had to be aborted. Thus, there are some results missing in this section. The results for groups of 5, 6, 7 and 8 slaves were taken as the means for the best 3 runs for each task size. However, the results for groups of 9 and 10 slaves were from only 2 (or even 1) runs, and some results were not obtained at all. Those results that were obtained for groups of 9 and 10 slaves show the same trends as those for the other groups.

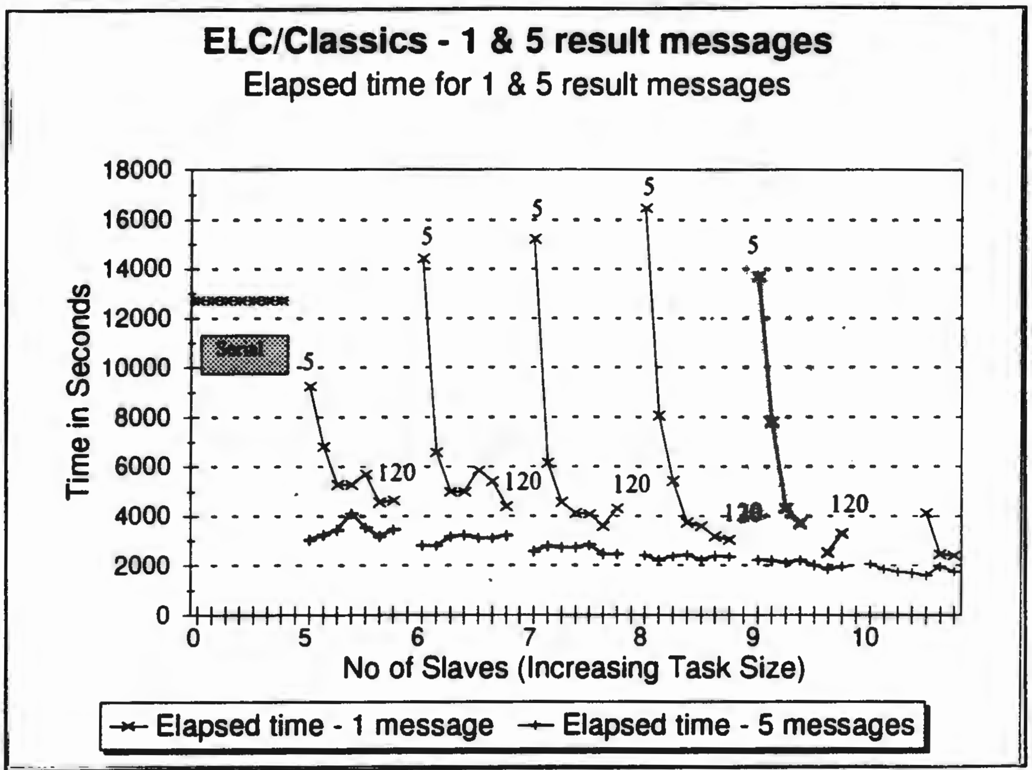

Figure 37: Elapsed times for **ELC/Classic** group - 1 & 5 result messages (grouped by number of slaves)

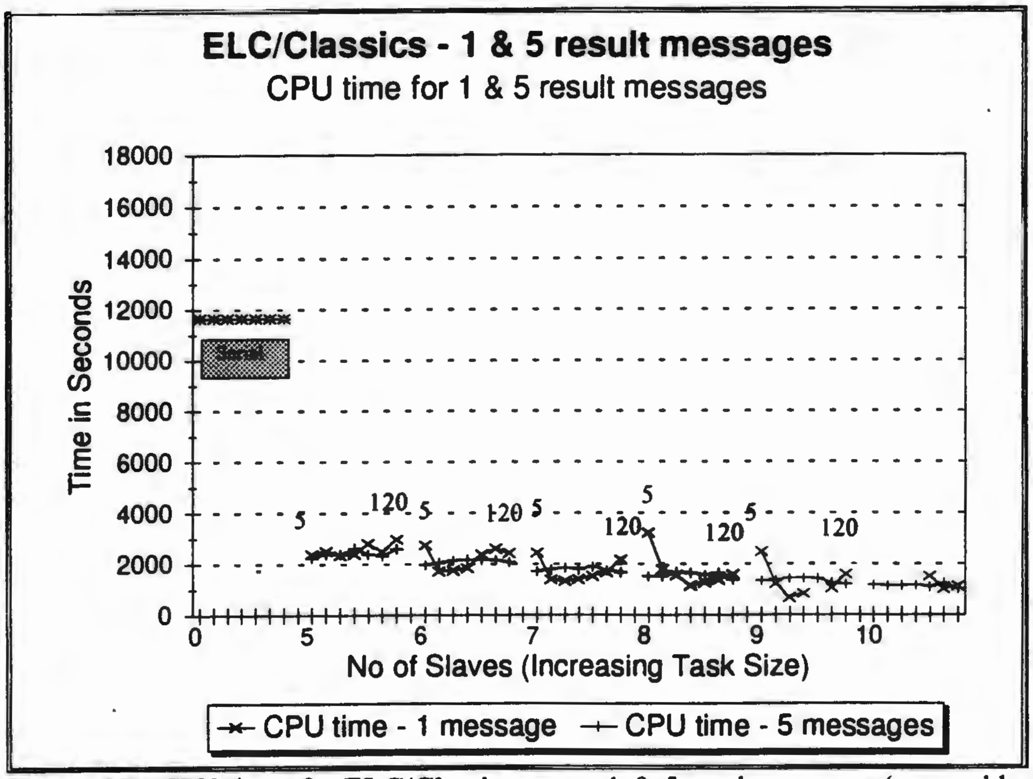

Figure 38: CPU times for **ELC/Classic** group - 1 & 5 result messages (grouped by number of slaves)

The overall elapsed times for the runs with a single results message, as compared to those with five results messages, are shown in Figure 37. In all cases the elapsed times, for the runs using a single results message, were much longer than for the runs using five results messages. The time for the task size of 10000 photons was approximately twice as long for the single message version as for the 5-message version, and that for the 5000 photons size took even longer than the serial run!

Figure 38 shows a comparison between the CPU times of the master, for the runs with a single results message, and those with five results messages. This graph showed that the CPU time for the master is similar for both versions of the program, but for the smaller task sizes the master uses more CPU time in the single results version, than in the version with five messages. This is probably because the master has spare time available to do Monte Carlo work, while waiting for results from the slaves which are trying to send results messages on a congested network. So the master does more tasks than in the 5-message version, when there is no network congestion. Figure 38 shows that the degradation in performance, of the single-message version, is not due to any difference in CPU time, but must therefore be entirely due to a large increase in system time.

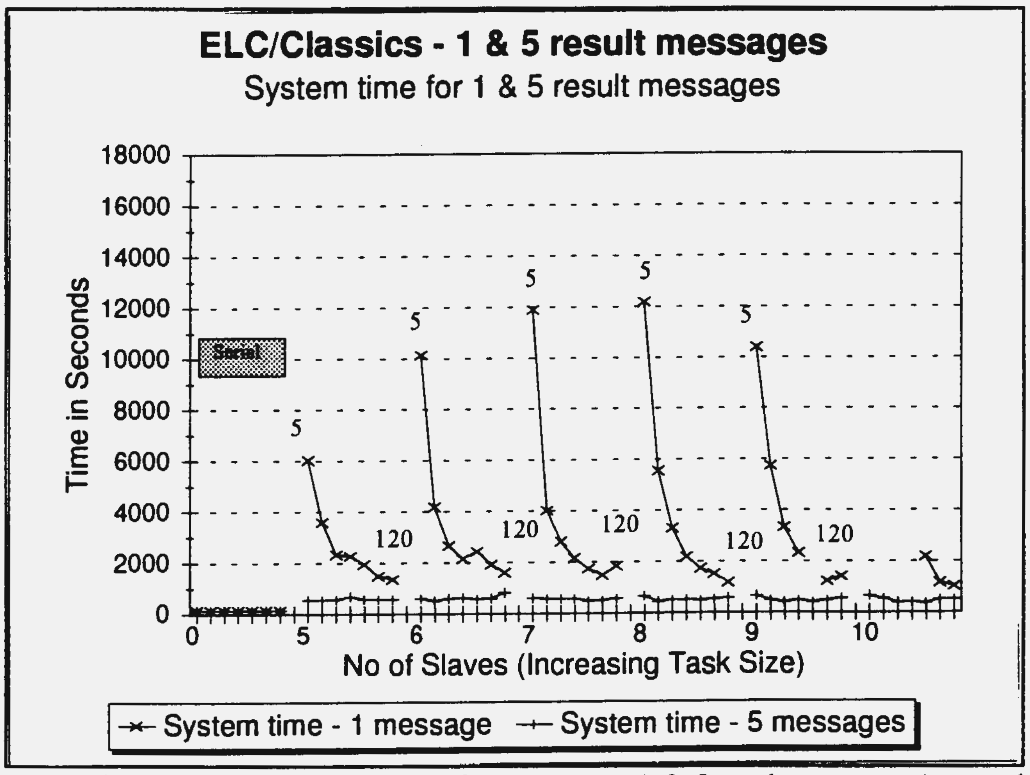

Figure 39: System times for **ELC/Classic** group - 1 & 5 result messages (grouped by number of slaves)

The system times of the master, for both versions of the program, are shown in Figure 39. This graph shows that the system time for the version with a single message is considerably greater than the corresponding system time for the 5-message version, which is relatively little, and remains fairly constant for all runs. The system time for the single message version also increases slightly as the number of slaves increases.

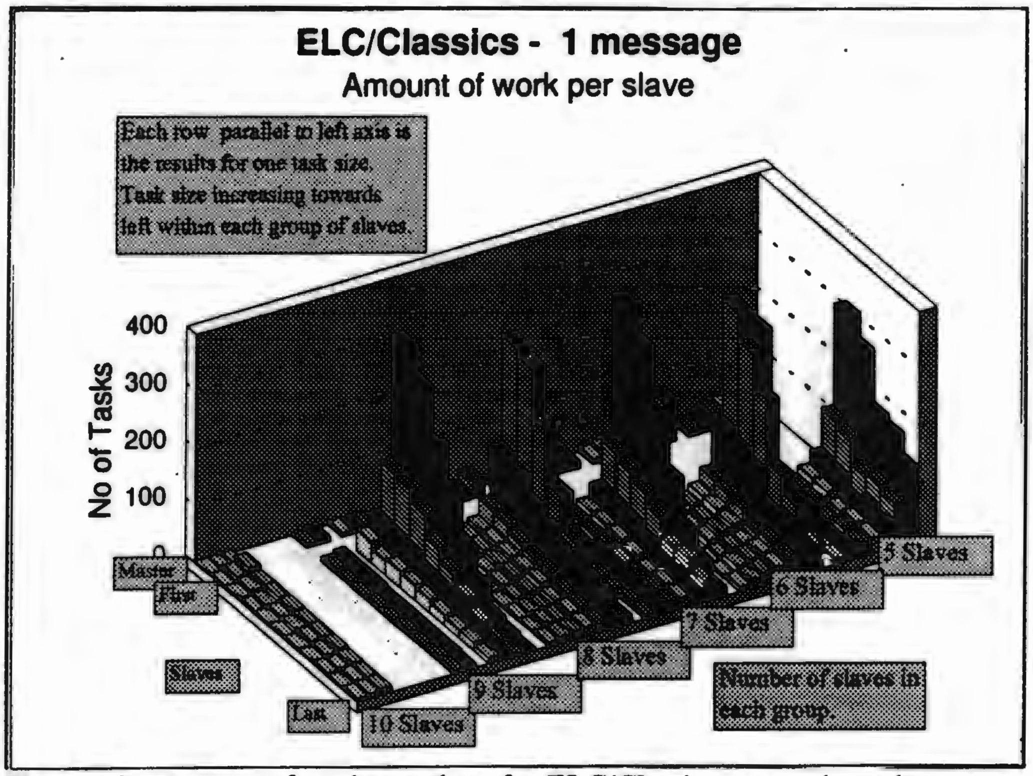

Figure 40: Amount of work per slave for **ELC/Classic** group - 1 result message (grouped by number of slaves)

The next graphs show that the serious degradation in performance, of the single message version, is due to network congestion, which is caused by Cloud itself.

Figure 40 shows the number of tasks executed by each processor, and indicates the load imbalance that arises because of communication bottlenecks. In this graph the number of tasks for the master are shown along the back wall of the graph. The number of tasks executed by the slaves is shown so that there is one row of slave data, from back to front, for each task size, for each number of slaves, and the slaves in each row are shown in the order in which they were started up. For each row, the slave that is nearest the back wall was the first slave started up by the master, and the slave nearest the front the last slave started up. This is also the order in which the master sends messages to the slaves. For each number of slaves, the rightmost row of slave data is for the smallest (5000 photon) task size, with each row of slave data, towards the left, the data for the next biggest task size.

Figure 40 shows that there is a drastic decrease in the number of tasks executed by each slave, according to the order the slaves were started up and received their tasks. For 5 slaves, although the decrease is quite apparent, slaves further "down the list" do still execute some tasks. For larger number of slaves, the graph shows that those slaves which are "last in the queue" for sending results messages, and for receiving further work, do almost no work. This is most serious for the small task sizes, where the time to execute one task is very short.

The corresponding graph for the five-message version is shown in Figure 31, in section 5.6.2. This graph shows, that for the five-message version, there was an even distribution of work between all the processors. Since the only difference between the versions of Cloud used was that one version returned five short results messages, and the other a single long results message, this proves that the difference in performance, and particularly the variation in the number of tasks executed by each slave, is caused by the different ways of returning the results messages.

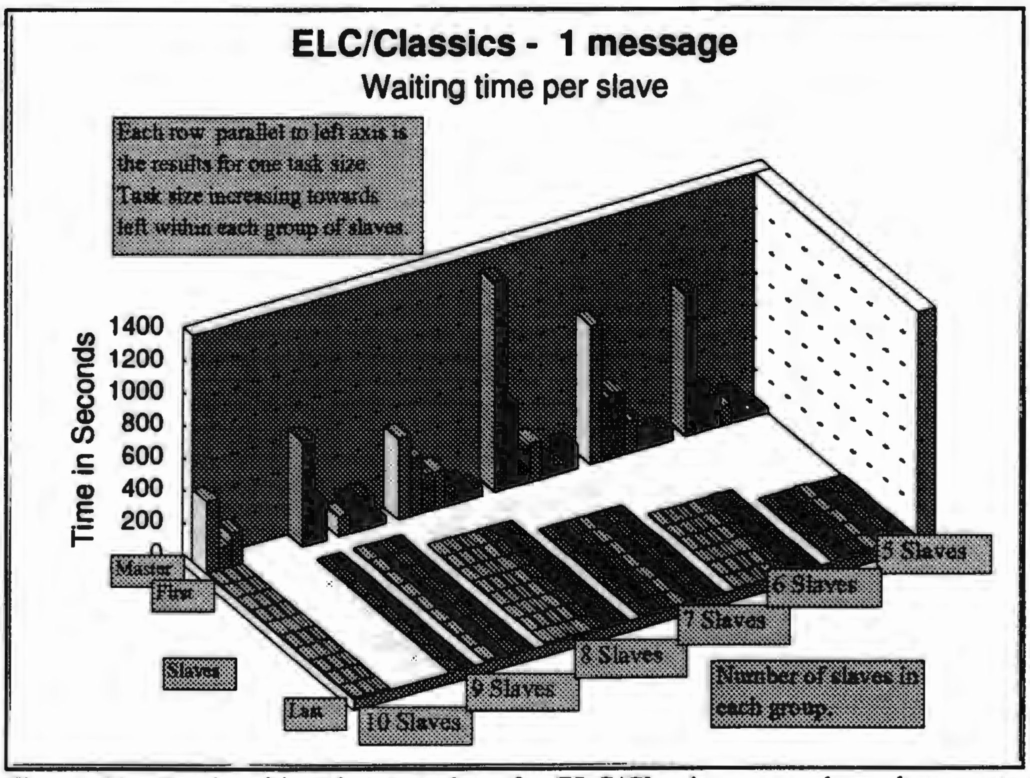

Figure 41: Total waiting time per slave for **ELC/Classic** group - 1 result message (grouped by number of slaves)

Figure 41 confirms that the degradation in performance for these slaves is not due to time spent waiting for messages from the master as, in all cases, the slaves spends almost no time waiting for messages. Therefore, it must be concluded that the degradation in performance is due to the communication delays, resulting from network congestion.

In a homogeneous system, such as the **ELC/Classic** group, all slaves would be expected to take about the same time to execute a task, and thus finish the tasks at the same time. Therefore, it is likely that the slaves will finish in the same order as they received their tasks, with a very small time separating them. Thus, the first slave to finish will be able to start sending its results immediately. Subsequent slaves must wait their turn to send. Ethernet requires that each processor must wait in between packets, to allow other processors a chance to transmit, so all slaves will have a chance. However, because they all finish at approximately the same time, there is increased chance of network contention. As soon as one slave pauses, after it has sent a packet, the remaining slaves will all try and send, as soon as they hear the pause, and this will result in collisions, with the processors backing off. For messages consisting of only a few packets, this will soon be resolved, but when all processors are trying to send very long messages, serious congestion arises. This is compounded when the time to process a task is very short, as for the small task sizes, as those slaves, which have succeeded in sending a results message, will complete processing the next task, and immediately try and send the results of this task, while the other slaves have not yet succeeded in sending the results of the previous task. This means, that for small task sizes, the network congestion will be continuous, throughout the entire run. This accounts for the exceptionally long elapsed run times, for the small task sizes, for the single-message version. (See Appendix B for further details on the operation of Ethernet).

The results in this section show that it is more efficient to send five shorter messages, than one long message. In the version with five shorter messages, there was a short time in between the sending of each message, when the data to be sent was copied into the message buffer, and then sent. This time gave the other slaves a chance to send a message, so there were fewer collisions. Also the messages were shorter, so each slave needed to send fewer packets, and any contention that did arise was resolved quickly. With the long messages the slave can do nothing else, while backing off and waiting for access to the network to send a message, and this is very inefficient. In this way, a library such as *PVM* is more efficient, since *PVM* is non-blocking. That is, *PVM* starts a daemon to sent the message and can meanwhile continue processing.

Keiser, in a comparison of Ethernet with a Token Ring, has also illustrated how, as the load on the network increases, performance deteriorates rapidly as a result of an increasing number of collisions [Keis89]. If the congestion is such, that the input rate of data at a node exceeds the output rate, this can result in buffer overflow, which may cause packets to be dropped and create the possibility of the whole network becoming deadlocked, with no packets getting through [Hamm88]. On Ethernet, if a packet is successfully received it can be assumed to be correct. However, according to the Ethernet specification, what happens to a packet, after the 16th collision in attempting to send this packet, depends on each implementation. In some implementations the packet may be abandoned.

The results in this section have shown that significant delays can occur when messages are very long, when there are too many messages, too short a time to process the messages, and too many slaves, and that these delays can be caused by the program itself. However, on a non-dedicated network congestion can also be caused by factors exterior to the program, such as other applications and users. Thus, congestion could occur even for small numbers of slaves and short messages. These problems must therefore be considered, when writing a program for a non-dedicated, distributed system.

5.7 Scalability and isoefficiency

When a real application is parallelized it is desirable that is should scale well if it is to be of practical use. Usually users will either want to add more processors to get a result in a shorter time, or run a larger problem, or both.

Gustafson's Law states that if the size of most problems is scaled up sufficiently, then any desired efficiency can be achieved on any number of processors [Wils93]. Essentially, if processors are added, a problem should be solved in a shorter time. However, as the number of processors increases, so do the communication overheads, which decreases the efficiency achieved. Gustafson's Law implies that if a desired level of efficiency is to be maintained then the communication/computation ratio must remain constant. Thus, if the communication overhead is increased, then the problem size must be increased, so as to maintain the proportions of communication and computation. For some applications this works well, but for others the algorithm used, or the amount of communication needed, may prevent linear scalability. Also, a larger problem size may require more memory, which may cause further constraints on performance [Sing93].

Isoefficiency means that the efficiency of a program remains constant as the number of processors is increased. Isoefficiency analysis is a means of determining the potential scalability of a program [Gram93][Kuma94] [Jako93].

It is obvious that if an application is scaled with the intent of obtaining a result in a fixed time, by increasing the number of processors, then, as the problem size increases, the potential scalability will eventually be bounded by such constraints as limited bandwidth. This is particularly likely in a network with low bandwidth such as Ethernet.

The issue of determining the scalability, or calculating the isoefficiency, of a program on a network of heterogenous workstations is a very difficult one. Singh and others show how this can be done for a homogenous system, but do not include the cost of bus or network contention. Kumar et al take contention into account in their formula for isoefficiency, but do not explain how to include heterogeneity in the formula [Kuma94].

Some other papers on this topic provide useful insight. Scaling is discussed in some detail in [Jako93] and [Müll91], who both develop formulae for predicting the scalability of a parallel program when considering the overheads arising from parallelism. Singh et al show that, instead of increasing the problem size, increased computing power can be used to reduce the errors arising from a simulation, so that scientists may achieve more accurate results ("scaling for error") [Sing93].

Cloud is potentially ideally scalable for several reasons. One reason is that the memory requirements for the master, and for each slave, are independent of both the task size and the number of processors used, so that the amount of memory used by the master and each slave will always be the same, regardless of changes in problem size or number of processors. Also the amount of memory required is relatively little, so even processors with small amounts of memory can be used. There is relatively little communication in this type of computation-intensive program, so the computation/communication ratio is favourable for good scalability. The size and number of messages required for communication between the master and slaves are also independent of the overall problem size, and number of processors, in the same way as the memory requirements. The number of messages is affected only by the granularity of the task size chosen.

The structure of the program is such that if the master has to communicate with too many slaves, this could easily become a bottleneck Indeed, these experiments have shown that network contention becomes an inhibiting factor for as few as 10 slaves when these are run by an ELC master. This is primarily because of the homogeneous nature of the slaves, which all try to send messages at approximately the same time. In this matter heterogeneity would be an advantage, as the sending of messages would be staggered and less likely to cause congestion, as described in section 5.5.4. This contention with 10 slaves occurs before the master has reached the stage where it cannot handle any more slaves thus causing the slaves to wait for the master. For the communication needed for 1 task the SGI2 Extreme spends on average 10% of the time the ELC master requires. This suggests that a network of SGI machines would scale better than one of Suns. If necessary, the program could be easily restructured so as to allow several "cluster masters", each handling some of the slaves, as described in section 6.7.2. This would enhance the scalability of the program, which would not then be limited by the number of slaves that one master could handle.

Despite the good potential scalability of this program, the small numbers of machines used in these experiments, and the widely different performance capabilities of these machines, make it extremely difficult to predict to what extent this program is scalable. Even if the potential scalability is calculated for various groups of machines, there are too few machines to confirm by experiment whether this is valid. It is also likely that network contention will limit the speedup that could be achieved. The potential scalability would be greatly improved if a higher speed communication network than Ethernet was used.

Chapter 6

Conclusions

This dissertation has shown that good parallel performance can be achieved by running a real scientific application on a network of heterogeneous workstations, but that this performance is affected by a number of factors.

The results of these experiments raised the question of what was the best way to evaluate the performance of a parallel program on a heterogeneous network. A number of methods of evaluating this performance were investigated. These included the conventional methods of speedup and efficiency, and also the alternative method of "linear speed" as proposed by Crowl [Crow94].

In addition, we proposed "perfect linear speed", and "linear efficiency" as extensions to Crowl's work in evaluating the parallel performance of heterogeneous systems.

This study has provided valuable insight into the performance of a real scientific application on a network of heterogeneous workstations. This is particularly important since most networks are heterogeneous, either because they consist of workstations of different makes and models, with different CPUs and differing amounts of memory and cache, or because they are shared environments, and the load on the workstations and the network is constantly changing.

These experiments have shown how such an existing, under-utilized network of workstations can provide a considerable amount of computing power, which can be exploited for parallel processing, with minimal impact on other users.

As described in this dissertation, Cloud has been parallelized for the Department of Meteorology at Pennsylvania State University in the USA. Scientists in this department are reported to be well satisfied with this parallel version of Cloud, and are now extending the meteorological sections of Cloud, so that it can be used for a number of research projects, which could not be done before because the original serial program just took too long. These projects are intended to take place soon, and the researchers at PennState expect to write a number of papers describing results obtained by using the parallel version of Cloud studied in this thesis.

This chapter also includes some conclusions regarding the usefulness of the *p4* library as a tool for the parallelization of an application for a distributed network of heterogeneous workstations.

6.1 Findings of this thesis

The findings of this thesis can be grouped into three sections: the overall performance achieved, the evaluation of the performance of heterogeneous systems, and the factors affecting this performance.

6.1.1 Performance achieved

This dissertation has shown that a considerable improvement in performance was achieved in the parallelization of a real scientific application for a network of heterogeneous workstations. The experiments with Cloud showed that:

- the performance range for this heterogeneous network was approximately 6.5, with the serial time of the fastest processor of 51 minutes, and the serial time of the slowest processor of 5.5 hours;
- the fastest parallel time was 16 minutes for all 18 processors, giving a speedup of 14 for 18 processors, and an efficiency of 0.8;
- the best performance will be achieved if the processors used are reasonably similar in performance capability. This study showed that adding four very slow processors as slaves only reduced the elapsed time from 18 minutes to 16 minutes, and that these four slow processors could have been better utilized for other work, as the reduction of under 2 minutes is too small to make it worthwhile to use these processors as slaves.

6.1.2 Evaluation of the parallel performance of heterogeneous networks

This study showed that there is considerable difficulty in evaluating the performance of a heterogeneous system. These experiments showed that:

- the performance of a parallel program should be measured by using elapsed time, and that measuring the CPU time alone can be very misleading. However, the CPU and system times may give useful insight in understanding the performance of an application, and may indicate ways in which performance can be improved;
- speedup and efficiency are inappropriate for evaluating the performance of a heterogeneous network, and may give misleading information. However, if speedup is to be used to measure the parallel performance of a heterogeneous system, then the most representative values will be obtained if the parallel performance of the system is compared to the mean of the serial performance of the processors used;
- alternative methods of measuring performance, such as linear speed, as proposed by Crowl [Crow94], and linear efficiency as suggested in this dissertation, give a more representative indication of the actual improvement in performance that is achieved.

6.1.3 Factors affecting the performance

There were a number of factors which affected the performance that could be achieved on this system. The experiments conducted for this study showed that:

- although good performance can be achieved for Cloud on a network of workstations, the network performance was very sensitive to a number of factors. The results of the experiments with Cloud indicated that this sensitivity occured when the network traffic was very "bursty", or when the network became saturated when there were too many messages being sent at the same time, or if the messages were very long (greater than 200kb). These factors caused a serious deterioration in the data rate, and the network degradation was so serious that messages were lost.

This poor performance of Ethernet can be alleviated to some extent by reducing the "burstiness" of the traffic that may occur, particularly on a homogeneous system, when all processors try and send at approximately the same time. This can be done by interspersing sending messages with computation, or by using heterogeneous machines which will take different times to finish a task, and thus transmit results at different times. Also, breaking long messages (greater than 30 kb) into more, shorter messages, of sizes between 10 kb and 30 kb will improve network throughput.

Network latency can be hidden by overlapping communication with computation. A reduction of 25%-30% in the overall elapsed time can be achieved by sending a spare task to each slave, so that each slave always has a spare task on "stand-by" and can immediately proceed with processing the next task, without having to wait for the time of the round-trip communication of results to the master, and receiving a new task from the master. No benefit was achieved by sending more than one spare task, and this can even reduce performance;

The results of the experiments also confirm that, as shown by others, the specified bandwidth for Ethernet of 10 Mbits/second cannot be achieved in practice. For this study the best data rate of 7-8 Mbits/sec was achieved by the Silicon Graphics machines. The SPARCstation 1+ machines had a data rate of about 6.5 Mbits/sec, and the ELC and Classics a rate of about 6 Mbits/sec. This was consistent with the findings of other researchers [Cap93][Nana93][Gärt93][Alte93];

- it is inefficient to run a slave process on the same processor as the master, and that better results can be achieved if the master does slave work within the master process, rather than in a separate process on the same processor.
- better load balancing is achieved for small task sizes, but higher efficiency is achieved for large task sizes. The best performance is obtained when there is the best tradeoff between the good communication/computation ratio possible for large task sizes, and the efficient load balancing for small task sizes. On a homogeneous system all machines will tend to finish at approximately the same time, so a relatively large task size can be used, and good load

balancing will still be achieved. On a heterogeneous system, with a wide range of performance capabilities, a large task size will lead to poor load balancing, and better results can be achieved with a small task size.

- it is important when choosing a task size that will ensure that the time taken to execute a task is longer than the time needed for the master to handle all the communication for tasks for all slaves, or else communication bottlenecks and delays will occur.

6.2 Usefulness of the *p4* library as a parallelization tool

This section gives a brief evaluation of the parallel library *p4*, and its usefulness in the parallelization of Cloud for a network of heterogeneous workstations.

p4 has proved to be simple to learn and implement, and efficient in its execution. The debugging tool, which allows a user to display error messages at different levels of complexity, was very useful during program development.

The portability of *p4* was well demonstrated in this study. The *p4* library was obtained from Argonne National Laboratories and installed at the University of Cape Town. Cloud was then parallelized, using *p4*, for five different models of workstation, of two different architectures. This parallel version of Cloud was then used successfully on both homogeneous and heterogeneous groups of up to 18 Sun and Silicon Graphics workstations. Cloud, and the *p4* library, were then successfully installed on Sun workstations at PennState, where Cloud was recompiled and ran successfully, with no problems at all.

The efficiency of *p4* was shown by the good performance results obtained for Cloud, and by the good data rates achieved on Ethernet with *p4*. These data rates were comparable to those achieved by other researchers using methods of communication other than *p4*, and according to other researchers, were close to being the optimal data rates that could be achieved on Ethernet.

There were a number of problems found in running a program implemented with *p4*, and this was also consistent with the experiences of Sukup [Suku94]. In both studies there were occasional problems with slave processes being lost, or crashing at the beginning of runs, and also with slaves occasionally hanging for unknown reasons. These problems show that a *p4* application must be written to be fault tolerant, so that if there is a problem with lost message, or insufficient memory, then both the slave and the master processes should be able to recover. If one or more slaves fail, the application should be able to continue, as long as at least the master, and possibly one slave, are still functioning.

There was also a slight problem with using *p4* on the SPARCclassics. The SPARCclassics functioned well as slaves, but could not be used as a master. This was because the *p4* library for the Sun workstations was implemented on a SPARCstation 1+, which is the sun4 architecture, and the SPARCclassics are sun4m architecture, and there is a difference in the way the SPARCstation 1+ and SPARCclassic machines start a remote shell. The ELCs (sun4 architecture) appear to use the same method of starting remote shells as the SPARCstation 1+s, and so could be used as masters. The *p4* library would need to be modified if SPARCclassics are to be used as masters.

One of the biggest advantages of *p4* is that it is the only parallel library that implements both a shared memory and a message-passing model, as well as being suitable for implementing a parallel program on networks of clusters of shared-memory multiprocessors. The library functions for shared-memory were not used in this study, but it is relatively simple to convert from a *p4* message-passing implementation to a *p4* shared-memory implementation. Another advantage of *p4* is that a message-passing implementation can also be run on a shared-memory multiprocessor, or on clusters of shared-memory multiprocessors connected by a network, as the send/receive procedures are generic and it does not matter whether a message must travel across a network, or through shared-memory, or via some other mechanism [Butl92].

6.2.1 *MPI*

With the development of the new standardized message-passing interface, *MPI* [Walk94], it is likely that most future work in parallelizing message-passing applications will be done using *MPI*. *MPI* incorporates many of the features of the more well-established parallel libraries such as *p4*, *PVM*, and *PARMACS*, and existing programs that were written using these libraries should be fairly easily reimplemented in *MPI*.

Future work, along the same lines as this thesis, should rather be implemented using *MPI*, since this is the new standard for message-passing systems.

6.3 Future work

This dissertation has shown that there are a number of factors that affect parallel performance. This section shows some ways in which the performance could be improved by redesigning of the program, and also suggests some possible research projects that could build on what has been learnt in this study.

6.3.1 Using a fast or a slow master?

Further work should be done to establish whether the best performance results from using a fast master or a slow master.

All experiments in this dissertation used a fast master because, even in a homogeneous group, the master executes more tasks in the same time, than can be executed by a slave. This is because the master has no communication overheads when doing Monte Carlo work. This suggests that it is advantageous to use the fastest machine as the master, because this machine can be used to do slave work whenever it is not otherwise engaged with administrative tasks, such as communication with slaves.

Also, as discussed in section 5.7, with the master/slave paradigm, the master is likely to become too busy with communication, thus causing a bottleneck. Thus, it would also be better if the master was the machine that can handle communication fastest, as the master has to do more communication than the slaves.

However, it may be better to use a slow master which handles communication only, and does not do any Monte Carlo work, and thus use the faster processors as slaves where their processing power can be better utilized, as a slave spends more time on processing than on communication.

6.3.2 Better utilization of idle workstations

The results in this dissertation have shown that, when the load balancing was poor, the master was idle for some time at the end of a run, while waiting for the slaves to complete their last tasks, and also, some of the faster processors finished before the slower processors had completed their last task. Also, Cloud is implemented so that the master initially sends two tasks to each slave, so each slave always has its next task immediately available. However, at the end of a run this can lead to even worse load balancing, as while the faster processors may have finished all the available work, the slower processors may still have spare tasks in their task queues.

The wasted time of the faster slaves, and idle time of the master, could be better utilized if these faster processors instead did some of the work allocated to other, still busy, slow slaves, especially if the tasks reallocated were those still in the standby queue of these slaves. In this case, the slaves must be constrained to receive a "CLOSE" message, if there is one, before any task in its queue, to prevent a slow slave from executing a task in its queue, when that task has now been reallocated, and completed by another faster processor.

As described in section 3.3.6, this feature to make better use of idle workstations was originally programmed into Cloud, but was not used in the performance tests. It should, however, be activated for real use. This feature may lead to some duplication of work, as sometimes a task may be executed by more than one processor, but the overall elapsed time will be shorter.

6.3.3 Cluster masters

This study has shown that communication bottlenecks can occur if there are too many slaves, and the master does not have sufficient time to handle all the communication from all slaves, for one set of tasks, in the time the slaves take to execute the next task.

This problem could be solved if the program was redesigned so that there are several "cluster masters", each administering a sub-group of slaves, and one overall master to collate the results from these "cluster masters". There would then be less chance of communication bottlenecks as each "cluster master" would only have to administer a few slaves.

6.3.4 Random size tasks

The results of the experiments described in section 5.5 have shown that in a homogeneous system all slaves will finish their tasks at approximately the same time, and this causes bursts of network traffic as all slaves try and transmit results at the same time.

This could be alleviated by sending tasks of different sizes to each slave, perhaps by getting the master to send tasks of "random" size.

6.3.5 Dynamic adaptive load balancing

This section describes a method of load balancing which could be implemented, in Cloud, to improve overall efficiency, and thus result in shorter elapsed times. The method is dynamic because the optimal size of the tasks is determined during the run, and adaptive because it takes into account the different performance capabilities of the processors, and the dynamically varying workloads on machines in a shared environment, and adapts the size of the tasks accordingly.

The results in this dissertation have shown that load balancing is best for small task sizes, but that efficiency is greatest for large task sizes, which have a better communication/computation ratio. The load balancing for large tasks was poor, resulting in a certain amount of processor idle time, which could be better utilized. The best overall performance is achieved when there is the best tradeoff between a good communication/computation ratio, and efficient load balancing.

If the program is changed to process large tasks for most of the run, this will improve the communication/computation ratio, and result in high efficiency. Then, the size of the task should be reduced in the last stages of the run, so the final load balancing will be good, and all processors will finish at approximately the same time.

The master can determine the comparative rates of the processors by measuring the time taken, by each machine, to perform each task. This time can easily be returned by the slave, as part of the results for each task. The master can then keep a table of the current performance rates for each slave, and use this in determining the size of the task to send to each slave. This is particularly useful in a shared environment when the load on a machine may vary considerably during a run.

The initial tasks sent by the master to the slaves should be relatively small, as the situation is unknown at that point. The times returned by the slaves, to the master, can then be used by the master for dynamic determination of the performance of each slave, so that the master allocates larger amounts of work to faster machines, and lesser to slow machines. As the situation changes, and the performance of the slaves varies, the master can adjust the amount of work sent to each slave. In some ways, this technique is similar to that of THE PARFORM which is described in [Cap93].

Towards the end of a run, it is extremely important to determine the time at which the master must change from sending out large tasks, to sending small tasks. This is so that a situation does not arise where all the faster processors have finished their tasks, while the slow machines are still working. The point at which the master must start sending smaller tasks can be established by comparing the speed of the fastest and slowest machines, and then not sending large tasks to a slow machine, when this machine will take longer to finish a task than the rest of the processors will take to finish all remaining work. At this stage, all remaining work should be divided into much smaller tasks, so the final load balancing will be good.

This combination of using large task sizes for most of the run, and small ones at the end to ensure good load balancing, will result in higher efficiency, and shorter elapsed times. The overheads introduced by this technique should be minimal, as no extra communication is required, and the extra memory requirements and computation required are small.

This method of load balancing, although ideally suited for the type of program where synchronization is not an issue, could be extended for use in programs with more complicated load balancing and synchronization requirements such as data-parallel programs.

Further suggestions of load balancing techniques can be found in [Kuma94].

6.3.6 Intelligent task allocation

In Cloud, the largest amount of communication is when the collection arrays are returned from a slave at the end of each sub-task. This section suggests a technique for allocating tasks to slaves, which will result in reduced communication time, thereby improving performance.

This technique can be implemented for Cloud, because for the Monte Carlo method used, the totals in the collection arrays for sub-tasks, for the same interval of the same wavelength, are added together before calculating the final results. In the current implementation of Cloud, the master receives the collection arrays for each sub-task, from each slave, and adds the totals for the same interval of the same wavelength together. The method in this section proposes that if a slave is specifically allocated subsequent sub-tasks, all for the same interval of the same wavelength, then the slave can add these totals together, without returning the collection arrays to the master at the end of every task, and this will reduce communication costs.

Since the input arrays total to about 210 kb, this technique would reduce the amount of communication to a fraction of that needed with the present implementation. Cloud has been programmed so that it is relatively trivial to implement this technique. However, there was insufficient time to repeat all the experiments using this method. The added processing cost would be minimal, and considerably less than the time otherwise needed for communication. In this way, relatively small sub-task sizes could be used, and the load balancing efficiency of small tasks could be achieved. However, a communication/computation ratio approaching that of the larger tasks could be achieved, because of the reduced communication overhead.

The simulation of **all** photons for one case of input conditions cannot be allocated to one slave initially, as it is unlikely that the work divided in this way would divide evenly among the slaves, and this would lead to poor load balancing.

References

[Alte93] Altevogt, P., and Linke, A.,
Parallelization of the two-dimensional Ising Model on a cluster of IBM RISC System/6000 workstations,
Parallel Computing,
19(9):1041-1052, September 1993.

[Bail92] Bailey, D.H., Barscz, E., Dagun, L., and Simon, H.D.,
NAS Parallel Benchmark Results,
RNR Technical Report RNR-92-002, NASA Ames Research Center, December 1992, quoted by [Bell94]

[Bail93] Bailey, D.H., and Barszcz, E.,
NAS Parallel Benchmark Results,
IEEE Parallel & Distributed Technology,
1(1):43-51, February 1993.

[Bers88] Bershad, B.N., Lazowska, E.D. and Levy, H.M.,
PRESTO: A System for Object-oriented Parallel Programming,
Software - Practice and Experience,
18(8):714-732, August 1988.

[Beck93] Becker, J.C., and Dagum, L.,
Particle simulation on heterogeneous distributed supercomputers,
Concurrency: Practice and Experience,
5(4):367-377, June 1993.

[Bell94] Bell, G.,
Scalable Parallel Computers: Alternatives, Issues, and Challenges, *International Journal of Parallel Programming,*
22(1):3-46, February 1994.

[Berr91] Berry, M., Cybenko, G., and Larson, J.,
Scientific benchmark characterizations,
Parallel Computing,
17(10&11):1173-1194, December 1991.

[Bete93] Betello, G., Richelli, G., Succhi, S., and Ruello, F.,
Lattice Boltzmann method on a cluster of IBM RISC System/6000 workstations,
Concurrency: Practice and Experience,
5(4):359-366, June 1993.

[Bird94] Birdsall, J. W.,
The Sun Hardware Reference,
comp.sys.sun.hardware,
Wed Sep 28 08:55:53 1994.
(jwbirdsa@picarefy.com)

[Butl94] Butler, R.M. & Lusk, E.L.,
Monitors, Messages, and Clusters: the p4 Parallel Programming System,
Parallel Computing,
20(4):547-564, April 1994.

[Calk94] Calkin, R., Hempel, R., Hoppe, H.-C., and Wypior, P.,
Portable programming with the PARMACS message-passing library,
Parallel Computing,
20(4):615-632, April 1994.

[Cap93] Cap, C.H. and Strumpen, V.,
Efficient parallel computing in distributed workstation environments,
Parallel Computing,
19(11):1221-1234, November 1993.

[Carr89] Carriero, N.J., Gelernter, D.,
Linda in Context,
Communications of the ACM,
32(4):444-458, 1989.

[Carr94] Carriero, N.J., Gelernter, D., Mattson, T.G., and Sherman, A.H.,
The Linda alternative to message-passing systems,
Parallel Computing,
20(4):633-656, April 1994.

[Crow94] Crowl, L.A.,
How to Measure, Present, and Compare Parallel Performance,
IEEE Parallel & Distributed Technology,
2(1):9-25, Spring 1994.

[Cybe90] Cybenko, G., Kipp, L., Pointer, L., and Kuck, D.,
Supercomputer Performance Evaluation and the Perfect Benchmarks,
ACM SIGARCH Computer Architecture News,
18(3):254-266,
Conference Proceedings, 1990 International Conference on Supercomputing, Amsterdam, Netherlands, June 1990.

[Denn94] Dennis, J.B.,
Machines and Models for Parallel Computing,
International Journal of Parallel Programming,
22(1):47-78, February 1994.

[deKe93] de Keyser, J. and Roose, D.,
Load balancing data parallel programs on distributed memory computers,
Parallel Computing,
19(11):1199-1220, November 1993.

[Dixi91] Dixit, K.M.,
The SPEC benchmarks,
Parallel Computing,
17(10&11):1195-1210, December 1991.

[Dona94] Donaldson, V., Berman, F., and Paturi, R.,
Program Speedup in a Heterogeneous Computing Network,
Journal of Parallel and Distributed Computing,
21(3):316-322, June 1994.

[Eage89] Eager, D.L., Zahorjan, J., and Lazowska, E.D.,
Speedup Versus Efficiency in Parallel Systems,
IEEE Transactions on Computers,
38(3):408-423, March 1989.

[Ewin94] Ewing, R.E., Sharpley, R.C., Mitchum, D., O'Leary, P. and Sochacki, J.S.,
Distributed Computation of Wave Propogation Models using PVM,
IEEE Parallel & Distributed Technology,
2(1):27-31, Spring 1994.

[Fisc91] Fischer, D.,
On superlinear speedups,
Parallel Computing,
17(9):695-697, September 1991.

[Gärt93] Gärtel, U., Joppich, W., and Schüller, A.,
Parallelizing the ECMWF's weather forecast program: the 2D case,
Parallel Computing,
19(11):1413-1425, November 1993.

[Gärt93a] Gärtel, U., Joppich, W., and Schüller, A.,
First results with a parallellized 3D weather prediction code,
Parallel Computing,
19(11):1427-1429, November 1993.

[Glei88] Gleick, J.,
Chaos, Making a New Science,
Sphere Books 1988.

[Gram93] Grama, A.Y., Gupta, Anshul, and Kumar, V.,
Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures,
IEEE Parallel & Distributed Technology,
1(3):13-21, August 1993.

[Grim93a] Grimshaw, A.S., Strayer, W.T. and Narayan, P.,
Dynamic Object-Oriented Parallel Processing
IEEE Parallel & Distributed Technology,
1(2):33-48, May 1993.

[Grim93b] Grimshaw, A.S., West, E.A. and Pearson, W.R.,
No Pain and Gain! - Experiences with Mentat on a Biological Application
Concurrency: Practice and Experience,
5(4):309-328, June 1993.

[Grim94] Grimshaw, A.S., Weissman, J.B., West, E.A. and Loyot, E.C., Jr.,
Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems
Journal of Parallel and Distributed Computing,
21(3):257-270, June 1994.

[Gust88] Gustafson, J.L.,
Reevaluating Amdahl's Law,
Communications of the ACM,
31(5):532-533, May 1988.

[Hals85] Halsall, F.,
Introduction to Data Communications and Computer Networks,
1985, Addison-Wesley Publishing Company Inc., 269pp.
ISBN 0-201-14547-2, ISBN 0-201-14540-5 pbk

[Hamm88] Hammond, J.L., & O'Reilly, P.J.P.,
Performance Analysis of Local Computer Networks,
1986, reprinted 1988 with corrections,
Addison-Wesley Publishing Company Inc., 410pp.
ISBN 0-201-11530-1

[Henn90] Hennessy, J., & Patterson, D.,
Computer Architecture: A Quantitative Approach,
1990, Morgan Kaufmann Publishers, Inc., 594pp.
ISBN 1-55880-069-8

[Hey91] Hey, A.J.G.,
The Genesis distributed memory benchmarks,
Parallel Computing,
17(10&11):1275-1283, December 1991.

[Hira94] Hiranandani, S., Kennedy, K, and Tseng, C-W,
Evaluating Compiler Optimizations for Fortran D,
Journal of Parallel and Distributed Computing,
21(1):24-45, April 1994.

[Hock91] Hockney, R.W.,
Performance parameters and benchmarking of supercomputers,
Parallel Computing,
17(10&11):1110-1130, December 1991.

[Home94] Homer, P.T. and Schlichting, R.D.,
A Software Platform for Constructing Scientific Applications from Heterogeneous Resources,
Journal of Parallel and Distributed Computing,
21(3):301-315, June 1994.

[Jako93] Jakobs, A., and Gerling, R.W.,
Scaling aspects for the performance of parallel algorithms,
Parallel Computing,
19(9):1063-1073, September 1993.

[Keis89] Keiser, G.E.,
Local Area Networks,
1989, McGraw-Hill Book Company, 420pp.
ISBN 0-07-033561-3

[Kenn94] Kennedy, K.,
Compiler Technology for Machine-Independent Parallel Programming,
International Journal of Parallel Programming,
22(1):79-98, February 1994.

[Klei92] Kleinrock, L., and Huang, J-H,
On Parallel Processing Systems: Amdahl's Law Generalized and Some Results on Optimal Design,
IEEE Transactions on Software Engineering,
18(5):434-447, May 1992.

[Krem93] Kremien, O., Kramer, J. and Magee, J.,
Scalable, Adaptive Load Sharing for Distributed Systems,
IEEE Parallel & Distributed Technology,
1(3):62-70, August 1993.

[Kuck94] Kuck, D. J.,
What Do Users of Parallel Computer Systems Really Need?
International Journal of Parallel Programming,
22(1):99-127, February 1994.

[Kuma94] Kumar, V., Grama, A.Y., and Vempaty, N.R.,
Scalable Load Balancing Techniques for Parallel Computers,
Journal of Parallel and Distributed Computing,
22(1):61-79, July 1994.

[Lin93] Lin, R-F.,
A Monte Carlo Simulation of the Reflectivity over a Stratocumulus Cloud Deck,
An M.Sc. thesis in Meteorology, Department of Meteorology, Pennsylvania State University,
May 1993.

[Ma94] Ma, C-m.,
Implementation of a Monte Carlo code on a parallel computer system,
Parallel Computing,
20(7):991-1005, July 1994.

[Mach92] Machanick, P.,
SpaceLib: A Library for Spatially Decomposed Shared Memory Multiprocessor Applications,
Computer Science Department, Stanford University, 1992.

[McBr94] McBryan, O.A.,
An overview of message passing environments,
Parallel Computing,
20(4):417-444, April 1994.

[Minn93] Minnich, R.G., and Pryor, D.V.,
Radiative Heat Transfer Simulation on a SPARCStation Farm, *Concurrency: Practice and Experience,*
5(4):345-357, June 1993.

[Müll91] Müller-Wichards, D.,
Problem size scaling in the presence of Parallel Overhead,
Parallel Computing,
17(11):1361-1376, December 1991.

[Nana93] Nanayakkara, A., Moncrieff, D. and Wilson, S.,
Performance of IBM RISC System/6000 workstation clusters in a quantum chemical application,
Parallel Computing,
19(9):1053-1062, September 1993.

[Nede93] Nedeljkovic, N., and Quinn, M.J.,
Data-parallel programming on a network of heterogeneous workstations,
Concurrency: Practice and Experience,
5(4):257-268, June 1993.

[Ponn93] Ponnusamy, R., Thakur, R., Choudhary, A., Velamakanni, K., Bozkus, Z. and Fox, G.,
Experimental Performance Evaluation of the CM-5,
Journal of Parallel and Distributed Computing,
19(3):192-202, November 1993.

[Scha93] Schaeffer, J., Szafron, D., Lobe, G. and Parsons, I.,
The Enterprise Model for Developing Distributed Applications,
IEEE Parallel & Distributed Technology,
1(3):85-96, August 1993.

[Schn93] Schneckenburger, T.,
Efficiency of Parallel Programs in Multi-tasking Environments,
in *PEPS Performance Evaluation of Parallel Systems,*
University of Warwick, Great Britain,
75-82, November 1993,

[Sela94] Sela, J.G., Anderson, P.B., Norton, D.W., and Young, M.A.,
Massive Parallelization of NMC's Spectral Model,
Journal of Parallel and Distributed Computing,
21(1):140-149, April 1994.

[Sing92] Singh, J.P., Weber, W-D, & Gupta, Anoop,
SPLASH: Stanford Parallel Applications for Shared-Memory, *Computer Architecture News,*
20(1):5-44, March 1992.
Also Stanford University Technical Report No. CSL-TR-92-526, June 1992.

[Sing93] Singh, J.P., Hennessy, J.L. and Gupta, Anoop,
Scaling Parallel Programs for Multiprocessors: Methodology and Examples,
Computer,
43-51, July 1993.

[Suku94] Sukup, F.,
Efficiency Evaluation of Some Parallelization Tools on a Workstation Cluster Using the NAS Parallel Benchmarks,
Computing Center, Vienna University of Technology, Austria, Technical Report No. ACPC/TR 94-2, January 1994.

[Sun91] Sun, X.-H., and Gustafson, J.L.,
Towards a better parallel performance metric,
Parallel Computing,
17(10&11):1093-1109, December 1991.

[Sund94] Sunderam, V.S., Geist, G.A., Dongarra, J., and Manchek, R.,
The PVM concurrent computing system: Evolution, experiences, and trends,
Parallel Computing,
20(4):531-546, April 1994.

[Walk94] Walker, D.W.,
The design of a standard message passing interface for distributed memory concurrent computers,
Parallel Computing,
20(4):657-674, April 1994.

[Weic91] Weicker, R.P.,
A detailed look at some popular benchmarks,
Parallel Computing,
17(10&11):1153-1172, December 1991.

[Wils93] Wilson, G.V.,
A Glossary of Parallel Computing Terminology,
IEEE Parallel & Distributed Technology,
1(1):52-67, February 1993.

[Yen93] Yen, I-L., Leiss, E.L., and Bastani, F.B.,
Exploiting Redundancy to Speed Up Parallel Systems,
IEEE Parallel & Distributed Technology,
1(3):51-60, August 1993.

Bibliography

Boyle, J., Butler, R., Disz, T., Glickfelt, B., Lusk, E., Overbeek, R., Paterson, J., & Stevens, R.,
Portable Programs for Parallel Processors,
Holt, Rinehart & Winston, Inc., New York, 1987.

Butler, R., & Lusk, E.,
User's Guide to the p4 Parallel Programming System,
Argonne National Laboratory, Argonne, IL 60439-4801, October 1992.

Carriero, N.J., and Gelernter, D.,
How to Write Parallel Programs: A First Course,
MIT Press, Cambridge, MA, 1990.

Herrarte V., & Lusk, E.,
Studying Parallel Program Behaviour with Upshot.

Machanick, P.,
SpaceLib: A Library for Shared Memory Parallel Applications,
Department of Computer Science, University of Cape Town, 1993.

Peixoto, J.H., and Oort, A.,
Physics of Climate,
pp 1-7, pp 450-479, 1992.

Tyson, P.D., & Preston-Whyte, R.A.,
The Atmosphere & Weather of Southern Africa,
1988, Oxford University Press, 374pp.

Appendix A

Description of the cloud radiation model

A stratocumulus cloud deck usually covers several hundred square kilometres. To study the reflectivity, transmissivity and absorptivity of such a cloud, a model has been set up. In this model a horizontal strip through the cloud is studied, with the cloud on either side of the strip assumed to continue to infinity. This strip is then divided into a number of identical vertical columns, each with a square horizontal cross-section. Each column is considered separately. In this way, different input data representing the atmospheric conditions, such as gas composition, water vapour content, and temperature, can be used for each column of cloud

As photons are incident on the top of the cloud column, they either penetrate the cloud, or are reflected off the top of the cloud. If a photon penetrates the cloud it is likely that within a certain distance, known as the mean free path, it will collide with a molecule of gas or water vapour. As a result of this collision it will either be absorbed in the cloud, or will continue passing through the cloud in a different direction until another collision. For all photons each photon is traced, until eventually it is either absorbed in the cloud, or reflected back out of the top of the cloud, or finally transmitted right through the cloud, and emerges from the bottom of the cloud. For this simulation it is assumed that the column of cloud studied is not at the edge of the cloud, so photons emerging from the sides of the column are 'wrapped round' to appear to come in from the other side of the column. Thus, all photons continue passing through the column of cloud until they are either absorbed, or emerge out the top (reflection), or the bottom (transmission) of the column.

Several collection arrays are used to count the photons reflected, transmitted or absorbed. Each cloud column has a square horizontal cross-section, and this is represented by a square grid divided into an equal number of collection cells in each direction. There is one such collection array representing the top of the cloud column, and one representing the bottom of the cloud column. As the photon passes through the cloud column, its new position after each collision is calculated. If it leaves the cloud through the bottom, or top, of the cloud column, it is counted in the collection array cell though which it emerges. At the same time the angle at which it leaves the cloud is recorded. For this purpose, there are another two collection arrays, one for the top, and one for the bottom of the cloud. These arrays have the same number of collection cells as those for counting the photons. As the photon leaves the cloud its angle (in radians) is added to the total in the corresponding cell. At the end of the run the mean angle of exit of photons leaving through that grid cell is calculated.

A schematic diagram showing a view of the top of the cloud column is shown in Figure 42. The strip of cloud is in the x direction, and the cloud is assumed to be infinite in the y direction. A square grid representing the collection array at the top of the cloud column is shown. This is schematic only. In the program the size of this array may be varied by the user. For the experiments described in this dissertation the array was 93 x 93. The view of the cloud column from the bottom will be similar.

These four collection arrays are used to compute the reflection of light from the top of the cloud and the transmissivity of light through the cloud.

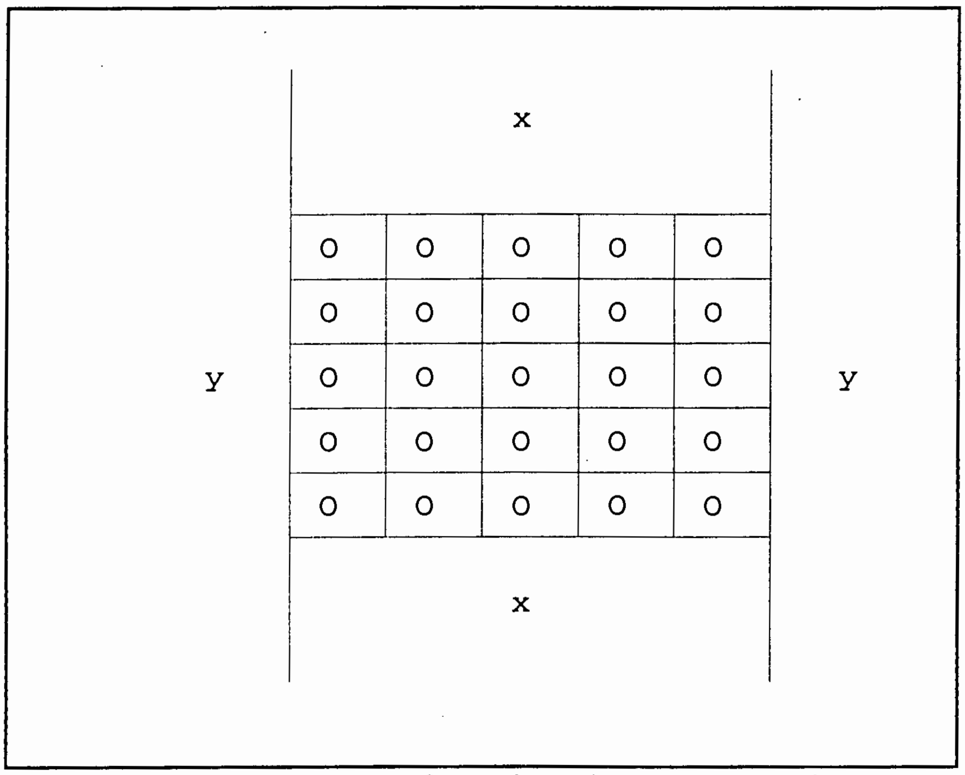

Figure 42: Schematic diagram of top view of cloud (bottom view will be similar)

Another array is used to count the number of photons absorbed into the cloud column. This array represents the cloud column vertically. The distance between the top and the bottom of the cloud is divided into a number of layers of equal height. The number of photons absorbed in each layer is counted. These totals are then used to calculate the heat absorbed by the cloud column. A schematic diagram showing how the cloud column is divided into layers is shown in Figure 43. For these experiments there were 1000 layers.

Light is composed of a number of different wavelengths, and each wavelength has different properties. For instance, certain gases absorb light of certain wavelengths, and other gases absorb light of a different wavelength. A cloud consists of a combination of different gases and water vapour. One of

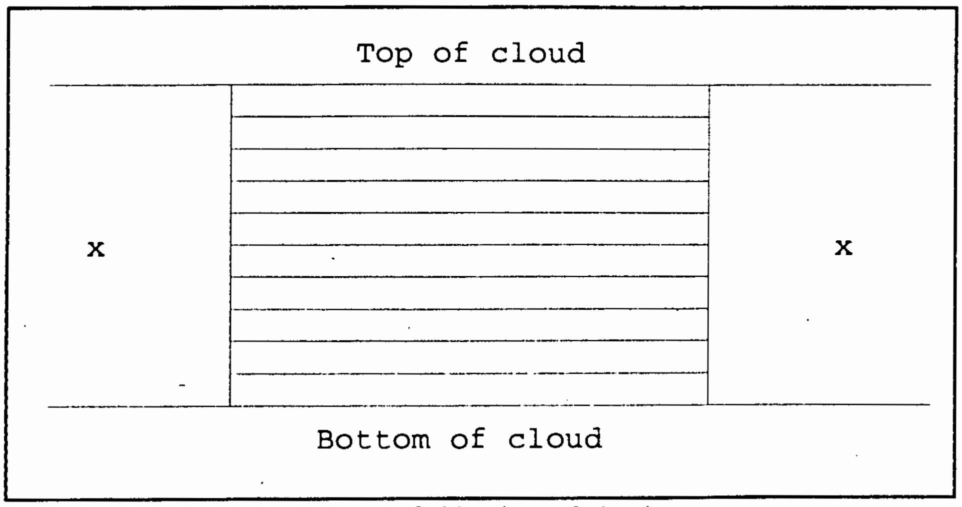

Figure 43: Schematic diagram of side view of cloud

the factors which will change according to the wavelength is the mean free path that a photon can be expected to travel before a collision, or being absorbed. Thus, to simulate the passage of light through a cloud it is necessary to simulate photons of all wavelengths, as photons of some wavelengths will be absorbed more than others, which may be reflected or transmitted. Also light may be either direct or diffuse, and will behave differently for each case. For this particular experiment there were in all 22 different wavelengths, some of which were divided into sub-intervals, making 50 sets of input conditions. Some cases were calculated for both diffuse and direct light. The properties of each wavelength are read in from file.

Monte Carlo simulation has been used to trace the paths of photons through the cloud, since other methods would take too long.

Appendix B

Hardware specifications

This section gives the technical specifications of the workstations and the network used in the experiments described in this dissertation. The workstations, described by the letters "a" to "r" as used elsewhere in this report, are grouped together with their common technical description.

First the workstations are described and then the Ethernet network.

B.1 Silicon Graphics workstations

The following information, except for the operating system information, was obtained by typing the command **hinv** on each workstation. The operating system version number is displayed at login.

Workstation "a" - Silicon Graphics Indigo2 Extreme

```
1 100 MHZ IP22 Processor
FPU: MIPS R4010 Floating Point Chip Revision: 0.0
CPU: MIPS R4000 Processor Chip Revision: 3.0
On-board serial ports: 2
Data cache size: 8 Kbytes
Instruction cache size: 8 Kbytes
Secondary unified instruction/data cache size: 1 Mbyte
Main memory size: 128 Mbytes
Integral Ethernet: ec0, version 1
Integral SCSI controller 1: Version WD33C93B, revision D
Disk drive: unit 1 on SCSI controller 0
Integral SCSI controller 0: Version WD33C93B, revision D
Iris Audio Processor: version A2 revision 0.1.0
Graphics board: GU1-Extreme
Operating system : IRIX 4.0.5H
```

Workstation "b" - Silicon Graphics Indigo - 24 Mb memory

```
1 33 MHZ IP12 Processor
FPU: MIPS R2010A/R3010 VLSI Floating Point Chip Revision: 4.0
CPU: MIPS R2000A/R3000 Processor Chip Revision: 3.0
On-board serial ports: 2
Data cache size: 32 Kbytes
Instruction cache size: 32 Kbytes
Main memory size: 24 Mbytes
Integral Ethernet: ec0, version 0
Disk drive: unit 1 on SCSI controller 0
Integral SCSI controller 0: Version WD33C93A, revision 9
Iris Audio Processor: revision 3
Graphics board: LG1
Operating system : IRIX 4.0.5F
```

Workstation "c" - Silicon Graphics Indigo - 16 Mb memory

```
1 33 MHZ IP12 Processor
FPU: MIPS R2010A/R3010 VLSI Floating Point Chip Revision: 4.0
CPU: MIPS R2000A/R3000 Processor Chip Revision: 3.0
On-board serial ports: 2
Data cache size: 32 Kbytes
Instruction cache size: 32 Kbytes
Main memory size: 16 Mbytes
Integral Ethernet: ec0, version 0
Tape drive: unit 4 on SCSI controller 0: DAT
Disk drive: unit 1 on SCSI controller 0
Integral SCSI controller 0: Version WD33C93B, revision C
Iris Audio Processor: revision 10
Graphics board: LG1
Operating system : IRIX 4.0.5F
```

B.2 Sun workstations

It has been extremely difficult to obtain any specifications of the Sun workstations used in these experiments. The following information was obtained from comp.sys.sun.hardware Wed Sep 28 08:55:53 1994. It is not known whether these specifications accurately describe the Sun workstations used. For example, from these descriptions one would expect the Classics to perform better than the ELCs whereas the serial performance tests showed the reverse to be true.

The document posted was:

```
                    THE SUN HARDWARE REFERENCE
                  compiled by James W. Birdsall
                     (jwbirdsa@picarefy.com)

                             PART I
                             ======
                            OVERVIEW
                          CPU/CHASSIS

                     Last updated: 09/09/1994

OVERVIEW
========

   This primary focus of this document is to cover Sun-badged hardware
in detail sufficient to be useful to buyers and collectors of used Sun
hardware, much of which comes without documentation. Details on
hardware commonly used with Suns, especially hardware specifically
designed for Suns, are also included where available.
```

An extract from this document describes the Sun-4/SPARCstations as follows: (All Suns used were part of this group)

```
    Sun-4/SPARCstation
    ------------------

OVERVIEW

   These machines were initially introduced with model designations in
the same pattern as previous lines: Sun 4/xxx. However, Sun departed
from their classic naming scheme with the name SPARCstation, and has
```

```
since experimented with alphabetic designations (e.g. "SPARCstation
SLC") before returning to numbered SPARCstations.

   This model line marks the introduction of Sun's own RISC chip, the
SPARC. There have been a number of different implementations of the chip
from various manufacturers, with varying degrees of hardware support for
the instruction set.

   Support for Sun-4's was introduced in SunOS 4.0. Since this product
line is still current, it is still supported by SunOS, which has mutated
to become Solaris.

   Some of the later models have pictures silkscreened on their CPU
boards.

   Note that MIP/GIP ratings for later models are deemed suspicious.
```

By taking extracts from this document the following specifications of the Sun workstations were obtained. Further notes from this document follow these specifications.

Workstations "d" (32Mb memory), and "e" (16Mb memory) -

```
    SPARCstation ELC (4/25)
        Processor(s):    Fujitsu MB86903 or Weitek W8701 @ 33MHz, FPU on
                         CPU chip, Sun-4c MMU, 8 hardware contexts,
                         21 MIPS, 3 MFLOPS
        CPU:             501-1730/1861
        Chassis type:    monitor
        Bus:             none
        Memory:          64M physical; 64K write-through cache,
                         direct-mapped, virtually indexed, virtually
                         tagged, 32-byte lines
        Notes:           Code name "Node Warrior" (?). 4M or 16M x 33 SIMMs.
                         No fan. 17" mono monitor built in. 8M standard.

        Operating System : SunOS 4.1.3 (ELC)
```

Workstations "g", "l", "m" (32Mb memory), and "f", "h", "i", "j", "k", "n" (16Mb memory) -

```
    SPARCclassic (SPARCclassic Server)
        Processor(s):    microSPARC @ 50MHz, 59.1 MIPS, 4.6 MFLOPS
                         (microSPARC - Texas Instruments TMS390S10.
                         On-chip 4K I-cache. On-chip 2K D-cache.
                         64 hardware contexts. FPU and SPARC Reference MMU
                         on chip. SPARC Reference MMU has in-memory
                         3-level page tables, similar to a
                         de-baroqued subset of the 68030 MMU, but with
                         Sun-MMU-style contexts.)
        Bus:             SBus
        Memory:          96M physical
        Notes:           Sun-4m, but no MBus. Code name "Sunergy".
                         Uniprocessor only. 16M standard. 1.44M 3.5"
                         floppy.

        Operating System : SunOS 5.3
```

Workstations "q" (28Mb memory), and "o", "p", "r" (16Mb memory) -

```
SPARCstation 1+ (4/65)
    Processor(s):    LSI L64801IU @ 25MHz, Weitek 3172, Sun-4c MMU,
                     8 hardware contexts, 15.8 MIPS, 1.7 MFLOPS
    CPU:             501-1632
    Chassis type:    square pizza box
    Bus:             SBus, 3 slots
    Memory:          64M (40M?) physical with synchronous parity,
                     512M/process virtual; 64K write-through cache,
                     direct-mapped, virtually indexed, virtually
                     tagged, 16-byte lines; 50ns cycle
    Notes:           Code name "Campus B". 1M x 9 30-pin 80ns SIMMs,
                     possibly higher capacities as well. 8M standard.
                     1.44M 3.5" floppy.

    Operating System : SunOS 4.1
```

The following information was part of this document and explains the preceding details.

```
CPU/CHASSIS
===========

   For each model listed above, whatever information is available is
given, in the following order:

   Processor: The microprocessor followed by its clock speed in MHz. The
floating point coprocessor (FPU), if any, followed by whatever
information is available about the MMU and number of hardware contexts
(in the MMU?). Lastly, the MIPS (Millions of Instructions Per Second,
aka Meaningless...) and MFLOPS (Millions of FLoating-point OPerations
per Second) ratings, if available. Note that some SPARC processors are
referred to by name; information on the SuperSPARC and microSPARC is
available in the "Processor Data" section.

   CPU: The Sun part number of the CPU board or motherboard.

   Chassis type: "Rackmount" chassis, as the name suggests, are designed
to fit into a standard 19" equipment rack. They usually require
clearance over and under the chassis for cooling. "Pizza box" chassis
are intended to sit on a desktop, typically underneath the monitor; they
are low, wide, and deep. Older pizza boxes (2/50, 3/75, 3/50, and 3/60)
are much wider than they are deep; newer ones are square (3/80,
SPARCstation 1, 1+, 2, etc.). Some older pizza boxes (mostly the 3/50)
have a 'dimple top', a case top with a circular depression that allows
the chassis to serve as a tilt/swivel monitor base directly. 9-slot
Multibus and 12-slot VME (and probably 6-slot VME as well) "deskside"
chassis are wide towers that must stand on the floor. 3-slot VME
"deskside" chassis can stand on the floor as narrow towers or lie on
their sides on a desktop as a tallish pizza box. "Shoebox" chassis are
small rectangular boxes the size of a couple large hardcover books
stacked. "Monitor" chassis (SPARCstation SLC, etc.) have the motherboard
in the back of the monitor.

   Bus: Whatever bus or busses the machine has. Sun has, at various
times, used Multibus, VMEbus, ISA, SBus, MBus, and XDBus.

   Memory: The amount of physical memory the machine can take, if known,
followed by the maximum size of the machine's virtual memory space, if
known, followed by the cycle time for physical memory, if known, and
finally details of any on-chip or off-chip caches, if known. The caches
on the Motorola 68020 and 68030 and the Intel 80386 are not described,
since information on these chips is widely known. To save space,
```

the on-chip caches of the SuperSPARC and microSPARC processors is described in the "Processor Data" section.

Notes: General information which does not belong under other headings.

Bibliography/Acknowledgements

Much of the information in CPU/CHASSIS was contributed by Al Kossow (aek@wiretap.spies.com).

"Guy" contributed notes on SF9010/MB86900 and 4/1xx and 4/2xx FPUs.

Additional information in CPU/CHASSIS confirmed by/added from Sun document "Cardcage Slot Assignments and Backplane Configuration Procedures", P/N 813-2004-10, Revision A of 5/13/87.

Additional information in CPU/CHASSIS (and all infomation in the Announcement Dates/List Prices section) confirmed by/added from Data Sources Reports on Computer Select CD-ROMs from February 1991, March 1991, April 1991, June 1992, July 1993, and July 1994.

Information on 3/2xx CPU boards added from Sun document "Sun 501-1206 CPU Board Configuration Procedures", P/N 813-2017-05, Revision A of 10 October 1986

Information on 3/50 motherboard added from Sun document "Sun 3/50 Desktop Workstation Hardware Installation Manual", P/N 800-1355-05, Revision A of 31 January 1986

Information on 3/60 motherboard added from Sun document "Hardware Installation Manual for the Sun-3/60 Workstation", P/N 800-1987-05, Revision 50 of 14 August 1987

Information on 2/120 CPU boards and other Multibus boards added from Sun document "Sun-2/120 Hardware Installation Manual", Revision A of 15 April 1985

Random facts contributed by or extracted from postings by:
Jon Mandrell (jon@amc.com)
Robert Dinse
Cave Newt (roe2@midway.uchicago.edu)
Chuck Cranor (chuck@maria.wustl.edu)
root@junior.apana.org.au
Bruce Orchard (orchard@eceservo.ece.wisc.edu)
Ren Tescher
Robert Tseng

SPEC Benchmark for Sun ELC

The following information was also obtained from comp.sys.sun.hardware and gives some benchmark information to give a rough idea of the performance of the ELC.

```
******** TABLE 4: SPECmark89 ******** SPEC89 Results:

Note:   - SPECmark 89 is an older figure derived from the results of
          a combined set of floating point and integer benchmarks, and
          is reported only because SPEC92 figures are not available for
          many older machines. The use of SPECmark 89 is strongly discouraged.
```

```
-----------------------------------------------------------------------------------------------
System        CPU          Clk MHz  Cache    SPEC  Info  Source
Name          Type         Ext/Int  Ext+I/D  Mrk89 Date  Obtained
------------  -----------  -------  -------- ----- ----- ----------------------
Sun SS ELC    SP/FuWe      33       64        20.3 Nov92 Sunflash
Intel 386/387 80386/7      33       64+0       4.3  1992 Intel
Intel 486DX   80486        25       128+8      8.7  1990 Intel
Intel 486DX   80486        33       0+8       11.1  1991 Intel
Intel 486DX   80486        50       256+8     21.9 Oct92 comp.arch
Intel 486DX2  80486        33/66    256+8     25.6  1992 Intel

So, ELC is roughly equal to 486/DX50, and much faster than DX33.
The above is from the table made by John DiMarco. But benchmarks are
misleading :-)
--
                                    Szymon Sokol -- Network Manager
U     U M     M M    M University of Mining and Metallurgy, Computer Center
U     U MM   MM MM  MM ave. Mickiewicza 30, 30-059 Krakow, POLAND
U     U M M M M M M M M TEL. +48 12 338100 EXT. 2885  FAX +48 12 338907
 UUUUU  M  M  M M  M  M finger szymon@galaxy.uci.agh.edu.pl for PGP key
                        WWW page: http://www.uci.agh.edu.pl/~szymon/
```

B.3 Ethernet technical summary

A full technical description of Ethernet can be obtained from a number of sources such as the IEEE-802 Standard. For the sake of brevity only those facts about Ethernet which are relevant to this thesis are described here. They were obtained from the **Ethernet Technical Summary** in Appendix C in [Keis89] unless otherwise specified.

B.3.1 Packet size

An Ethernet packet consists of an 8-byte preamble, a 14-byte header, from 46-1500 bytes of data and a 4-byte CRC. Thus the minimum packet size is 72 bytes and the maximum packet size is 1526 bytes.

The minimum length of the data field is 46 bytes in order to ensure that valid packets are distinguishable from collision fragments. If the data supplied is less than the 46 bytes required for proper operation of the Ethernet protocol this an integer number of padding bytes will be added by the Logical Link Control protocol layer to bring the length of the data field to 46 bytes.

B.3.2 Data rate

The specified data rate of Ethernet is 10 Mbits/second so that the bit cell is 100 ns ± 0.01%.

B.3.3 Inter-packet spacing

The minimum time that must elapse after one transmission before another transmission is started is 9.6 μs.

B.3.4 Carrier

The presence of data transitions indicates that a carrier is present. If a transition is not seen between 0.75 and 1.25 bit times since the center of the last bit cell, then carrier has been lost, indicating the end of a packet. For purposes of deferring, carrier means any activity on the cable independent of the signal being properly formatted. Specifically, it is any activity of either receive or collision-detect signals in the last 160 ns.

B.3.5 Control procedure

The control procedure defines when and how a station may transmit packets.

Defer	A station must not transmit when a carrier is present or within the minimum interpacket spacing time.
Transmit	A station may transmit if it is not deferring. It may continue transmitting until either the end of the packet is reached or a collision is detected.
Abort	If a collision is detected, transmission of a packet must terminate, and a jam signal (4 to 6 bytes of arbitrary data) is transmitted to ensure that all involved participants are notified of the collision.
Retransmit	After a station has detected a collision and then aborted, it must wait for a random retransmission delay, defer as usual, and then attempt to retransmit the packet.
Backoff	Retransmission delays are computed using the truncated binary exponential backoff algorithm, with the aim of resolving contention among up to 1024 stations. The basic unit of backoff is 51.2 μs. [Hamm86] One version of the truncated binary exponential backoff algorithm allows an initial attempt plus 15 retransmissions each delayed by an integer r times the base backoff time. The integer r is selected at random from the discrete distribution, uniform on the set of integers $\{0,...,2^k-1\}$, where k is the minimum of the number of retransmissions to date and the integer 10. i.e. For the 11th through the 15th retransmission attempts the upper limit of the set of values for r is fixed at $2^{10}-1 = 1023$. After 16 attempts the Ethernet algorithm reports an error and a higher level protocol

must decide whether to discard the packet or to continue the attempt to access the network. [Hamm86]

B.3.6 Network node congestion

The following notes on network congestion were taken from [Hamm86].

Congestion occurs at a node in a computer network when the resources of the node are stretched to capacity. This happens when the total input traffic rate exceed the output rate so that all available buffers become full. As a consequence of buffer overflow packets will have to be dropped and there is a likelihood that the whole network will become deadlocked with no packets getting through.