INFORMATION TO USERS

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

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

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

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

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

UMI

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

Management of Parallel I/O.

By
Luca Diaconescu

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University,
Ottawa, Ontario, Canada

July 1997

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-26992-2

The undersigned hereby recommend to the Faculty of Graduate Studies and
Research acceptance of the thesis

Management of Parallel I/O

submitted by Luca Diaconescu, B. Eng., B.Sc.,
in partial fulfillment of the requirements for the degree of
Master of Engineering

Chair, Department of Systems and Computer Engineering

Professor Shikharesh Majumdar, Thesis Supervisor

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
26 August 1997

Abstract

In the last two decades the computation times for applications have been significantly reduced by the continuous growth in CPU and memory speeds as well as by the deployment of parallel processing technology. However, the decrease in access times for hard disks was minimal over the same period. Currently, there exists a large discrepancy between the performance of the I/O subsystem and the compute subsystem. A way to improve the I/O subsystem performance is the use of software which performs I/O in a parallel manner. However, to harness the power of parallel I/O requires appropriate management of the I/O devices, as well as the deployment of an appropriate style, or I/O model for the application program.

This thesis investigates the performance of different I/O models from a theoretical and experimental point of view. Each I/O model differs from others in the way I/O and computation are interleaved during application execution. The theoretical part consists in obtaining bounds on speedup and response time for parallel I/O application in a uniprogrammed environment. This work is based on existing techniques for obtaining the bounds on speedup and response time of parallel application that do not consider I/O.

The experimental work has two major parts. The first part is concerned with identifying which I/O model is more suitable for a given set of system parameters. The second part investigates the effect of data replication on the performance of parallel I/O applications and shows that there is a trade-off between the amount of data replication and performance. For the experimental part, both a multiprogrammed and a uniprogrammed environments are considered.

The experimental work was accomplished by implementing the I/O models introduced as C programs and doing measurements on a cluster of workstations running under PVM (Parallel Virtual Machine).

The results of this research can be used for understanding the behavior and performance of parallel I/O as well as for the design of appropriate management strategies for the operating system in a multiprogrammed parallel environment.

Acknowledgments

I would like to thank my supervisor Dr. Shikharesh Majumdar for his advice and guidance throughout the last couple of years.

I would also like to thank Dr. Eric Masson for the time spent in helping me with some of my computing problems.

The financial assistance of the Natural Sciences and Engineering Research Council is gratefully acknowledged.

I would also like to take this opportunity to thank my wife and my parents for their continued support and patience during my studies and research.

Table of Contents

Abstract	II
Acknowledgments	IV
Table of Contents	V
List of Figures	VIII
List of Tables	X
List of Terms and Definitions	XI
1.0 Introduction	1
1.1 Why Parallel I/O?	1
1.2 Motivation for the Thesis	2
1.3 Goals of the Thesis	3
1.4 Outline of the Thesis.	5
2.0 Parallel I/O Systems	6
2.1 Overview	6
2.2 Miscellaneous Definitions	9
2.3 Workload Parameters Affecting Performance of Parallel I/O Jobs	10
2.3.1 Average Parallelism (A)	10
2.3.2 I/O Factor (F_i)	12
2.3.3 Coefficient of Variation	12
2.3.4 Number of Devices (N_d and N_i)	12
2.3.5 Number of Threads (n_d and n_i)	13
2.3.6 I/O Models	13
2.4 Resource Management Strategies	16
2.4.1 Scheduling	16

2.4.2 Data Distribution.	16
2.5 Summary	17
3.0 Performance Bounds	19
3.1 Introduction	19
3.2 Bounds on Completion Time and Speedup	20
3.2.1 Case I: No overlap between CPU and I/O operations	23
3.2.2 Case II: Overlap between CPU and I/O operations.	26
3.2.3 Case III: General Case.	30
3.3 Sample Numerical Result.	33
4.0 Experimental Environment	38
4.1 Introduction	38
4.2 Parallel Virtual Machine	39
4.3 Workstations in the Virtual Machine	41
4.4 Implementation of the I/O Models	43
4.4.1 Synthetic - NOP	44
4.4.2 Synthetic - O1	45
4.4.3 Synthetic - O2	46
4.4.4 Matrix Multiplication - O1	47
4.4.5 Matrix Multiplication - O2	48
4.5 Measurement Accuracy.	48
5.0 Performance of I/O Models	50
5.1 Introduction	50
5.2 Uniprogrammed System	51
5.2.1 I/O factor (Synthetic)	51
5.2.2 Variability in I/O Demand.	55
5.2.3 Number of O2 slices	58
5.2.4 The Effect of Network Traffic	61
5.3 Multiprogrammed System	63
5.3.1 Multiple Parallel I/O Jobs	63
5.3.2 Single Parallel I/O Job, and Multiple Sequential Jobs	68

5.3.3 Value of Multiprogramming	71
5.4 Discussion.	75
6.0 Performance of Data Replication Strategies	76
6.1 Introduction	76
6.2 Data Replication Strategies	77
6.2.1 Full Replication (Replicated-C)	78
6.2.2 Limited Replication (ReplicatedL-C)	79
6.2.3 No Replication (FixedR-C)	79
6.3 Comparison of the Data Replication Strategies	80
7.0 Conclusions	83
7.1 Summary of Investigations	83
7.2 Discussion of Results	84
7.3 Contributions	86
7.4 Future Research	86
References	88
Appendix A: Pseudo-Code for I/O Model Implementations.	93
A.1 Pseudo-Code for NOP Threads	93
A.2 Pseudo-Code for O1 Threads	95
A.3 Pseudo-Code for O2 Threads	96
A.4 Matrix Multiplication O1 and O2	97
Appendix B: Results of Synthetic Programs	99
B.1 Comparison of I/O Models - Uniprogrammed Case	99
B.2 Effect of S on O2 Performance.	101
B.3 Comparison of I/O Models - Multiprogrammed Case	102
Appendix C: Results of Matrix Multiplication	106
C.1 Results for the multiprogrammed case	106
Appendix D: Generation of CV_i	107
Appendix E: Lower Bound on Speedup	108

List of Figures

Figure 1: CPU and disk speeds	1
Figure 2: I/O Models	14
Figure 3: The two types of parallel I/O applications	22
Figure 4: Reduction of Case III to Case II	32
Figure 5: The No Overlap Case	34
Figure 6: The Overlap Case	35
Figure 7: Upper Bound (No Overlap) and Lower Bound (Overlap) as a function of P for $N_i=3$ case	36
Figure 8: Upper Bound (No Overlap) and Lower Bound (Overlap) as a function of P for $N_i=6$ case	37
Figure 9: Results for the Synthetic Programs with $F_i=1$, $CV_i=0$	52
Figure 10: Results for the Synthetic Programs with $F_i=2$, $CV_i=0$	53
Figure 11: Results for the Synthetic Programs with $F_i=1$, $CV_i=0.5$	56
Figure 12: Results for the Synthetic Programs with $F_i=2$, $CV_i=0.5$	57
Figure 13: Results with $F_i=0.5$, $CV_i=0$ for Different Number of O2 Slices	59
Figure 14: Results with $F_i=4$, $CV_i=0$ for Different Number of O2 Slices.	60
Figure 15: Results with $F_i=1.1$, $CV_i=0$ for Matrix. Mult. Programs	62
Figure 16: Performance of O1 and O2 in a Multiprog. System. ($F_i=1$, $CV_i=0$, $N_i=3$)	64
Figure 17: Performance of O1 and O2 in a Multiprog. System. ($F_i=2$, $CV_i=0$, $N_i=3$)	64
Figure 18: Performance of O1 and O2 in a Multiprog. System. ($F_i=1$, $CV_i=0.5$, $N_i=3$)	66
Figure 19: Performance of O1 and O2 in a Multiprog. System. ($F_i=2$, $CV_i=0.5$, $N_i=3$)	66
Figure 20: Performance of the Matrix Multiplication Programs in a Multiprogrammed System ($F_i=1.1$, $CV_i=0.5$, $N_i=3$).	67
Figure 21: Performance Under Different System Loads ($F_i=1$, $CV_i=0$)	69
Figure 22: Performance Under Different System Loads ($F_i=2$, $CV_i=0$)	69
Figure 23: Value of Multiprogramming vs. Degree of Multiprogramming for $F_i=1$ (Synthetic)	73

Figure 24:Value of Multiprogramming vs. Degree of Multiprogramming for $F_i=1$ (Matrix Mult.)	73
Figure 25:Performance of Data Replication Strategies for Parameter Set 1	81
Figure 26:Performance of Data Replication Strategies for Parameter Set 2	82
Figure 27:Graph of NOP, O1 and O2 for $F_i=0.5$ and $CV_i=0$	99
Figure 28:Graph of NOP, O1 and O2 for $F_i=0.5$ and $CV_i=0.5$	99
Figure 29:Graph of NOP,O1 and O2 for $F_i=4$ and $CV_i=0$	100
Figure 30:Graph of O2 with different number of slices for $F_i=1$ and $CV_i=0$	101
Figure 31:Graph of O2 with different number of slices for $F_i=2$ and $CV_i=0$	101
Figure 32:Histograms of O1 and O2 for the Multiprogrammed Case for $F_i=1$ $CV_i=0$	102
Figure 33:Histograms of O1 and O2 for the Multiprogrammed Case for $F_i=1$ $CV_i=0.5$	103
Figure 34:Histograms of O1 and O2 for the Multiprogrammed Case for $F_i=2$ $CV_i=0$	104
Figure 35:Histograms of O1 and O2 for the Multiprogrammed Case for $F_i=2$ $CV_i=0.5$	105
Figure 36:Matrix Multiplication Program, Multiprogrammed Case. $F_i=1.1$	106

List of Tables

Table 1:Job Model Parameters 17

Table 2:System Parameters 18

Table 3:Job Model Parameters 19

Table 4:Parameters for the Sample Bounds Calculation 33

Table 5:Value of Multiprogramming for Different Parameters 72

Table 6:Data Replication Parameters 77

List of Terms

F_i : Normalized I/O Factor

F_i' : I/O Factor

n_i : Number of I/O threads

n_d : Number of CPU threads

N_i : Number of I/O devices

N_d : Number of CPU's

A_i : Average I/O parallelism

A_d : Average CPU parallelism

A_o : Average overlap parallelism

m_i : Maximum I/O parallelism

m_d : Maximum CPU parallelism

m_o : Maximum overlap parallelism

S : Number of O2 slices.

CV_i : Coefficient of variation of I/O demand

CV_d : Coefficient of variation of CPU demand

M : Degree of multiprogramming.

$S(N)$: Speedup factor

T_X : Response time of application X

$Time_X$: Response time of section X of an application

T_i : sum of execution times of all I/O threads

T_d : sum of execution times of all I/O threads

S_X : Speedup of application X

P: Fraction of the total I/O time done in the overlap section

$V(M)$: Value of multiprogramming

1.0 Introduction

1.1 Why Parallel I/O?

In the past twenty years the computation times for applications have been significantly reduced by the continuous growth in CPU and memory speeds as well as by the deployment of parallel processing technology. The improvements in hard disk technology, however, have mainly achieved increases in storage density and size [Chou93]. Due to their electro-mechanical components, the decrease in access time for these devices was much smaller in comparison to the increase in CPU speed. Figure 1 presents graphs showing the increase in CPU speeds and decrease in disk seek times for a number of different manufacturers.

Figure 1 CPU and disk speeds

For Figure 1a the y-axis is measured in MIPS/pin which is a measure of processor bandwidth. These graphs were obtained from [Burg96] and [Hugh91]. As seen from Figure 1, there exists a huge disparity between the I/O subsystem's performance and the processing power of the compute subsystem. [Gibbs91] estimates that there is an increase of 37% per year for CPU speeds whereas the disk speeds increase by a factor of only 5% per year. As a result of these large differences, in many situations the performance bottleneck has shifted from the processor subsystem to the I/O subsystem.

A number of studies show that significant I/O is done by a variety of classes of parallel applications. Many scientific and graphics applications [Cyph93] [Kotz94] require transfer of large amounts of data between the I/O and the computation subsystems. Grand challenge problems [Patt94] are large scientific applications which require a lot of CPU and I/O power. For such applications it is critical that the I/O subsystem has a high performance.

To address the problem of the slow electro-mechanical components used in the building of I/O devices, disk caches and array of disks have been introduced. Different data layout strategies for optimizing disk performance have also been considered for multimedia applications [KwTG97] [Ghan97]. A different way to improve the I/O subsystem performance is the use of software which performs I/O in a parallel manner. This requires appropriate management of the I/O devices, as well as appropriate I/O models for the application in order to harness the power of the I/O subsystem.

1.2 Motivation for the Thesis

The large discrepancy between the I/O subsystem performance and the processor subsystem performance is an important issue in system design. One way to reduce this performance gap is to introduce parallel I/O applications. Little work has been done in the area of parallel I/O performance on multiprogrammed systems. Most of this existing work (see Section 2.1) is based

on simulation results. This thesis investigates a number of issues underlining the performance of parallel I/O in uniprogrammed and multiprogrammed environments using measurements made on a real system.

Many parallel systems are used for running a single application in isolation. However, the usage of multiprocessor systems is becoming more and more widespread, and general purpose systems that run a number of different parallel I/O applications are becoming popular. The availability of inexpensive parallel hardware and tools such as PVM and restructuring compilers is the reason for this increase in popularity. Environments running such a variety of different applications need to multiprogram the system to provide user satisfaction and enhance resource utilization [Sevc94]. Both uniprogrammed and multiprogrammed environments are investigated in this thesis.

Most of the existing work on parallel I/O application focuses on applications running on large and expensive multiprocessors, like the Intel Paragon, which are therefore rare and inaccessible to many researchers. Also, to the best of our knowledge, there are no studies on how such applications with parallel I/O perform in a more common environment like a network of workstations. Such a system is available to most researchers, and therefore an investigation of how parallel I/O application perform in such an environment where the nodes of the parallel machine are loosely coupled is required. Furthermore, applications can be written in standard C, and therefore are fully portable to networks that can run PVM. PVM is a tool that enables workstations on a network to form a parallel virtual machine. This thesis focuses on parallel I/O performed on such a network of workstations.

1.3 Goals of the Thesis

The main goal of the thesis is to determine which of the I/O models discussed performs better under a given set of system parameters. Although this issue has been addressed in other work [Kwon96] by running simulations of the different parallel I/O models, the goal here is to confirm

the simulation results by making measurements on a real system. Also, the question of whether there is any potential gain from dynamically changing the style in which an application performs I/O as system load changes is investigated. The goal is to determine if such a dynamic change can produce a significant performance increase. To accomplish the above, real measurements are performed on a cluster of workstations running PVM.

In particular, the following issues are investigated in this thesis:

- Which are the most important parameters that affect the relative performance of the I/O models discussed?
- Why is one model performing better than another under a given set of system parameters?
- How does variability in the total I/O demands of applications influence system performance, and which model is mostly affected by such variability?
- What is the effect of data replication on the relative performance of the I/O models?
- Is there any advantage in running parallel I/O application in a multiprogrammed mode rather than in a uniprogrammed mode?
- How is the performance of a parallel I/O application affected by the presence of sequential applications in the system, and which I/O model performs better in this case?
- What is the effect of having different data replications strategies on system performance?
- Is it possible to obtain bounds on the response time and speedup for parallel I/O applications?

Another important goal of the thesis is to demonstrate that PVM can be used for running parallel I/O applications on a network of heterogenous workstations. In this thesis, however, the effect of having different types of machines (different CPU speeds and different disk access times) as the nodes in the virtual machine, is not investigated.

1.4 Outline of the Thesis.

This thesis is divided into six chapters. An introduction to parallel I/O issues and current research is given in Chapter 2 (Parallel I/O Systems). Chapter 3 (Performance Bounds) presents the bounds on response time and speedup obtained for parallel I/O applications. The environment in which the measurements were taken and a description of how the I/O models were implemented is found in Chapter 4 (Experimental Environment). The results of all investigations are presented in Chapter 5 (Performance of I/O Models) and Chapter 6 (Performance of Data Replication Strategies). Finally, Chapter 7 (Conclusions) summarizes the major results and discusses further research possibilities.

2.0 Parallel I/O Systems

2.1 Overview

A lot of work has been done in the area of parallel applications not involving a large I/O demand. Some of this work is relevant to this thesis since it introduces some parameters that will be used later on. The trade-off between speedup and efficiency of a parallel computation is discussed in [Eage89]. A set of parameters characterizing the parallelism in application is introduced. In this paper the most important parameter is the average parallelism. The paper also presents bounds on speedup of a parallel application based on this new parameter.

Significant work exists in the area of processor scheduling in multiprogrammed parallel systems. Only a few examples of such work are presented. [Maju91] is concerned with scheduling of programs in multiprogrammed parallel systems. Both static and semi-static approaches to the global scheduling problem are discussed. [Sevc94] also addresses the issue of scheduling and processor allocation in multiprogrammed parallel systems. Here, the focus is on scheduling a batch of applications with deterministically known characteristics. Extensions to handle uncertainties in service times, priorities and the arrival stream of applications are also considered.

A large body of work exists on the allocation of processors to single concurrent applications. For example [Coff73] presents an algorithm for processor allocation for applications in which processes do not communicate with one another. The question of processor allocation in distributed and parallel systems running programs with inter-communicating processes is addressed in [Wood90]. This paper presents a static task allocator which could be incorporated in an automated compiler/linker/loader for distributed processing systems.

Management of memory in parallel systems has also received significant attention from

researchers. For example, several algorithms for implementing distributed shared memory are compared in [Stum90]. This paper shows that the performance of the algorithms under consideration is sensitive to the memory access behavior of the applications. Distributed shared memory is also considered in [Nitz91], where it is shown that distributed shared memory combines the scalability of network based architectures with the convenience of shared memory programming.

A presentation of the I/O requirements of large parallel applications (grand challenge problems, imaging applications) is done in [Ross94], where the need for having a high performance I/O subsystems is presented. An implementation of an I/O intensive solution to a system of differential Navier-Stokes flowfield equations is presented in [Ryan93].

Redundant arrays of inexpensive disks (RAID) have been proposed in [Patt88]. In this paper six different RAID organizations are defined, each with different degree of protection against disk failure and each with different degree of read/write concurrency. The design and performance of RAID architectures is considered in [Chen93] and [Chen95]. In a RAID architecture, the I/O requests are sent to an I/O node that has many disks. Each I/O request is then divided into a number of disk-requests that are serviced in parallel according to a given schedule. The application that has issued the I/O request may not be a parallel I/O application since parallelization is achieved at the I/O subsystem, independent of the initiating application. In [Chen95] the performance of two of the most promising RAID architectures, the mirrored array and the rotated parity array, are considered. Several scheduling policies are introduced for the mirror array, and a new data layout, called group-rotate declustering, is also considered. The performance of the scheduling schemes and data layouts are compared. Similar issues for arrays of disks are investigated in [Vita95], where the problems related to parallelizing I/O access in a disk array system are discussed. A new, flexible architecture called partial dynamic declustering, which is fault tolerant and offers higher levels of performance and reliability is also introduced.

Comparison of different disk configurations for a transaction workload system as well as for a scientific application workload, is presented in [Redd89]. The paper considers disk synchronization, data declustering/disk striping, and a combination of both of these approaches. The effects of block size and other parameters of the system are also evaluated.

Some work has been done in the area of management and scheduling of parallel applications that perform significant I/O on a multiprogrammed parallel system. Basic issues that underline processor scheduling in multiprogrammed parallel environments running applications with I/O is presented in [Maju94]. Two I/O management issues that affect system performance in multiprogrammed parallel environments are considered in [KwoM96]. In this paper four different I/O models are introduced and their I/O behavior is characterized. Also, the performance of different data distribution strategies is investigated. This work is carried further in [Kwon96]. Here different scheduling policies are evaluated, and the strategy for data distribution is also considered. The intricate relations between data distribution, scheduling policy and I/O model are investigated. An evaluation of workload characteristics of intensive parallel I/O applications is presented in [Smir97]. Here the different I/O access patterns commonly found in I/O intensive parallel applications are presented.

A multi-threaded runtime library for parallel I/O (MTIO) has been designed and evaluated in [More97]. Here the concept of multi-threading is extended in order to separate the compute and I/O tasks in two separate threads of control. This multi-threaded design permits asynchronous I/O even if the underlying file system does not support this. It also provides a capability to perform collective I/O asynchronously without blocking the compute threads. Performance results for various data distributions and access patterns on an IBM SP2 are presented.

Research on parallel file systems is also under way. The Galley file system that is composed of a standard fixed core and a set of platform dependent libraries is discussed in [Kotz95]. The Vesta file system is discussed in [Corb93]. A scalable parallel file system for the Intel Paragon (SPIFFT) is discussed in [Free96]. The Hurricane file system that provides a collection of build-

ing block objects that can be plugged together differently for different systems needs is described in [Krie96]. A parallel out-of-core LU-factorization computation is presented in [Kotz95b]. Here the use of collective I/O requests permits the implementation of techniques like two-phase I/O and disk-directed I/O. A collective I/O request is a technique which allows processes in an application to cooperate and make a single, large request to the I/O subsystem. Two-phase I/O is a technique for optimizing high level, collective I/O interfaces, by splitting the large collective I/O request into two phases, an I/O phase and a redistribution phase. This means that when reading, the compute processors cooperate to read a matrix in a way chosen for best disk performance (I/O phase), and then the data is redistributed to its ultimate destination (redistribution phase). Disk-directed I/O is a technique for optimizing high level, collective I/O interfaces, by generating a list of the disk-blocks required in the transfer and then using this information to optimize the transfer. These techniques significantly improve the performance over traditional file systems and interfaces. The disk-directed I/O technique for Multiple Instructions Multiple Data (MIMD) multiprocessors is discussed in [Kotz97].

2.2 Miscellaneous Definitions

The following is a set of definitions for terms used throughout the thesis.

a) The degree of multiprogramming M is equal to the number of applications that are running in parallel on a system at a given time.

b) The response time of an application is defined as the time interval required for an application to complete.

Response time = Finish time - Start time

The response time is measured in seconds throughout this thesis.

c) An I/O thread is a thread that services I/O requests. A single I/O thread runs on a given I/O node.

d) A compute thread is a thread that performs computations and issues I/O requests. A compute thread runs on a compute node.

2.3 Workload Parameters Affecting Performance of Parallel I/O Jobs

From the above mentioned work it is evident that a number of parameters play an important role in the performance of parallel I/O applications. Below is a description of the factors introduced in the previously mentioned papers that affect the performance of the parallel I/O applications.

2.3.1 Average Parallelism (A)

This parameter was first introduced in [Eag89]. It can be defined in many different ways, one of which is:

the average number of processors that are busy during the execution time of a parallel application, given an unbounded number of available processors.

The average parallelism is a measure on “how parallel” an application is. This definition does not apply directly to a parallel I/O application, since in this case a lot of time is spent at the I/O devices (also referred as I/O nodes in the thesis). For such an application we will define two

parameters, the average I/O parallelism (A_i) and the average CPU parallelism (A_d). A_d is defined as above, while A_i is defined as:

the average number of I/O nodes (disks) that are busy during the execution time of the parallel application in question, given an unbounded number of available I/O nodes.

Bounds on speedup are calculated using the average parallelism parameter in [Eage89]. However, in order to obtain tighter bounds, another parameter is required. This is the maximum parallelism denoted by $m_{\max}$ which is defined as:

the maximum number of processors that are simultaneously busy when an unlimited number of processors are available.

For an application with parallel I/O, two such parameters are required: the *maximum I/O parallelism* m_i and the *maximum compute parallelism* m_d . The maximum compute parallelism is the same as $m_{\max}$ defined above. The maximum I/O parallelism is defined as:

the maximum number of I/O devices(nodes) that are simultaneously busy when an unbounded number of such devices is available.

Each I/O operation requested by the application is serviced by one I/O node.

In this thesis m_i and m_d are used just for the purpose of obtaining bounds on response time and speedup for the parallel I/O application. They will be constant for the experimental part of the thesis.

2.3.2 I/O Factor (F_i)

The I/O factor is introduced in [Maju94] and it is a measure of the total I/O demand relative to the total CPU demand. Its definition is:

The I/O factor is the ratio between the mean total I/O demand and the mean total processor demand for jobs.

Clearly, if the application is running on a one CPU-one disk system, the I/O factor F_i is equal to the total I/O time divided by the total CPU time.

[Maju94] shows that F_i has a strong impact on the performance of applications with parallel I/O. The I/O factor is one of the parameters used extensively throughout the thesis.

2.3.3 Coefficient of Variation

Another important factor for parallel I/O applications is the variability present in the CPU and I/O demands. These coefficients of variation are denoted by CV_d and CV_i , respectively.

[Maju94] also addresses the interaction between the I/O factor and the variability in the process execution time (CV_d). The interaction between the I/O factor and the coefficient of variation in the I/O time (CV_i) will be investigated in this thesis.

2.3.4 Number of Devices (N_d and N_i)

The impact on performance of the number of devices available to a job was discussed in [Eage89]. For the purpose of this thesis we have to introduce two such parameters: the number

of compute nodes (devices) N_d and the number of I/O nodes (devices) N_i . Both of these have a strong impact on the time of completion of a parallel I/O application, and this impact will be investigated in this thesis. Each I/O thread is serviced by one I/O device

2.3.5 Number of Threads (n_d and n_i)

A parallel I/O application is composed of a number of CPU threads (threads that perform computations only) and I/O threads (threads that service the I/O requests issued by the CPU threads). The number of CPU threads and I/O threads are labeled with n_d and n_i respectively.

2.3.6 I/O Models

Four I/O models were introduced in [Maju94] and [KwonM96]. Figure 2 below shows a graphical representation of these models.

The NO Model (Figure 2 a): An NO job begins by first reading some data using a single read operation from a single I/O device. When the read is done, the job forks into several children threads, each of which performs only computations. When all these threads complete, they join and the job performs a single write operation to a single I/O device. The job terminates when the I/O completes. This job model does not have CPU-I/O overlap because the job can only do I/O or computation, but not both simultaneously.

The NOP Model (Figure 2 b): An NOP job begins like an NO job by first reading in some data. The read, however, is done using parallel I/O operations. When all these read operations have completed, the job forks several children threads, each of which performs only computation. When all threads complete, the computations they join and the job performs a parallel write

Figure 2 I/O Models

operation. After all writes are complete the job terminates. There is no CPU-I/O overlap present in this model.

The O1 Model (Figure 2 c): This represents jobs which overlap CPU with I/O. A job starts by forking a number of children threads. Each thread reads data from one I/O device, performs some computation and then writes data to one I/O device. When all threads complete they join and the job terminates. This job model has CPU-I/O overlap because the I/O performed by a thread is independent of the other threads, and thus it can overlap with computations from a different thread. Note that this is similar to the NOP model, except for the fact that the synchronization between the threads after the initial read and before the final write operations has been removed. Also note that for the O1 model the number of compute threads n_d is equal to the maximum number of simultaneous I/O operations. This is different from the NOP case where one can have any values for n_i and n_d .

The O2 Model (Figure 2 d): This is similar to O1, but has more overlap between I/O and computation. A thread alternates between performing I/O and doing computations. The last operation a thread does is a write. When all these writes are complete, the threads join and the job

terminates. The number of times a thread performs I/O is called the number of slices S and is the same (deterministically) for all threads of a given job. Clearly for $S=2$ this degenerates into the O1 model. This model has CPU-I/O overlap like the O1 model. Furthermore, the overlap is higher in this case since the probability of having overlap grows as S increases. For the O2 model there is another parameter called the read-write ratio. This ratio is a measure of the relative amount of reading and writing performed by the application. In this thesis, the amount of reading and writing performed by an application will be the same.

Many scientific programs read their input data, fork a number of processes which perform the necessary computations and then join. The output results are then written by the parent process to the disk. Such an application running on a single disk system can be modeled by the NO model. When such an application runs on a system that consists of a number of parallel disks it can be modeled as an NOP model. In a number of applications, such as matrix multiplication, each thread behaves independently. It reads its input data, performs the necessary computations and writes the results on to the disk. For an example of such a program see section 4.4.4. An O1 model is used for modeling such applications. Many applications such as LU factorization [Kotz95b] and [Ryan93] use a number of threads, each of which alternates between CPU and I/O. An O2 model is appropriate for modeling such a system. Note that in this research the same number of I/O operations are performed by each thread in the O2 model. However different number of I/O operations for threads can be easily incorporated into the model.

In the thesis, the NOP, O1, and O2 models will be investigated, with special attention given to the O1 and O2 models. Synthetic programs are created for the NOP, O1, and O2 models and their performance under different conditions is reevaluated. Also an O1 and an O2 version of a real matrix multiplication program is also implemented and evaluated.

The NO model is not investigated since it does not implement parallel I/O and in [Kwon96] it was found that NO does not perform well.

2.4 Resource Management Strategies

The management of resources for parallel I/O application was investigated in [Kwon96] and in [KwoM96], where various scheduling policies and data replication strategies were investigated. In this thesis, the scheduling of parallel I/O application will not be investigated, however some data distribution strategies will be discussed.

2.4.1 Scheduling

In [Kwon96] scheduling issues in I/O multiprogrammed parallel systems are addressed. The I/O models considered for this work are the NO, NOP, O1, and O2 models described above. The paper is concerned with devising of appropriate I/O scheduling strategies. Both policies that use I/O characteristics of jobs and policies that do not use such characteristics are considered. Based on a simulation model, the performance of a number of policies are investigated.

The simulation results reveal the effect of different workload characteristics on performance, and demonstrate the existence of complex interactions among data distribution strategies, workload characteristics, and scheduling. Policies which use the I/O characteristics of jobs are observed to perform better.

Although scheduling can have a significant effect on performance the focus of this thesis is on the I/O models and data distribution strategies that are shown to have a strong impact on system performance. A simple FCFS I/O scheduling policy is implemented for all the measurements performed.

2.4.2 Data Distribution

Data distribution has a strong impact on performance since it affects m_i , the maximum I/O parallelism. For the best case, full replication, m_i is not affected and its value is at a maximum. In

the worst case, when all data is found on one disk only and there are no copies of it on any other disk, it is clear that $m_i=1$ and therefore we are dealing with a sequential I/O subsystem. [KwoM96] addresses this issue of data replication.

In this thesis the impact of data distribution is also considered. Two of the policies from [KwoM96] are discussed along with another distribution policy which will be introduced in Chapter 6. The goal is to determine which of the I/O models performs best under different data distribution policies. All these policies are presented and discussed in detail in Chapter 6.

2.5 Summary

From the above survey of existing work on parallel I/O application we can conclude that although there is some research done on I/O management in multiprogrammed system, none of the existing work describes measurements made on real multiprogrammed parallel I/O environments. Most of the work done previously is performed in a simulation environment. Furthermore, all real measurements that are performed in the surveyed papers are done on expensive multiprocessors that run single applications. This thesis extends this work by making investigations of real programs running on a parallel machine. The parallel environment is composed of a set of workstations running under PVM.

A summary of a number of important parameters that describe a parallel application and the system on which it runs are presented in Table 1 and Table 2.

TABLE 1. System Parameters

Parameter	Description
M	NO, NOP, O1, O2
N_i	Number of I/O devices
N_d	Number of CPU devices

TABLE 2. Job Parameters

Parameter	Description
I/O Model	NO, NOP, O1, O2
F_i	I/O Factor
n_i	Number of I/O threads
n_d	Number of CPU threads
A_i	Average I/O parallelism
A_d	Average CPU parallelism
m_i	Maximum I/O parallelism
m_d	Maximum CPU parallelism
S	Number of O2 slices.
CV_i	Coefficient of variation of I/O demand
CV_d	Coefficient of variation of CPU demand
T_d	Sum of the execution time of all compute threads.
T_i	Sum of the execution time of all I/O threads.

The notation used in the above table is consistently used throughout the thesis.

The following terms used throughout the thesis are understood to be synonymous:

- a) Number of I/O devices and Number of disks.*
- b) Number of compute nodes and Number of CPU's*
- c) Application and job*

3.0 Performance Bounds

3.1 Introduction

The goal of this chapter is to get a better understanding of some high level bounds on the response times and speedup of a parallel I/O application. The discussion is based on the work described in [Eage89]. In this paper, bounds on speedup and efficiency are given in terms of average parallelism and maximum parallelism. However, this is done for an application that does no I/O and therefore only the CPU execution times are considered. In this thesis this result is extended to parallel I/O applications.

TABLE 3. Job Model Parameters

Parameter	Description
A_o	Average overlap parallelism defined as the average I/O parallelism in the overlap section.
m_o	Maximum overlap parallelism defined as the maximum I/O parallelism in the overlap section.
T_X	Response time of an application. X stands for NoOverlap or Overlap depending on the case considered.
$Time_X$	Time required to complete a section of an application. X can be: CPU, I/O or Overlap
S_X	Speedup of an application
P	Fraction of the total I/O time done in the overlap section

Table 3 shows all the parameters that enter in the derivations to follow and are not present in Table 2. The parameters that have not been defined yet will be described when they are introduced in the discussion

3.2 Bounds on Response Time and Speedup

For parallel applications, speedup is a measure of how much faster an application completes in a parallel environment compared to a sequential environment. It is defined as the response time of the application running on one processor, divided by the response time of the application running on N processors. However, for the case of parallel I/O application, the speed-up will be given by the ratio between the response time of the application on a “one CPU - one disk” system, to the response time on a “ N CPU - M disk” system.

It is known that for a single application running under any work-conserving scheduling strategy, the lower bound on speedup is given by [Eage89] (a derivation of this formula is also given in Appendix E):

$$S(N) \geq \frac{NA}{N+A-1} \quad (1)$$

where A is the average parallelism of the application, and N is the number of processors on which it is run. Knowledge of maximum parallelism can be used to refine the bounds in (1). This is captured in the following inequality [Eage89]:

$$S(N) \geq \min \left(A, \frac{NA}{N+A-1 - \frac{(N-1)(A-1)}{m_{max}-1}} \right) \quad (2)$$

where $m_{\max}$ is the maximum parallelism of the application.

The inequalities (1) and (2) apply to a compute only parallel application. However, a similar argument can be made for a I/O only parallel application, and the exact same formulae will hold for such an application given that A_i is the average I/O parallelism, N_i is the number of I/O devices and m_i is the maximum I/O parallelism. Therefore, for a parallel I/O application that uses both CPU and I/O, one has two inequations that correspond to (2). These are:

$$S_i(N_i) \geq \min \left(A_i, \frac{N_i A_i}{N_i + A_i - 1 - \frac{(N_i - 1)(A_i - 1)}{m_i - 1}} \right) \quad (3)$$

$$S_d(N_d) \geq \min \left(A_d, \frac{N_d A_d}{N_d + A_d - 1 - \frac{(N_d - 1)(A_d - 1)}{m_d - 1}} \right) \quad (4)$$

The above represent lower bounds on the speedup of the I/O and compute section of a parallel I/O application.

The upper bound on speedup (lower bound on response time) has two components: a hardware bound and a software bound. Consider a compute only system first. Clearly if a system has only N processors, the shortest response time of an application is T/N , where T is the response time with only one processor. This is called the hardware bound.

The software bound captures the fact that although an application may have N processors at its disposal, it may not use them all at the same time. On an average, an application will use A processors, where A is the average parallelism. The software bound on response time is given by T/A . The derivation of these bounds are discussed in detail in [Eage89]. Putting the hardware and software bounds together we get:

$$S(N) \leq \min(N, A) \quad (5)$$

This represents an upper bound on speedup. As with the case of lower bounds, we obtain two inequalities for applications with parallel I/O:

$$S_i(N_i) \leq \min(N_p, A_i) \quad (6)$$

$$S_d(N_d) \leq \min(N_p, A_d) \quad (7)$$

In order to obtain a single upper bound and a single lower bound on the speedup of an application that performs both CPU and I/O operations, we need to investigate two cases. Figure 3 shows the two cases for a parallel I/O application. In Figure 3a there is no overlap between I/O and CPU operations and in the Figure 3b there is overlap. Each of these cases is discussed in detail.

Figure 3 The two types of parallel I/O applications

3.2.1 Case I: No overlap between CPU and I/O operations

Let us begin by considering the lower bound on speedup. For any given number of processors and I/O devices, the response time is just given by the sum of the response time of the compute section and the I/O section. Therefore, for the no overlap case (see Figure 3a) we have:

$$T_{NoOverlap} = Time_{I/O} + Time_{CPU} \quad (8)$$

However, the response time of an application is given as the time the application would take if it was running on only one device, divided by the speed-up. In this thesis, the response time with only one device is denoted by T_d or T_i and it is called total CPU or total I/O time respectively. Clearly, the ratio between T_i and T_d is the I/O factor. Therefore the above equation becomes:

$$T_{NoOverlap} = \frac{T_i}{S_i(N_i)} + \frac{T_d}{S_d(N_d)} = T_d \left(\frac{F_i}{S_i(N_i)} + \frac{1}{S_d(N_d)} \right) \quad (9)$$

and substituting for the speedup from (3) and (4) we get:

$$T_{No} \leq T_d \left(\frac{F_i}{\min \left(A_i, \frac{N_i A_i}{N_i + A_i - 1 - \frac{(N_i - 1)(A_i - 1)}{m_i - 1}} \right)} + \frac{1}{\min \left(A_d, \frac{N_d A_d}{N_d + A_d - 1 - \frac{(N_d - 1)(A_d - 1)}{m_d - 1}} \right)} \right) \quad (10)$$

Where NO stands for “No Overlap”

The above gives the response time of the application. The time to complete on one CPU-one disk system is given by:

$$T_i + T_d = T_d (1 + F_i) \quad (11)$$

where T_d is the response time of the compute section on one CPU, and T_i is the response time of the I/O section on one disk. Speedup by definition is given by:

$$S_{NO} = \frac{T_i + T_d}{T_{NO}} \quad (12)$$

and substituting from (10) and (11) we get

$$S_{NO} \geq \frac{1 + F_i}{\frac{F_i}{\min \left(A_i, \frac{N_i A_i}{N_i + A_i - 1 - \frac{(N_i - 1)(A_i - 1)}{m_i - 1}} \right)} + \frac{1}{\min \left(A_d, \frac{N_d A_d}{N_d + A_d - 1 - \frac{(N_d - 1)(A_d - 1)}{m_d - 1}} \right)}} \quad (13)$$

Turning our attention to the upper bound on speedup, we have to determine the hardware and software bounds that were discussed earlier. Beginning with the hardware bound of the system we have:

$$T_{NOhardware} \geq \text{HardwareBound}(IO) + \text{HardwareBound}(CPU) \quad (14)$$

where $\text{HardwareBound}(IO)$ and $\text{HardwareBound}(CPU)$ are the hardware bounds for the I/O and CPU sections respectively. Thus, substituting for the hardware bound discussed earlier:

$$T_{NOhardware} \geq \frac{T_d}{N_d} + \frac{T_i}{N_i} = T_d \left(\frac{1}{N_d} + \frac{F_i}{N_i} \right) \quad (15)$$

And the speedup is computed as before:

$$S_{NOhardware} = \frac{T_i + T_d}{T(NO)} \leq \frac{1 + F_i}{\frac{1}{N_d} + \frac{F_i}{N_i}} \quad (16)$$

The above are for the hardware bound only. The software bound is similar, except we replace N_d and N_i with A_d and A_i . We therefore get:

$$T_{NOsoftware} \geq \frac{T_d}{A_d} + \frac{T_i}{A_i} = T_d \left(\frac{1}{A_d} + \frac{F_i}{A_i} \right) \quad (17)$$

and as before:

$$S_{NOsoftware} = \frac{T_i + T_d}{T_{NO}} \leq \frac{1 + F_i}{\frac{1}{A_d} + \frac{F_i}{A_i}} \quad (18)$$

where the subscript NO stands for no overlap. Combining the hardware and software bound together, we obtain:

$$S_{NO} \leq \min(S_{NOhardware}, S_{NOsoftware}) \quad (19)$$

and substituting for the hardware and software bounds from (16) and (18) and taking into account that both bounds have the same constant numerator, we obtain for the speedup the following expression.

$$S_{NO} \leq \frac{1 + F_i}{\max\left(\frac{1}{N_d} + \frac{F_i}{N_i}, \frac{1}{A_d} + \frac{F_i}{A_i}\right)} \quad (20)$$

And for the response time we immediately obtain:

$$T_{NO} = \frac{T_d + T_i}{S_{NO}} \geq \max\left(T_d \left(\frac{1}{N_d} + \frac{F_i}{N_i} \right), T_d \left(\frac{1}{A_d} + \frac{F_i}{A_i} \right)\right) \quad (21)$$

Equations (20) and (21) together with equations (10) and (13) form a set of upper and lower bounds on speedup and response time for the no overlap case.

3.2.2 Case II: Overlap between CPU and I/O operations

Now let us consider the case where there is an overlap between I/O and computation. As, before, the discussion that follows is valid for any number of processors and I/O devices. In this case, equation (8) becomes:

$$T_o = Time_{IO} + Time_{CPU} - Time_{Overlap} \quad (22)$$

Where O stands for "Overlap".

The first two terms are the same as for the previous case. For the third term, we require another application characteristic. We first need to know the fraction of the total I/O time that is executed in the overlap section (denoted by P), and also the average overlap parallelism of the I/O in this overlap section (denoted by A_o). Although the value of P may depend on system parameters, for the following discussion P is assumed to be known for a given system. The average parallelism in the overlap section is required, since the average I/O parallelism of this section can be very different from the average parallelism of the entire I/O section. If these two parameters are known, then the third term can be expressed as the ratio between the time in the overlap section to the speedup in the overlap section:

$$Time_{Overlap} = \frac{TimeInOverlapSection}{SpeedupInOverlapSection} = \frac{PT_i}{SpeedupInOverlapSection} \quad (23)$$

and substituting for speedup from (2) we obtain:

$$Time_{Overlap} \leq \frac{PT_i}{\min \left(A_o, \frac{N_i A_o}{N_i + A_o - 1 - \frac{(N_i - 1)(A_o - 1)}{m_o - 1}} \right)} = \frac{PF_i T_d}{\min \left(A_o, \frac{N_i A_o}{N_i + A_o - 1 - \frac{(N_i - 1)(A_o - 1)}{m_o - 1}} \right)} \quad (24)$$

Thus, the response time of the application is bounded as given by the following inequality:

$$T_o \leq T_d \left(\frac{1}{\min \left(A_d, \frac{N_d A_d}{N_d + A_d - 1 - \frac{(N_d - 1)(A_d - 1)}{m_d - 1}} \right)} + \frac{F_i}{\min \left(A_i, \frac{N_i A_i}{N_i + A_i - 1 - \frac{(N_i - 1)(A_i - 1)}{m_i - 1}} \right)} \right. \\ \left. - \frac{P F_i}{\min \left(A_o, \frac{N_i A_o}{N_i + A_o - 1 - \frac{(N_i - 1)(A_o - 1)}{m_o - 1}} \right)} \right) \quad (25)$$

Turning our attention to the speedup, the total time on a one CPU - one disk system is still the same, and therefore the speed-up is given by:

$$S_o = \frac{T_d + T_i}{T_o} \quad (26)$$

and substituting from (25) we get:

$$S_o \geq \frac{1 + F_i}{\frac{F_i}{\min \left(A_i, \frac{N_i A_i}{N_i + A_i - 1 - \frac{(N_i - 1)(A_i - 1)}{m_i - 1}} \right)} - \frac{P F_i}{\min \left(A_o, \frac{N_i A_o}{N_i + A_o - 1 - \frac{(N_i - 1)(A_o - 1)}{m_o - 1}} \right)}} \\ + \frac{1}{\min \left(A_d, \frac{N_d A_d}{N_d + A_d - 1 - \frac{(N_d - 1)(A_d - 1)}{m_d - 1}} \right)} \quad (27)$$

As a special case, one can consider an application in which the average I/O parallelism and the

maximum I/O parallelism in the overlap section are the same as those of the entire I/O section respectively ($A_i = A_o$ and $m_i = m_o$). In this case, equation (25) for response time becomes:

$$T_o \leq T_d \left(\frac{1}{\min \left(A_d, \frac{N_d A_d}{N_d + A_d - 1 - \frac{(N_d - 1)(A_d - 1)}{m_d - 1}} \right)} + \frac{F_i - P F_i}{\min \left(A_i, \frac{N_i A_i}{N_i + A_i - 1 - \frac{(N_i - 1)(A_i - 1)}{m_i - 1}} \right)} \right)$$

and equation (27) for speedup becomes:

$$S_o \geq \frac{1 + F_i}{\frac{F_i - P F_i}{\min \left(A_i, \frac{N_i A_i}{N_i + A_i - 1 - \frac{(N_i - 1)(A_i - 1)}{m_i - 1}} \right)} + \frac{1}{\min \left(A_d, \frac{N_d A_d}{N_d + A_d - 1 - \frac{(N_d - 1)(A_d - 1)}{m_d - 1}} \right)}}$$

It is also clear that when the fraction of I/O time done in the overlap section (P) is zero, equations (25) and (27) become (10) and (13), which was expected.

Now, the upper bound on speedup can be obtained in the same manner as for the no overlap case:

$$T_{O_{hardware}} \geq \text{HardwareBound}(IO) + \text{HardwareBound}(CPU) - \text{HardwareBound}(Overlap) \quad (28)$$

Substituting for the hardware bound in a similar fashion as in the no overlap case we obtain:

$$T_{O_{hardware}} \geq T_d \left(\frac{1}{N_d} + \frac{F_i}{N_i} - \frac{P F_i}{N_i} \right) = T_d \left(\frac{1}{N_d} + \frac{F_i(1 - P)}{N_i} \right) \quad (29)$$

and:

$$S_{Ohardware} = \frac{T_d + T_i}{T_o} \leq \frac{1 + F_i}{\frac{1}{N_d} + \frac{F_i}{N_i} - \frac{PF_i}{N_i}} = \frac{1 + F_i}{\frac{1}{N_d} + \frac{F_i(1-P)}{N_i}} \quad (30)$$

Again, as before, the software bound is obtained just by replacing N by A. However, in this case we must take into account the fact that we have considered the average parallelism in the overlap region to be different than the average parallelism of the I/O region. We therefore have:

$$T_{Osoftware} \geq T_d \left(\frac{1}{A_d} + \frac{F_i}{A_i} - \frac{PF_i}{A_o} \right) \quad (31)$$

and

$$S_{Osoftware} = \frac{T_d + T_i}{T_o} \leq \frac{1 + F_i}{\frac{1}{A_d} + \frac{F_i}{A_i} - \frac{PF_i}{A_o}} \quad (32)$$

Putting both the software and hardware bounds together, we obtain:

$$S_o \leq \min(S_{Ohardware}, S_{Osoftware}) \quad (33)$$

Substituting from (32) and (30) and noting that both equations have the same constant numerator we get:

$$S_o \leq \frac{1 + F_i}{\max\left(\left(\frac{1}{N_d} + \frac{F_i(1-P)}{N_i}\right), \left(\frac{1}{A_d} + \frac{F_i}{A_i} - \frac{PF_i}{A_o}\right)\right)} \quad (34)$$

and

$$T_o = \frac{T_d + T_i}{S_o} \geq \max\left(T_d \left(\frac{1}{N_d} + \frac{F_i(1-P)}{N_i}\right), T_d \left(\frac{1}{A_d} + \frac{F_i}{A_i} - \frac{PF_i}{A_o}\right)\right) \quad (35)$$

Equations (34) and (35) together with equations (25) and (27) form a set of upper and lower bounds on speedup and response time for the overlap case.

3.2.3 Case III: General Case

We would also like to know what happens when the I/O and the computations are not done in the manner shown in Figure 3, but when I/O and CPU operations can overlap in arbitrary fashion during the execution of the application. We will show that such a system is equivalent to one in which both the CPU and I/O operations are performed in a single burst (see Figure 4) given a set of conditions (outlined in equations 38 and 41) are true.

An application with multiple CPU and I/O bursts can be considered as a sum of overlap and no overlap bounds and could easily be expressed mathematically. For such an application, the response time will be given by the sum of all the response times of the CPU bursts plus the sum of all the response times for the I/O operations, minus the sum of all overlap sections.

$$T_{Arbitrary} = \sum_k Time_k(CPU) + \sum_k Time_k(IO) - \sum_k Time_k(Overlap) \quad (36)$$

where k is an index that runs over all the different sections of the application. Now, the first sum, as shown before (see equation 26), is of the form:

$$\sum_k Time_k(CPU) \leq \sum_k \frac{T_{dk}}{\min \left(A_{dk}, \frac{N_{dk} A_{dk}}{N_{dk} + A_{dk} - 1 - \frac{(N_{dk} - 1)(A_{dk} - 1)}{m_{dk} - 1}} \right)} \quad (37)$$

Where the subscript k associated to a parameter means the value of that parameter for a given section of the application. Now, if all the following are true:

$$\begin{aligned} A_{dk} &= A_d \\ m_{dk} &= m_d \end{aligned} \quad (38)$$

then (37) becomes:

$$\sum_k Time_k (CPU) \leq \sum_k \frac{T_{dk}}{\min \left(A_d, \frac{N_d A_d}{N_d + A_d - 1 - \frac{(N_d - 1)(A_d - 1)}{m_d - 1}} \right)} \quad (39)$$

and since the denominator is the same for all terms, the equation becomes:

$$\sum_k Time_k (CPU) \leq \frac{\sum_k T_{dk}}{\min \left(A_d, \frac{N_d A_d}{N_d + A_d - 1 - \frac{(N_d - 1)(A_d - 1)}{m_d - 1}} \right)} = \frac{T_d}{\min \left(A_d, \frac{N_d A_d}{N_d + A_d - 1 - \frac{(N_d - 1)(A_d - 1)}{m_d - 1}} \right)} \quad (40)$$

Clearly the same applies to the other two terms, if for any k:

$$\begin{aligned} A_{ik} &= A_i \\ m_{ik} &= m_i \\ A_{ok} &= A_o \\ m_{ok} &= m_o \end{aligned} \quad (41)$$

and thus, in this case, equation (36) reduces to equation (25). Therefore, the general case reduces to the overlap case if the conditions (38) and (41) stated above are true.

Figure 4 Reduction of Case III to Case II

3.3 Sample Numerical Result.

To better illustrate the bounds obtained above, a graph of the bounds for both the overlap case and no overlap case is plotted below. The performance bounds are plotted as a function of N_i with all other parameters being constants. For all graphs the values presented in Table 4 are chosen for these fixed parameters.

Table 4: Parameters for the Sample Bounds Calculation (Figure 5 & 6)

Parameter and value
I/O factor = 1
$N_d = 6$
$A_d = 3$
$A_i = 4$
$A_o = 5$
$m_d = 6$
$m_i = 6$
$m_o = 4$
$P = 0.1$ (constant)
$T_d = 100$ sec

Substituting these values in equations (10), (13), (20), (21), (25), (27), (34), and (35) and plotting the resulting equations as functions of N_i we obtain the results plotted in Figure 5 and Figure 6.

a) Response time versus Number of Disks

b) Speedup versus number of disks

Figure 5 The No Overlap Case

a) Response time versus Number of Disks

b) Speedup versus number of disks

Figure 6 The Overlap Case

From the above graphs we notice that for the same parameters both the upper and lower bounds on speedup are higher for the overlap case. Therefore, as expected, such an application can potentially perform better than an application that does not overlap CPU and I/O operations.

What is the minimum degree of overlap for which one can guarantee the speedup of the overlap application to be higher than the speedup of an application without overlap is an interesting question. Let us determine this minimum value of P by finding out where the lower bound on speedup for the overlap case is higher than the upper bound on speedup for the no overlap case. For this, we choose the same parameters as for the previous test and we add $N_i = 3$ and $N_i = 6$. Plotting the lower bound on speedup for the overlap case and the upper bound on speedup for the no overlap case we get:

Figure 7 Upper Bound (No Overlap) and Lower Bound (Overlap) as a function of P for the $N_i = 3$ case.

From Figure 7 we immediately see that for P greater than 75% the lower bound on speedup for

the overlap case becomes higher then the upper bound on speedup for the no overlap case.

Figure 8 Upper Bound (No Overlap) and Lower Bound (Overlap) as a function of P for the $N_i = 6$ case.

From Figure 8 we see that in this case the overlap case always performs better than the no overlap case.

4.0 Experimental Environment

4.1 Introduction

Results presented in the subsequent chapters of this thesis are based on measurements of real parallel I/O applications. These measurements were performed on a cluster of workstations connected by a 10Mb Ethernet. Parallelism is achieved by using PVM [PVM], a tool that allows workstations on the network to be configured into a parallel virtual machine. Each node (workstation) of the parallel virtual machine is configured to be either an I/O node (the I/O operations are done using the local disk of that workstation) or a compute node (only the CPU of the corresponding workstation is used). Synchronization is done using the PVM communication primitives.

For this virtual machine, synthetic applications written in C that implement the discussed I/O models were developed. Each application has two distinct parts to it: threads that run on the compute nodes (compute threads), and one dispatcher thread that runs separately on a different workstation in the virtual machine. Also, on every I/O node there is a thread that services all the I/O for that device (I/O thread). The dispatcher thread is responsible for scheduling the I/O requests and performing the coordination among the CPU and I/O operation. It also keeps track of the response time of the application. A description of PVM is presented first, followed by a discussion of the C programs that implement the different I/O models.

An important characteristic of the application is the I/O factor. The compute threads consume CPU time by executing dummy for loops. The amount of work to be done by each compute thread was fixed to 15.8 seconds. With this value PVM overhead is just a small fraction of the execution time and the run times for the experiments are also reasonable. This enabled us to perform a large number of measurements that led to a good confidence interval. The amount of I/O work is a function of the number of bytes read/written. The length of the data block was

adjusted in order to produce an I/O factor close to 1. Once this was done, many measurements were taken in order to measure the exact mean value of the I/O factor which was found to be 0.94. This measurement has an accuracy such that the confidence interval is within 1% of the average, at a confidence level of 95%. This measured value of the I/O factor is denoted by F_i' . All the F_i values used for the synthetic programs are normalized with respect to F_i' . This means that an F_i of 1 corresponds to a measured I/O factor of 0.94. F_i values greater than one are created by scaling the number of bytes used in the data transfer. For example $F_i=2$ corresponds to transferring twice the number of bytes used for $F_i=1$ when the compute threads in the two cases perform identical amount of work.

The number of bytes to be transferred is divided into a number of equal parts that depends on the I/O model (O1 or O2) and the number of compute threads in the application.

The determination of the number of iterations in the for loop for generating the 15.8 second compute thread was done on a SUN-2 machine. The determination of the number of bytes to be transferred to and from the I/O disks to produce the I/O factor discussed in the previous paragraphs was made on a SUN-5 machine, equipped with an ST1480 disk. A given number of bytes was read and written and the average time measured over both operations was used in the determination of the I/O factor. The same count in the for loop and the same number of bytes for the I/O operations were used on each machine to construct a synthetic application with a given value of F_i . However, even when the same hardware is used, some differences in execution times of the compute threads and I/O threads are expected on different machines. These differences accrue from the different levels of interactions with operating system processes, different initial positions of disk heads, etc.

4.2 Parallel Virtual Machine

The PVM software developed at MIT [PVM] provides a unified framework within which the development of efficient parallel programs can be done in a straightforward manner using

existing hardware. PVM allows a collection of heterogeneous machines to function as a single high-performance parallel virtual machine. PVM transparently handles all message routing, data conversion, and task scheduling across a network of heterogeneous systems.

The computing model for PVM is simple and accommodates a large variety of program structures. The programming interface is straightforward and easy to use. The user writes his/her application as a collection of cooperating tasks which access PVM resources through a library of standard interface routines. These library routines support initiation, termination, synchronization, and communication between tasks. The message passing routines are oriented towards heterogeneous operation, involving strongly typed constructs for buffering and transmission. Communication constructs include low level mechanisms for sending and receiving data structures as well as high level primitives such as broadcast, barrier synchronization, and global sum. Of importance in this thesis are the simple data communication primitives and the broadcasting primitives. They will be used to achieve communication and synchronization among the CPU nodes and the I/O nodes.

PVM tasks may possess arbitrary control, that is, at any point during execution a task can start or stop other tasks or add or delete computers (nodes) from the virtual machines. Any process may communicate and/or synchronize with any other process running on the virtual machine. Any specific control structure to synchronize the threads in the virtual machine may be implemented using PVM by the use of appropriate constructs.

Given the above characteristics, PVM is a good tool for the task at hand since it provides all the features required for implementing the parallel I/O models described.

4.3 Workstations in the Virtual Machine

For all tests presented in this thesis a set of 13 workstations were used. These workstations were running PVM and thus formed a parallel virtual machine. Six of the workstations were configured as I/O nodes (i.e. they were running threads that service I/O requests by reading or writing to the workstation local disk). Another six workstations were configured as compute nodes, and they were running compute threads. Finally, a single workstation was used for running a dispatcher thread that keeps track of the response time of the application and also takes care of the scheduling of I/O.

As mentioned above the workstations used in this thesis were divided in two groups, one for the compute nodes and one for the I/O nodes. All six compute nodes, were SUN-2 workstations running at the same speed. For the I/O nodes a variety of workstations (SUN-2, SUN-5 and SUN-10) were used. These were chosen such that the local disks on these machines were identical. However due to workstation limitations only five such workstations were available. These five workstations had an ST1480 disk. The sixth workstation had a slower CP30548 disk. This slower disk was only used for the tests where $n_i=6$. As a result the I/O factor may be higher than the desired F_i when the slower disk is used. For a n_i value smaller than 6 only identical disks were used. All twelve workstations used for the compute and I/O nodes were running the Solaris operating system.

The scheduling of all the threads on their corresponding node was done by the Unix scheduler running on that particular machine, and therefore it was not controlled by the application. This may cause problems since all 13 workstations are part of Carleton University Systems and Computer Engineering Department research LAN. Thus, students in the department have access to these machines and could start jobs on them at any time, jobs which could interfere with the experiments. Any such job would be detrimental to any measurement since it will share the processor with the thread running on that particular machine and therefore it will increase the response time of the application. Furthermore, these jobs, that are started by other

users, can happen at random times and are therefore hard to account for. To minimize this problem, a Unix shell script was developed. This script checks for the load on all 13 workstations, and if the load on all of them is below a given threshold (less than 0.05), a measurement is started. At the end of the measurement, the script is invoked again, and if it passes, the measurement is accepted as being valid, else the measurement is rejected. Clearly, the possibility of a short job (i.e. that starts after the measurement started and completes before measurement ends) interfering with the measurement still exists. To minimize the risk of having such interfering jobs, all measurements were done at night, when the system is likely to be idle.

Another problem arises from the fact that there are many other workstations on the LAN discussed above, and some of these workstations could be running applications that involve large data transfers through the network. Although these jobs run on different machines, they use the same Ethernet, and therefore such jobs may interfere with the communication of the threads running in the virtual machine. The measure to guard against this effect was to run all measurements at night.

Yet another problem is the fact that all these workstations have a set of demons that grab the CPU from time to time. Again, this is hard to guard against. To eliminate this problem, many measurements with same parameters were done. From the set of values obtained, the ones that were significantly different from the average were discarded.

A solution to most of the above problems would be to run the experiments on a quiet network, i.e. where there are no other users in the system. For such a system one would be faced only with the problem of having system daemons competing for the CPU. Although such a network is available at Carleton University it was not used since the number of workstations in this network is small.

4.4 Implementation of the I/O Models

In this thesis only the NOP, O1 and O2 models are considered. As stated before, due to the poor performance of the NO model observed in [Kwong96], the NO model will not be discussed here. For each of them a “synthetic” C application was developed for the virtual machine. A “synthetic” application does not compute any useful result. It just mimics the corresponding I/O model by doing real read/writes of some random data to simulate the I/O stages for the application, and doing dummy computations to account for the computation stages of the application. This was done because one would need to transfer data between the compute and the I/O devices in order to perform some meaningful computations. However, if this is done, the delays through the network come into play since all these programs deal with reading and writing large amounts of data (between 5 and 25 Mb per thread). These network delays are significant for the parallel virtual machine under consideration due to the slow Ethernet connection between the nodes. However, a number of commercially available parallel computers, like the Intel Paragon, use faster interconnection networks. These delays are minimal on such systems. Moreover, we are primarily interested in the effect of I/O characteristics of jobs on performance. The larger network delays of the Ethernet tend to muffle the effect of I/O characteristics of applications. Thus to eliminate network delays and get a better understanding of the I/O models, the communication implemented was restricted only to the communication required for scheduling and synchronization. Actual data transfers were implemented in a different class of applications that are described in the next paragraph.

Two other real applications were also implemented. Both of them accomplish the same task: the squaring of a 10Mb matrix. However one of them does so using an O1 I/O model, while the other uses an O2 model. In these two programs, actual transfer of data takes place, and therefore meaningful computations can be performed by the compute nodes on the available data. Thus, these results also include the effect of the network, which is not present in the case of the synthetic applications.

A detailed description of all five programs follows. The actual C code is not included, but Appendix A contains pseudo-code for the threads of all the programs.

4.4.1 Synthetic - NOP

In the Synthetic-NOP case when a job starts the dispatcher thread is started. The dispatcher thread then spawns six compute threads and a variable number (between 1 and 6) of I/O threads depending on how many I/O devices are required for a particular experiment.

When all these setup tasks are complete, the dispatcher starts recording the time and signals all compute threads to begin execution. The dispatcher then waits for I/O requests from the compute threads and sends each of them to a free I/O node that is capable of servicing the particular read/write request. In certain experiments, only one disk can service requests from a particular compute thread. In other experiments, data replication is used and read/write requests can be serviced by one of many disks. If there is no such free I/O node, then the request is queued at the dispatcher until a I/O node is freed. After all initial reads of the NOP model are complete the dispatcher again signals all the compute threads to begin the compute stage of the NOP model. This compute stage consists in performing some dummy computations inside a for loop. Once a thread finishes the compute stage, it signals the dispatcher. When the dispatcher receives signals from all compute threads it multicasts to all compute threads the signal to begin the writing stage of the NOP model. Here again, the dispatcher waits for I/O requests and passes them to the I/O subsystem in a way similar to that described earlier. When every compute thread has finished the write, the dispatcher records the time, cleans up the system by killing all threads in the virtual machine, and exits.

The I/O thread is started by the dispatcher. It starts by writing to the local disk a file of corresponding size, large enough to accommodate an I/O transfer of the size imposed by the I/O factor at which the application is run. The time required for this operation will not be recorded by

the dispatcher. When the dispatcher signals the execution to begin it goes into an infinite loop waiting for requests. When a request arrives from the dispatcher thread, it figures out what kind of request it is, and either reads from the file or writes to the file already created on the local disk. When this is done it signals the compute thread that initiated the request that the I/O is complete and it goes back to waiting for requests.

The compute thread is also started by the dispatcher. After it starts, it waits for the dispatcher to signal the start of the measurement, then issues an I/O request for a read and waits for a response. After the response is received (the reading is complete) it waits for the dispatcher to signal the beginning of the execution stage. Once this signal is received, the thread begins computations. When it finishes, it signals the dispatcher that it is done, and waits for the dispatcher to signal for the beginning of the writing stage. Once this signal is received, the thread issues a write request. The compute thread then waits for an acknowledgment from the I/O subsystem. Once this is received the thread terminates.

4.4.2 Synthetic - O1

When a job starts the dispatcher thread is created first. The dispatcher thread then spawns six compute threads and a variable number (between 1 and 6) of I/O threads depending on how many I/O devices are required for a particular experiment.

When all these setup tasks are complete, the dispatcher starts recording time and signals all compute threads to begin execution. The dispatcher then waits for I/O requests from the compute threads and sends them to a free I/O node that is capable of servicing the request. If there is no such a free I/O node, the request is queued at the dispatcher until one I/O node is freed. For this model, the threads synchronize only at the beginning and at the end of the program, therefore in this case the dispatcher just waits until all I/O requests (reads and writes) are received and processed. When every compute thread has finished, the dispatcher records the

response time. Finally, it cleans up the system by killing all threads in the virtual machine and then dies.

The I/O threads are exactly the same as for the synthetic - NOP case.

A compute thread is started by the dispatcher as for the synthetic - NOP case. After it starts, the thread waits for the dispatcher to signal the start of measurement. After receiving the signal, it issues an I/O request for a read and waits for a response. After the response is received (the reading is complete) the thread begins computations. When it finishes the computations, the thread issues an I/O write request, and waits for an acknowledgment from the I/O subsystem. Once this is received the thread terminates.

4.4.3 Synthetic - O2

In this case, when the job starts the dispatcher thread is started. The dispatcher thread then spawns six compute threads and a variable number (between 1 and 6) of I/O threads depending on how many I/O devices are required for a particular experiment.

When all these setup tasks are complete the dispatcher starts recording time and signals all compute threads to begin execution. The dispatcher then waits for I/O requests from the compute threads and sends them to a free I/O nodes capable of servicing the request. If there is no such free I/O node, the request is queued at the dispatcher, until one I/O node is freed. In this case, there will be S times more requests to be serviced compared to the synthetic - O1 case. Finally, after all the requests are serviced, the dispatcher cleans up the system by killing all threads in the virtual machine, records the time and then exits.

The I/O threads are exactly the same as for the NOP case.

The compute thread is similar to the compute thread for the synthetic - O1 model. The only difference is that in this case the read, compute, write cycle described in the synthetic - O1 case is repeated a number of times in order to obtain the required value for S . Half of the S operations are reads while the other half of the operations are writes. However, each I/O request is a factor of $1/S$ smaller in size relative to their synthetic - O1 counterparts.

4.4.4 Matrix Multiplication - O1

In this case when a job starts the dispatcher thread is started. The dispatcher thread is exactly the same as for the synthetic O1 model.

The compute thread has the same structure as the synthetic - O1 compute thread. After it starts the compute thread issues a read request and waits for an acknowledgment from the I/O subsystem. With the acknowledgment it also receives some rows of a matrix that must be multiplied by its transpose. The exact number of rows received is the total number of rows of the matrix divided by the number of compute threads. The compute thread then proceeds to perform the required multiplications. When this is done, the results are sent for writing to the I/O subsystem. Finally, the compute thread signals the dispatcher and dies.

The I/O thread begins by generating a file containing the matrix on the local disk. The time required for this operation will not be recorded by the dispatcher. Since we are doing matrix multiplications, there are no conditions for the entries of this matrix (i.e. we do not have to verify that the determinant is not zero or anything of the sort). After this is complete, the I/O thread waits for I/O requests in an infinite loop. If a read request arrives then it reads the corresponding data and sends it to the compute thread that originated the request. If it receives a write request, it performs the writing and sends an acknowledgment to the thread that issued the request.

4.4.5 Matrix Multiplication - O2

In this case when a job starts the dispatcher thread is started. The dispatcher thread is exactly the same as in the case of the synthetic - O2 model.

The compute thread has the same structure as the O2 compute thread. After it starts, the following is repeated S (number of slices) times: the thread issues a read request and waits for an acknowledgment from the I/O subsystem. With the acknowledgment it also receives some rows (the exact number is the total number of rows of the matrix divided by the number of compute threads divided by the number of slices) of a matrix that must be multiplied by its transpose. It then proceeds to perform the required multiplications. When this is done, it sends the results to the I/O subsystem and goes back to the top of the loop. When all iterations are completed the thread signals the dispatcher and dies.

The I/O thread begins by generating on the local disk a file containing the matrix (the time required for this operation will not be recorded by the dispatcher). Since we are doing matrix multiplications there are no conditions (i.e. we do not have to verify that the determinant is not zero or anything of the sort) for the entries of this matrix. After this is complete the I/O thread waits for I/O requests in an infinite loop. If a read request arrives then it reads the corresponding data and sends it to the compute thread that originated the request. If it receives a write request it performs the write and sends an acknowledgment to the compute thread.

4.5 Measurement Accuracy.

All measurements presented in this thesis are measurements of the average response time of an application running in the virtual machine. These average responses time were obtained by measuring the response time of an application many times and then computing the average. All

average response time values presented here demonstrate a confidence interval of 5% of the average, at a confidence level of 95%.

The value of the standard deviation was computed from the set of values measured using a Perl script and assuming that the distribution of the values is a normal distribution. This is appropriate since more than forty measurements were performed for each case.

Furthermore, to improve the accuracy of the measurement, and therefore to reduce the number of measurements required to achieve the desired confidence level, the set of experimental results was first inspected to remove all experiments that produced drastically different values (a factor of at least 1.5) from the mean. The inspection was done manually, and usually a maximum of two such values were found for every fifty samples. These “abnormal” values were attributed to a number of factors discussed in Section 4.3. This removal of measurement was done only for those experiments with C_{vi} and C_{vd} values of zero, since in such cases we expect all the measurements to be comparable with one another. To achieve this for C_{vi} and C_{vd} values greater than zero, one has to run an experiment with a given combination of random parameter values multiple times. This would lead to prohibitively long run times for experiments. That is why the removal of abnormal values was restricted to the experiments with $C_{vi} = C_{vd} = 0$.

The same measurement procedure was followed for both the uniprogrammed case and the multiprogrammed system.

5.0 Performance of I/O Models

5.1 Introduction

As described in previous chapters, many parameters affect the performance of a parallel I/O application. For all measurements discussed in Chapter 5 and Chapter 6 only the parameter N_i (the number of I/O devices) was varied at a time, since we are interested in getting a better understanding of the performance of the parallel I/O models under consideration as a function of the number of I/O devices.

For all experiments presented in this chapter, full read and write data replication is implemented. That means that a thread can read or write on any disk at any time.

The performance of the I/O models is measured using two metrics. One is the response time (see section 2.5 for definition) of an application, and the other is a measure of the relative performance of the I/O models called “ratio”. The response time of an application is measured by the dispatcher thread. The ratio metric is based on the average response time for two or more I/O models with the same values of F_i and CV_i .

$$Ratio(IOModelA) = \frac{ResponseTime(IOModelA)}{\max(ResponseTime)}$$

Clearly, the smaller the value of the ratio for a particular I/O model, the better this I/O model performs relative to the other I/O models. This ratio metric has the advantage that it “normalizes” the results, thus enabling us to compare relative performance under different conditions. All results presented in this section are based on measurements performed on the virtual system. Both uniprogrammed and multiprogrammed systems are considered. The results for the

uniprogrammed system are presented first followed by the results for the multiprogrammed system

For all results presented in this chapter, the following parameters were maintained constant:

$$n_d = N_d = 6$$

$$n_i = N_i$$

$$CV_d = 0$$

Also note that for the data presented in Chapter 5 and Chapter 6 the number of I/O threads in a job will always be equal to the total number of disks in the system, and the number of compute threads in a job will always be equal to the number of CPU's in the system.

5.2 Uniprogrammed System

A number of experiments were performed for a uniprogrammed environment in order to understand the effects of different parameters on the performance of the different I/O models. These experiments are presented below.

5.2.1 I/O factor (Synthetic)

First of all, the three I/O models under consideration NOP, O1 and O2 (as stated earlier, the NO model was not investigated since it does not implement parallel I/O) were evaluated in a uniprogrammed environment, that is, only one application is run on the system. A number of measurements were performed for different sets of parameters of the I/O models. Only a few of these results are presented here. More data is available in Appendix B and Appendix C.

The first experiment performed is concerned with evaluating the effects of F_i (the I/O factor) on the performance of the synthetic programs. Below are the graphs obtained for $F_i = 1$ (Figure 9) and $F_i = 2$ (Figure 10).

a) Response Time vs. Number of I/O Devices.

b) Ratio vs. Number of I/O Devices.

Figure 9 Results for the Synthetic Programs with $F_I=1$, $CV_I=0$

a) Response Time vs. Number of I/O Devices.

b) Ratio vs. Number of I/O Devices.

Figure 10 Results for the Synthetic Programs with $F_i=2$, $CV_i=0$

As can be seen from these figures, the NOP model has the worst performance, while the O2 model performs the best. This is because the O2 model has the most CPU-I/O overlap while the NOP model has no CPU-I/O overlap. Note that for $N_i=1$ (sequential I/O on one disk) O1 performs better than O2. In this borderline case, the difference in CPU - I/O overlap between O1 and O2 is smaller in comparison to the $N_i = 6$ case. Moreover, the O2 model has more overhead associated with the setting up and servicing of the extra I/O requests (there are S times more I/O requests in the O2 model than in the O1 model). Overall, O1 exhibits a slightly better performance than O2 for $N_i = 1$.

From Figure 9 and Figure 10 one can also see that there is no change in the relative performance of the models when the I/O factor is increased from 1 to 2.

Another important result is the fact that the gain in going from two I/O devices to three I/O devices is very small (as can be seen from Figure 7a and Figure 8a).

One should also note that the number of I/O requests on an I/O device is a discrete value. Therefore, if we increase the number of I/O devices from N_i to N_i+1 , for some specific values of N_i we may for example have the following scenario:

With 3 devices:

Disk A services 2 requests
 Disk B services 2 requests
 Disk C services 2 requests.

Increasing to 4 devices:

Disk A services 2 requests
 Disk B services 2 requests
 Disk C services 1 request
 Disk D services 1 request.

For the above scenario, the bottleneck is at Disk A, and there is no improvement when we add one more disk, since the total number of requests to be serviced by disk A is the same in both cases. Clearly there is no way to distribute the I/O requests such that we have 1.5 I/O requests per disk, which would be the optimal value, since the number of request must be an integer value.

Note that the above results are in agreement with the simulation results of the same system obtained by P. Kwong and presented in [KwoM96].

5.2.2 Variability in I/O Demand

The next experiment performed was to determine the effect of introducing a coefficient of variation in the I/O request size. A CV_i value of 0.5 was obtained by using a uniform distribution for the amount of data to be transferred in the I/O operations performed at the I/O nodes. Some of the results obtained for the synthetic programs are shown in Figure 11 and Figure 12. (See Appendix B for a complete set of results).

Clearly, for a CV_i value of 0.5, O2 still performs better than O1 which in turn performs better than NOP. For any given I/O model, number of I/O devices, and F_i , the average response time in this case is greater than the average response time in the $CV_i=0$. This is because with a higher CV_i some of the I/O requests have larger I/O demands in comparison to their $CV_i=0$ counterparts. However, the relative performance of the models did not change significantly. Again, note that for $N_i=1$, O1 performs better than O2 as in the previous case.

These results indicate that although a CV_i greater than zero has a small negative impact on the performance of the I/O models, it has little effect on the relative performance of the I/O models.

a) Response Time vs. Number of I/O Devices.

b) Ratio vs. Number of I/O Devices.

Figure 11 Results for the Synthetic Programs with $F_i=1$, $CV_i=0.5$

a) Response Time vs. Number of I/O Devices.

b) Ratio vs. Number of I/O Devices.

Figure 12 Results for the Synthetic Programs with $F_1=2$, $CV_1=0.5$

Also from Figure 11 and Figure 12 we see that increasing the I/O factor has no impact on the relative performance of the applications.

5.2.3 Number of O2 slices

The effect of changing S (the number of slices) for the O2 model are discussed in this section. Measurements were taken for different values of S and F_i while CV_i was set to zero. A representative set of results are presented here (Figure 13 and Figure 14). More data is available in Appendix B. The notation S2, S5 etc. used in these figures refers to the O2 models having 2, 5, etc. slices respectively.

From these two figures we see that there is a relationship between the I/O factor and the number of slices used. For a small value of F_i , the best value of S is two. As F_i is increased, the best value of S shifts from two to five or ten, depending on the number of disks used. For small values of F_i there is little opportunity of having CPU-I/O overlap, since there is little I/O to be done. Thus, for such a case, the use of a large number of slices is detrimental because as S increases so does the overhead associated with setting up the extra I/O requests. However, as F_i increases, a higher degree of CPU-I/O overlap is possible, and this leads to a higher performance of the model. In this case, the benefit of having more slices (more CPU-I/O overlap) offsets the cost of having more overhead and thus, the best value of S increases with an increase in F_i .

From the above experiments we conclude that there is a direct relationship between F_i and the optimum number of O2 slices.

a) Response Time vs. Number of I/O Devices.

b) Ratio vs. Number of I/O Devices.

Figure 13 Results with $F_1=0.5$, $CV_1=0$ for Different Number of O2 Slices

a) Response Time vs. Number of I/O Devices.

b) Ratio vs. Number of I/O Devices.

Figure 14 Results with $F_i=4$, $CV_i=0$ for Different Number of O2 Slices.

5.2.4 The Effect of Network Traffic

The results presented in the previous section were based on the execution of synthetic programs. In order to study the performance of the I/O models with real network traffic a matrix multiplication program (see sections 4.4.4 and 4.4.5 for a description of the program) is executed in a uniprogrammed system. Since the performance of the NOP model was observed to be much inferior to the O1 and O2 models in case of the synthetic models, we have only experimented with the O1 and O2 models for the matrix multiplication programs. The results are shown in Figure 15. As in the case of synthetic programs, O2 performs better for large N_i values, while O1 is best for an N_i of one. The explanation of why this is so is similar to that provided in Section 5.2.1.

Comparing the above with the results of the synthetic workload presented in Section 5.2.1 we can also understand the effect of passing long messages between the I/O subsystem and the compute subsystem through the network.

By comparing the performance of these programs with their synthetic counterparts, we note that the difference in performance between O1 and O2 is smaller in the case of the matrix multiplication program. A rationale for this behavior is briefly presented. Both CPU-I/O overlap and waiting time for message transfer contribute to system performance. O2 exhibits better CPU-I/O overlap than O1 and demonstrates a superior performance in the synthetic case where the message transfer times are small. The delays for message transfer become large in the matrix multiplication program which transfer real data. Thus the effect of message transfer on performance increases in the case of the matrix multiplication program. As a result the relative impact of CPU-I/O overlap on overall performance decreases and the difference between O2 and O1 performance tends to diminish as we go from the synthetic system to the matrix multiplication applications.

Note that in this case the I/O factor was measured directly on the system.

a) Response Time vs. Number of I/O Devices.

b) Ratio vs. Number of I/O Devices.

Figure 15 Results with $F'_i=1.1$, $CV_i=0$ for Matrix. Mult. Programs

5.3 Multiprogrammed System

The next step in evaluating the performance of the I/O model was to investigate how these models perform in a multiprogrammed environment. To achieve this, more than one job was present in the system at all times. This was done by starting the desired number of applications in the parallel virtual machine, and having all these applications restart a statistically identical copy of themselves right before terminating. This ensured that the total number of applications running on the parallel virtual machine was constant at all times. Thus the average response time metric is to be measured over a larger period of time (i.e. a large number of measurements) in comparison to a uniprogrammed system in order to achieve good accuracy.

Note that in the multiprogrammed case, there are M (degree of multiprogramming) rather than one dispatcher threads. Each such dispatcher manages its associated compute threads and is responsible for measuring the response time for that particular application. This implementation was chosen since it is easier to keep track of which compute thread belongs to which application. Furthermore, it is also easy to maintain the desired degree of multiprogramming, since each dispatcher can start a copy of itself right before terminating.

5.3.1 Multiple Parallel I/O Jobs

The first experiment performed in this environment was to observe the effects of having multiple parallel I/O jobs (synthetic programs) running at the same time. As mentioned earlier, the fixed parameters presented in Section 5.1 are valid. The interest here was to investigate which I/O model performs better for different values of M (the degree of mutliprograming). The results for the synthetic programs are shown in Figure 16 and Figure 17. Only the "ratio" metric is shown here since the average response time metric would have little meaning when comparing results for different values of M . Appendix B contains more results for the multiple parallel I/O jobs.

Figure 16 Performance of O1 and O2 in a Multiprog. System. $F_1=1$, $CV_1=0$, $N_1=3$

Figure 17 Performance of O1 and O2 in a Multiprog. System. $F_1=2$, $CV_1=0$, $N_1=3$

The results for $N_i = 3$ were chosen since they best show the effects we are interested in. Appendix B contains results for other values of N_i ranging from 1 to 6.

As can be seen from Figure 16 and Figure 17, an interesting effect appears once the degree of mutliprograming is varied. The performance of O1 starts off by being worse than the performance of O2 for $M=1$, but as M is increased, O1 starts to perform better then O2. This “swing” in relative performance is quite large and thus suggests that there would be a benefit in performance if an application was able to dynamically switch between doing its I/O in O1 or O2 manner, depending on the load on the system.

Two factors are important in determining the relative performance of O1 relative to O2: the waiting time experienced by the I/O requests and the CPU-I/O overlap. O2 exhibits better CPU-I/O overlap. The waiting time experienced by an I/O request has two components: waiting time for an I/O device to become free and waiting time due to the network. Note that the contribution of the network to the delay experienced by the I/O request is very small because only short messages (a few bytes in length) are exchanged in the synthetic program. Measurement of queueing delays at the dispatcher have shown that the I/O requests of the O2 model spend more time waiting to be serviced than the I/O requests of the O1 model. On an average, an O2 request waits 10% longer than an O1 request when $M=1$. At $M=1$ the effect of CPU-I/O overlap dominates and O2 performs better than O1. At $M=5$ this additional waiting time for an O2 request is observed to increase to 90% and O2 performs worse. A similar result is obtained when the coefficient of variation is set to 0.5, and the measurement repeated. The results are shown in Figure 18 and Figure 19.

The relative performance for this case (as seen from Figure 18 and Figure 19) is not very different from the $CV_i=0$ case. From the above tests, we can conclude that it is better to use the O2 model in a uniprogrammed environment, while O1 is better suited for a multiprogrammed system. Similar conclusions are derived by P. Kwong [Kwon96] based on a simulation based investigation.

Figure 18 Performance of O1 and O2 in a Multiprog. System. $F_1=1$, $CV_1=0.5$, $N_1=3$

Figure 19 Performance of O1 and O2 in a Multiprog. System. $F_1=2$, $CV_1=0.5$, $N_1=3$

Next, the same experiment was performed with the matrix multiplication programs. The results are shown below (Figure 20). Here again, the same effect is present, however O1 starts performing better than O2 for values of M smaller than in the previous case. Moreover, for large degree of multiprogramming, there is a significant performance gain in doing the I/O in a O1 fashion. As can be seen from the graph, a swing in relative performance of about 40% exists between the system with M=1 and the system with M=5.

The difference between the performance of the matrix multiplication programs and the synthetic programs may be attributed to the fact that in the matrix multiplication case the threads wait a long time for the Ethernet to become free in order to be able to transmit messages. Every such wait contributes to a significant increase in response time. Therefore, for the matrix multiplication programs the swing in performance will be larger, since for every message sent we incur a larger performance hit than in the synthetic case. Furthermore, the probability of waiting for the Ethernet to become free is larger due to the larger message size.

Figure 20 Performance of the Matrix Multiplication Programs in a Multiprogrammed System ($F_i' = 1.1$, $CV_i = 0.5$, $N_i = 3$).

In this case, the possibility of performance improvement by dynamically switching between O1 and O2 is even larger than in the case of synthetic programs, since the swing in relative performance is larger.

5.3.2 Single Parallel I/O Job, and Multiple Sequential Jobs

Since one of the goals of this thesis was to investigate how these large parallel I/O applications perform in a “real” environment, the next experiment performed was to measure the performance of the different models in a system where we have one large parallel I/O job using all the nodes, and a number of sequential jobs each running on a given node. These sequential jobs require only a few seconds to complete, however they start a statistically identical copy of themselves before exiting. The sequential jobs are jobs that perform some computations and then perform some I/O. These I/O operations use the network file system, and thus they do not use the local disk of that node. This was done because the experiment is intended to simulate the effect of having different users doing their text processing and/or other small jobs on the nodes of the virtual machine, while a large I/O application is also running. Only the relative performance of the different I/O models for the parallel applications is the focus of attention.

In the following discussion the number of sequential jobs running on the system is called “System Load” The sequential jobs are distributed evenly among the workstations that are running compute threads. Also, for the results presented in this section, the number of O2 slices was set to five.

Figure 21 Performance Under Different System Loads ($F_i=1$, $CV_i=0$)

Figure 22 Performance Under Different System Loads ($F_i=2$, $CV_i=0$)

We see From Figure 21 and Figure 22 that the impact on system performance due to other sequential jobs running on the system produces a greater increase in O1 response time. Moreover, the response time of the O2 model increases only by a small factor even for relatively large system loads. To explain this, we have to consider both the compute threads and the sequential jobs in the system which share some of the processors. On these processors, the UNIX scheduler allocates time slices to each process, and at the end of the time slice it performs a context switch. Every time a context switch is performed, we lose the processor and thus the response time increases. In the O2 case, the compute bursts are much shorter than in the O1 case, and therefore O2 will experience a fewer number of preemptions by the scheduler. If the CPU burst of the O2 model is smaller than the time slice allocated to it, it may be possible for the O2 compute thread to never experience a preemption by the scheduler. This explains why the increase in O2 response time is smaller than the increase in O1 response time. We can also conclude that for such a system it is beneficial to run the O2 model with a higher value of S , since this will result in a smaller number of preemptions at higher system loads.

The experimental results from Figure 21 and Figure 22 show that for eight sequential jobs (system load of 8), and one parallel job with the O1 model running in the system the increase in response time is a factor of 1.9 for an F_i value of one, and 1.5 for an F_i value of two. For smaller system loads this increase is even lower. In the case of the O2 model, we see that the increase in response time with system load is minimal. One explanation for these effects follows. The processors running the compute threads of the large parallel I/O application are mostly idle due to the fact that the compute threads of the parallel I/O job are blocked while waiting for the I/O threads to finish reading and/or writing. Thus, another process on such a processor can time share the processor with the compute thread without a large negative impact on performance.

In conclusion, we can say that for parallel I/O applications with large values of F_i the compute nodes are idle for a large fraction of time. Thus, running small sequential jobs on these nodes has a minimal impact of the performance of a large parallel I/O application. Only for very large system loads the response time of the parallel I/O application starts to increase drastically. Fur-

thermore, we can also conclude that the O1 model is more sensitive than the O2 model due to the fact that its CPU bursts are larger.

5.3.3 Value of Multiprogramming

The value of multiprogramming is a metric that enables us to measure if it is more efficient to run a number of parallel applications in a sequential way or in a multiprogrammed way. Furthermore, this metric also points out what is the best value of the multiprogramming parameter M . The exact definition is:

$$\text{ValueOfMultiprogramming} = V(M) = \frac{\text{AverageResponseTimeMulti}(M)}{\text{AverageResponseTimeUni}(M)}$$

where the $\text{AverageResponseTimeMulti}(M)$ is the average response time of an application running on a system with a degree of multiprogramming M , and the $\text{AverageResponseTimeUni}(M)$ is the average response time of M applications running one after another on a system with only one application running to completion on the system. Clearly, the $\text{AverageResponseTimeMulti}(M)$ is just the average response time measured on a system with the degree of multiprogramming equal to M .

a) Homogeneous Multiprogramming Mix.

The multiprogrammed mixes considered so far were homogenous, i.e. they consisted of statistically identical applications running in parallel. For such a multiprogrammed mix, the $\text{AverageResponseTimeUni}(M)$ can be computed from the average completion time of an application :

$$\text{AverageResponseTimeUni}(M) = \frac{\text{AverageTimeOfCompletion}(1) (1 + 2 + \dots + M)}{M}$$

Therefore, the Value of Multiprogramming is:

$$V(M) = \frac{\text{AverageTimeOfCompletion}(M)}{\text{AverageTimeOfCompletion}(1)} \cdot \frac{M}{1 + 2 + \dots + M}$$

With the available data, V as a function of M was computed for a number of cases using the above formula. The table below summarizes some of the results.

Table 5: Value of Multiprogramming for Different Parameters

M	1	3	5	10
Synthetic with $F_i=1$ and $CV_i=0$ (O1)	1	1.32	1.48	1.67
Synthetic with $F_i=1$ and $CV_i=0$ (O2)	1	1.45	1.63	1.81
Synthetic with $F_i=2$ and $CV_i=0$ (O1)	1	1.60	1.48	1.57
Synthetic with $F_i=2$ and $CV_i=0$ (O2)	1	1.52	1.72	1.92
Matrix Mult with $F'_i=1.1$ (O1)	1	1.12	1.15	N/A
Matrix Mult with $F'_i=1.1$ (O2)	1	1.57	1.75	N/A

The results for the synthetic programs with $F_i=1$ and for the matrix multiplication are presented in Figure 23 and Figure 24. From these figures we can see that the best value of M for all cases is one. This implies that there is some performance loss if we run the parallel I/O applications in a multiprogrammed environment, rather than one after another. This performance loss is attributed to the fact that the extra overhead associated with having more programs running at the same time is large, and impacts negatively on performance.

Figure 23 Value of Multiprogramming vs. Degree of Multiprogramming for $F_1=1$ (Synthetic)

Figure 24 Value of Multiprogramming vs. Degree of Multiprogramming for $F_1=1$ (Matrix Mult.)

Moreover, a multiprogrammed environment is beneficial only when the nodes of the uniprogrammed system are idle, as was the case in section 5.3.1. In this case, the compute nodes are still mostly idle like before, but the I/O nodes are utilized heavily since all the applications are performing extensive I/O. Thus, running more programs on this system is detrimental to performance.

We also note that the O2 model has a larger V value compared to the O1 model. This is to be expected from the results in section 5.3.1 where we found that O2 performs worse than O1 in a multiprogrammed environment.

In conclusion, we can say that there is a loss of performance for a homogenous multiprogramming mix if the parallel I/O applications are run in a multiprogrammed environment.

Heterogeneous Multiprogramming Mix.

The next experiment undertaken was to find out if there is some gain in running parallel I/O applications on a system with a heterogenous multiprogramming mix. Clearly, a vast number of combinations exist in this case. A preliminary investigation based on one heterogenous multiprogrammed mix is considered. The degree of multiprogramming is four. The applications are chosen such that variability exists in a number of application characteristics that are presented next.

Application 1: An O2 model with $F_i = 2$, $n_i = 6$ and $CV_i = 0.5$

Application 2: An O1 model with $F_i = 1$, $n_i = 6$ and $CV_i = 0$

Application 3: An O2 model with $F_i = 1$, $n_i = 3$ and $CV_i = 0$

Application 4: An O1 model with $F_i = 2$, $n_i = 3$ and $CV_i = 0.5$

All the other parameters are common to all applications and they are:

$$N_i = N_d = n_d = 6$$

$$CV_d = 0$$

For this heterogeneous set of applications the $AverageResponseTimeUni(M)$ is given by:

$$AverageResponseTimeUni(M) = \frac{T1 + T2 + T3 + T4}{M}$$

where $T1$ is the response time of the first application to complete and so on. Clearly, this value depends on the order in which the applications are run. To get a good estimate of this value, the four programs were run in a shortest to longest order and a longest to shortest order. The result for the first order is $V(4)=0.86$ while a $V(4)$ of 0.94 was found for the second case. The mean of both these cases is therefore given by a $V(4)$ of 0.9. Thus, in a heterogenous multiprogramming mix there is some advantage in running the applications in parallel. Clearly more experiments are required in this area.

5.4 Discussion

This section is intended to provide a summary of the major conclusions found in this chapter:

- O2 performs better in most uniprogrammed environments.
- O1 performs better in most multiprogrammed environments with many similar parallel I/O jobs.
- In most multiprogrammed environments with one parallel I/O job and many sequential jobs O2 performs better.
- The value of multiprogramming is observed to be greater than 1 for a homogenous multiprogrammed mix and can be less than one for a heterogeneous multiprogrammed mix.

6.0 Performance of Data Replication Strategies

6.1 Introduction

For all measurements in Chapter 5, all the data was replicated on all the devices used. Since the parallel applications deal with large amounts of data, full replication is costly in terms of disk space. For example, an application that requires 1Gb of data in a full replicated system will use $N_i \times 1\text{Gb}$ of disk space, since all data is replicated on all devices. A way to reduce the amount of disk space required, is to reduce data replication. This, in turn, may have a negative impact on performance since reducing data replication reduces the I/O parallelism. This is because operations that were performed in parallel for full replication case, must now be done sequentially since the data they access is located on the same disk. Therefore, there is a trade-off between data replication and performance. An investigation of this trade-off is presented in this section. We are concerned with the determination of a reasonable degree of replication such that the negative impact on performance is not significant.

As seen from the previous sections, a number of parameters affect the performance of the I/O models described. Many combinations are therefore possible for making measurements. However, due to time constraints, experiments were performed for two sets of parameters only. The parameters used are listed in Table 6 below. The constant parameters listed in Section 5 apply for this entire chapter as well.

Table 6: Data Replication Parameters

Parameter Set 1	Parameter Set 2
$F_i = 1$	$F_i = 2$
$n_i = 6$	$n_i = 6$
$M = 3$	$M = 3$
$C_v = 0$	$C_v = 0$
$S = 5$	$S = 5$

Both the O1 and O2 models were investigated using these parameters.

These same sets of parameters are used over and over for many of the experiments performed. In this way, a comparison of these results with the full replication results from the previous chapter is easier to perform.

6.2 Data Replication Strategies

Some work on data replication was done by P. Kwong and is presented in [KwoM96]. In Kwong's work the replication strategies are evaluated in a simulation environment. In this thesis two of the strategies introduced in the mentioned paper, namely Replicated-C and FixedR-C are evaluated. Another strategy (ReplicatedL-C) that limits the degree of replication is introduced here and evaluated under the above mentioned set of parameters.

A description of each of the three strategies is presented.

6.2.1 Full Replication (Replicated-C)

In the Replicated-C strategy the input data read by the programs is replicated on all disks. Thus, a read operation can be performed on any disk. However, only the data read is replicated. The data that is written, can be written anywhere, but it is not replicated. This lack of write data replication would be appropriate for scientific applications where the outputs are written to the I/O devices by one thread and are not needed by other threads for further processing. For example, this is clearly the case for the matrix multiplication program presented in this thesis. Once a thread has completed the computation of some of the entries in the matrix, they are not needed by other threads for further computations. This thesis is concerned only with the parallel performance of such applications in which the data written by any thread is not needed by another. For other applications such as databases writing of the same data to multiple disks may be required in order to maintain data consistency.

This strategy is implemented as follows. All I/O requests are inserted into a central FIFO queue. A central router which knows the busy/idle status of all the disks in the system, sends the I/O request to the first idle I/O device it finds. If no device is idle, then the requests remain in the queue until a device becomes idle.

From the above, we can see that the Replicate-C strategy implements full replication for the read data and both read and write service requests can be serviced by any disk at any time.

The C suffix in the Replicated-C stands for “Centralized” and refers to the fact that there is only a central FIFO where all I/O requests queue. In the tests shown here the FIFO and the router are implemented by the dispatcher thread which was described earlier.

Note that all the results presented in Chapter 5 use the Replicated-C strategy.

6.2.2 Limited Replication (ReplicatedL-C)

This strategy is similar to the Replicated-C strategy except for the fact that the read data is not replicated on all disks, but on a subset of the available disks. Therefore, in this case, when an I/O request is issued it is queued as before in a central FIFO queue. The router then takes requests from this queue and if the request is a write request, it sends it to a free I/O device. If there is no free I/O device, it waits until one becomes free. If the request is a read request, then the router sends this request to a free disk from the subset of disks that store the required data locally. If no such device is free, the request is sent to one of this devices where it is queued. The selection of the device to receive this new request is done according to the length of the queue at each device. The one with the shortest queue gets the new request. If there is a tie, the new requests are distributed in a random fashion.

The L suffix in the ReplicatedL-C stands for “Limited”. In the experiments described in this thesis each piece of data is replicated on two disks which corresponds to the minimum degree of replication. This strategy will be compared with Replicated-C.

6.2.3 No Replication (FixedR-C)

The FixerR-C strategy is a subset of ReplicatedL-C. Here, there is no replication for the read data, thus there is only one disk that can service a particular read request. This simplifies the implementation of the dispatcher since it is no longer necessary to check the length of queues at the I/O nodes since there is only one I/O node that can service a read request. Such a read request is therefore sent to its corresponding node where it queues if the node is busy, or begins execution if the node is idle. The write requests are handled as in the ReplicatedL-C case.

6.3 Comparison of the Data Replication Strategies

The results obtained for the O1 and O2 models with different data replication strategies are presented in Figure 25 and Figure 26. From these graphs, we see that Replicated-C performs the best, while FixedR-C performs the worst. This is to be expected, since the I/O parallelism is decreased as replication is decreased. However, we see that for the cases considered, the performance of the ReplicatedL-C strategy compares well to the performance of the Replicated-C strategy. This is due to the fact that for $M=3$ (the degree of multiprogramming at which the test was run) the number of I/O requests for a particular disk is not very large.

As an example, for the O1 model with a multiprogramming degree of three, there can be a maximum of only six requests for a disk. Clearly, two I/O devices can handle these requests with relative ease, and thus there is little gain of having six disks (full replication) able to service the requests. On the other hand, if the degree of multiprogramming increases, we expect the ReplicatedL-C strategy to perform worse and worse relative to Replicated-C, since in such a case the two I/O devices are no longer able to adequately service the increasing number of requests. From Figure 25 and Figure 26 we can see that, indeed, this effect is present for a large degree of multiprogramming.

The data also indicates that it is possible to reduce the amount of replication without incurring a large penalty in performance, as long as the degree of multiprogramming is not very large. More experiments with other sets of parameter values are required to confirm this observation. Also this presents an opportunity for future research in devising other data distribution strategies.

a) Data Replication Strategy for O1

b) Data Replication Strategy for O2

Figure 25 Performance of Data Replication Strategies for Parameter Set 1

a) Data Replication Strategy for O1

b) Data Replication Strategy for O2

Figure 26 Performance of Data Replication Strategies for Parameter Set 2

7.0 Conclusions

7.1 Summary of Investigations

In this research a number of issues regarding the performance of three parallel I/O models and data replication strategies were investigated. A major part of the research was based on experiments made on a cluster of workstations running under PVM (parallel virtual machine). Both uniprogrammed and multiprogrammed environments were considered. For the multiprogrammed case different types of the multiprogramming mix were investigated as well.

First of all, a set of upper and lower bounds on the response time and speed-up of a parallel I/O application were derived in Chapter 3. This research extends the work described in [Eage89] to include I/O operations in parallel programs.

A large part of the investigation reported in this thesis is based on measurements of the performance of programs with parallel I/O running under PVM in a network of workstations. In order to make these measurements, C programs that implement the I/O models under consideration were implemented. Both synthetic and real applications were experimented with. These programs are implemented in such a way that the I/O factor, the coefficient of variation in the I/O demand, the number of devices used (both I/O and compute devices) and the scheduling policy can be easily varied.

Three different I/O models, NOP, O1 and O2 were investigated. The performance of these models was evaluated with respect to a number of parameters. The effects of the I/O factor, the coefficient of variation in the I/O demand, the number of I/O devices and the number of slices used in the O2 model on the performance of the models were investigated for a uniprogrammed environment. The performance of the I/O models were also analyzed for two types of multiprogrammed environments: one where there are sequential jobs running concurrently with one parallel application, and one where there are multiple parallel applications present in the

system.

The last part of the thesis focuses on the effect of data replication on performance. Three different data replication strategies were considered, and the results of the investigation are described in Chapter 6.

7.2 Discussion of Results

Out of the three I/O models considered, the NOP model performed the worst for every experiment performed. The performance of the other two models, O1 and O2 however, was found to depend on the environment in which the application runs. The research therefore focussed on evaluating which of these two I/O models (O1 or O2) performs best under various conditions.

For a uniprogrammed environment it was found that O2 performs better than O1 for all values of the I/O factor. This was attributed to the fact that the O2 model has more CPU - I/O overlap relative to the O1 model. We have experimented with a coefficient of variation in the I/O demand of 0 and 0.5. It was also found that at these values, the coefficient of variation in the I/O demand did not affect the relative performance of the I/O models.

For a multiprogrammed environment, where many I/O parallel applications are present in the multiprogramming mix, it was found that for small values of the degree of multiprogramming the O2 model performed better than the O1 model. On the other hand, for medium and large values of M , O1 is a better choice. This is due to the fact that at large degree of multiprogramming, the I/O requests of the O2 model experience more queueing delay at the dispatcher in comparison to the I/O of the O1 model. For this case it was also found that changing the I/O factor from 0.5 to 1, 2 and 4 did not affect the relative performance of the two models.

However, in a multiprogrammed environment, where one large parallel I/O application coexists with many sequential jobs running on the nodes of the virtual machine it was found that O2 performs better than O1 for all the values of M that were experimented with. This was attributed to the fact that the O2 model has smaller compute bursts than the O1 model and as such, it is less likely for its compute threads to be preempted by the scheduler. Furthermore, it was found that for such a system, the loss in performance for the large parallel job due to these sequential jobs was relatively small even for a large number of sequential jobs. This is because for parallel I/O applications with large I/O factors, the compute nodes are idle for a large fraction of time, and thus other applications can be run on these processors without incurring a severe performance penalty.

The above two paragraphs make a case for creating applications that can dynamically change between the I/O models used depending on the load in the system and the type of applications in the multiprogramming mix.

For the multiprogrammed environment the “value of multiprogramming” for an I/O application was also determined. It was found that when considering similar parallel I/O applications in a multiprogrammed ($M > 1$) environment (homogenous multiprogramming mix) there is loss of performance of these applications relative to a system where the jobs are running in sequence ($M = 1$). However, this performance loss is not large, and therefore multiple users could run similar parallel I/O application at the same time without incurring a large performance hit. Preliminary investigations indicate that with a heterogenous set of parallel I/O applications, performance gain can result by multiprogramming the system ($M > 1$). More investigations are required in this direction.

Also, for the multiprogrammed environment, the performance of three different data replication strategies was investigated. It was found that although the full replication strategy performs best, one can reduce the amount of replication drastically without significant loss of performance.

Throughout the thesis it was found that although having variability in the I/O request size impacts negatively on the performance, it does not affect the relative performance of the I/O models.

7.3 Contributions

The main contributions of this thesis are:

- Based on implementation and measurements, it presents a thorough investigation of different styles that can be employed by a parallel application for performing I/O.
- It makes a strong case for designing applications capable of dynamically switching between the O1 and O2 models according to system parameters.
- It demonstrates that even a limited degree of data replication can result in a significant improvement in performance.
- It establishes that PVM can be used to build a parallel machine on which parallel I/O applications can run with significant performance gain in comparison to sequential systems.
- It provides a set of theoretical upper and lower bounds on the performance of parallel I/O applications.
- It provides insights into system behavior and performance of the systems using parallel I/O.

7.4 Future Research

An important area for future research is to expand the data replication work presented here. In this thesis, the evaluation of replication strategies was a secondary goal and as such, only a few

measurements were performed to evaluate them. Measurements at higher I/O factors and the devising of different strategies may give a better understanding of the effects of data replication on performance.

I/O scheduling can have a significant impact on performance and investigation of different types of scheduling policies represents another area of future research. Some simulation results exist in this area, however there are no real measurements of the different scheduling policies for multiprogrammed systems.

Finally, research on the adaptive I/O strategy suggested in Section 5.3.1 would be of interest.

References

- [Burg96] D. Burger, J. R. Goodman, A. Kagi, "Memory Bandwidth Limitations of Future Microprocessors", 23rd Annual International Symposium on Computer Architecture, May 22-24 1996, Philadelphia, pp 78-89.
- [Chen93] S. Chen, D. Townsley, "The Design and Evaluation of RAID 5 and Parity Striping Disk Array Architectures", Journal of Parallel & Distributed Computing, vol 17, 1993, pp. 58-74.
- [Chen95] S. Chen, D. Townsley, "A Performance Evaluation of RAID Architectures", IEEE Transactions on Computers, 1996, Oct, Vol. 45, No. 10, pp 1116-1130.
- [Chou93] A. Choudhary, "Parallel I/O Systems", Journal of Parallel and Distributed Computing, vol. 17, pp 1-3, 1993.
- [Coff73] E.G. Coffman, P. J. Denning, *Operating Systems Theory*, Prentice Hall, 1973.
- [Corb93] P. F. Corbett, D. G. Feitelson, J.-P. Prost, S. J. Baylor, "Parallel Access to Files in the Vesta File System", Proc. Supercomputing SIGARCH 1993, pp. 472-481.
- [Cyph93] R. Cypher, A. Ho, S. Konstantinidou, P. Messina, "Architectural Requirements for Parallel Scientific Applications with Explicit Communication", Proceedings of the 20th Annual International Symp. on Computer Architecture '93, 1993, pp. 462-471.
- [Eage89] D. Eager, J. Zahorjan, E. Lazowska, "Speedup Versus Efficiency in Parallel Systems", IEEE Transactions on Computers, Vol 38, No 3, pp. 408-422, March 1989.

-
- [Free96] C. Freedman, J. Burger, D. DeWitt, "SPIFFI-A Scalable Parallel File System for the Intel Paragon", *IEEE Transactions on Parallel and Distributed Systems*, Vol 7, No 11, pp. 1185-1197, November 1996.
- [Ghan97] S. Ghandeharizadeh, "MITRA-A Scalable Continuous Media Server", *Multimedia Tools and Applications*, 1997, Jul, Vol. 5, No. 1, pp. 79-108.
- [Gibs91] G. A. Gibson, *Redundant Disk Arrays; Reliable, Parallel Secondary Storage*, The MIT Press, Cambridge Massachusetts, 1991.
- [Hugh91] Hugh Sierra, *An Introduction to Direct Access Storage Devices*, Academic Press Inc., San Diego, 1990.
- [Kotz94] D. Kotz, N. Nieuwejaar, "Dynamic File Access Characteristics of a Production Parallel Scientific Workload", *Proc. of Supercomputing 1994*, November 1994, pp. 640-649.
- [Kotz95] D. Kotz, N. Nieuwejaar, et al. "File-Access Characterization of Parallel Scientific Workloads", Tech. Report PCS-TR95-263, Dept. of Math. and Comp. Science, Dartmouth College, Hanover, USA.
- [Kotz95b] D. Kotz, N. Nieuwejaar, "Disk-directed I/O for an Out-of-core Computation", Tech. Report PCS-TR95-251, Dept. of Math. and Comp. Science, Dartmouth College, Hanover, USA.
- [Kotz97] D. Kotz, "Disk-directed I/O for MIMD Multiprocessors", *ACM Transactions on Computer Systems*, February 1997, Vol. 15, Number 1, pp. 41-74. Available at URL <ftp://ftp.cs.dartmouth.edu/pub/CS-papers/Kotz/kotz:jdiskdir.ps.Z>

-
- [Krie96] O. Krieger and M. Stumm, "HFS: a Performance Oriented Flexible File System Based on Building-Block Composition", Proc. 4th Workshop I/O in Parallel and Distributed Systems, Philadelphia, May 1996, pp. 95-108.
- [KwoM96] P. Kwong, S. Majumdar, "Study of Data Distribution Strategies for Parallel I/O Management", Proc. Third International Conf. of The Austrian Committee on Parallel Computing (published as Lecture Notes in Computer Science by Springer Verlag) September 1996, pp. 12-23.
- [Kwon96] P. Kwong, "Management of I/O in Multiprogrammed Parallel Systems", M. Eng Thesis, Carleton University, Department. of Systems and Computer Engineering, 1996, Ottawa, Canada.
- [Kwon96b] P. Kwong, S. Majumdar, "Scheduling of I/O in Multiprogrammed Parallel Systems", Tech. Report SCE-96-12, Dept. of Systems and Comp. Engineering, Carleton University, Ottawa, Canada.
- [KwTG97] T. G. Kwong, "Disk Placement for Arbitrary-Rate Playback in an Interactive Video Server", Multimedia Systems, 1997, Jul., Vol. 5, No. 4, pp. 271-281.
- [Maju91] S. Majumdar, D. Eager, R. Bunt, "Characterization of programs for scheduling in multiprogrammed parallel systems", Performance Evaluation, Vol 13, 1991, pp. 109-130.
- [Maju94] S. Majumdar, Y.M. Leung, "Characterization of Programs for Scheduling in Multiprogrammed Parallel Systems", Proc. Sixth IEEE Symp. on Parallel and Distributed Processing, Dallas, 1994, pp. 298-307.

-
- [More97] S. More, A. Choudhary, I. Foster, M. Xu, "MTIO A Multi-Threaded Parallel I/O System", Tech. Report, Dept. of Electrical and Computer Engineering, Northwestern Univ, 1997.
- [Nitz91] B. Nitzberg, V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms", IEEE Computers, August 1991, pp. 52-60
- [Patt88] D. A. Patterson, G. Gibson, R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", Proceedings of the ACM SIGMOD Conference, June 1988, pp. 109-116.
- [Patt94] Y.N. Patt, "The I/O Subsystem: a Candidate for Improvement", IEEE Computer, March 1994, pp. 15-16.
- [Redd89] A.L.N. Reddy, P. Banerjee, "An Evaluation of Multiple-Disk I/O Systems", IEEE Trans. on Computers, Vol. 38, No. 12, December 1989, pp. 1680-1690.
- [Ryan93] James Ryan, Sisira Weeratunga, "Parallel Computation of a 3-D Navier-Stokes Flowfields for Supersonic Vehicles", NASA Ames Research Center, Moffett Field, California, 1993. Available at
URL <ftp://ftp.cs.dartmouth.edu/pub/pario/examples/CFS3D.INFO>
- [Ross94] J. M. del Rosario, A. N. Choudhary, "High-Performance I/O for Massively Parallel Computers" Proc. of the IEEE Symposium on Mass Storage Systems, June 1994, Annecy, France, pp59 - 68.
- [Sevc94] K. Sevcik, "Application Scheduling and Processor Allocation in Multiprogrammed Parallel Systems", Performance Evaluation, March 1994, Vol. 19, No. 2, pp. 107-140.

-
- [Smir97] E. Smirni, D. A. Reed, "Workload Characterization of Input/Output Intensive Parallel Applications", Lecture Notes in Computer Science 1245, Computer Performance Evaluation, Modeling Techniques and Tools, Proc. of the 9th International conference, St. Malo, France, June 1997, pp. 169-180.
- [Stum90] M. Stumm, S. Zhou, "Algorithms Implementing Distributed Shared Memory", IEEE Computer, May 1990, pp. 54-62
- [Vita95] V. Catania, A. Puliafito, S. Riccobene, L. Vita, "Design and Performance Analysis of a Disk Array System", IEEE Transactions on Computers, Vol. 44, No. 10, October 1995, pp. 1236-1245.
- [Wood90] C.M. Woodside and G.G. Monforton, "Fast Allocation of Processes in Distributed and Parallel Systems", IEEE Trans. on Parallel and Distributed Systems, Vol. 4, No. 2, pp. 164-174, Feb. 1993.M.
- [PVM] A. Geist. A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderan, *PVM: Parallel Virtual Machine*, The MIT Press, Cambridge, Massachusetts, 1995.
- [Stat] J. Pitman, *Probability*, Springer-Verlag, New York, 1993.
- [Mat] S. Fisher, S. Reges, *Pascal and Beyond...Data Abstraction and Data Structures Using Turbo Pascal*, John Wiley & Sons, Toronto, 1992.

Appendix A: Pseudo-Code for I/O Model Implementations.

A.1 Pseudo-Code for NOP Threads

Dispatcher Thread:

Start

Initialize virtual machine

Start compute and writer threads on the different nodes of the virtual machine

Mark all I/O nodes as free

Send to the compute threads their thread id (used to distinguish between compute threads)

Record start time

Signal compute threads to begin execution

do

 wait for a message

 if message is a read request {

 if there are free I/O nodes {

 send request to free I/O node

 mark I/O node as busy

 }

 else wait until one I/O node becomes free {

 send request to free I/O node

 mark I/O node as busy

 }

 }

 if message is an acknowledgment from disk mark disk as free

until all read requests have been serviced.

Signal compute threads to begin computation

Wait for acknowledgment from all compute threads.

Signal all compute threads to begin writing

do

 wait for a message

 if message is a write request {

 if there are free I/O nodes {

 send request to free I/O node

 mark I/O node as busy

 }

```

        else wait until one I/O node becomes free {
            send request to free I/O node
            mark I/O node as busy
        }
    }
    if message is acknowledgment from disk mark disk as free
until all write requests have been serviced

```

```

Wait for acknowledgment from all compute threads
Record end time
Compute response time.
Kill all children
Kill self.

```

Writer Thread:

```

Start.
Write to local disk a large file.
Get the parent (dispatcher) process id (pid)
forever do {
    wait for a message from parent
    unpack message and extract request type and pid of the thread who initiated message
    generate a uniformly distributed random number

    if a  $CV_i$  greater than zero was selected {
        use the generated random number to compute the amount of data to transfer
    }
    else use fixed amount of data to transfer

    if request type is a read {
        read computed amount of data from the large file.
        send an acknowledgment to the thread who initiated the request.
    }
    else {
        write computed amount of data from the large file.
        send an acknowledgment to the thread who initiated the request.
    }
} end of forever loop

```

Compute Thread:

Start.
 Get parent (dispatcher) pid and self pid.
 Wait for message from parent
 Unpack thread number form message.
 Issue a read request to parent.
 Wait for acknowledgment from I/O subsystem
 Wait for message from parent.
 Begin computation
 Finish computation and send message to parent.
 Wait for message from parent
 Issue write request.
 Wait for acknowledgment from I/O subsystem
 Send message to parent
 Kill self

A.2 Pseudo-Code for O1 ThreadsDispatcher Thread:

Start
 Initialize virtual machine
 Start compute and writer threads on the different nodes of the virtual machine
 Mark all I/O nodes as free
 Send to the compute threads their thread id
 Start the clock
 Signal compute threads to begin execution

```

do
  wait for a message
  if message is an I/O request {
    if there are free I/O nodes {
      send request to free I/O node
      mark I/O node as busy
    }
    else wait until one I/O node becomes free {
      send request to free I/O node
      mark I/O node as busy
    }
  }
}

```

if message is an acknowledgment from disk mark disk as free
until all I/O requests have been serviced.

wait for acknowledgment from all compute threads
Record stop time
Compute response time.
Kill all children
Kill self.

Writer thread:

Same as for the NOP case

Compute thread:

Start.
Get parent (dispatcher) pid and self pid.
Wait for message from parent
Unpack thread number from message.
Issue a read request to parent.
Wait for acknowledgment from I/O subsystem
Begin computation
Finish computation
Issue write request.
Wait for acknowledgment from I/O subsystem
Send message to parent
Kill self

A.3 Pseudo-Code for O2 Threads

Dispatcher thread:

Same as for the O1 case

Writer thread:

Same as for the NOP case

Compute thread:

Start.

Get parent pid and self pid.

Wait for message from parent

Unpack thread number form message.

do S times:

 Issue a read request to parent.

 Wait for acknowledgment from I/O subsystem

 Begin computation

 Finish computation

 Issue write request.

 Wait for acknowledgment from I/O subsystem

end do loop

Send message to parent

Kill self

A.4 Matrix Multiplication O1 and O2

The pseudo-code for these programs is identical to their synthetic counter parts except for the fact that in this case the data is packed (therefore sent) along with the message. Furthermore the dummy computations in the compute thread are replaced by the following matrix multiplication code which is discussed in [Mat].

```
for all new elements to be computed (given by the indices j and k ) {
    for i=1 to number of elements in a row
        new_element[j][k] = element[i][j] * element[k][j]
    }
```

where `element` is an element of the matrix to be multiplied and `new_element` is an element of the resulting matrix.

Appendix B: Results of Synthetic Programs

For this appendix the constant parameters stated in Section 5.1 are valid.

B.1 Comparison of I/O Models - Uniprogrammed Case

The following are all the results obtained for the comparison of the I/O models in an uniprogrammed environment (Synthetic applications).

Figure 27 Graph of NOP, O1 and O2 for $F_1=0.5$ and $CV_1=0$

Figure 28 Graph of NOP, O1 and O2 for $F_1=0.5$ and $CV_1=0.5$

Figure 29 Graph of NOP,O1 and O2 for $F_1=4$ and $CV_1=0$

B.2 Effect of S on O2 Performance.

The following results present the effect of the number of slices on the performance of the O2 I/O model.

Figure 30 Graph of O2 with different number of slices for $F_1=1$ and $CV_1=0$

Figure 31 Graph of O2 with different number of slices for $F_1=2$ and $CV_1=0$

B.3 Comparison of I/O Models - Multiprogrammed Case

The following are the results obtained in a multiprogrammed environment.

Figure 32 Histograms of O1 and O2 for the Multiprogrammed Case for $F_i=1$
 $CV_i=0$

Figure 33 Histograms of O1 and O2 for the Multiprogrammed Case for $F_i=1$ $CV_i=0.5$

Figure 34 Histograms of O1 and O2 for the Multiprogrammed Case for $F_i=2$
 $CV_i=0$

Figure 35 Histograms of O1 and O2 for the Multiprogrammed Case for $F_i=2$
 $CV_i=0.5$

Appendix C: Results of Matrix Multiplication

C.1 Results for the multiprogrammed case

Figure 36 Matrix Multiplication Program, Multiprogrammed Case. $F_i=1.1$

Appendix D: Generation of CV_i

The purpose of this appendix is to describe how a coefficient of variation in the I/O demand was generated.

Since it was decided that only a CV_i of 0.5 will be used for testing, only a uniform distribution between a and b [a,b] was used in order to get the desired result. This uniform distribution was generated using the drand() function.

Now, for a uniform distribution we have [Stat]:

$$Mean = \frac{a + b}{2}$$

and

$$Variance = \frac{(b - a)^2}{12}$$

The above form a system of two equations with two unknowns a and b. This system can be solved as long as we know the Mean and the Variance. The Mean is obtained from the value of the I/O factor, while the variance is set to the desired values. This therefore determines the numbers a and b, and thus using drand() with an appropriate scaling, a uniform distribution between a and b can be easily generated. The scaling is required since drand() produces a uniform distribution between 0 and 1.

Appendix E: Lower Bound on Speedup

The following derivation is extracted from the paper “Speedup Versus Efficiency in Parallel Systems” by D. Eager, J. Zahorjan and E. Lazowska published in IEEE Transactions on Computers, vol 38, No 3, March 1989.

Theorem 1: Let A denote the average parallelism of a software structure and $S(N)$ denote the speedup with N processors. For any work-conserving scheduling policy we have

$$S(N) \geq \frac{NA}{N + A - 1}$$

Proof: Consider an arbitrary software structure with average parallelism A . Let T_u denote the elapsed time of its execution given an unlimited number of processors. It follows from the definition of average parallelism that the total processor busy time accumulated during such an execution (summed over all N processors) is $T_u A$.

Now, suppose that this software structure is executed using N processors and some arbitrary work-conserving scheduling policy. Since the scheduling policy is work conserving, it follows that the total busy time is as before, $T_u A$. The execution time is then given by $T_u A + I(N)/N$ where $I(N)$ denotes the total idle time that is accumulated during the execution. From the definition of speedup we then have:

$$S(N) = \frac{NA}{A + \frac{I(N)}{T_u}}$$

To establish the desired result we only have to show that

$$I(N) \leq T_u (N - 1)$$

To this end define $w(t)$ to be the portion of the original software structure graph that has not

completed execution at time t . $w(t)$ includes those tasks that have not yet been initiated, and those tasks that have been initiated but not yet completed. The service demand of each task in $w(t)$ is its original service demand diminished by the amount of service (if any) already provided to the task.

Define $L(t)$ to be the length (i.e. the total service demand) of the longest path within $w(t)$. Note that the value of $L(t)$ varies from T_u (at the start of the execution) to zero (at the end of the execution). If, at some time t , $L(t)$ is not decreasing, there must be some task at the head of the longest path in $w(t)$ that is not being executed. Since no precedence constraints prevent the execution of such task, and since the scheduling is work-conserving, it must be the case that there are no idle processors at time t . Thus, processors can only be idle during those periods of time when $L(t)$ is decreasing. Since $L(t)$ decreases for a total length of time T_u , and since at most $N-1$ processors can be idle at any point in time the total idle time $I(N)$ is at most $T_u(N-1)$ which establishes the result.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved