Performance of Parallel I/O Data Distribution Strategies for Space-Sharing Policies

By

Liuhong Wang, B.Eng. (Shanghai)

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

School of Computer Science
Carleton University
Ottawa, Ontario

August 2004

© Copyright
2004, Liuhong Wang B.Eng (Shanghai)

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-97485-5

Our file Notre référence

ISBN: 0-612-97485-5

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives 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.

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 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.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

PAGINATION ERROR.

TEXT COMPLETE.

ERREUR DE PAGINATION.

LE TEXTE EST COMPLET.

Abstract

The I/O subsystem is becoming the performance bottleneck of parallel applications. In order to address this bottleneck, integrated approaches at application, system and architecture levels need to be applied to parallel programs. In this thesis, the modified adaptive processor (MAP) scheduling policy is used to study the performance sensitivity of selected parallel job structures to four I/O models and different types of parallel data access and distribution policies.

Four types of I/O models used in this thesis are the NO, NOP, O1 and O2 modules. These modules are studied in combination with three parallel job structures of Fork-and-Join, Divide-and-Conquer, and GE. Our results show that under most data access and distribution policies, the O1 I/O model provides the best performance for the Fork-and-Join and Divide-and-Conquer parallel job structures, while the NOP I/O model gives the best for the GE job structure.

In terms of the data access and distribution policy types, the central replication technique can provide significant performance improvements. Any other data policies that balance the I/O devices when routing the I/O requests can achieve comparable performance result as the central replication.

Acknowledgments

I would like to especially thank my supervisor, Dr. Sivarama P. Dandamudi, for his research guidance, and personal encouragement throughout this thesis.

I would like to dedicate this thesis to my parents, my husband Dingyi Jin and daughter Laura Jin for their love and encouragement for all these years.

Contents

ABSTRACT	III
ACKNOWLEDGEMENTS	IV
LIST OF FIGURES	VIII
1 INTRODUCTION.....	1
1.1 MOTIVATION.....	1
1.2 PARALLEL SYSTEMS AND I/O INFRASTRUCTURE.....	2
1.3 THESIS OBJECTIVE	5
1.4 THESIS ORGANIZATION	5
1.5 CONTRIBUTION	6
2 BACKGROUND	7
2.1 SPACE-SHARING PROCESSOR ALLOCATION POLICY	7
2.1.1 The Original Adaptive Policy	9
2.1.2 The Modified Adaptive Policy	10
2.2 PARALLEL I/O.....	11
2.2.1 Parallel I/O Overview.....	11
2.2.2 I/O Models.....	13
2.2.3 Discussion	17
3 SYSTEM MODEL AND IMPLEMENTATION.....	19
3.1 MULTI-PROGRAMMING SYSTEM DESIGN.....	19

3.1.1	System Architecture	19
3.1.2	Simulation Model	21
3.2	WORKLOAD MODEL	24
3.2.1	Parallel Job Programs	24
3.2.1.1	Fork-and-join	25
3.2.1.2	Divide-and-Conquer	25
3.2.1.3	GE	26
3.2.2	PARALLEL JOB STRUCTURES WITH I/O OPERATIONS	28
3.2.2.1	Job Structures with NO I/O model	29
3.2.2.2	Job Structures with NOP I/O Model	31
3.2.2.3	Job Structures with O1 I/O Model	32
3.2.2.4	Job Structures with O2 I/O Model	34
3.2.3	WORKLOAD MODEL PARAMETER	35
4	DATA ACCESS AND DISTRIBUTION POLICIES	38
4.1	TYPE OF DATA POLICIES	38
4.1.1	Replicated	39
4.1.2	FixedRW	41
4.1.3	FixedR	42
5	SIMULATION RESULTS AND ANALYSIS	45
5.1	PERFORMANCE SENSITIVITY TO PARALLEL JOB STRUCTURE	46
5.1.1	Performance of All Fork-And-Join Job Structures	46
5.1.2	Performance of All Divide-And-Conquer Job Structures	53
5.1.3	Performance of All GE Job Structure	60
5.2	PERFORMANCE SENSITIVITY TO DATA POLICIES	67
5.3	PERFORMANCE SENSITIVITY TO CPU SERVICE TIME VARIANCE (CV_D)	74
5.4	PERFORMANCE SENSITIVITY TO JOB ARRIVAL RATE VARIANCE (CV_A)	75
5.5	DISCUSSION ON THE SHORTEST WAITING TIME ALGORITHM	76
6	CONCLUSIONS AND FUTURE WORK	78
6.1	SUMMARY	78

6.2 SUGGESTIONS FOR FUTURE WORK	80
---------------------------------------	----

List of Figures

Figure 1 The NO Model.....	14
Figure 2 The NOP Model	15
Figure 3 The O1 Model.....	16
Figure 4 The O2 Model.....	17
Figure 5 System Architecture	20
Figure 6 Fork-and-join job structure.....	25
Figure 7 Divide-and-conquer job structure.....	26
Figure 8 GE parallel program structure	27
Figure 9 Matrix factorization job structure.....	28
Figure 10 Three job structures with NO I/O Model	30
Figure 11 Three job structures with the NOP model	32
Figure 12 Three job structures with the O1 model	33
Figure 13 Three job structures with the O2 model	34
Figure 14 Replicated Data Access/Distribution.....	41
Figure 15 FixedR Data policy.....	44
Figure 16 Job response time for the Fork-and-join job structures under the ReplicatedC data policy	46
Figure 17 Job response time for the Fork-and-join job structures under the ReplicatedD_rr data policy	47
Figure 18 Job response time for the Fork-and-join job structures under the ReplicatedD_shttime data policy	48
Figure 19 Job response time for the Fork-and-join job structures under the FixedR_c data policy.....	49

Figure 20 Job response time for the Fork-and-join job structures under the FixedR_d_rr data policy	50
Figure 21 Job response time for the Fork-and-join job structures under the FixedRW_rr data policy	50
Figure 22 Job response time for the Fork-and-join job structures under the FixedR_d_shttime data policy.....	51
Figure 23 Job response time for the Fork-and-join job structures under the FixedRW_shttime data policy	51
Figure 24 Job response time for the Divide-and-conquer job structures under the ReplicatedD_rr data policy	53
Figure 25 Job response time for the Divide-and-conquer job structures under the ReplicatedC data policy	54
Figure 26 Job response time for the Divide-and-conquer job structures under the ReplicatedD_shttime data policy	55
Figure 27 Job response time for the Divide-and-conquer job structures under the FixedRW_shttime data policy	56
Figure 28 Job response time for the Divide-and-conquer job structures under the FixedR_c data policy.....	57
Figure 29 Job response time for the Divide-and-conquer job structures under the FixedR_d_rr data policy.....	58
Figure 30 Job response time for the Divide-and-conquer job structures under the FixedR_d_shttime data policy.....	58
Figure 31 Job response time for the Divide-and-conquer job structures under the FixedRW_rr data policy	59
Figure 32 Job response time for the GE job structures under FixedR_c data policy.....	61
Figure 33 Job response time for the GE job structures under ReplicatedC data policy ..	62
Figure 34 Job response time for the GE job structures under ReplicatedD_rr data policy	62
Figure 35 Job response time for the GE job structures under ReplicatedD_shttime data policy.....	63
Figure 36 Job response time for the GE job structures under the FixedRW_rr data policy	64
Figure 37 Job response time for the GE job structures under the FixedRW_shttime data policy.....	64
Figure 38 Job response time for the GE job structures under the FixedR_d_shttime data policy.....	65
Figure 39 Job response time for the GE job structures under the FixedR_d_rr data policy	66

Figure 40 Job response time for all data policies under the NO_Fork-And-Join job structure.....	67
Figure 41 Job response time for all data policies under the NO_Divide-And-Conquer job structure.....	68
Figure 42 Job response time for all data policies under the NO_GE job structure	68
Figure 43 Job response time for all data policies under the NOP_Fork-and-join job structure.....	69
Figure 44 Job response time for all data policies under the O1_Fork-and-join job structure.....	70
Figure 45 Job response time for all data policies under the O2_Divide-and-conquer job structure.....	70
Figure 46 Job response time for all data policies under the O1_GE job structure	71
Figure 47 Job response time for all data policies under O2_GE job structure	72
Figure 48 Performance sensitivity of CPU service time variance	74
Figure 49 Performance sensitivity to inter-arrival time variance	76
Figure 50 Response time comparison for NOP_Divide-and-conquer job structures	77

Chapter 1

Introduction

1.1 Motivation

In recent years, substantial improvement has been obtained in processor and communication performance of parallel computers. However, advances of similar magnitude have not taken place in the input/output (I/O) area as far as the access times are concerned. Most of the parallel applications such as climate prediction, water pollution control, and galaxy formation require significant computational capacity as well as data intensive I/O operations [3, 27]. The computational speed of parallel computers can reach more than 1 Tflops/sec but the real applications running on these parallel machines usually achieve a I/O bandwidth of at most a few hundred Mbytes/sec. Some applications can only achieve less than 10 Mbytes/sec [24].

Thus, performance bottleneck caused by I/O subsystem becomes a more dominant factor than the bottleneck of CPU processing in overall performance of parallel applications. To improve I/O performance, various optimization techniques such as disk

caches/arrays and parallel/concurrent I/O access to multiple disks have been introduced [26].

For parallel applications, optimization in the I/O subsystem alone is not enough to improve the overall performance of such applications. The applications must adopt parallel algorithms that optimally utilize different subsystems such as CPU, communication, and I/O to achieve scalable and balanced performance. Without this system-level coordination, starvation and congestion may occur in some subsystems and thus degrade the overall performance. In the parallel I/O research area, most effort has been focused on the design of parallel file systems and management of parallel disks. Some classic strategies include the two-phase data access strategy [28], data sieving [29], and collective I/O [30]. Comparatively less work has been done on combining parallel I/O algorithms with parallel job structures with various parallelism levels. The objective of this work is to study the performance impact of parallel I/O and data access methods on parallel job structures.

1.2 Parallel Systems and I/O Infrastructure

Most parallel systems provide hardware and software support for parallel I/O operations. This allows processors to make concurrent requests to I/O subsystems. I/O nodes in parallel systems can be either dedicated or non-dedicated for I/O operations. Dedicated I/O nodes can only perform I/O requests while non-dedicated I/O nodes can process both I/O and computation operations. For read operations of an application, the I/O nodes read the requested data from the disk and distribute them to the computing nodes that

issue the read request. Similarly, data are gathered from the computing nodes and written to disk for the write operations. An I/O node in general uses several disks in parallel (e.g., a redundant arrays of inexpensive disks (RAID) system) to increase the I/O bandwidth and to ensure data integrity. Typical architectural designs of parallel systems along with I/O strategies are summarized as follows:

□ Shared-Memory Systems

The shared-memory parallel systems such as SGI Origin 2000 and HP/Convex Exemplar do not have separate I/O nodes. The disks are connected to the computational unit via a single bus line. The operating system schedules the file systems server on the computing nodes [16]. There may have multiple disks and a file system that stripes files across the disk.

□ Distributed-Memory Systems

Distributed-memory systems do not share memory among the processors. Processors communicate with each other explicitly through message passing. Usually distributed-memory systems such as Intel Paragon and IBM SP have a set of separate I/O nodes dedicated to perform I/O requests. These I/O nodes act as servers for the parallel file system and are normally connected to one or more disks or RAID's. The file system normally divides a file into a number of smaller units called striping units and assigns those striping units to disks in a round-robin manner. The striping technique allows multiple processors to access distinct portions of a file concurrently and thus provides higher bandwidth [16]. Some machines also provide local disks for computing nodes. These local disks are not accessible by other nodes.

□ Cluster Systems

With the availability of high-speed processors and the scalability of networks, it is becoming more popular to use cluster systems to run parallel applications. In comparison with classic super computers, cluster systems are easily scalable at low cost but typically have slower interconnections between processors. This latter issue is addressed by new and user-friendly techniques such as 1Gbps Ethernet and Myrinet. While a cluster system has less elaborate interconnections between units and thus maximum system-wide I/O throughput is bounded by the interconnection bandwidth, it has an advantage in that each node has one or more local disks and certain operations can be completed on these local disks without accessing the shared interconnection facilities. When a local disk operation is not sufficient and data has to be exchanged among different nodes, the relatively slow interconnections become performance bottlenecks. Following strategies are devised to address this issue [31]:

- Replication: A copy of data is maintained on each disk.
- Caching: Read data from a single disk, broadcast them to all units and keep them in the memory of all processors.
- Distribution: Distribute data among local disks. Each processor will keep part of the data. Whenever the part of the data is needed during program execution, the processor that has the part is responsible for reading the data and broadcasting it to other units.

1.3 Thesis Objective

It is observed that integrated approaches at application, system software, and architecture levels need to be applied in parallel applications in order to address the I/O performance bottleneck. The Modified Adaptive Policy (MAP) for processor scheduling policy has been shown to provide better performance in [1]. The MAP processor scheduling policy is thus used for processor resource management in this research. The objectives of this thesis are summarized as follow:

- ❑ To understand the performance sensitivity to the parallel job structure with different models of I/O requests. Three parallel job structures are studied to evaluate the performance impact when different types of I/O job requests are explicitly introduced on a sub-set of processors.
- ❑ To identify data access and distribution strategies in I/O models to achieve better performance. Different data access and distribution strategies in I/O models affect the I/O operation performance in parallel applications. It is important to find out a proper data access and distribution method in I/O operations for different types of parallel applications.
- ❑ To evaluate performance sensitivity of the system model on parallel job structure with certain data access and distribution methods.

1.4 Thesis organization

The remainder of this thesis is organized as follows. Literature on parallel I/O research, three parallel job structures used in this thesis, and the MAP processor scheduling policy

are described in Chapter 2. Chapter 3 covers the implementation of models that include parallel job structures in combination with various I/O models used in this research. Simulation models and high-level implementation design are presented along with an introduction to the simulation engine CSIM19. Chapter 4 discusses data storage and distribution policies used in the experiments. Chapter 5 presents experimental results on various job models. Performance results for various I/O models and data access and distribution methods for each job structure are compared in this chapter. Chapter 6 gives conclusions of this thesis and points to future research topics.

1.5 Contribution

A number of contributions have been made by this research activity to understand the performance impact of parallel I/O on different parallel job structures.

1. A new data access and storage algorithm is proposed. In this algorithm, device allocation is based on shortest service waiting time for an I/O request. Significant performance improvement has been obtained in applying this algorithm to ReplicatedD, FixedR and FixedRW data access/distribution policies [2].
2. Data access and distribution strategies in I/O models have been identified to achieve better performance.
3. We evaluated the performance sensitivity to the parallel job structures with different models of I/O requests.
4. We studied the performance sensitivity of the system model to parallel job structure under certain data access and distribution methods.

Chapter 2

Background

With the dramatic improvement in processor speed, I/O subsystem has become significant parallel performance bottleneck for some parallel applications. As a result, the I/O bottleneck problem has been receiving increasing attention from academic research communities. This chapter reviews strategies in addressing I/O performance issues for parallel applications along with processor scheduling policies.

2.1 Space-Sharing Processor Allocation Policy

Distributed-memory systems are an important class of parallel systems. The Intel Paragon, Teraflop, and NCUBE2S systems are some examples of commercial systems in this category. Distributed-memory multicomputer systems do not share memory among processors. Processors communicate with each other by explicit message passing. Job scheduling policy plays an important role in the overall performance of such systems. The process of job scheduling can be divided into two steps. The first step is to allocate a set of processors for a job. Second, the tasks of the job can be executed either

sequentially or in parallel on the given processors [7]. We will discuss the processor allocation policies next.

To obtain good performance, the processors' capability should be equally allocated to jobs in multiprocessor systems [5, 6]. The processor scheduling policies can be briefly divided into *time-sharing* and *space-sharing* policies. In a time-sharing policy, different jobs share the same processors in a round robin fashion. No processor is given exclusively to a job. In the space-sharing policies, on the other hand, the processors are divided into distinct sets. Each set of processors is given to a job for its execution.

The space-sharing policies can be further classified into *fixed*, *variable*, *adaptive*, and *dynamic* policies [8]. In the fixed space-sharing policies, the partition size is fixed on a long-term basis and is normally modified only at system reboot time. This policy is simple to implement. However, its major disadvantage is that it does not adapt to changes in system load conditions and resource requirements of jobs. The variable policies allow the user to change partition size at job submission time. In adaptive partition policies, the system determines partition size by taking system load conditions and user requirements into account. Partition sizes in the above three policies remain the same during job lifetime. This type of processor allocation policies could cause resource wastage if a job's parallelism varies during its execution. Dynamic policies address this inefficiency. Dynamic policies allow partition size to change during a job's execution. Though there is no performance loss due to fragmentation and idled processors, there is still performance overhead due to job preemption, data migration, and reconfiguration of application. As the performance of dynamic policies depends on the scheduling cost and

system workload characteristics, they are not widely used for parallel job scheduling in cluster systems [9].

2.1.1 The Original Adaptive Policy

It is observed that in order to achieve better performance, job scheduling policies should facilitate sharing of the processing power equally among jobs [5, 6]. The space-sharing policies proposed by Rosti et al. in [10] attempt to generate equal-sized partitions. The algorithm of the basic adaptive policy is shown below:

Given: a specific implementation of `compute_target_size`

```
target_size ← compute_target_size
while (queue_length > 0) and (free_processors ≥ target_size)
do
    free_processors ← free_processors - target_size
    queue_length ← queue_length - 1
    schedule_a_job(target_size)
od
```

A set of five adaptive (AP1 through AP5) space-sharing policies have been proposed by Rosti et al. [10]. These five adaptive policies (AP1 through AP5) use different algorithms to calculate the partition size for a job. These policies adjust the partition size of each job at the scheduling time by taking workload and system conditions into account. Overall, the AP2 policy provides best performance under various workload and system conditions. Thus we focus on the AP2 policy in this study.

Among the five policies, the original AP2 proposed by Rosti et al. is the only policy that reserves a partition for future jobs. The processor partition size is calculated as follows:

$$\text{partition size} = \max \left(1, \text{ceil} \left(\frac{\text{total processors}}{\text{queue length} + 1} \right) \right)$$

This policy reserves one partition for a future job by adding 1 to the queue length. However, this policy only relies on the queued jobs to determine the partition size but not the total number of jobs in the system. For instance, on a 32-node system, when two jobs arrive at the idle system, the first job will get 11 processors according to the AP policy. The second job will be allocated 16 processors. When a third job arrives, a partition size of 16 should be assigned to the job. However, there are only 5 free processors available in the system. Thus, the third job has to wait until a partition is freed. Since this policy considers only the queued jobs to determine the partition size, it conflicts with the principle of sharing processor power equally among the jobs. The MAP scheduling policy proposed in [1] addresses this limitation by considering the total number of jobs in the system when allocating a partition to a job.

2.1.2 The Modified Adaptive Policy

This policy, proposed in [1], modifies the original adaptive policy described in previous section by taking into account the total number of jobs, no matter whether the job is already scheduled or just waiting in the job queue. The modified partition size calculation formula is shown below:

$$\text{partition size} = \max \left(1, \text{ceil} \left(\frac{\text{total processors}}{\text{queue length} + (f * \text{number scheduled jobs}) + 1} \right) \right)$$

where $0 \leq f \leq 1$. This modified policy is called MAP (modified adaptive policy) in [1]. The parameter f is used to control the contribution of scheduled jobs to the partition size.

When the f value is set to 0, the MAP policy reduces to the original adaptive policy. When f is set to 1, it takes all the jobs in the system into account to calculate the partition size for each job. By looking at the previous example, if MAP is being used to determine the partition size, the third job will be allocated 8 processors when f is set to 1. Thus MAP utilizes processor resources better than the original policy. The objective of equal partition for each job is better realized by taking all jobs into account. Results in [1] show that the modified policy provides significant performance improvement. In terms of performance sensitivity of this policy to f parameter, it has been found that $f = 0.75$ provides the best performance in most cases.

2.2 Parallel I/O

2.2.1 Parallel I/O Overview

An integrated approach that includes the application, system architecture, and software solutions is normally required to obtain good I/O performance for parallel applications. Hardware improvement such as multiple disks, faster I/O processors, and larger bandwidth I/O buses can enhance I/O performance [3, 12]. Other I/O techniques to improve parallel I/O performance can be classified into three categories: application-level methods, device-level methods and access-anticipation methods. These I/O techniques try to achieve one or more of the following goals [31]:

- To use disk devices in parallel in order to increase the I/O bandwidth
- To reduce the number of disk access operations to avoid multiple latencies

- To reduce the application run time by overlapping computation, communication operations with the I/O operations.
- To minimize the reorganization and administration costs.

The two-phase method proposed by del Rosario, Bordawekar and Choudhary [28] is an example of the application-level technique. In this method, data are collectively accessed locally in the first phase. In the second phase, data are transferred to the requesting nodes over the network. Another commonly used application-level technique is data sieving, suggested in [29]. To avoid disk latency, a large sequential block of disk is read at once into the main memory and then a smaller block of data is extracted upon request from the large block.

Device-level methods normally reorganize I/O requests to improve overall performance. Representative methods of this type are disk-directed I/O in [20] and server-directed I/O in [33].

Access-anticipation methods attempt to optimize I/O performance by using proper data access and distribution strategies for the I/O requests in parallel applications. Informed pre-fetching technique [34] and two-phase data administration [35] are two examples.

In addition, several optimization techniques such as data striping, caching, and pre-fetching for parallel file systems are studied in order to reduce the I/O latencies [13, 14, 15]. Other important techniques include collective I/O and two-phased I/O [28]. To facilitate easy use of parallel I/O techniques, high-level parallel I/O libraries and runtime systems have been proposed by several groups. Runtime systems allow a user to input the access pattern of their codes using program-level data structures [17, 18, 19]. A new

parallel I/O interface standard MPI-2 has been proposed [25]. This interface allows programmers to access several noncontiguous pieces of data from a shared, disk-resident file using a single I/O call. Though most of the research and studies on parallel I/O focus on the design of file system, as well as the coordination and management of parallel disk, scheduling of parallel I/O operations has also drawn some attention as in [21, 22, 23].

2.2.2 I/O Models

The impact of four different I/O models with fork-and-join job structure and various data access and distribution strategies on system performance were studied in [2]. These I/O models are detailed below.

□ NO (No Overlap) Model

In this model, there is only one read operation and only one write operation during the whole job execution time. The read and write operations are performed from a single I/O device. The read request is carried out at the beginning of the job execution. After the read operation finishes, the job then forks into multiple child threads to perform the computation work in parallel. A write operation is executed after the computation work finishes. The job terminates after completion of the write operation. This I/O model is shown in Figure 1.

Figure 1 The NO Model

□ NOP (No Overlap Parallel) Model

This model is very similar to NO model except that the read and write operations are performed in parallel. There are multiple child threads forked to perform the read request at the beginning of the job execution. The computation phase is the same as in the NO model. Before a job terminates, multiple threads run in parallel to write the result data to the I/O devices. The parallel read and write operations can be executed either on a single I/O device or on more than one I/O device depending on the data access/distribution policy used by the application. There are two synchronization points that need to be noted. The first one is at the end of the parallel read operations. The computation work cannot be performed in parallel until all the read operations are completed. The second one is at the end of the computation operation. Only when all the child threads finish computation tasks can the write operations start in parallel. This I/O model is shown in Figure 2.

Figure 2 The NOP Model

□ O1 Model

In the NO and NOP models, jobs can either perform I/O operation or computation but not simultaneously. There is no operation overlap between I/O operation and computation. The O1 model introduces a job structure that overlaps I/O operations with computation work. In this model, a job forks multiple threads to implement tasks in parallel at the beginning of execution. Each thread reads its input data at the beginning of the execution and then performs some computation. After finishing its own computation work, each thread writes the result to an I/O device. A job terminates after all of its write operations are completed. The I/O operation of one thread in O1 model can be overlapped with computation performed by another thread. As the threads in this model run independently, each thread doesn't have to wait for all the other reading operations to complete before starting its own computation. This I/O model is shown in Figure 3.

Figure 3 The O1 Model

□ O2 Model

This model introduces more overlap between I/O and computation operations. After a job forks multiple threads to perform tasks in parallel, a thread can alternate between performing computation and I/O operations. The I/O operations during the task execution can be either reading data or writing results. The first operation in the task is always a data read. The last operation of a child thread is to write the computation result. A job completes when all of its write operations are done. The O2 I/O model is shown in Figure 4.

Figure 4 The O2 Model

2.2.3 Discussion

As mentioned before, I/O performance bottleneck needs to be addressed at the system level by integrating application, software, and architecture into the potential solutions. This research explores several areas to further our understanding of the parallel I/O impact on various parallel application models.

1. The MAP processor scheduling policy: The study in [2] uses static processor scheduling policy. In our study, we use a modified adaptive processor scheduling policy (MAP policy) for processor allocation. The adaptive processor scheduling policy takes system load and job resource requirement into account when processors are assigned to an incoming job. The MAP policy is expected to achieve better performance since it is shown to be more efficient than the original adaptive policy.

2. More application models: In addition to the fork-and-join job structure that is used with the parallel I/O, we use two additional job structures: divide-and-conquer and GE. These job structures are introduced in the next Chapter.

3. Data access/distribution strategies: Research has shown that data distribution methods have a significant influence on I/O parallelism and I/O workload. It is useful to understand the I/O workload and parallelism introduced by different data distribution methods in order to select a better distribution scheme to improve performance. In this thesis, a new shortest waiting time algorithm is proposed and studied under several data access/distribution categories.

Chapter 3

System Model and Implementation

The overall design and system model are discussed in Section 3.1. The implementation part in Section 3.1 also introduces the simulation tool used in this research. Section 3.2 explains three job structures incorporating the parallel I/O models discussed in the last Chapter.

3.1 Multi-programming System Design

3.1.1 System Architecture

The simulation program is based on the discrete event driven simulation technique. There are two subsystems in the simulation program. One is the multiple processors subsystem and the other is the I/O devices subsystem. The total number of processors and I/O devices are pre-defined in the system. When a job enters the system, it is placed

in a central FIFO job queue. Each job has a predefined minimum required number of processors. If there are enough free processors available in the system, the job scheduler will de-queue the first job and assign a certain number of processors to it. If there are not enough free processors available, the job scheduler will wait for processors to be freed. When a job has completed, the assigned processors are freed. An I/O request of a job is routed to an I/O device and processed immediately if the device is idle. Otherwise the I/O request is placed in a queue depending on the data access/distribution method used.

Three types of daemons are used in the simulation model: a job generator daemon, a job scheduler daemon, and a job execution daemon. The relationship among these daemons and subsystems is illustrated in Figure 5.

Figure 5 System Architecture

A job is generated with an arrival rate λ by the job generator. The number of processors assigned to the job is determined by the MAP policy [1]. A job has access to the appropriate I/O devices and processors assigned to it. Processors allocated to a job remain with the job during execution and are returned to the scheduler after the job completes. The processors assigned to a job are not shared during the job execution.

3.1.2 Simulation Model

A simulation software package CSIM19 by Mesquite Software is used to implement the design of this research study. The CSIM19 package is implemented as a library of classes and procedures with necessary structures and operations that allows the user to create process-oriented and discrete-event simulation models for complex systems.

The CISM19 program models a system as a collection of interacting processors. It provides a complete set of objects that can be used to construct models of almost any kind of system at any level of complexity and detail. A high-level description of the CSIM19 objects is presented in the following section.

- **Processes**

Processes are used to model the active entities of a system. Depending on the type of model, processes may model different elements of the workload. For example, in this study several processes are created to model the job generator, job scheduler, and job executor. In CSIM19, a process is a procedure which is executed with the create statement. For example, the job generator process is defined as:


```

void jobGenerator()      //defines job generator
{
    create("jobGenerator");
    // generator job code here
}

```

A CSIM19 process is started by another process; it is called using the usual procedural call. Processes in CSIM19 are operated in a “simulated environment” under the control of CSIM19 execution supervisor during the simulation time. Multiple instances of the same process can be “active” at same time in this simulated environment. A process can be in one of four states:

- Computing ---- using CPU
- Ready ---- ready to start computing
- Holding ---- allowing simulated time to pass
- Waiting ---- allowing external events to occur

The above job generator process can be invoked in a periodic manner by other processes as shown below:

```

void otherProcess() //the starting process
{
    // other statement for example set to during simulated time
    jobGenerator();
    hold(10.0);
    // other statement
}

```

The hold(10.0) statement suspends the ‘otherProcess’ and schedules it to enter the Ready state when 10.0 unit of simulated time have passed.

A process can also invoke multiple processes by using a “for” loop. At the end of the “for” loop, those multiple instances of invoked processes are ready to enter the Computing state.

▪ Resources

In many systems, processes must use or occupy a series of resources such as CPU processor and I/O devices. CSIM19 software offers two kinds of resources: facility and storage. A facility consists of a single queue (for waiting processes) and one or more servers. The operations that apply to facilities by processes include:

- reserve - wait for and then gain access to a “free” server,
- release - release a reserved server, and
- use - a combination of a reserve, hold and release.

A storage consists of a queue and a pool of storage units. A process can be allocated to one or more storage units. If a number of units is not sufficient for a process, the process is suspended and placed in queue. The queued process then gets access to the given units when other processes have de-allocated their storage units.

▪ Process Interactions

There are two mechanisms to facilitate and control interactions and communications among different processes:

- Events
- Mailboxes

An event can be either in OCCURRED state or NOT_OCCURRED state. A process can wait for a specified event. Another process can set the event to cause the waiting process to resume execution.

A mailbox is a mechanism for processes to exchange messages. A process can send a message to a mailbox or receive a message from a particular mailbox. If there is a message in the mailbox, the receiving process gets the message and continues

execution. Otherwise the receiving process will wait on the mailbox until there is a message sent to the mailbox.

- **Other features**

All of the data collected at facilities and storages are accessible via a group of inspector functions. These functions can be used to produce output reports tailored to specific reporting needs. CSIM19 also provides random number generator function to be used for various probability distributions. Multiple streams of random numbers are available for some specific requirement to have each stream operations in a repeatable manner. It also provides a mechanism for running a model until a desired confidence level is achieved for a specific statistic. The automatic run length control can be used with tables, qtables, meters and boxes.

3.2 Workload Model

3.2.1 Parallel Job Programs

Three types of parallel job programs are selected for this research – fork-and-join, divide-and-conquer and Gaussian elimination (GE). These three parallel programs cover the spectrum of variable parallelism job structures. The fork-and-join program exhibits constant parallelism. The parallelism of divide-and-conquer program increases initially to a maximum and then decreases progressively. In GE job program, a job starts with its maximum parallelism and progressively decreases to 1.

3.2.1.1 Fork-and-join

In this type of program, work can be decomposed into independent sub-works. Each sub-work can be executed by parallel tasks. Figure 6 shows one phase of such a job program.

Figure 6 Fork-and-join job structure

3.2.1.2 Divide-and-Conquer

In general, a divide-and-conquer program does not restrict the number of sub-work into which a given job is partitioned. However, we only focus on the binary case in which a given job is divided into two sub-works in this study. The parallelism steps through the sequence of $2^0, 2^1, 2^2, \dots, 2^{n-1}, 2^n, 2^{n-1}, \dots, 2^1, 2^0$ during job executions. Figure 7 shows this job structure.

Figure 7 Divide-and-conquer job structure

The binary search/sort program is one of the examples that use the divide-and-conquer algorithm. This type of algorithm performs one of the following three operations: DIVIDE, WORK and MERGE. In Figure 7, the input job is split into 2 sub-tasks. This is the DIVIDE operation. Then each sub-divided task is decomposed into independent sub-computations in the WORK stage. Then the computation results are combined together at the MERGE stage. Generally, execution time of the tasks decreases level-by-level over the upper half of the structure and increases level-by-level over the lower half of the structure.

3.2.1.3 GE (Gaussian Elimination)

The matrix equation of the form $Ax = b$ can be solved by decomposing A into upper U and lower L triangular matrices and resolving $Ly = b$ and $Ux = y$. The most computationally intensive part of the algorithm is to find the L/U decomposition. Thus

the L/U decomposition has received considerable attention. Much research has been done to derive a task system for the L/U decomposition algorithm and find an optimal static scheduling on this task system. Other related algorithms include the Cholesky factorization and Gauss-Jordan algorithms. The basic idea of these decomposition algorithms is to compute and find the k th pivot column and then to modify all the columns that are right of the pivot column. This computation work continues until the last element A_k^k is reached. The following diagram describes details of this process.

Figure 8 GE parallel program structure

The structure of these parallel programs can be represented by the task system shown in Figure 9. T_i^j represents the task that works on column i in the j th iteration. However, the actual function of the tasks is different for the above- mentioned algorithms. In the L/U decomposition and Gauss-Jordan algorithms, T_k^k computes to find the pivot element. For the Cholesky factorization algorithm, T_k^k computes the square root of the diagonal

element and then divides the lower part of column k by this square root. In all the three algorithms, T_k^j uses the elements of column k to modify those in column j .

Figure 9 Matrix factorization job structure

Since matrix factorization is a way of performing Gaussian elimination, we refer this class of programs as the Gaussian elimination (GE) programs.

3.2.2 Parallel job structures with I/O operations

The parallel job structures studied in this research are formed by combining the three parallel programs with the four I/O models. These four I/O models are NO (No overlap), NOP (NO overlap-parallel), O1 (overlap #1) and O2 (overlap #2) discussed in Chapter 2.

The new job structures are named such that both the I/O model and the job structure type appear in the name.

3.2.2.1 Job Structures with NO I/O model

In this job structure category, there is no overlap between I/O operations and computation works. A job can either perform the I/O operation or do the computation work but not perform both operations simultaneously. As we consider only a single phase in the experiments, there will be a total of two I/O operations during the execution of a job. During the job execution, a read operation will take place at the beginning. A write operation will be performed to write the data on the disk after completing the computation work phase. The job leaves the system when the write operation is completed. Figure 10 shows the three job structures with NO I/O model.

write operation takes place at the MERGE stage. Synchronization is done at each MERGE stage.

In the NO_GE model, the I/O operations are done in the same manner as in the other two NO I/O job structures. The read operation gets all the data from disk. Maximum number of tasks is forked in the second stage and the number decreases by one at each subsequent level until it reaches 1. A single write operation completes the whole job.

3.2.2.2 Job Structures with NOP I/O Model

The NOP I/O model is similar to the NO I/O model. There are read operations at the beginning of the job and write operations at the end of the job. The difference between the NOP and NO model is that the I/O operations in NOP are implemented in parallel. Multiple tasks are forked at the beginning of the job to perform the read operations simultaneously. Similar operations are done for the write operation at last stage of job execution as shown in Figure 11.

Though the I/O operations in the NOP model are implemented in parallel, there is no overlap between the I/O operations and computation works. All the read operations must be finished before any computation can start. No write operation can be started until all computations are completed. Figure 11 shows the three job structures with the NOP I/O model.

Figure 11 Three job structures with the NOP model

(a) NOP_Fork-and-join (b) NOP_Divide-and-conquer (c) NOP_Gaussian-elimination

3.2.2.3 Job Structures with O1 I/O Model

The O1 model introduces the I/O operation overlap with computation. The read and write operations take place within each child thread. In each thread, the read operation is performed at the beginning of the thread execution before it performs the computation

3.2.2.4 Job Structures with O2 I/O Model

This model introduces more overlap between the I/O and computation. This model is similar to the O1 model as described in the last section. Unlike the O1 model, each thread can have only one read operation and one write operation. This model introduces more I/O operations within each child thread in that each thread in this model can alternate the I/O and computation during its execution. Figure 13 shows the O2 I/O models for the three job structures.

Figure 13 Three job structures with the O2 model

(a) O2_Fork-and-join

(b) O2_Divide-and-conquer

(c) O2_Gaussian-elimination

Each thread in the three O2 job structures starts by reading some data from an I/O device. The last operation of each thread is the write operation. The thread performs

computation and I/O operations alternatively during its execution. The I/O operations can be either read or write operations.

3.2.3 Workload model parameters

This section presents an abstract workload model for parallel programs. The abstract model consists of a set of input parameters along with a job structure.

Job arrival rate (λ): This is the mean arrival rate of a job. The Poisson job arrival model is used in our experiments.

Coefficient of variation of job arrival time (CV_a): CV_a refers to the coefficient of variation of the job arrival time.

Processors (P): The number of processors in the system.

Maximum parallelism (mp): The maximum parallelism of a parallel program refers to the maximum number of threads that can be run in parallel. In a fork-and-join job structure, the maximum parallelism is always equal to the number of threads of the job. For the divide-and-conquer job structure, the maximum parallelism is achieved when the job is at its WORK stage. For the Gaussian elimination job model, the maximum parallelism is at the second stage of the program when the independent threads are created to work on each column of the matrix.

I/O devices (I): The number of I/O devices in the system.

Mean Computation Service Demand (D): The mean computation service demand refers to the mean CPU service demand for a parallel program. The uncorrelated model is used

for the computation service demand in this study. That means the overall service demand of a parallel program is computed before this service demand is distributed among the threads of the program. The overall service demand of a parallel program in this model is independent of the number of threads in the program.

Coefficient of variation of CPU service demand (CV_d): CV_d refers to the coefficient of variation of the CPU service demand of a CPU request. In our study CV_d refers to the coefficient of variation of service demand of parallel programs as the uncorrelated model is used in this research.

Number of I/O operations in O2 model (k): This parameter represents the number of I/O operations per O2 child thread ($k \geq 2$).

Read Probability in O2 model (p): This is the probability of read operations in an O2 child thread's I/O request.

I/O factor (F_i): This represents the I/O request factor in a job.

Coefficient of variation of an I/O demand service time (CV_i): This is the coefficient of variation of the I/O demand service time.

The mean total I/O demand of a job (IOD) is a fraction of the mean CPU demand D for a parallel program. The IOD can be expressed as:

$$IOD = D * F_i$$

Synchronization cost (syn): This parameter represents the cost (i.e., processor service time required) of performing synchronization. In the divide-and-conquer workload, both the divide and merge operations are assumed to take the same amount of processor time and is represented by this parameter. For the GE model, this parameter represents the processor service demand for each serial phase T_k^k .

These parameters along with a job structure define the workload. Job mean response time (R), server utilization (U), and server throughput (T) are used to measure the system performance.

Chapter 4

Data Access and Distribution Policies

The parallel application data can be fetched from or distributed to disks using various algorithms. These algorithms are discussed in this chapter. We call the data access and distribution policies as the data policy to simplify the terminology.

4.1 Type of Data Policies

Three types of data access and distribution patterns are studied in this research. Depending on which I/O device is used to perform write or read request, the access and distribution patterns can be classified as Replicated (write on any device and read from any device), Fixed Read and Write (FixedRW), and Fixed Read (FixedR) [2]. The Replicated and FixedR classes can be further divided into centralized and distributed approaches. The centralized approach is suitable for small to medium scale shared-memory systems or small distributed systems. For larger, especially distributed systems, distributed data storage/distribution strategy is more appropriate.

Two types of device allocation algorithms, round robin and shortest waiting time, are studied for the above data policies in our experiments. Random selection of I/O devices is only applied to the FixedR data policy for I/O device selection for read operations. The round robin device allocation algorithm tries to enhance the application I/O parallelism by allocating maximum number of different I/O devices. The shortest service waiting time device allocation algorithm minimizes the waiting time at a device queue. It helps reduce the I/O bottleneck by getting the I/O request to be scheduled in the shortest possible time.

4.1.1 Replicated

When the Replicated method is used, data for the read operation is available on all I/O devices. In other words, data are replicated on each I/O device for read. So the read operation can be performed on any I/O device. The data to be written is not replicated for the write operation. The Replicated data access and distribution method can be further divided as Centralized Replicated (ReplicatedC) and Distributed Replicated (ReplicatedD) methods depending on how the read or write requests are scheduled.

- ReplicatedC

There is a centralized I/O scheduler in this approach recording the status of all I/O devices. A central FIFO queue is maintained to keep all the I/O requests. When an I/O request, regardless of the read or write operation arrives at the scheduler, it will be sent to an idle device. If all I/O devices are busy, then the request will be inserted into the FIFO queue. The request will be processed when an I/O device is free.

- ReplicatedD

In this approach, there is no centralized scheduler monitoring the status of the I/O devices. No central FIFO queue is maintained in this case. An I/O request is sent to a device immediately regardless of the status of the device. The I/O device can be selected either in a round robin fashion or by the shortest waiting time algorithm. For the round robin approach, the first I/O device number is selected randomly at j . The remaining I/O requests for different tasks will then be processed on device ' $j+1$ ', ' $j+2$ ' and so on. After all I/O devices in this round have been assigned, the scheduler will then start another round of round robin I/O device assignment starting from reselecting an I/O device number randomly again. For the shortest service waiting time approach, an I/O request is sent to an I/O device for which the service time of all the I/O requests waiting for that device is the shortest. There shall be a global table that is associated with the I/O router to maintain the I/O request waiting time on each I/O device in order to select which I/O device shall be used for the next request. We name this two distribution approaches as ReplicatedD_rr (round robin) and ReplicatedD_shttime (shortest service waiting time). For both the distributed approaches there is a local FIFO queue at each I/O device. The I/O request might be performed immediately if the device is idle or it might be inserted into the local FIFO queue. Figure 14 shows the Replicated policies.

Figure 14 Replicated Data Access/Distribution

In NOP, O1 and O2 parallel job structures, the replicated approach is suitable to those applications that need to write the temporary computation data of each thread on an I/O device.

4.1.2 FixedRW

Unlike the Replicated method, there is no data replication in FixedRW approach. A specific device will be used to perform the read and write operations for a certain thread or processor. The I/O request is sent to the specific I/O device without considering its status. The request will be inserted into the local FIFO queue if the device is busy. Same as in the replicated distributed approach, the device for an I/O request can be selected in a round robin fashion or by the shortest service waiting time algorithm. Thus

the FixedRW data access and distributed policy can be further divided into FixedRW_rr (Round Robin) and FixedRW_shttime (shortest service waiting time).

- FixedRW_rr (Round Robin)

The device is selected in a round robin fashion for each task to undertake read and write requests. The algorithm is the same as that described in the replicated distributed approach. Once the device for read operation is chosen for a task, all the remaining tasks will use the same device for both read and write operations.

- FixedRW_shttime (shortest service waiting time)

An I/O device with the shortest service time of the I/O requests queued up at its local queue is selected for the incoming I/O request. But once an I/O device is selected for a job, all other tasks in the job will use the same device for the I/O operations.

4.1.3 FixedR

In this data access and storage mode, data for the read operation can only be performed on a specific I/O device for a task but data result is not restricted to be written onto the same device. A read request of a task is sent to a randomly selected device selected. Once the device for a read operation is selected, all the read operations for the tasks will use the same device. If the device is busy, the read request will be served in FIFO manner. The device for the write operation of a task is selected based on different algorithms. The write operation in the FixedR method can be further divided into FixedR_c (centralized) and FixedR_d (distributed) categories.

- FixedR_c (centralized)

Similar to in the ReplicatedC policy, there is a central FIFO queue for the write requests. A central router monitors the status of all I/O devices. A write request will be routed only when an I/O device is idle. If all devices are busy when a write request arrives, the write request will be inserted into the central FIFO queue.

- FixedR_d (Distributed)

There is no centralized FIFO queue for write requests either for this data policy. A write request is sent to a selected device based on a predefined algorithm. The local FIFO queue of an I/O device can also be used to insert write requests if the device is busy. In this approach, the I/O devices for the write operations of a job are chosen either by round robin or shortest service waiting time algorithm. We name them as FixedR_d_rr and FixedR_d_shttime.

Figure 15 shows the FixedR data access/distribution methods.

FixedR_c Data access/distribution policy

FixedR_d Data access/distribution policy

Figure 15 FixedR Data policy

Chapter 5

Simulation Results and Analysis

We have conducted a detailed simulation study of the four parallel job structure types described in Chapter 3 combined with the various data access and distribution policies discussed in Chapter 4. This section presents the results of the simulation experiments to show performance sensitivity of different parallel job models and data access/distribution policies. Unless otherwise stated, the following default parameter values are used : $P = 64$, $mp = 32$, $f = 0.5$, $D = 16$, $I = 4$, $k = 5$, $p = 0.5$, $syn = 0.1$, $CV_d = 1.0$, $CV_a = 1.0$, $CV_i = 1.0$, $F_i = 0.5$. The number of processors in the simulation experiments is set to 64. The maximum parallelism of a job is varied from 1 to mp . The I/O factor of a job is set to 50% of the total CPU demand. We use Poisson job arrival model in our study. The CV_d is set to 1. The performance sensitivity to variance CV_d and arrival variance CV_a is discussed in Section 5.3 and 5.4, respectively.

5.1 Performance Sensitivity to Parallel Job Structure

5.1.1 Performance of All Fork-And-Join Job Structures

Overall the O1_Fork-and-join model provides the best performance under all data policies in all the Fork-and-join parallel structure applications. Figure 16 shows the mean job response time obtained by the Fork-and-join job structures with NO, NOP, O1 and O2 I/O model under ReplicatedC data policy. The data in Figure 16 shows the improvement in performance with O1 I/O model can be as high as 60% over NO I/O model when the job arrival rate is 0.05.

Figure 16 Job response time for the Fork-and-join job structures under the ReplicatedC data policy

Even at high job arrival rate $\lambda = 0.45$, the O1_Fork-and-Join model still improves the job response time by 30% over the NO model.

Figures 17 and 18 show the job response times under the ReplicatedD_rr and ReplicatedD_shttime data policies, respectively. Also the O1_Fork-and-join model provides the best performance over the other Fork-and-join models.

Figure 17 Job response time for the Fork-and-join job structures under the ReplicatedD_rr data policy

Figure 18 Job response time for the Fork-and-join job structures under the ReplicatedD_shttime data policy

Obviously introducing CPU-I/O operations overlap produces a significant benefit in performance for those jobs with the Fork-and-join structure. Without the CPU-I/O operations overlap, the CPUs assigned to a job remain idle while the I/O requests are in progress and vice versa. The unnecessary idle waiting time on CPU leads to longer mean job response time. Though the NOP I/O model introduces parallel operations in processing I/O request, it still doesn't have the CPU-I/O operations overlap in the structure. Thus the NOP I/O model job structure doesn't perform as well as the job structure with the O1 I/O model. The data in Figure16 indicates that the NOP model provides worse performance than the O1_Fork-and-join job structure. Under the FixedR_c data policies, the NOP job model provides the worst performance result for low

to medium job arrival rates. Figure 19 shows the job response times under FixedR_c data policies for all the Fork-and-join job structures.

Figure 19 Job response time for the Fork-and-join job structures under the FixedR_c data policy

Previous results show that proper level of CPU-I/O operations overlaps as in O1 structure can improve application performance under most of the data policies. But high-level of CPU-I/O operations overlap can cause a negative impact on the application performance under certain data policies. Figure 19 indicates that O2_Fork-and-join job structure provides the worst performance under the FixedR_c data policies when the job arrival rate is high. Our experiment results also show that the O2_Fork-and-joining job structure provides the worst performance result under the FixedR_d_rr and the FixedRW_rr data policies. Figures 20 and 21 show the job response times under these two data policies for all the Fork-and-join job structures.

Figure 20 Job response time for the Fork-and-join job structures under the FixedR_d_rr data policy

Figure 21 Job response time for the Fork-and-join job structures under the FixedRW_rr data policy

Figures 22 and 23 show job response times under the shortest waiting time approach for the FixedR_d and FixedRW data policies.

Figure 22 Job response time for the Fork-and-join job structures under the FixedR_d_shttime data policy

Figure 23 Job response time for the Fork-and-join job structures under the FixedRW_shttime data policy

Normally the O2 I/O model tends to generate large number of I/O operations with small device demand than the other three I/O models and thus causes more contention on the I/O devices when the system load is high. This defect of generating excessive I/O requests has a big impact on the application performance, especially under those data policies that can't adjust the workload on I/O device.

The above results also show that O1_Fork-and-join model provides the best performance in terms of job mean response time under most of the data policies. The only exception is under FixedR_d_rr data policy as indicated in Figure 20. It is interesting to note that under FixedR_d_rr data policy the NO_Fork-and-join model provides the best performance over the other three structures at medium to high job arrival rate. For NO I/O job structures, the FixedR_d_rr data policy is the same as the Replicated_d_rr data policy. Similarly the FixedR_d_shttime data policy is equivalent to Replicated_d_shttime data policy. At low job arrival rate, NO_Fork-and-join job model performs similar to the other three I/O job models. When the job arrival rate is high ($\lambda = 0.45$), NO job model improves performance over other job structures by more than 50%.

Under the FixedR_d_rr data policy, the device for the read operations is fixed once it is selected randomly. The I/O device for the write operations is also selected by the round robin algorithm. The write operation of each task is routed to an I/O device regardless of its status. The experimental result shows that the overall delay accumulated by each write I/O request of a job is more severe than the delay caused by a single read and write operations for a job. If the FixedR_d_rr data policy has to be applied for the parallel application, reducing the number of I/O requests per job should result in significant improvement in performance.

5.1.2 Performance of All Divide-And-Conquer Job Structures

Similar experimental results are obtained for the Divide-and-conquer job structures as in Fork-and-join. Job structure with O1 I/O model (O1_Divide-and-conquer) overall provides the best performance on job mean response time from low to high job arrival rate under all data policies except FixeR_d_rr. Like the results of the Fork-and-join job structure experiment, application performance can be improved by introducing CPU-I/O operation overlap. The performance improvement obtained with the O1_Divide-and-conquer job structure over NO_Divide-and-conquer job structure under ReplicatedD_rr data policy can be as high as 50% when the job arrival rate is 0.45.

Figure 24 Job response time for the Divide-and-conquer job structures under the ReplicatedD_rr data policy

As NOP_Divide-and-conquer job structure has no CPU-I/O overlap, it provides the worst job response time under the ReplicatedC and ReplicatedD_shttime data policies when the job arrival rate is from medium to high. Figures 25 and 26 show the job response times under these two policies.

Figure 25 Job response time for the Divide-and-conquer job structures under the ReplicatedC data policy

Figure 26 Job response time for the Divide-and-conquer job structures under the ReplicatedD_shttime data policy

The O1_Divide-and-conquer job structure provides approximately 20% performance improvement when job arrival is 0.45 under both data policies.

The NOP_Divide-and-conquer job structure provides the worst performance under the FixedRW_shttime data policy. Its performance is almost the same as that of NO I/O model as the I/O device for read and write operation under NOP model is fixed. Figure 27 shows the job response time of all the Divide-and-conquer job structures under the FixedRW_shttime data policy.

Figure 27 Job response time for the Divide-and-conquer job structures under the FixedRW_shttime data policy

The NOP_Divide-and-conquer job structure provides the worst job response time under the FixedR_c data policies at low to medium job arrival rate. Figure 28 shows the job response times for all Divide-and-conquer structures under the FixedR_c data policy. The results are consistent with those obtained in the Fork-and-join job structure under the same data policies.

Figure 28 Job response time for the Divide-and-conquer job structures under the FixedR_c data policy

Figure 29 shows the job response time of NOP_Divide-and-conquer job structure is about 20% to 30% worse compared to what is achieved under the O1-Divide_and_conquer job structure when the job arrival rate is 0.05.

Results obtained in the Fork-and-join job structures indicate that excessive CPU-I/O operation overlap can degrade job response time severely under certain data policies, such as FixedRW_rr and FixedR_d. Figures 29, 30 and 31 show the job response times obtained under these data policies.

Figure 29 Job response time for the Divide-and-conquer job structures under the FixedR_d_rr data policy

Figure 30 Job response time for the Divide-and-conquer job structures under the FixedR_d_shttime data policy

Figure 31 Job response time for the Divide-and-conquer job structures under the FixedRW_rr data policy

There are two reasons for the poor performance of the O2 I/O model under certain data policies. On one hand, data policies have their own limitation on balancing workload on I/O devices as the read operations of these data policies is restricted to be processed on a particular I/O device regardless of the status of selected device. On the other hand, excessive number of small I/O operations generated by O2 job structure tends to cause more serious contention on the I/O devices when system load is high.

The O1_Divide-and-conquer job structure generally achieves best performance under most of the data policies. The only exception is the FixedR_d_rr data policy. Under this data policy, the NO_Divide-and-conquer job structure provides the best performance compared to the other three job structures. Figure 29 shows the job response time for the Divide-and-conquer job structure under the FixedR_d_rr data policy. The

NO_Divide-and-conquer provides more than 50% performance improvement over the second best O1-Divide_and_conquer job structure when the job arrival rate is 0.45. This result is consistent with what is achieved in the NO_Fork-and-join job structure under the same data policy.

5.1.3 Performance of All GE Job Structure

Unlike the results obtained in the Fork-and-join and Divide-and-conquer job structures which indicate O1 I/O model achieves the best performance under most data policies, O1 job structure in the GE model no longer provides such an advantage. The O1_GE/O2_GE job structure provides comparatively better performance under the FixedR_c data policies when the job arrival rate is from low to medium. The NOP_GE job model provides better performance under all the data policies other than the FixedR_c and FixedR_d_rr.

Job response times of all the GE job structures under the FixedR_c are shown in Figures 32. Under the FixedR_c data policy, O1_GE and O2_GE models still provide better performance than the NO_GE and NOP_GE models when the job arrival rate is from low to medium high. The NO_GE and NOP_GE models achieve slightly better performance result than the O1_GE and O2_GE models when the job arrival rate is 0.45.

Figure 32 Job response time for the GE job structures under FixedR_c data policy

NOP_GE model provides the best performance under the Replicated, FixedRW and FixedR_d_shttime data policies. Figures 33, 34 and 35 show the job response times for all the GE job structures under the ReplicatedC, ReplicatedD_rr and ReplicatedD_shttime data policies, respectively.

Figure 33 Job response time for the GE job structures under ReplicatedC data policy

Figure 34 Job response time for the GE job structures under ReplicatedD_rr data policy

Figure 35 Job response time for the GE job structures under ReplicatedD_shttime data policy

It can be observed that introducing parallel I/O operations in the GE model can improve performance over single I/O operation model under the non-FixedRW data policies. But using CPU-I/O operation overlap technique doesn't necessarily bring performance improvement under most data policies except FixedR_c. From O1_GE and O2_GE job structure results, it can be observed that a large number of small service demand I/O requests is generated during job execution. These I/O requests can cause severe contention at the I/O devices when the system load is high. This problem has been observed under the FixedRW and FixedR_d_shttime data policies. Figures 36, 37 and 38 show the job response times of all the GE job models under the FixedRW_rr, FixedRW_shttime, and FixedR_d_shttime data policies, respectively.

Figure 36 Job response time for the GE job structures under the FixedRW_rr data policy

Figure 37 Job response time for the GE job structures under the FixedRW_shttime data policy

Figure 38 Job response time for the GE job structures under the FixedR_d_shttime data policy

The NOP_GE model achieves the best performance under most data policies except FixedR_c and FixedR_d_rr. Under the FixedR_d_rr data policy, the NO_GE job model provides the best performance compared to the other three GE job models. Figure 39 shows job response times for all GE job structures under the FixedR_d_rr data policy.

Figure 39 Job response time for the GE job structures under the FixedR_d_rr data policy

The job response time results shown in Figure 39 indicate that the NO_GE job model provides the best performance among the four GE job structures under the FixedR_d_rr data policy. The NO_GE model offers over 100% better performance over O1_GE or O2_GE when job arrival rate is 0.42. The results achieved under the FixedR_d_rr data policy with the GE job structure is consistent with the results obtained from the experiments of the Fork-and-join and Divide-and-conquer job structures. This implies that reducing the number of I/O operations under the FixedR_d_rr data policy for the GE job structures can significantly improve application performance, especially when the system load is from medium to high.

5.2 Performance Sensitivity to Data Policies

It is observed that ReplicatedC and ReplicatedD_shttime data policies provide the best performance under various parallel job structures. These two policies achieve very close results for each job structure considered in this thesis.

For the job structures with the NO I/O model, FixedRW_rr and ReplicatedD_rr data policies tend to achieve the worst job response time under various NO job structures. Figures 40, 41 and 42 show job response times for various data policies under the NO_Fork-and-join, NO_Divide-and-conquer and NO_GE job structures, respectively.

Figure 40 Job response time for all data policies under the NO_Fork-and-join job structure

Figure 41 Job response time for all data policies under the NO_Divide-and-conquer job structure

Figure 42 Job response time for all data policies under the NO_GE job structure

FixedRW_rr data policy provides the worst performance for the job structures with NO I/O model. ReplicatedC and ReplicatedD_shttime provide the best performance result for the NO I/O job models. Comparable performance results are obtained for ReplicatedC/ReplicatedD_shttime by using the FixedRW and FixedRC data policies.

Under the job structures with NOP, O1 and O2 I/O models, FixedRW_rr and FixedR_d_rr policies provide worst performance result. Figures 43, 44 and 45 show job response times for all data policies under the NOP_Fork-and-join, O1_Fork-and-join and O2_Divide-and-conquer job structures, respectively.

Figure 43 Job response time for all data policies under the NOP_Fork-and-join job structure

Figure 44 Job response time for all data policies under the O1_Fork-and-join job structure

Figure 45 Job response time for all data policies under the O2_Divide-and-conquer job structure

Figures 43, 44 and 45 show that the job response times for FixedR_d_rr and FixedRW_rr under NOP, O1 and O2 I/O model job structures are much higher than the results obtained by the other data policies. The worst job response time generated by FixedR_d_rr data policy is 70% more than the best performance result produced by ReplicatedC/ReplicatedD_rr. It is even 50% worst than the result achieved by ReplicatedD_rr data policy.

For the O1_GE and O2_GE job structures, ReplicatedD_rr data policy provides worst performance among the data policies studied. Figures 46 and 47 show the job response times for all data policies under the O1_GE and O2_GE job structures.

Figure 46 Job response time for all data policies under the O1_GE job structure

Figure 47 Job response time for all data policies under O2_GE job structure

The FixedR_d_rr data policy provides the worst performance for the O1_GE and O2_GE job structures. ReplicatedD_rr and FixedRW_rr data policies provide similar performance that is also much worse than the results obtained by the other data policies under these two job structures. ReplicatedC and ReplicatedD_shttime data policies provide the shortest job response times. Other data policies such as FixedRW, FixedR_c, and FixedR_d_shttime generate comparable performance to ReplicatedC and ReplicatedD_shttime.

The reason that the ReplicatedC and ReplicatedD_shttime data policies provide the best performance among all the job structures is that they take device status into account when the I/O router sends an I/O request to a device. It also has the flexibility to select any device to perform the read and write I/O operations. This helps the system to reduce I/O bottleneck on the performance by balancing workload on the I/O devices and

maximize parallelism level of I/O operations. On the contrary, FixedRW_rr and FixedR_d_rr policies do not consider the device status when routing an I/O request. Besides, read or write I/O operations for these two policies are restricted on one I/O device. For the FixedRW_rr policy, both the read and write operations of all tasks of a job are sent to the chosen device regardless of its status. The device of the read operations in the FixedR_d_rr policy is also fixed once a device is chosen for a job. The write operations of a job are sent to different devices in a round robin fashion without considering the device status. These two policies can introduce significant job response time delay under various job structures when the system load is medium to high. The performance of the other data policies is worse off by 25% to 40% in comparison to the ReplicatedC policy, depending on the job structure.

As a large number of small service demand I/O requests is generated during job execution by O1_GE and O2_GE job structure, FixedRW_rr, FixedR_d_rr and ReplicatedD_rr data policies are unable to balance workload on the I/O devices well in order to reduce I/O performance bottleneck. Thus they demonstrate the worst performance, which is similar to the performance obtained by FixedRW_rr. This is mainly due to the fact that these three data policies distribute I/O requests to I/O devices without taking the device status into account. They fail to balance the workload on the I/O devices when the system load is high.

5.3 Performance Sensitivity to CPU Service Time Variance (CV_d)

All the NO, NOP, O1 and O2 parallel job structures are studied for relative performance of ReplicatedC and FixedR_c data policies as CV_d increases. Figure 48 shows the job response times of all four Fork_and_join job structures under the ReplicatedC data policy. The experiment is done when job arrival rate is 0.45 and CV_a is set to 1. The result shows that performance is not sensitive to CPU service time CV. The service response time increases by less than 2% as CPU service time CV increases from 1 to 10.

Figure 48 Performance sensitivity of CPU service time variance

Further research can be carried out on the other data policies to study the performance sensitivity to the CPU service time variance.

5.4 Performance Sensitivity to Job Arrival Rate Variance (CV_a)

The NO job structure is studied for performance of the ReplicatedC data policies as CV_a increases. Figure 49 shows job response time of NO models as inter-arrival CV increases. The CPU service time variance is set to 1 and the job arrival rate is fixed at 0.45. It can be seen that the arrival CV has a significant impact on the response times of all the NO job structures. The job response time increases from 30 to 260 for the NO-Fork_and_join job structure when the CV_a increases from 1 to 5. The increased variance on job inter-arrival time implies burst arrival of jobs into the system. This affects the system performance in three ways:

1. The burst arrival of jobs means jobs will be waiting in the job queue for a longer time in order to get executed. Thus, the job queuing time increases, which is part of the job response time.
2. Burst arrival jobs also affect the number of processors assigned to a job. As the MAP policy takes into account both the queued jobs and scheduled jobs in the system into account during the processor assignment, most jobs tend to get smaller partitions. Thus it takes longer for a job to complete.
3. The high arrival CV also implies longer gaps between job arrival after the burst of job (The higher the CV the longer the gaps). Thus some of the processors become idle

when they finish a job due to the long gaps while the other jobs get smaller number of processors due to the clustered arrival.

Figure 49 Performance sensitivity to inter-arrival time variance

5.5 Discussion on the Shortest Waiting Time Algorithm

The shortest waiting time algorithm is proposed in this thesis. The shortest service waiting time approach has been applied in the FixedRW, FixedR_d and ReplicatedD data policies. For all the parallel job structures, these data policies with the shortest waiting time data algorithm provide significant performance improvement over the same data policies with the round robin approach under various system load conditions.

Figure 50 Response time comparison for NOP_Divide-and-conquer job structures

Figure 50 shows the job response time for the NOP_Divide-and-conquer structure under all data policies. The performance improvement by the shortest waiting time algorithm can be as high as 70% over FixedRW_rr and FixedR_d_rr data policies. The reason is that the three round robin policies (i.e., FixedRW_rr, FixedR_d_rr, ReplicatedD_rr) do not consider device status when they dispatch new I/O requests. Since shortest waiting time approach takes device status into account, it makes significant performance improvement over round robin approaches.

When the FixedRW, FixedR_d, or ReplicatedR_d policy is selected for a parallel application, the shortest waiting time algorithm of routing an I/O request should be used over the round robin method. With the shortest waiting time approach of the FixedRW, FixedR_D and Replicated_D data policies, they provide comparable job response time under various parallel job structures to ReplicatedC data policy.

Chapter 6

Conclusions and Future Work

6.1 Summary

With a given number of I/O devices, significant performance improvement can be obtained by introducing parallel I/O operations and CPU-I/O overlap (O1 job model) during job execution for the Fork_and_join and Divide_and_conquer parallel job structures under most data policies except FixedR_d_rr. A job which only does I/O operations in parallel tends to take longer time to complete than a job which has CPU-I/O operation overlap in these two job structure categories. Introducing more than one read and one write I/O operations per task in a job under these two job structures might result in poor performance under certain data policies such as the FixedR_d_rr and ReplicatedD_rr data policies.

Due to the special structure of the GE jobs, introducing CPU-I/O operation overlap may cause severe contention on I/O devices and leads to longer job response time though it provides minor performance improvement under certain data policies. Under the GE

parallel job category, NOP job structure normally achieves better performance than most other data policies except FixedR_d_rr.

All job structures with the least number of I/O operations (i.e. NO I/O model) tend to achieve better performance under the FixedR_d_rr data policy.

In terms of data access/distribution policies, ReplicatedC data policy provides the best performance among all the data policies studied. The replication of input data alone doesn't guarantee a good performance as seen by the differences between ReplicatedC and ReplicatedD_rr. Although both data policies allow any read or write request to be performed on any device, the ReplicatedD_rr policy has more restrictions on selecting a device to service an I/O request.

Other data policies that take device status into account when routing an I/O request tend to achieve comparable performance results as ReplicatedC. FixedR_c, FixedRW_shttime and ReplicatedD_shttime all belong to this type of data policy. Even though these data policies do not have replicated input data, they can provide better performance than ReplicatedD_rr policy. Policies such as ReplicatedD_rr, FixedR_d_rr, and FixedRW_rr tend to produce poor performance since they do not consider device status when distributing new I/O requests.

When I/O requests have to be performed on fixed I/O devices, using the shortest waiting time algorithm to select a device can significantly improve performance over the round robin algorithm. Experimental results show that the FixedRW_shttime policy provides superior performance improvement over the FixedRW_rr data policy under all Fork-and-join, Divide-and-conquer and GE parallel job structures with various I/O models.

6.2 Suggestions for future work

When an I/O device is busy upon arrival of a new I/O request, the I/O request is queued in the I/O device local queue in a FCFS manner in this study. Instead of using the FCFS policy, a different queuing discipline such as smallest request first can be adopted as an alternative approach. Further investigation can be carried out to study performance impact of such I/O request queuing policies.

The data policy can also be expanded in future research activities. New data policies such as data fetching from any device but limiting the write operation to a particular device can be further studied.

The MAP processor scheduling policy is used in this study. Further research may take static or AP processor scheduling policy together with I/O management strategies to compare the performance impact of various processor scheduling policies.

Experiments in this study are carried out by simulation. Implementation of I/O management with the MAP processor scheduling policy on a real system could be carried out to verify the promising results shown by simulation.

References

1. S. Dandamudi and H. Yu, "Performance of Adaptive Space Sharing Processor Allocation Policies for Distributed-Memory Multicomputers", *Journal of Parallel and Distributed Computing*, vol. 58, pp. 109-125, 1999.
2. P. Kwong and S. Majumdar, "Study of Data Distribution Strategies for Parallel I/O Management", In *Proceedings of Third International Conference of the ACPC Parallel Databases and Parallel I/O*, pp. 112-123, 1996.
3. J. Rosario and A. Choudhary, "High-Performance I/O for Massively Parallel Computers", *Computer* vol.27, pp. 59-68, 1994.
4. M. A. Kandaswamy, M. Kandemir, A. Choudhary, and D. Bernholdt, "An Experimental Evaluation of I/O Optimizations on Different Applications", *IEEE Transactions on Parallel And Distributed Systems*, vol. 13, no.7, pp. 728-744 2002.
5. S. T. Leutenegger and M.K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies", In *Proc. Of ACM Sigmetrics Conference*, Boulder, pp. 226-236 1990.

6. S. Majumdar, D.L. Eager, and R.B. Bunt, "Scheduling in Multiprogrammed Parallel Systems", In *Proc. ACM Sigmetrics Conf.*, Santa Fe, NM May, pp. 104-113, 1988.
7. L. Shi and S. Olafsson, "Hybrid Equipartitioning Job Scheduling Policies for Parallel Computer System", *Proceedings of the 37th IEEE Conference on Decision and Control*, pp. 1704, 1998.
8. D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K.C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling", *Proceedings of 3rd Workshop on Job Scheduling Strategies for Parallel Processing (Held in Conj. with IPPS'97)*, pp. 1-34, 1997.
9. D. G. Feitelson and L. Rudolph, "Parallel job scheduling: Issues and approaches", *Proceedings of Workshop on Job Scheduling Strategies for Parallel Processing (in conjunction with IPPS'95)*, pp. 1-18, 1995.
10. E. Rosti, E. Smirni, G. Serazzi, L. W. Dowdy, and B. M. Carlson, "Robust partitioning policies for multiprocessor systems", *Perform. Eval.* 19, pp. 141-165, 1994.
11. E. Rosti, E. Smirni, L.W. Dowdy, G. Serazzi, and K. C. Sevcik, "Processor saving scheduling policies for multiprocessor systems", *IEEE Trans. Comput.* 47, no.2, pp. 178-189, 1998.

12. D.G. Feitelson, P.F. Corbett, S. J. Baylor, and Y. Hsu, "Parallel I/O Subsystems in Massively Parallel Supercomputers", *IEEE Parallel and Distributed Technology*, pp. 33-47, 1995.
13. B. Rullman, "Paragon Parallel File System", *External Product Specification, Intel Supercomputer Systems Division*, 1993.
14. P. F. Corbett and D.G. Feitelson, "The Vesta Parallel File System", *ACM Trans. Computer Systems*, vol. 14, no.3, pp. 225-264, 1996.
15. N. Nieuwejaar and D. Kotz, "The Galley Parallel File System", *Parallel Computing*, vol. 23, no.4-5, pp. 447-476, 1997.
16. R. Thakur, E. Lusk, and W. Gropp, "I/O in Parallel Application: The Weakest Link", *The International of Journal of High Performance Computing Applications*, vol. 12, no. 4, pp. 389-395, 1998.
17. S. Toledo and F.G. Gustavson, "The Design and Implementation of SOLAR, a Portable Library for Scalable Out-of-Core Linear Algebra Computations", *Proc. Fourth Ann. Workshop I/O in Parallel and Distributed Systems*, May 1996.
18. R. Thakur, A. Choudhary, R. Bordawekar, S. More, and K. Sivaramakrishna, "PASSION: Optimized I/O for Parallel Applications", *Computer*, vol. 29, no. 6, pp. 70-78, 1996.

19. K.E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett; "Server-Directed Collective I/O in Panda", *Proc Supercomputing 95*, vol. 2, pp. 1580-1599, 1995.
20. D. Kotz; "Disk-directed I/O for MIMD Multiprocessors", *ACM Transactions on Computer Systems*, vol.14, no. 1, pp. 41-74, 1997.
21. R. Jain, K. Somalwar, J. Werth, and J.C. Brown, "Heuristics for Scheduling I/O Operations", *Proc. IEEE Trans. On Parallel & Distributed Systems (IOPADS)*, Atlanta, pp. 68-77, 1997.
22. F. Chen and S. Majumdar, "Performance of Parallel I/O Scheduling Strategies on a Network of Workstations", *Parallel and Distributed Systems, IEEE Transactions*, vol. 13, 2002.
23. P. Kwong and S. Majumdar; "Scheduling of I/O in Multiprogrammed Parallel Systems", *Informatica*, vol. 23, no. 1, pp. 67-76, 1999.
24. P. Crandall, R. Aydt, A. Chien, and D. Reed, "Input/output Characteristics of Scalable Parallel Applications", *Proceedings of Supercomputing*, vol. 2, pp.1643-1673, 1995
25. Message Passing Interface Forum "MPI-2: Extensions to the Message-Passing Interface", 1997.
26. D. A. Patterson, G. Gibson, and R. H. Katz, "A case for Redundant Arrays of Inexpensive Disks (RAID)", *Proc. ACM SIGMOD Conference*, pp. 109-116, 1988.

27. R. Thakur, E. Lusk, and W. Gropp, "I/O in parallel applications: the weakest link", *The International Journal of High Performance Computing Applications*, vol. 12, pp. 389 – 395, 1998
28. J. M. Del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a Two-phase Run-Time Access Strategy", *ACM Computer Architecture News*, vol. 21, no. 5, pp. 31- 38, 1993.
29. A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thankur, "PASSION: Parallel and scalable software for input-output", SCCS-636, ECE Department, NPAC and CASE Center, Syracuse University, 1994.
30. R. Bordawekar, "Implementation of collective I/O in the Intel paragon parallel file system: Initial experiences", *Proceedings of the 11th ACM International Conference*, pp. 20-27, 1997.
31. E. Schikuta and H. Wanek, "Parallel I/O", *The International Journal of High Performance Computing Applications*, vol. 15, no. 2, pp. 162-168, 2001.
32. K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-directed collective I/O in Panda", *Proceedings of Supercomputing*, San Diego, pp. 1580-1599, 1995.
33. A. Tomkins, R. Patterson, and G. Gibson, "Informed multi-process prefetching and caching", *Proceedings of the ACM International Conference on Measurement and*

Modeling of Computer Systems (Sigmetrics), Washington DC, vol. 25, no. 1, pp.100-114, 1997.

34. T. Fuerle, O. Jorns, E. Schikuta, and H. Wanek, "Meta-ViPIOS: Harness distributed I/O resources with ViPIOs", *Journal of Research Computing and Systems*", vol. 4, issue 2, pp. 124-141, 2000.