University of Alberta

Library Release Form

Name of Author: Limin Zhang

Title of Thesis: A User-Level Library for Parallel Disk Input-Output

Degree: Master of Science

Year this Degree Granted: 2003

Limin Zhang
(Address withheld.)
Edmonton, AB
Canada

Date: ____________________

University of Alberta

A User-Level Library for Parallel Disk Input-Output

by

Limin Zhang

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

Department of Computing Science

Edmonton, Alberta
Fall 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A User-Level Library for Parallel Disk Input-Output** submitted by Limin Zhang in partial fulfillment of the requirements for the degree of **Master of Science**.

Paul Lu
Supervisor

Duane Szafron

Peter Minev

Date: ____________________

Abstract

The performance mismatch between storage subsystems and microprocessors in computer systems forms a bottleneck in high-performance computing. The causes for the mismatch are the lower bandwidth and higher latency of hard disk drives as compared to main memory. Three techniques – prefetching, write-behind, and parallelism – are utilized to solve this problem.

In this thesis, we design and implement a user-level Parallel Disk Input-Output Library (PDIOL). The goal of PDIOL is to improve the performance of sequential applications through the parallelization of I/O operations across all workstations in a cluster. Prefetching and write-behind are used in PDIOL as well. We evaluate the performance of PDIOL with a suite of application benchmarks, which include *grep*, *sort* and *bzip2*. From the results, we find that I/O-intensive applications benefit most while computation-intensive applications benefit least, which is consistent with our intuition.

Acknowledgements

First of all, I would like to express the greatest gratitude to my parents. It was because of their unselfish love that I have had the opportunity to be educated. Born in a small country village, it would have been impossible for me to get a good education without their support. Thank you, my beloved parents.

Second, I must express my most sincere thanks to my wife. Without her, it would have been impossible for me to live the happy life that I have been able to lead. Since our marriage in 2000, she has given me so much, far more than I was able to give her back. Henceforth I will do my best to let her live a happier, brighter life. Thank you, my beloved wife.

Third, I wish to thank my supervisor, Dr. Paul Lu, who gave me a great deal of invaluable guidance. All that I have learned under his tutelage has given me a level of confidence that has served me well in this research, and will continue to serve me in the years to come. I cannot adequately stress the importance of the skills I acquired. Furthermore, his consideration and compassion as a human being made this study period that much more comfortable. Thank you, Paul. May your great achievements continue, as I am sure they will.

Fourth, I have to thank my friends, old and new. Without them, I would have led a very lonely and different life.

Finally, I would like to express my gratitude to the Computing Science department and the University of Alberta. I am grateful for the opportunity to study here.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we will first discuss the development trends in microprocessors, hard disks, and networks. The techniques used to make up for the increasing mismatch between disk I/O systems and microprocessors will then be presented. Finally, we will introduce a user-level Library for Parallel Disk Input-Output (PDIOL), that provides a UNIX I/O Application Programmers Interface (API) for accessing files striped across a workstation cluster.

1.1 Motivation

The mismatch between the storage subsystem and CPU in computers forms a bottleneck, especially in disk I/O-intensive applications, such as text retrieval systems. This mismatch has been increasing ever since the first appearance of computers. Much of the effort in storage design has been devoted to finding ways of masking this enormous discrepancy in bandwidth and access latency. The demand for large storage capacity and high performance in more and more data-intensive applications exacerbates the problem. The introduction of the high-speed switched local network creates challenges and opportunities for storage systems. In the following paragraphs, the development of main hardware components of the computer system are described.

1. Microprocessors

   The improvements in microprocessors conform to Moore's Law, i.e., the transistor density roughly doubles every 18 months. Table 1.1 shows the number of transistors in one microprocessor, and the processing power of various types since the 1970s. Moore's Law has been maintained and still holds true today. It is expected to continue to be valid at least through the end of this decade.

CPU Model	Year of Introduction	Number of transistors	Millions of Instructions per Second
4004	1971	2,300	0.06
8008	1972	3,500	0.06
8080	1974	4,500	0.64
8086	1978	29,000	0.33
286	1982	134,000	0.90
386	1985	275,000	5.00
486	1989	1,200,000	20.00
Pentium	1993	3,100,000	60.00
Pentium Pro	1995	5,500,000	200.00
Pentium II	1997	7,500,000	300.00
Pentium III	1999	28,100,000	733.00
Pentium 4	2000	42,000,000	155.00
Itanium 2	2002	291,000,000	6000.00

Table 1.1: **The trend in the number of transistors and processing power of Intel microprocessors**. The table is adapted from [3].

2. Disk storage systems

   With the developments in hard disk drives and hard disk drive connections, the performance of disk storage systems in terms of capacity has improved continually over the last few decades. From the 10MB hard disks installed in IBM/XTs in the early 1980s to the current 120GB hard disks, the capacity has expanded 12,000 times – a factor which is even greater than the proportional improvement in microprocessors over the same period. However, the bandwidth and (especially) the data access latency, which are limited by mechanical components – notably spindle speed and read/write head movement – has not improved much. Table 1.2 shows the most common PC spindle speeds and the average rotational latency in early 2000 [5]. From 3,600 RPM in the early 1980s to the currently most popular 7,200 RPM, the spindle speed has only doubled. Even for the high-end SCSI disks with 15,500 RPM, the improvement is only 5 times, compared with early 3,600 RPM hard disks. The average rotational latency of a 15,550 RPM disk is: 60 seconds / 15500 / 2 = 1.93 milli seconds. Compared with the access latency of main memory, which is tens of nanoseconds [29], the mismatch is huge, without even taking into account the speed of microprocessors. Furthermore, disk head movement latency brings more overhead. The bad news is that the performance of read/write heads has not improved a great deal either.

   The mechanically-related overhead causes disk performance to lag far behind micro-

Spindle Speed (RPM)	Average Latency Half Rotational Time(ms)	Seek Time (ms)	Typical Applications
5,400	5.6	12.0	Low-end IDE/ATA
7,200	4.2	9.0	High-end IDE/ATA, Low-end SCSI
10,000	3.0	5.2	High-end SCSI
12,000	2.5	5.0	High-end SCSI
15,000	2.0	3.6	Top-of-the-line SCSI

Table 1.2: **The performance of contemporary hard disks [5]**.

processors – a situation which is expected to continue in the current situation.

3. Network systems

   Switched local area networks, such as Myrinet [8], provide high bandwidth in the hundreds of megabytes/second range [29] – a rate which can scale further with the number of machines in a network systems. Furthermore, new network protocols with less computational overhead and lower latency (less than a few microseconds), enable machines to cooperate to perform finer granularity work than previously. For example, 178 Myrinet clusters were listed in the June 2003 TOP500 List [23].

   Such innovations bring about new challenges for storage systems. For example, only a few workstations processing multimedia data can overflow a central file system.

4. Demand from disk I/O-intensive applications

   I/O-intensive applications, such as satellite data processing, medical image databases, high-performance relational databases, data mining, and detailed scientific modeling of complex phenomena, not only require a huge capacity to store data, but also have a need for high-access speed. Disk I/O systems must be designed to accommodate such requirements.

1.2 Techniques

Amdahl's Law tells us that without tracking computing performance, disk I/O systems will limit the improvements to high-performance I/O-intensive computation. There are several techniques available to address this problem:

1. Prefetching and Write-behind

Prefetching is a technique to delegate a proxy, such as an operating system, to load data prior to processing it and write-behind is a technique to delegate a proxy to store data after processing it. Through prefetching and write-behind, applications can overlap computation time with I/O waiting time, and pipeline I/O operations.

2. Caching and Buffering

   Data buffered in caches can be reused without accessing disks. However, this approach depends on the locality of data access. For applications with poor locality, buffering brings little or no benefit because few data are reused. However, when combined with prefetching, caches can be used to buffer the data that will be used and thus play an important role, even in applications with poor locality. The prefetching and cache approaches need to be be considered together.

3. Parallelism through Disk Striping

   Through disk striping, a body of data is divided into blocks and the data blocks are spread across several partitions on several hard disks. Disk I/O bandwidth can be improved by utilizing the aggregate bandwidth of multiple disks. At the same time, the decreasing price of disks makes it cost effective to utilize parallel disk systems. However, parallel disk systems are more complex than single disk systems.

1.3 Parallel Disk Input-Output Library

With the development of commodity computer hardware and open-source systems, it has become cost-effective to utilize workstation cluster systems to implement a parallel disk system [14, 10]. In this thesis, we implement a user-level library for parallel disk input-output (PDIOL) on a workstation cluster.

The contributions of the thesis include:

1. Implement a user-level library based on the UNIX API for parallel disk input-output;

2. Use a windows-based prefetching algorithm and write-behind technique in the library;

3. Experiment with benchmarking the performance of network and disk drives, and benchmarking and tuning the performance of PDIOL.

The contents of this thesis are as follows: The background knowledge and related research will be discussed in Chapter 2. The design and implementation of the PDIOL will be described in Chapter 3. In Chapter 4, the microbenchmark for PDIOL and several benchmarks of real applications will be described. Chapter 5 serves as the conclusion and will discuss possible directions of future research. The usage of the PDIOL library and APIs will be described in Appendix A and Appendix B. During the development of the library, an auxiliary tool called Thread Log Viewer (TLV) was developed to aid with tuning performance. It will be described in Appendix C.

Chapter 2

Background and Related Work

The gap between the performance of microprocessors and disk I/O systems continues to widen [12]. Improvements in microprocessors will result in marginal performance improvements in overall system performance if there are no accompanying improvements in disk I/O systems. One of the most important solutions to these problems is that of distributing data, for example, striping, across parallel disks.

In order to improve parallel disk storage system performance, much research has been done in the following areas.

1. Increasing aggregate bandwidth from disks through parallel disk systems;

2. Exploring I/O concurrency through prefetching, combined with cache management and effective scheduling of data-access techniques;

3. Exploiting the parallelism of parallel storage systems with parallel file systems.

In this chapter, some background material will be presented and some related research will be discussed.

2.1 Parallel Disk Systems

Like multi-processors, multiple disks can improve capacity, performance, and availability in storage systems. There are two main kinds of disk parallelism: Redundant Array of Independent (or Inexpensive) Disks (RAID) and network storage.

2.1.1 Redundant Array of Independent Disks (RAID)

RAID [31] began as a research project at the University of California, Berkeley in the 1980s. It utilizes parallelism between multiple disks to improve aggregate I/O performance.

By striping data across parallel disks, RAID increases the bandwidth of disk systems. At the same time, RAID improves reliability and availability through redundancy. RAID uses device virtualization to represent internal disks as a larger virtual drive, and the server and application view the RAID system as a single disk system. There are seven configurations of RAID, ranging from RAID 0 to RAID 6, which differ in interleaving granularity, their algorithm for redundancy, and their placement of redundant data. The 7 RAID levels are shown in Figure 2.1.

Although the different RAID levels are optimal for different applications, RAID 5 is one of the most common implementations of RAID in the market. By computing a parity block and striping data across an array of disks, RAID 5 improves the bandwidth and reliability of storage systems.

However, RAID systems improve performance primarily by increasing throughput via multiple read-write operations. For applications with a throughput bottleneck, RAID improves performance. Except for RAID level 0, which utilizes only the striping technique, the other RAID levels may increase the latency of writing due to the need for more writing operations for redundancy, and the computation overhead for parity data generation.

Another shortcoming of RAID is that the file server which manages a RAID system may form a bottleneck. If the speed-up of the disk array exceeds the processing power of the server, the high performance of RAID cannot be delivered to the client side. It can be said that RAID is moderately scalable. For RAID systems, the striping unit, which is the maximum amount of consecutive data assigned to a single disk, is an important characteristic. Much research [16, 36, 15] has been done on how to select appropriate striping units, according to the workload.

By striping data across workstation clusters, PDIOL implements a kind of RAID 0 library and utilizes the aggregate bandwidth of multiple disks. Note that PDIOL does not currently support redundancy.

2.1.2 Network Storage

The demand for high-performance storage and the introduction of high-speed switched local area networks bring about the use of network storage. By comparing the network speed and direct-attached disk speed, we can draw the conclusion that by distributing data across a bundle of storage servers connected on a network, storage system performance can be improved. At the same time, scalability and reliability can also be enhanced.

The architecture of the network storage is shown in Figure 2.2.

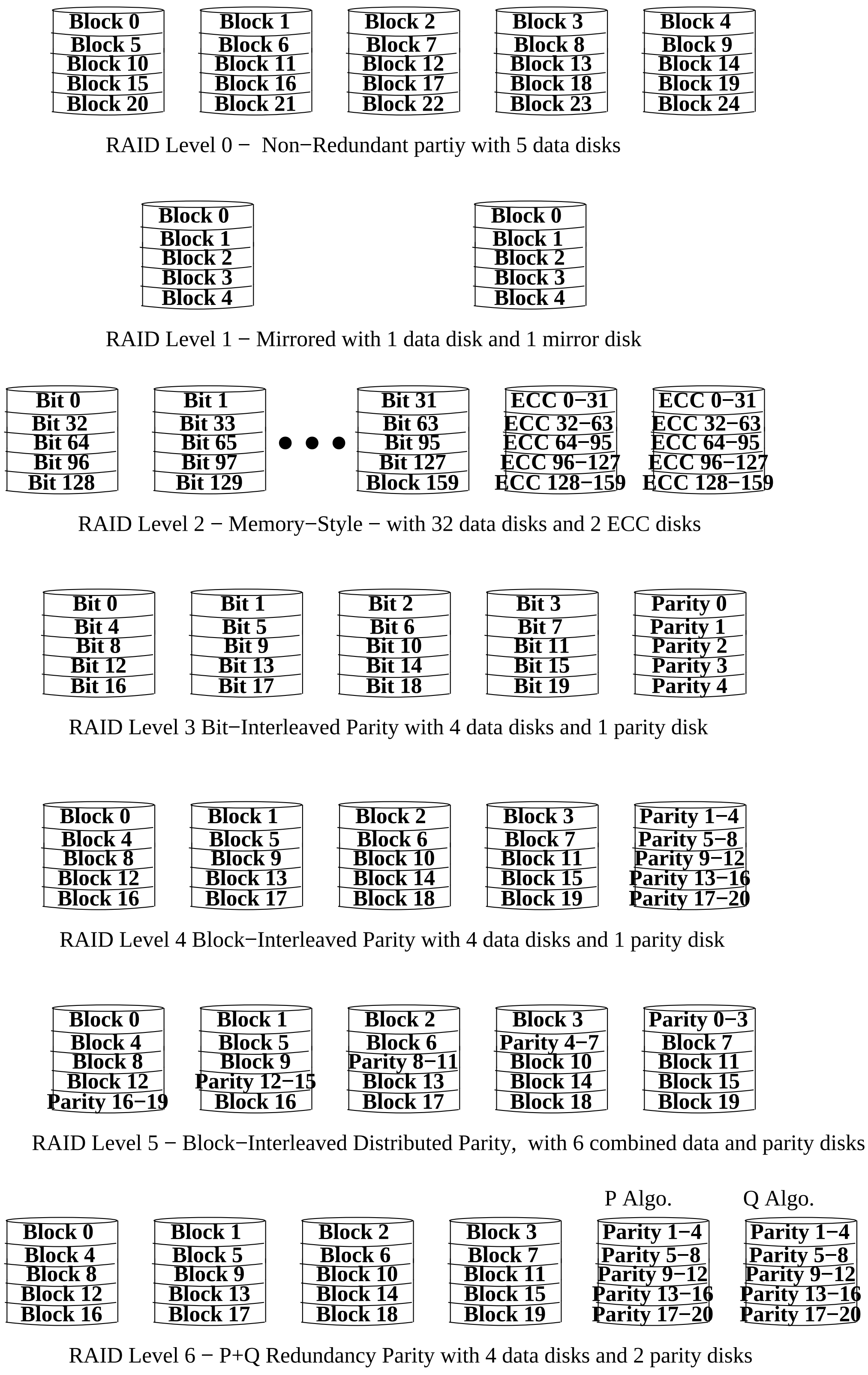

Block 0 Block 5 Block 10 Block 15 Block 20
Block 1 Block 6 Block 11 Block 16 Block 21
Block 2 Block 7 Block 12 Block 17 Block 22
Block 3 Block 8 Block 13 Block 18 Block 23
Block 4 Block 9 Block 14 Block 19 Block 24
RAID Level 0 − Non−Redundant partiy with 5 data disks
Block 0 Block 1 Block 2 Block 3 Block 4
Block 0 Block 1 Block 2 Block 3 Block 4
RAID Level 1 − Mirrored with 1 data disk and 1 mirror disk
Bit 0 Bit 32 Bit 64 Bit 96 Bit 128
Bit 1 Bit 33 Bit 65 Bit 97 Bit 129
● ● ●
Bit 31 Bit 63 Bit 95 Bit 127 Block 159
ECC 0−31 ECC 32−63 ECC 64−95 ECC 96−127 ECC 128−159
ECC 0−31 ECC 32−63 ECC 64−95 ECC 96−127 ECC 128−159
RAID Level 2 − Memory−Style − with 32 data disks and 2 ECC disks
Bit 0 Bit 4 Bit 8 Bit 12 Bit 16
Bit 1 Bit 5 Bit 9 Bit 13 Bit 17
Bit 2 Bit 6 Bit 10 Bit 14 Bit 18
Bit 3 Bit 7 Bit 11 Bit 15 Bit 19
Parity 0 Parity 1 Parity 2 Parity 3 Parity 4
RAID Level 3 Bit−Interleaved Parity with 4 data disks and 1 parity disk
Block 0 Block 4 Block 8 Block 12 Block 16
Block 1 Block 5 Block 9 Block 13 Block 17
Block 2 Block 6 Block 10 Block 14 Block 18
Block 3 Block 7 Block 11 Block 15 Block 19
Parity 1−4 Parity 5−8 Parity 9−12 Parity 13−16 Parity 17−20
RAID Level 4 Block−Interleaved Parity with 4 data disks and 1 parity disk
Block 0 Block 4 Block 8 Block 12 Parity 16−19
Block 1 Block 5 Block 9 Parity 12−15 Block 16
Block 2 Block 6 Parity 8−11 Block 13 Block 17
Block 3 Parity 4−7 Block 10 Block 14 Block 18
Parity 0−3 Block 7 Block 11 Block 15 Block 19
RAID Level 5 − Block−Interleaved Distributed Parity, with 6 combined data and parity disks
P Algo.
Q Algo.
Block 0 Block 4 Block 8 Block 12 Block 16
Block 1 Block 5 Block 9 Block 13 Block 17
Block 2 Block 6 Block 10 Block 14 Block 18
Block 3 Block 7 Block 11 Block 15 Block 19
Parity 1−4 Parity 5−8 Parity 9−12 Parity 13−16 Parity 17−20
Parity 1−4 Parity 5−8 Parity 9−12 Parity 13−16 Parity 17−20
RAID Level 6 − P+Q Redundancy Parity with 4 data disks and 2 parity disks

Figure 2.1: **RAID level 0 - RAID level 6 data distribution.**

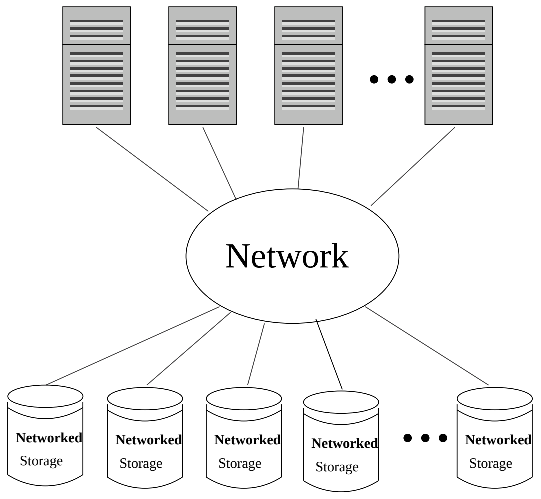


Figure 2.2: **Architecture for the network storage.**

The Storage Area Network (SAN) and Network Attached Storage (NAS) [20, 19, 21] are two approaches to network storage. SAN is a network of disks that interact with a file server via SCSI operations across a network and NAS is a set of disk servers that interact with other services via file system operation over a network. The difference between them is that NAS systems offer file system functionality while SAN systems do not. SAN systems are more like traditional attached disks, while NAS systems resemble a traditional local file system. Usually, NAS systems are built upon SAN systems; however, it is worth noting that there is considerable cross-development between SAN and NAS.

Compared with direct-attached storage (DAS) systems, where disks are attached to a single server not via a network, network storage has the following advantages:

1. Scalability. All the storage servers in a network storage system are connected through a Local Area Network (LAN). In theory, there is no limit on the number of storage servers. A traditional DAS system, however, has a limit on the number of devices that can be attached to a typical server (i.e., limited number of SCSI ports).

2. Availability. The availability in a network storage system is inherent. With redundant storage servers, network storage can maintain high availability, since a failed server does not prevent other servers from accessing the same disk across a network.

The main disadvantage of network storage systems is their complexity. In addition to the complexity of the data network, network storage systems have to address problems such as, data integrity after faults and data consistency among storage servers. Much research has been done on network storage systems in terms of performance improvement.

Gibson *et al.* [20] proposed a Network-Attached Secure Disk (NASD) storage architecture to shorten the data transfer path. The idea of utilizing the aggregate processing power of networked disks was proposed [34]. With on-drive processing and software downloadability, the disks can execute application-level processing and reduce data traffic. This technique is especially useful for some basic data processing tasks. A kernel-based network memory system was also developed [6]. It manages the memory of cluster nodes as a shared distributed page cache in the kernel, and improves data-intensive applications by replacing disk I/O operations with memory-to-memory transfer across a high-speed network.

Implementing a storage network on a workstation cluster is also possible. A workstation cluster-based storage network has several advantages: maximizing the utilization of the disks in a workstation cluster; using the main memory in the workstations as caches and replacing disk I/O operations with memory copy; and improving the utilization of idle workstations in the cluster. The increasing network bandwidth and new protocols [8] make this approach feasible.

2.2 Prefetch and Cache Management

Through prefetching, the latency of disk data access time can be overlapped with computation. Especially for sequential applications, it is difficult to utilize the parallelism of parallel disks without prefetching, since there is always only one I/O operation in progress. Since prefetched data must be stored in caches, these two techniques should be considered together. With parallel disk systems, cache management becomes more complex [24].

In theory, prefetching can be done on three levels:

1. The application level. With asynchronous I/O commands, applications can prefetch data. The advantage of this approach is that only the data needed will be prefetched. However, this approach creates burdens for the application developer.

2. The file system level. Almost all file systems provide some kind of prefetching algorithm. However, without accurate access-patterns information, it is rather difficult to prefetch data precisely. How to provide an interface to pass application access-pattern information to file systems and how to utilize such information remain important research topics.

3. The storage system level. The disks have no knowledge of the data distribution. One

case is that adjacent blocks on the storage system do not necessarily belong to the same file or object. Thus, the storage system cannot prefetch data for sequential access, let alone for a more complex access pattern.

It is critical for file systems to know the access patterns of applications in order to prefetch data precisely. In the following section, we will review some research on how file systems utilize access patterns to prefetch data, and how applications pass such access-pattern information to the file system below.

There are three ways for file systems to obtain access patterns:

1. Prediction from past access patterns.

   All file systems have some mechanism for deducing access patterns from past data accesses, and for predicting future accesses. This approach is completely transparent to application developers. However, most of them focus only on sequential prefetching. For example, if block N and N+1 of a file have been referenced, block N+2 will be prefetched. Although file systems which predict more complex access patterns exist [26, 22], without hints from applications, these system may not prefetch data correctly and may reduce performance.

2. Application-provided access patterns.

   Application developers may know the prefetch access patterns of their programs in detail. Through hints, such information can be passed to file systems. With the advance knowledge of future reference, Cao *et al*. [13] proposed a two-level cache management strategy. In the kernel, an algorithm called LRU-SP (Least-Recently-Used with Swapping and Placeholder) is used to manage the cache among multiple processes. With a policy named controlled aggressive policy, each process uses application-specific information to manage its own caching and prefetching. A prototype file system, Application Controlled File System (ACFS), utilizing this algorithm was implemented. However, this algorithm does not address the issue of unbalanced disk workload.

   In Informed Prefetching [33], a cost-benefit analysis model is used to allocate buffers among three competing demands: prefetching hinted blocks, caching hinted blocks for reuse, and caching recently-used data for unhinted accesses. Differences between Informed Prefetching and ACFS include: (1) Different levels of abstractions. The hints in ACFS specify the to-be-prefetched blocks and files, while hints in Informed

Prefetching specify the access pattern of files. (2) Informed Prefetching addresses the problem of how far ahead a prefetch should be done, with respect to a certain disk workload, while ACFS focuses on prefetching and caching according to the data block assess sequence.

Collaboration among researchers [13, 33, 24] has brought about more advanced algorithms. In Kimbrel *et al.* [24], four algorithms are compared under the multiple disk environment, in which a single process is running with full advance knowledge of references. The first three algorithms presented are: fixed horizon [13], aggressive prefetching, and reverse aggressive prefetching [13]. The reverse aggressive prefetching algorithm works best under different situations but needs to know the complete access sequence in advance. Forestall, a new algorithm which combines the above three algorithms is then proposed. The idea behind forestall is that, through estimating the point to prefetch, the shortcomings of aggressive prefetching and fixed horizon are overcome.

3. Compiler-found access patterns

   Mowry *et al.* [28] used compiler technology to statically derive the file access pattern especially for loops and issue prefetching commands, accordingly. The compiler, OS, and a run-time library cooperate to make prefetching work. A. Brown *et al.* [11] improved the earlier work [28] by letting applications release pages pro-actively, which is better cache management.

Yang *et al.* [39] propose a method to dynamically prefetch through a prefetching thread. Their method converts original source code into a computation thread, which executes all the instructions of the original program, and a prefetch thread, which executes only the disk access-related instructions. At run-time, the prefetch thread runs far ahead of the computation thread, and prefetches all data blocks into the cache before the computation thread. Although limitations exist because of the complexity of data-dependency analysis, the idea is distinct from that of prefetching with knowledge of access patterns.

2.3 Parallel File Systems

In terms of level of abstraction, PDIOL is most comparable to other parallel file system, so we make more explicit comparisons between PDIOL and the related work in this section.

Through parallel file systems, the parallelism of network storage can be exposed and utilized. From the simplest file sharing system, Network File System (NFS), to more complex systems, such as VaxClusters [25], there are many kinds of parallel file systems. With respect to granularity, VaxClusters allows block-level striping in a cluster environment, while the mount unit of NFS is a directory and files cannot span over multiple workstations. Although much research [14, 10, 7] on parallel file systems is in progress, there are no standard, widely available parallel file systems on Linux operating systems. Here, we discuss several typical parallel file systems.

2.3.1 Network File System (NFS)

The NFS file system is the most common distributed file system in the UNIX world. NFS uses a basic client-server approach, and has the following limitations for high-performance computation.

1. Coarse granularity. Although NFS distributes a file system across many servers by partitioning directory trees, a file is always at one server, therefore a hot spot can develop for a single file.

2. Weak cache consistency. In NFS, the client is responsible for caching data. No consistency is guaranteed as NFS does not define strict semantics for the cache consistency among clients.

Unlike NFS, PDIOL always spans a single logical file across multiple cluster nodes.

2.3.2 Serverless Network File System (xFS)

xFS [7], a serverless network file systems has been proposed. Without a centralized server, xFS tries to break the bottleneck of traditional centralized file servers. The techniques used in xFS include: distribution of the control of file system metadata control over systems; implementation of a software RAID across storage servers; and utilization of cooperative caching to form a global file cache.

The difference between xFS and PDIOL is that while xFS provides a complete file system, PDIOL provides a simple user-level library to utilize parallel disks over a workstation cluster.

2.3.3 Parallel Virtual File System (PVFS)

PVFS [14] is a parallel file system for Linux clusters, implemented by Argonne National Laboratory. Its goal is to provide high-speed file data access for parallel applications and a

	NFS	xFS	PVFS	GPFS	PDIOL
Interface	OS	OS	OS	OS	User-level API
Deployment	Inside OS	Inside OS	Inside OS	Inside OS	User-level library
Striping File	N	Y	Y	Y	Y
Meta Data	N/A	Distributed	Centralized	Distributed	N/A
Support concu-rrent process	Y	Y	Y	Y	N
Shared Disk	N	N	N	Y	N

Table 2.1: **Comparison of NFS, xFS, PVFS, GPFS, and PDIOL.**

cluster-wide consistent name space. It is designed as a client-server system with multiple servers. The client interacts with a PVFS server through the PVFS library. In this respect, PDIOL is similar to PVFS. However, the goal of PVFS is to improve the access speed for parallel applications through a concurrent read/write interface, while PDIOL is designed to improve the performance for sequential applications. In PVFS, a single manager is responsible for the PVFS file metadata, which describe the characteristics of a file, such as the physical distribution of the file data. The distribution and stripe size are specific to certain PVFS files, providing more flexibility than PDIOL.

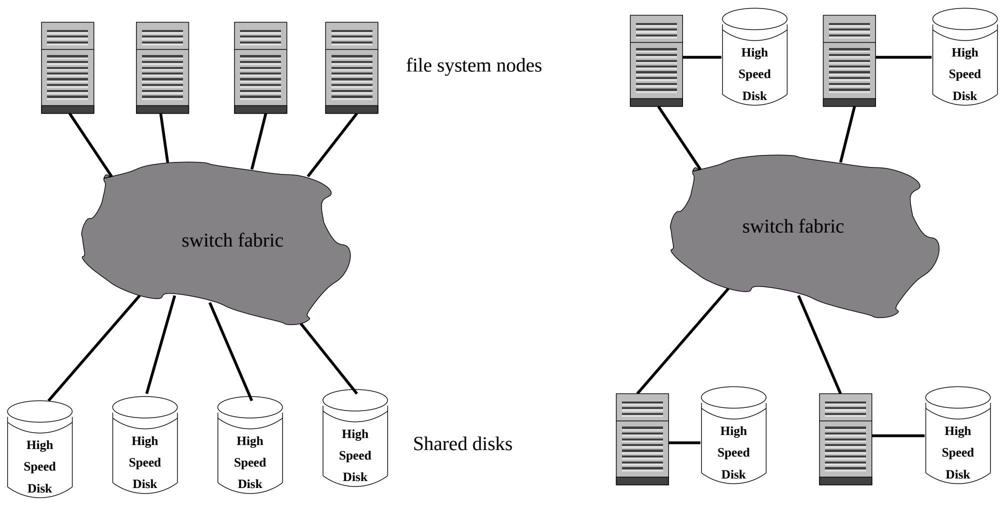
2.3.4 General Parallel File System (GPFS)

GPFS [35] for Linux is a shared-disk file system (i.e. SAN), which has been run on IBM RS/6000 SP and provides an interface as close as possible to the standard UNIX I/O file interface. It uses a data-striping technique to distribute data across multiple disks and multiple nodes, and to improve I/O performance through prefetching and write-behind. With distributed locks and centralized management for consistency of file data and metadata consistency, it achieves data consistency among multiple nodes. The difference between GPFS and PDIOL is that the nodes in the GPFS file system share common disk pools connected through a SAN, and every node has equal access to every disk, whereas every node of PDIOL has its own local file system and cannot access disks of other nodes directly.

2.3.5 Concluding Remarks

The differences between the above file systems and the PDIOL library are shown in Figure 2.3 and Table 2.1

Although PDIOL is implemented at user-level and lacks some flexibility and functionalities, the user-level property makes the implementation and use of PDIOL simple, as it



(a) Network storage with shared disk model.
GPFS uses this model

(b) Cluster of workstations with directly attached disks.
NFS, xFS, PVFS and PDIOL use this model.

Figure 2.3: **Comparison of architectures of GPFS, PVFS, xFS, and PDIOL.**

does not require kernel modification.

Chapter 3

Design and Implementation

3.1 Overview

PDIOL is a user-level library in C for improving the I/O performance of sequential applications. It improves I/O performance in two ways: improving effective bandwidth by striping data sets across a workstation cluster, and reducing disk I/O operations by utilizing the buffers of multiple computers. With prefetch and write-behind, PDIOL delivers high performance to applications.

The main design goals for PDIOL are to:

1. Improve disk I/O bandwidth by striping data across parallel disks in a workstation cluster;
2. Decrease disk I/O latency with prefetching, caching, and write-behind;
3. Provide good usability with a UNIX I/O API wrapper. (Appendix B).

3.2 Architecture

PDIOL consists of two parts – a PDIOL library and PDIOL agents (local and remote) – which are shown in Figure 3.1. The PDIOL library implements routines for PDIOL cache management and communication. PDIOL applications need to be compiled and linked with the PDIOL library in order to utilize the prefetch ability of PDIOL. The data paths between remote disks and PDIOL applications are shown in Figure 3.2.

The PDIOL cache is a large partition of memory allocated in user-space, and stores data read from remote agents. The size of the PDIOL cache is currently configured to be 256M bytes, which is one half of the physical memory in one node of our test platform. The

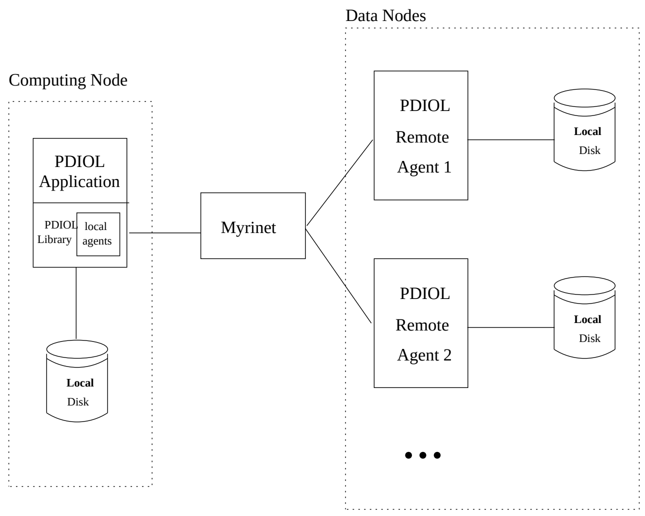


Figure 3.1: **The architecture of PDIOL.**

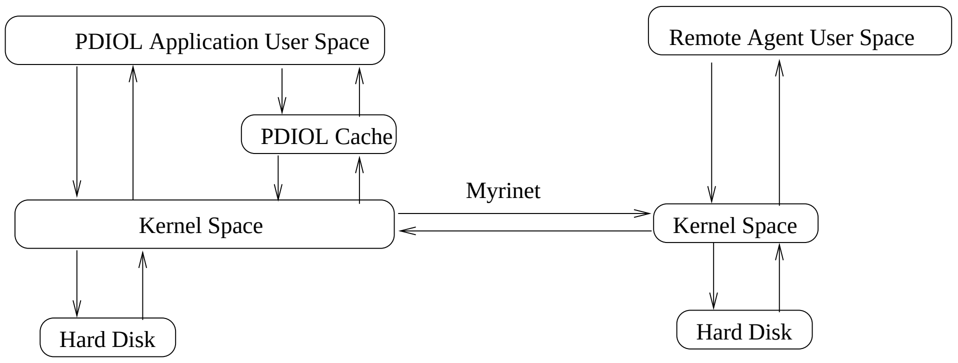


Figure 3.2: **The PDIOL data path.**

PDIOL cache is divided into fixed-size cache pages, each with a size of 64KB, which are managed via a cache page table. We will discuss the cache pages in Section 3.4.2. Only files located on remote disks are cached in the PDIOL cache. Data of the files on the local disk are managed by the local operating system's file system. When accessing files on remote disks, PDIOL applications call PDIOL library functions which, in turn, send requests to remote agents. Upon arrival, fetched data are stored in the PDIOL cache. When accessing files on the local disk, PDIOL applications bypass the PDIOL cache by calling the standard I/O APIs directly.

In the following discussion, we will use a sample PDIOL logical file *mydir/mypdiolfile* with PDIOL applications operating on it.

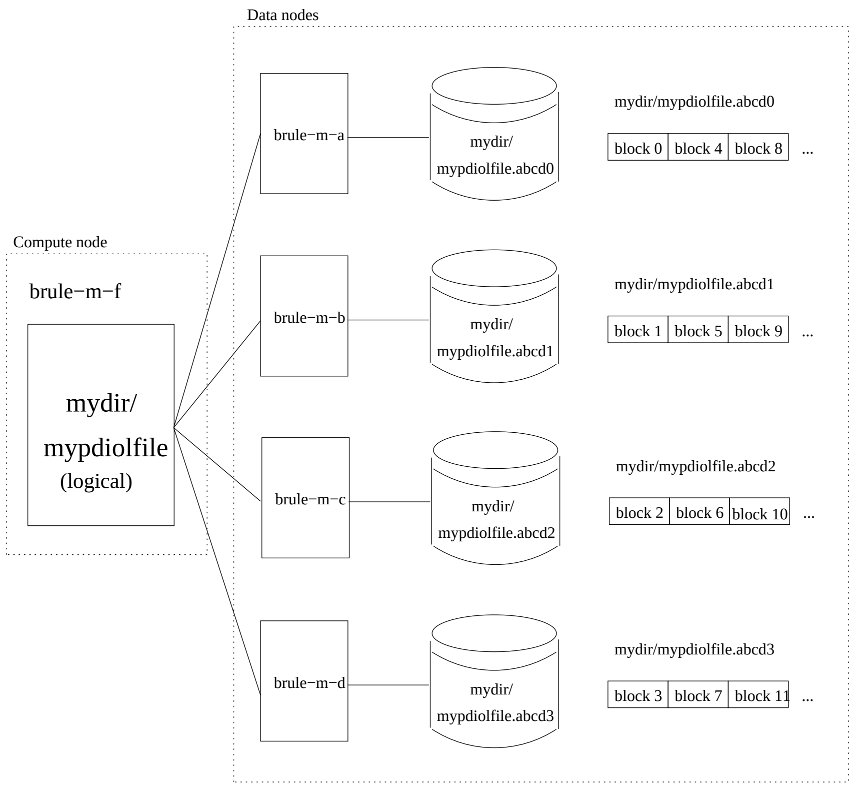


Figure 3.3: **PDIOL data distribution and naming scheme.**

3.3 PDIOL File Structure

A PDIOL logical file is distributed (i.e., striped), therefore a PDIOL logical file consists of multiple local files, as shown in Figure 3.3. The partitioning of a logical file into many physical files is analogous to RAID 0 systems. The number of sub-files is determined by the PDIOL configuration file *$HOME/PDIOLconf*. The data distribution, naming scheme, and PDIOL logical file property are discussed below. We assume that the computing node is *brule-m-f* and the data nodes are *brule-m-a*, *brule-m-b*, *brule-m-c*, and *brule-m-d*.

3.3.1 Data Distribution

The data in a PDIOL logical file are distributed cyclically to multiple sub-files residing on different disks, as shown in Figure 3.3. With this distribution scheme, sequential access, which occurs in most cases, can obtain maximal concurrency. The reason for this is that the sequential access pattern can prefetch and pipeline the data access to multiple disks, when data are distributed over disks in this way. The block size of distribution affects concurrency: a bigger block size results in less concurrency but also incurs less per-block cache management overhead. In one extreme case, if the block size is the same as the size of a file, sequential access will get no concurrency. In our implementation, we used 64KB as the block size – a value which is the default socket send/receive buffer size – and had reasonable performance results. The PDIOL cache page size is the same as the distribution block size, which makes mapping the PDIOL cache pages to disks easy to do.

3.3.2 Naming Scheme

In order to simplify the implementation, all the sub-files of one PDIOL logical file reside in the same local-file-system paths, with the same name as the PDIOL logical file. In the example shown in Figure 3.3, the name of the PDIOL logical file is *mypdiolfile* in the directory *mydir/*. The data nodes consists of *brule-m-a*, *brule-m-b*, *brule-m-c* and *brule-m-d*. The sub-files are named *mypdiolfile.abcd0*, *mypdiolfile.abcd1*, *mypdiolfile.abcd2* and *mypdiolfile.abcd3*, and reside in *mydir/* of *brule-m-a* to *brule-m-d*, respectively. Since the names of sub-files encode information about the PDIOL logical file, remote disks, and distribution sequence, there will be no name conflict for different PDIOL logical files and distributions. Thus, this naming scheme makes it convenient to debug the PDIOL library on a single computer. To open the file on *brule-m-f*, the application would call, for example, *pdiol_open("mydir/mypdiolfile", O_RDONLY)*

3.3.3 PDIOL Logical File Property

In PDIOL, there is no explicit metadata associated with each PDIOL logical file, which makes PDIOL easy to implement. The PDIOL logical file has the same property – such as ownership and access time – as the first sub-file. The size of a PDIOL logical file is the sum of the sizes of all sub-files.

3.4 In-Memory Data Structure and Cache Management

In this section, several data structures, with which PDIOL files and PDIOL cache are managed, will be shown in detail. The algorithm for cache management is given as well.

3.4.1 In-Memory Data Structure

The main in-memory data structures of the PDIOL library are (as shown in Figure 3.4):

1. The PDIOL logical file table. The table stores PDIOL logical file properties, such as file path, size, and file position.

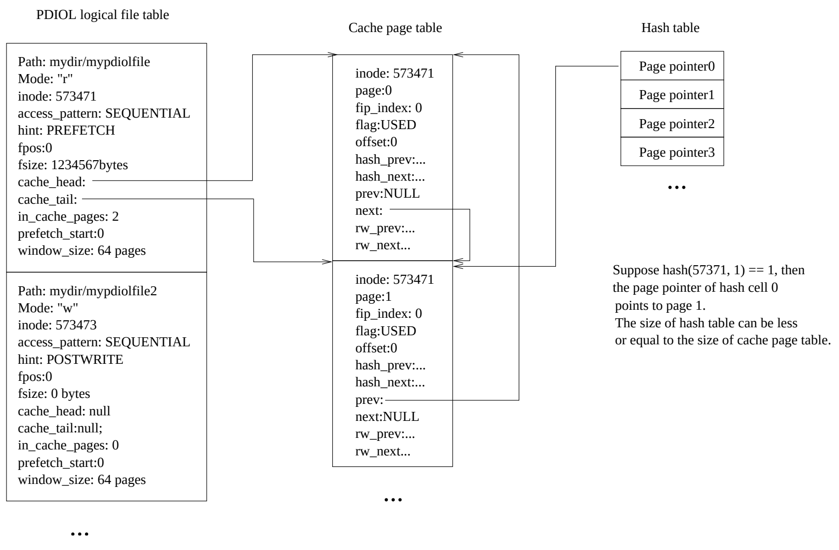


Figure 3.4: **PDIOL logical file table, cache page table, and hash table data structure.**

2. The cache page table. The page table for the PDIOL cache pages.

3. The hash table. The hash table provides a fast way to find pages through file descriptor and page index. The cell of the hash table stores the page pointer pointing to a page whose hash value is the index of the hash table cell. The hash conflicts are handled by chaining.

Since remote agents only read/write according to the requests from the computing node and do not have a PDIOL cache management mechanism, the cache page table and hash table exist only on the computing node, and the Linux buffer cache performs the cache management on the remote nodes [9]. The PDIOL logical file table exists on all remote I/O nodes and the computing node.

3.4.2 Cache Management

Prefetching and caching should be considered together because prefetching must cache prefetched data. Several page replacement algorithms exist: First-In, First-Out (FIFO), Second Chance, Clock Page, and Least Recently Used (LRU) page replacement algorithm [37].

In PDIOL, we selected the Second Chance algorithm because of its small overhead and easy implementation. Since our experiments with PDIOL emphasizes flexibility and diversity with respect to prefetching, instead of cache management, our experiment uses only the Second Chance Replacement algorithm, with which we achieve satisfactory results. If

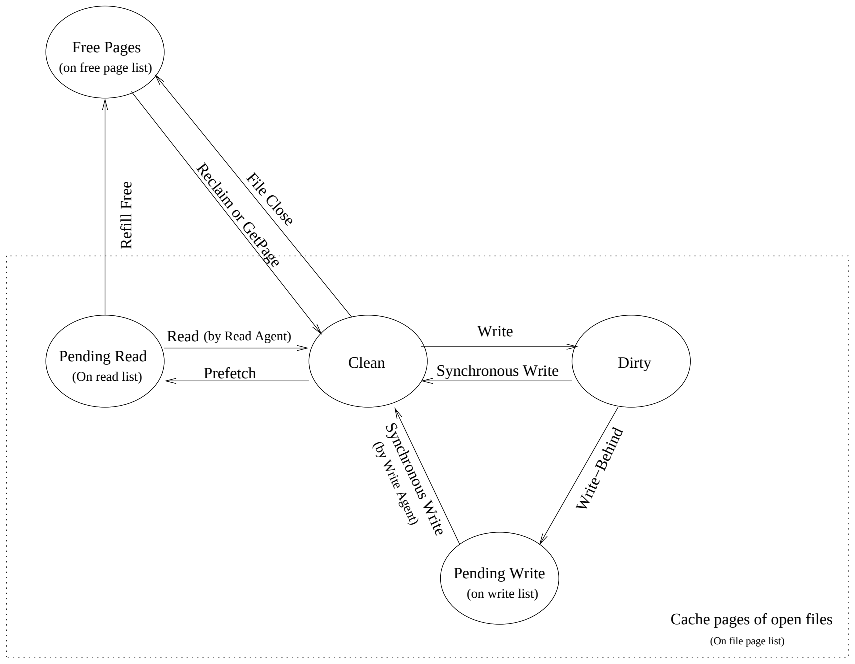


Figure 3.5: **PDIOL cache page state transition.**

multiple files are opened, the Second Chance algorithm will be applied globally across all open files.

The cache management structure is described below. Figure 3.5 shows the state transition of the PDIOL cache pages. Each of the five states of Figure 3.5 corresponds to a linked list.

Four kinds of lists – the free page list, the read list, the write list, and the file-page lists – are used to group the PDIOL cache pages. The **Free** pages belong to the free page list. The **Pending Read** pages belong to both the read list and file-page lists. **Pending Write** pages belong to the write list and file-page lists. **Clean** and **Dirty** pages belong to file-page lists only.

1. The free page list (A), which contains all pages that do not belong to any opened file. After the initialization of the PDIOL environment, all PDIOL pages are on this list. When a file is closed, all its pages will be put on this list by the function **FileClose**. When one page of a file is accessed for the first time, the page will be allocated from this list by the function **GetPage**, when the data is not buffered in the free pages, or reclaimed by the function **Reclaim**, when the data is buffered in the free pages.

2. The read list (B), which contains all the pages which are in **Pending-Read** state and

to be prefetched. The to-be-prefetched page is put on the prefetch list by the function **Prefetch**. After being **Read**, the page will changed to **Clean** state. The page on the read list may be moved to the free page list by the function **RefillFree** when the free page list is empty and higher priority request (e.g. demand-read request) arrives.

3. The write list (C), which contains all the pages which are in **Pending-Write** state and to be written out to remote nodes. A to-be-written-behind page is put on the write list by the function **WriteBehind**. After **Synchronous Writing**, the state of the page on write list will be changed to **Clean**.

4. The file-page list (D), which contains all the pages of an opened file. It includes **Pending-Read** page, **Pending-Write** page, **Clean** page, and **Dirty** pages. Every PDIOL file has its own file-page list.

The relationships between them are as follows:

1. $A \cap (B \cup C \cup D) = \emptyset$. This means that pages in the free page list do not belong to any open file;

2. $B \cap C$ = empty. No pages can be written out and read in at the same time;

3. $B \subset D, C \subset D$. Only pages belonging to an opened file can be read or written out.

3.5 Communication Mechanisms

As shown in Figure 3.6, PDIOL applications communicate with remote agents through sockets. Each remote agent has two sockets connected to the PDIOL application. One socket, the control socket, serves as a channel to transfer file management commands or other system commands. Another socket, the data socket, serves as a channel through which data pages of PDIOL logical files are transferred. If there are N remote agents, there are 2*N sockets on the PDIOL application side, and two sockets on every remote agent side. Command messages have high priority and are often small messages; they should not be blocked by long data messages.

In PDIOL, we selected the TCP-based connection-oriented socket, instead of a UDP-based connection-less communication mechanism. The reason is that with Myrinet, a high-performance network, bandwidth and latency over the network is no longer as much a bottleneck as it is with disk I/O operations. Although TCP has a little higher overhead, it is still suitable for PDIOL implementation. Furthermore, if we use UDP, PDIOL has to

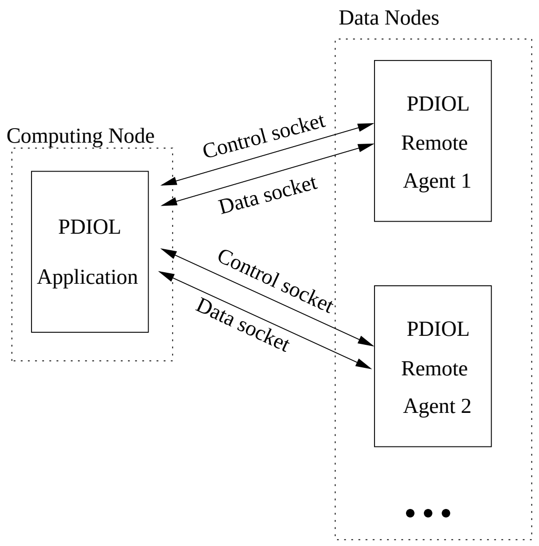


Figure 3.6: **Sockets between the PDIOL application and remote agents.**

reimplement some of the buffering functions (e.g. retransmit) of TCP in order to make sure the data will be sent reliably, making PDIOL more complex to implement.

The messages between PDIOL applications and remote agents can be divided into three groups which are transferred through different sockets:

1. System control/status messages, which carry system initialization/termination commands and system status information, are transferred through control sockets;

2. File control/status messages, which pass information about file I/O commands and file status information, are transferred through control sockets;

3. Data messages, which carry file data, go through data sockets.

3.6 Prefetching and the Local Read Agent

Prefetching is the most important technique in PDIOL. This section will detail the window-based prefetching algorithm. The local agent, which prefetches pages according to the algorithm, will be discussed as well.

3.6.1 Prefetching

PDIOL provides an API for enabling sequential prefetching, through which PDIOL applications enable prefetching for certain files. For sequentially-accessed files, the default prefetching algorithm is a window-based prefetching algorithm. The algorithm works in the following way: Suppose the window size is *W* pages and the reference sequence is $r_1, r_2, ...r_i, r_{i+1}, ...r_N, .$ If r_i is the currently accessed page, the pages from page r_{i+1} to page r_{i+W-1} will be prefetched. When selecting the window size, several factors need to be considered:

1. The number of disks.

   If there are N disks, the window size should not be less than N in order to obtain maximal concurrency. For example, if page 1 is currently accessed and the window size is N-1, page N will not be prefetched and disk N is idle.

2. Latency of message passing.

   The window size should be big enough in order to hide the latency of message passing, especially since the latency of messages passing between two user-level processes on different workstations is so high. Thus, even I/O-intensive applications which do not benefit from overlapping computation with communication, can still benefit from a large window size, via pipelining and parallelizing disk accesses. This can be demonstrated by the micro-benchmark results in Chapter 4.

3. Free buffer size.

   If the number of free pages in the system is small, a larger window size may deteriorate performance by evicting pages before they are used, due to page replacement. Therefore, there are factors that force a lower and an upper bound on the window size. In practice, one has to find the optimal window size.

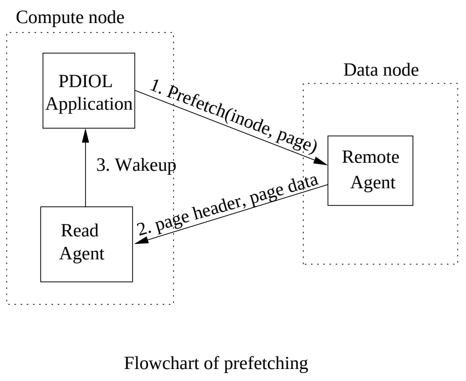
3.6.2 Application-side Read Agent

The application-side read agent is responsible for reading data from data sockets. Prefetching is done in the following steps, as shown in Figure 3.7.

1. The PDIOL application sends a prefetch message to a remote agent through a data socket;

2. The read agent on a data node reads the page from the local physical file and writes to data sockets;

3. If a PDIOL application is waiting on the page, it will be signaled via a wakeup message. If no PDIOL application is waiting for it, the read agent will leave the prefetched data in the PDIOL cache.



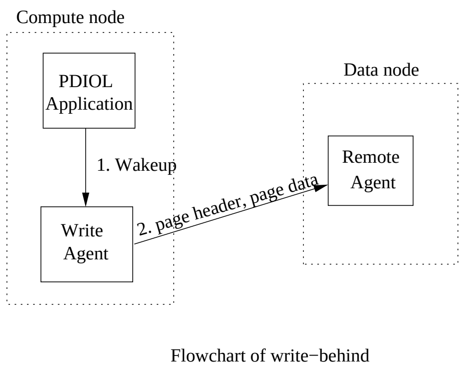


Figure 3.7: **Flowchart of prefetching and write-behind.**

The PDIOL application sends many prefetch messages (depending on the window size and the user prefetching command) at one time. The prefetching operations of the read agent can overlap with the PDIOL application's computation. Even if there is no overlap between computation and communication, the prefetching can still greatly decrease the latency by pipelining and parallelizing disk accesses.

To make sure that remote agents start prefetching as early as possible, it is necessary to send prefetch messages without any delay on the part of the buffer of the socket. The TCP_NO_NODELAY option is applied to the sockets to pass prefetching messages.

3.7 Write-behind and the Application-side Write Agent

The data can be written to disk two different ways – synchronous write or asynchronous write. A synchronous write function blocks until the writing is completed while an asynchronous write function delegates the writing to a proxy and returns immediately. The latency of synchronous writes data to disk is very high. Traditional file systems address this problem with asynchronous writes or write-behind. With write-behind, applications write data to buffers in main memory, and continue running. The data in the buffer will be written to disk later on by the local/remote file systems. The latency of writing is thus hidden from applications.

PDIOL needs to address the latency of writing as well, especially because the latency of writing to remote disks is higher than that of writing to local disks. Figure 3.7 shows the flowchart of write-behind implemented in PDIOL.

In PDIOL, write-behind is implemented in two ways:

1. The data to be written out to a remote agent will be put on the write list and the

PDIOL application can continue computing. The local write agent will send the data to the remote agent later.

2. The local write agent will delay sending data of a temporary file to remote agents until the cache pages of the temporary file are evicted. If the temporary file is created, accessed, and then deleted, the data of a temporary file may not have to written out to remote disks at all, and the amount of data to be transferred decreases.

Once created, the local write agent keeps checking the write list. If the write list is not empty, it sends the pages on the write list to the remote agents. The PDIOL application communicates with the local write agent through Pthread [38] signals. When the write list is empty, the local write agent enters a waiting state until signaled by the PDIOL application.

3.8 Remote Agents

Remote agents reside on remote workstations and process requests from PDIOL applications. They do not have their own cache management mechanisms and rely on local file systems to provide prefetching and write-behind abilities. Remote agents process three kinds of requests:

1. File data reading/prefetching/writing;
2. File level operations such as opening file, closing file, unlinking file, and other file operations;
3. PDIOL system-level operations, including initialization and finalizing.

In order to give higher priority to the last two kinds of requests, and to provide a clear interface, every remote agent spawns a read/write daemon which only reads-from / writes-to a data socket. The main thread of a remote agent is responsible for file level operations and other PDIOL system-level operations. In this way, the bulk data transfer of files will not block the processing of higher-level operations.

Processing commands from the PDIOL application consists of the steps shown in Figure 3.8.

1. When a read/write daemon receives a prefetching message, it puts it into the prefetching list;

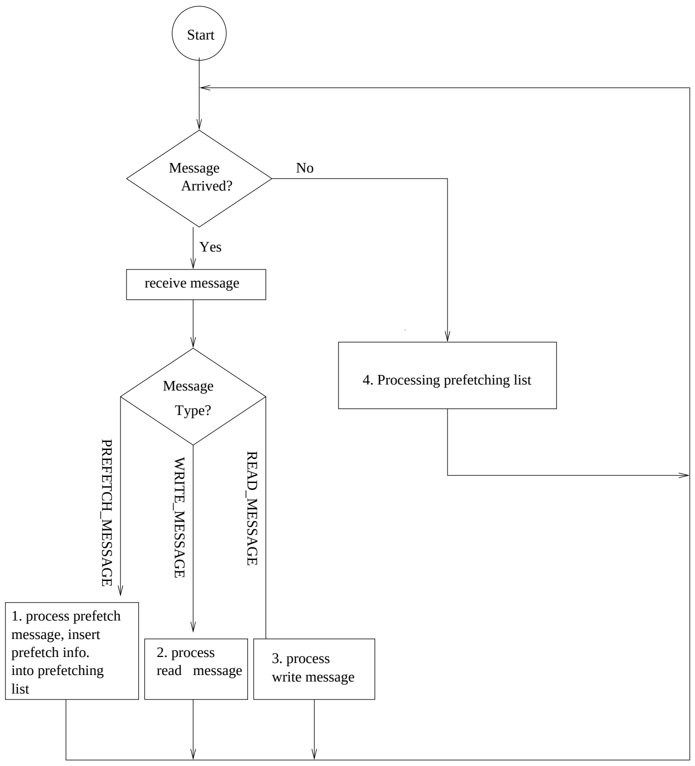


Figure 3.8: **The message processing of a remote agent.**

2. When a read/write daemon receives a demand-read message, it will check whether it is in the prefetch list or not. If it is, it will remove the page from the prefetch list. It then reads data from local sub-files and writes data to the data socket;

3. When a read/write daemon receives a write message, it will read data from a data socket and write data to local sub-files;

4. When no incoming message arrives and the prefetching list is not empty, a read/write daemon will obtain one page from the prefetch list, read it from local sub-files, and write it to the data socket.

   Only when there is no demand-read/write message, will an agent read/write daemon process a prefetching message, as demand read/write requests have higher priority than a prefetching request.

The main thread synchronizes with the read/write daemon through Pthread signals.

3.9 Concluding Remarks

In this chapter, we discussed the architecture of PDIOL, which includes the PDIOL library, local agents, and remote agents. The data structure and cache management in the PDIOL library were examined in detail. The communication mechanism, implementation of prefetch, write-behind, and remote agents were analyzed as well. With the cooperation between PDIOL applications and remote agents, the aggregate bandwidth of parallel disks over a workstation cluster can be utilized. In the next chapter, we will evaluate PDIOL with microbenchmarks and real applications.

Chapter 4

Performance Evaluation

In this chapter, we will evaluate the ideas of prefetching and disk parallelism as implemented in PDIOL. As the goal of PDIOL is to overcome the I/O bottleneck by increasing the bandwidth and decreasing the latency of disk systems with disk parallelism, we plan to conduct experiments in the following way:

1. We test the bandwidth and latency of the network and the disks in the testbed to show the feasibility of PDIOL;

2. We use modified version of *cat* and *grep*, which are I/O intensive applications, and *sort* and *bzip2*, which are compute-intensive applications, to evaluate and analyze the performance of PDIOL.

With PDIOL, *cat* and *grep* eliminate much of the I/O waiting time and obtain a speedup of 2.6 with 4 remote disks, while *sort* and *bzip2* gain a marginal speedup because of their already-low I/O waiting times. From the results, we can see that I/O-intensive applications gain more benefit, while compute-intensive applications benefit less – a result which is consistent with our intuition.

In order to give a clear view of the experiments, Table 4.1 summarizes the purpose and the results.

4.1 Experimental Testbed

Our testbed is a workstation cluster which consists of 8 nodes connected by a Myrinet network. Each node has dual 800MHz Pentium III processors, 512 MB (ECC) main memory, and a 18 GB SCSI disk (Seagate 18.4GB Barracuda USCSI ST318416W). Note that the PDIOL uses multiple threads for agents, but the applications are single-threaded. The de-

Exp.	Benchmark	Result	Note
1	The bandwidth and latency of **disk access**	Figure 4.3 and 4.4	Network bandwidth and latency should be better than disks to support the use of PDIOL.
2	The bandwidth and latency of the **network**	Figure 4.5, 4.6 and 4.9	Since the network is faster than disks, PDIOL can be useful.
3	*mycat* and PDIOL *mycat*	Figure 4.12 and 4.15, Table 4.3 and 4.4	PDIOL improves performance by reducing waiting time through disk parallelism and prefetching.
4	Effect of **prefetch window size**	Figure 4.16 and Table 4.5	Performance cannot improve beyond a certain size of the prefetch window. As the number of disks varies, the minimum and maximum window size for improved performance also changes.
5	*grep*, *sort* and *bzip2* on the local disk	Figure 4.17, 4.18, and 4.19, 4.20 and Table 4.6	When is PDIOL useful? It is useful when the application is bottlenecked on I/O waiting time (i.e., *grep*), but not when it is bottlenecked on computation (i.e. *sort* and *bzip2*).
6	PDIOL *grep*	Figure 4.24 and 4.25, Table 4.7 and 4.8	PDIOL improves the the performance of *grep* by a factor of 2.6 with 4 disks.
7	PDIOL *sort*	Figure 4.27, 4.28, 4.29, 4.32, and 4.33, Table 4.9 and 4.10	*sort* gets marginal speedup with more than 1 disk.
8	PDIOL *bzip2*	Figure 4.36 and 4.37, Table 4.11 and 4.12	*bzip2* gets little speedup with more than 1 disk, similar to Experiment 7.

Table 4.1: **Description of experiments.**

Vendor_id	Intel
Model name	Pentium III (Coppermine)
CPU MHz	799.665
Cache size	256 KB

Table 4.2: Details concerning the CPUs of the computers in the testbed.

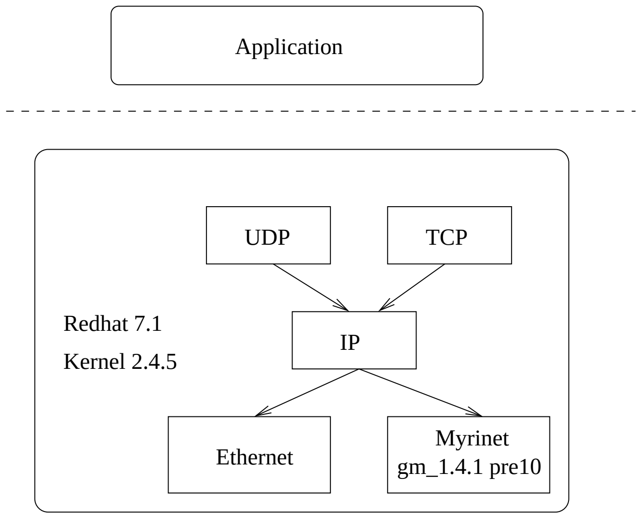


Figure 4.1: **The software structure of the testbed.**

tails are shown in Table 4.2. The operating system is Linux with the kernel version 2.4.5 SMP. The low-level message-passing system for the Myrinet is GM version 1.4.1 pre10.

The topology of the cluster is shown in Figure 3.3 and the software layers are shown in Figure 4.1:

To justify PDIOL, the following two conditions must be considered:

1. The total bandwidth of the network for one computer must be higher than that of local disks. Otherwise, the network will form a new bottleneck and make it impossible to obtain performance improvement through disk parallelism;

2. The latency of the network should not be too high. If the latency of the network is much higher than that of the disk access, a PDIOL cache miss will cause I/O operations to wait a longer time than that of local disks.

In the following sections, we benchmark the disk and network systems in the testbed.

```
                  -------Sequential Output-------- ---Sequential Input-- --Random--
                  -Per Char- --Block--- -Rewrite-- -Per Char- --Block--- --Seeks---
Machine    MB K/sec %CPU K/sec %CPU    K/sec %CPU  K/sec %CPU K/sec %CPU  K/sec %CPU
brule-a    63  8933 99.1 230613 100.0 178413 99.6  9914 100.0 517632 96.3 35225.  82.5
```

Figure 4.2: **A sample output of *Bonnie*.**

4.1.1 Disk System

There are many disk I/O benchmark tools [1, 30, 32, 27, 17]. *Bonnie* [1] benchmarks sequential output, sequential input, and random seeks of file systems with access units ranging from one character to blocks of different sizes. One sample output of *Bonnie* is shown in Figure 4.2

We select *Bonnie* as our benchmarking tool for the following reasons:

1. *Bonnie* is a simple program and it is easy to configure its code to benchmark certain access patterns;

2. The sequential access pattern of PDIOL is similar to one of *Bonnie*'s test cases.

However, *Bonnie* has some shortcomings. If the file size is less than available physical RAM, all operations will be done on physical RAM instead of on disks. This is shown in Figures 4.3 and 4.4.

Because the size of a PDIOL page is 64KB, the chunk size in *Bonnie* is updated from the default value of 16KB to 64KB in order to compare the result with the PDIOL benchmarks.

From Figures 4.3 and 4.4, it can be seen that when the file size is less than 256MB, the Linux operating system itself caches all file data reside in the physical memory, and there is almost no disk access. When the file size is greater than 256MB, disk accesses are required to fetch data from disk to the physical memory. The larger the file size, the more representative the result is of the actual hard disk performance. In the following discussion, we will use the benchmark results for the 2GB file.

Because the I/O waiting time cannot be obtained from the output of *Bonnie* directly, we calculate the I/O waiting time in the following way, with the assumption that CPU utilization is zero when applications are waiting for I/O operations:

1. Obtain the CPU utilization ratio R_{cpu_util} of disk operations through the *Bonnie* benchmark;

2. Obtain the total access time T_{total};

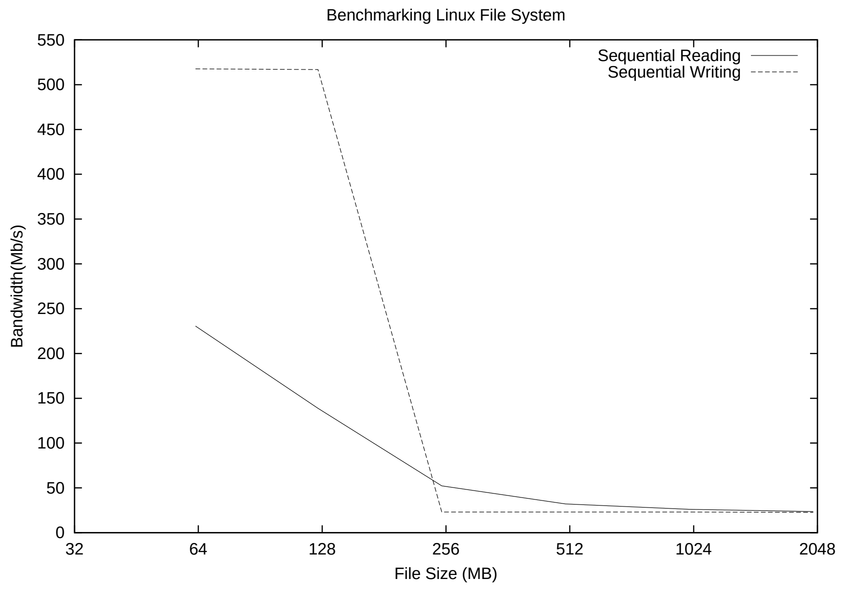


Figure 4.3: **Disk access bandwidth** – The experiment is done on brule-m-a:/usr/scratch and the benchmark tool is *Bonnie*.

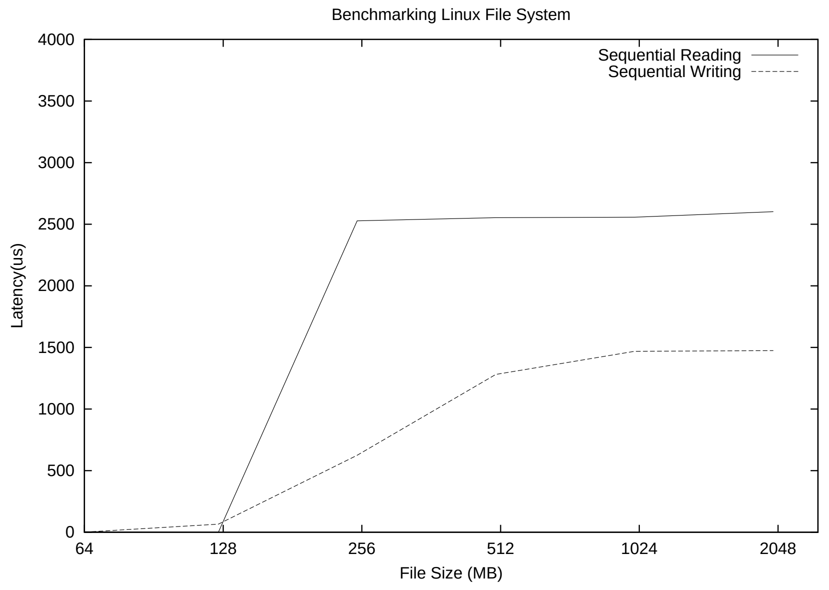


Figure 4.4: **Disk access waiting time** – The experiment is done on brule-m-a:/usr/scratch and the benchmark tool is *Bonnie*.

3. Calculate waiting time per page:

$$T_{waiting\_time\_per\_page} = T_{total} * (1 - R_{cpu\_util})/N_{number\_of\_pages}$$

   $T_{waiting\_time\_per\_page}$ is shown by the Y-axis in Figure 4.4. R_{cpu_util} is obtained from the output of Bonnie.

4.1.2 Network

Netperf [18] is a benchmark tool which measures the performance of many different types of networks, such as: TCP and UDP via BSD Sockets, DLPI, UNIX Domain Sockets, Fore ATM API, and HP HiPPI Link Level Access. *Netperf* can test both unidirectional throughput and end-to-end latency.

We select *Netperf* as our benchmarking tool for the following reasons:

1. *Netperf* is a popular and standard tool for benchmarking networks;

2. *Netperf* provides all the measurements, such as bandwidth and latency that we need in our experiments.

The version of *Netperf* used in our experiments is 2.1 [4]

Two experiments have been carried out. The first is for one-to-one connections, and tests the bandwidth and latency for a single connection. The second experiment is for one-to-many connections, in order to test the scalability of the network.

One-to-one Connection

The experiments have been performed on *brule-m-a*, which serves as a server, and *brule-m-f*, which serves as a client. See Figure 3.3 for the topology of the testbed. The bandwidth and latency of the network are shown in Figures 4.5 and 4.6, respectively. The confidence interval for +/-2.5% is 99% for all results. The result in Figure 4.5 is obtained through an updated version of the `tcp_stream_script` in *Netperf*, which tests the performance of TCP/IP stream over Myrinet. The `tcp_stream_script` measures bulk-data transfer performance, i.e., ”unidirectional stream” performance. Each test lasts 60 seconds.

Except for the buffer sizes for sending and receiving, all other TCP settings are at their default settings. Figure 4.7 shows the settings of the sockets. With respect to Figure 4.6, network request/response performance is expressed as the rate of the transaction, which is the exchange of a request and a response of certain sizes. Given the transaction rate obtained

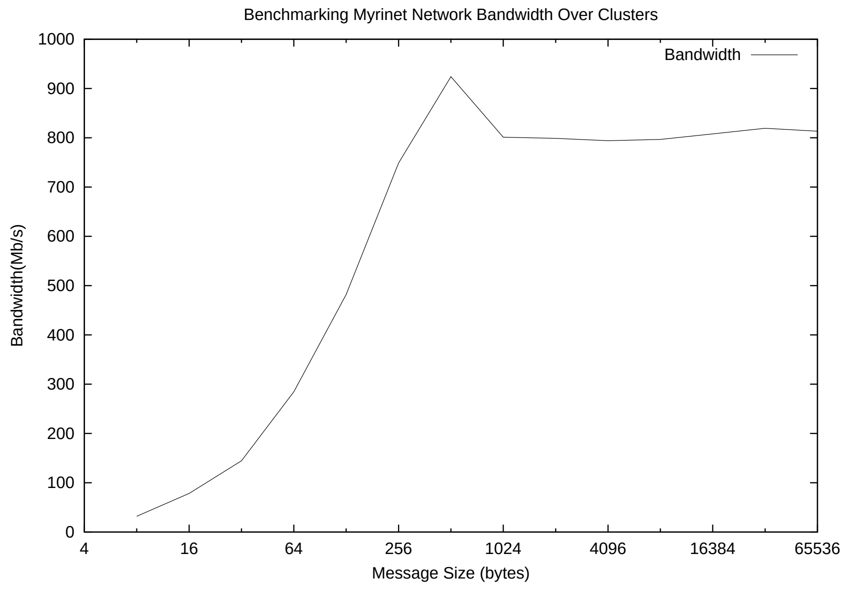


Figure 4.5: **The bandwidth of the Myrinet network for one-to-one connection**. The result is obtained for the TCP stream. The benchmarking tool is *Netperf* 2.1 and the `tcp_stream_script`.

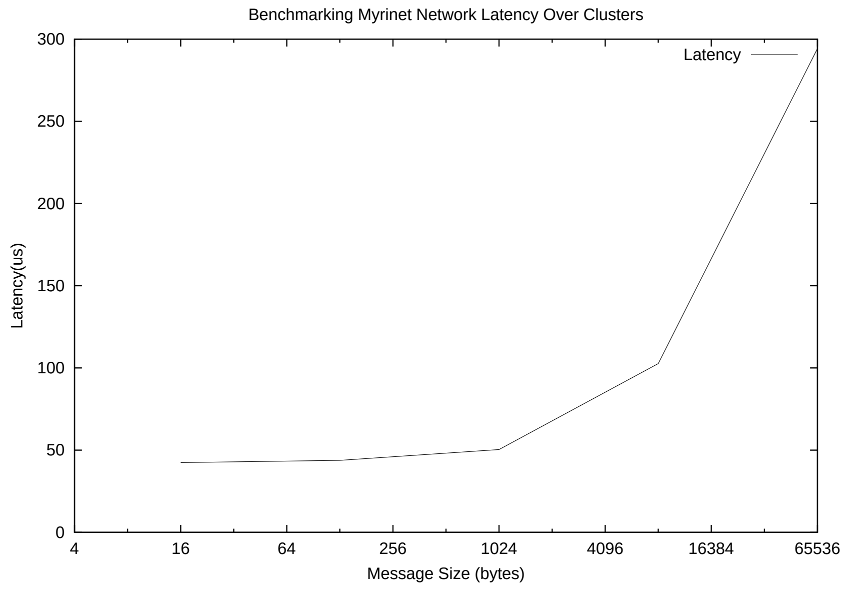


Figure 4.6: **One-way latency** The result is for the TCP stream. The benchmarking tool is *Netperf* 2.1 and `tcp_rr_script`. The request size is fixed at 128 bytes. The message sizes are shown in the X-axis, and latencies are shown in the Y-axis.

```
SO_BROADCAST: default = off
SO_DEBUG: default = off
SO_DONTROUTE: default = off
SO_ERROR: default = 0
SO_KEEPALIVE: default = off
SO_LINGER: default = l_onoff = 0, l_linger = 0
SO_OOBINLINE: default = off
SO_RCVBUF: default = 131070
SO_SNDBUF: default = 131070
SO_RCVLOWAT: default = 1
SO_SNDLOWAT: default = 1
SO_RCVTIMEO: default = 0 sec, 0 usec
SO_SNDTIMEO: default = 0 sec, 0 usec
SO_REUSEADDR: default = off
SO_REUSEPORT: (undefined)
SO_TYPE (tcp,udp): default = 1
IP_TOS: default = 0
IP_TTL: default = 64
TCP_MAXSEG: default = 8948
TCP_NODELAY: default = on
```

Figure 4.7: **Socket settings**.

from the `tcp_rr_script`, the latency (Figure 4.6) of the network can be calculated with the following method:

$$T_{round-trip} = 1/transaction\_rate$$
$$T_{one-way} = 1/transaction\_rate/2$$

In this experiment, we set the request size at 128 bytes, which is about the size of the reading, prefetching, and writing messages of PDIOL; we set the response size at 64KB bytes, which is the page size of PDIOL.

From the above experiment and those in Section 4.1.1, it can be seen that the network has much better one-to-one bandwidth than that of disks (e.g., Figure 4.3).

One-to-many Connections

In order to evaluate the scalability of the Myrinet for one-to-many connections, which is the scenario if we use disk parallelism (Chapter 3.2), the following experiment is done on *brule-m-f*, which serves as a client, and *brule-m-a*, *brule-m-b*, *brule-m-c*, *brule-m-d*, the four of which serve as servers (Figure 4.11).

Because it is difficult to get the aggregate bandwidth for multi-connections with *netperf*, a benchmark tool – *pdiol_nettest* – is developed. *pdiol_nettest* consists of two parts – *pdiol_nettest_client* and *pdiol_nettest_server*. The algorithms of *netperf* and *pdiol_nettest* are

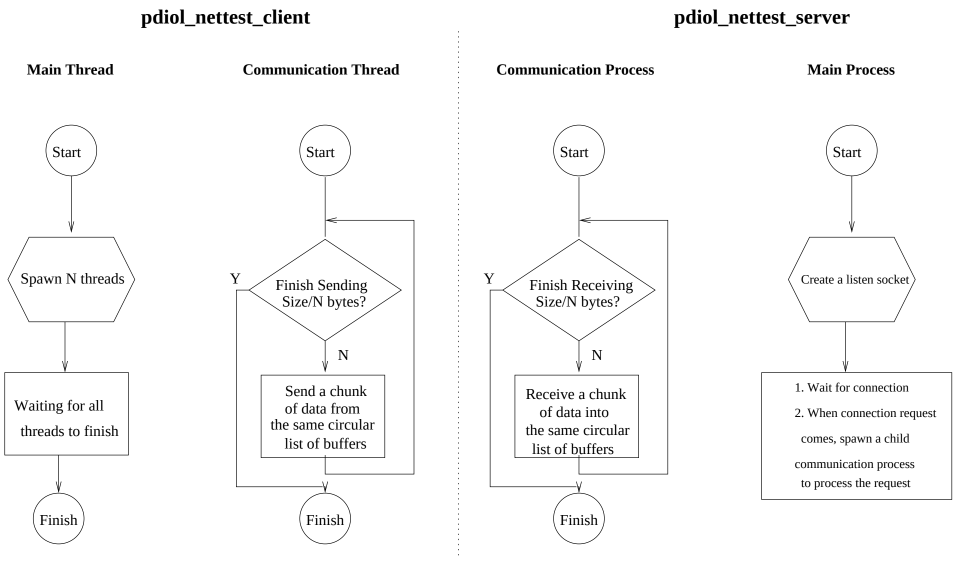

pdiol_nettest_client
pdiol_nettest_server
Main Thread
Communication Thread
Communication Process
Main Process
Start
Spawn N threads
Waiting for all threads to finish
Finish
Finish Sending Size/N bytes?
Y
N
Send a chunk of data from the same circular list of buffers
Finish Receiving Size/N bytes?
Receive a chunk of data into the same circular list of buffers
Create a listen socket
1. Wait for connection
2. When connection request comes, spawn a child communication process to process the request

Figure 4.8: **The algorithm of *pdiol-nettest*.**

shown in Figure 4.8. The experiments are done on *brule-m-f*, on which *pdiol_nettest_client* is run, and *brule-m-a*, *brule-m-b*, *brule-m-c* and *brule-m-d*, on which *pdiol_nettest_server* is run.

Figure 4.9 shows that the bandwidth for 4 nodes is not higher than 2 nodes. We speculate that 2 streams of data are enough to saturate the bandwidth of the PCI bus of the server, *brule-m-f*. However, as we will see, it will take more than 2 disks to generate that much data traffic.

Careful readers will notice that the results of *pdiol_nettest* and *Netperf* for one-to-one connection are a little different. This is because when sending and receiving data, *Netperf* and *pdiol_nettest* uses different buffer management mechanisms. *Netperf* uses different circular lists of buffers while *pdiol_nettest* keeps using the same circular list of buffers. As the purpose of *pdiol_nettest* is to measure the raw performance of PDIOL, it is designed to work in the same way as PDIOL.

Since it is more complex to update *Netperf* to simulate the PDIOL behavior, we leave it unchanged. As there is not too much difference between *Netperf* and *pdiol_nettest*, we speculate that the results are still useful.

From the results shown in Figures 4.5, 4.6, and 4.9, it can be seen that the aggregate bandwidth reaches the maximal value when the connection number is 2. We can expect that when the aggregate bandwidth of all disks outnumbers the aggregate bandwidth of the network, no further improvement can be obtained by adding more disks.

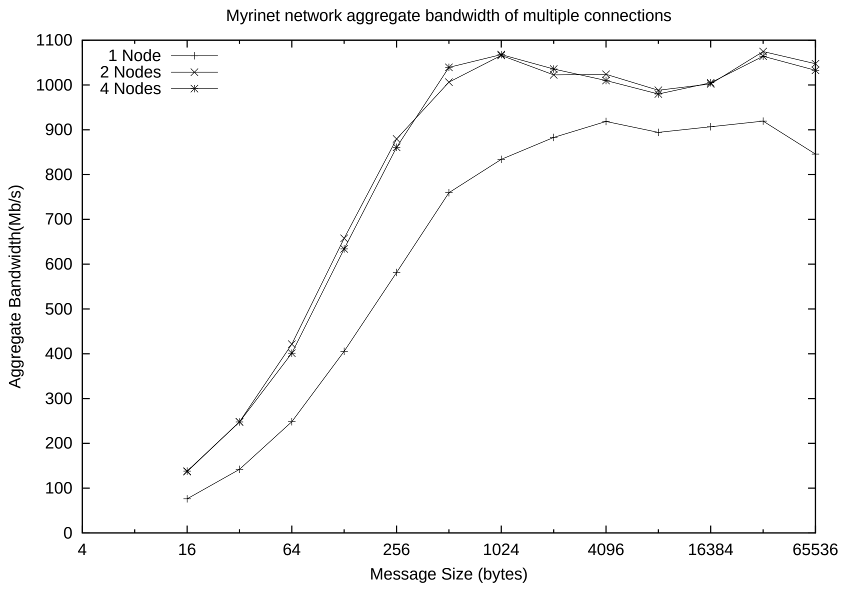


Figure 4.9: **One-to-many aggregate bandwidth** The result is obtained for the TCP stream. The benchmarking tool is *pdiol_nettest* The request size is fixed at 128 bytes. The response sizes are shown in the X-axis, and the latency is shown in the Y-axis.

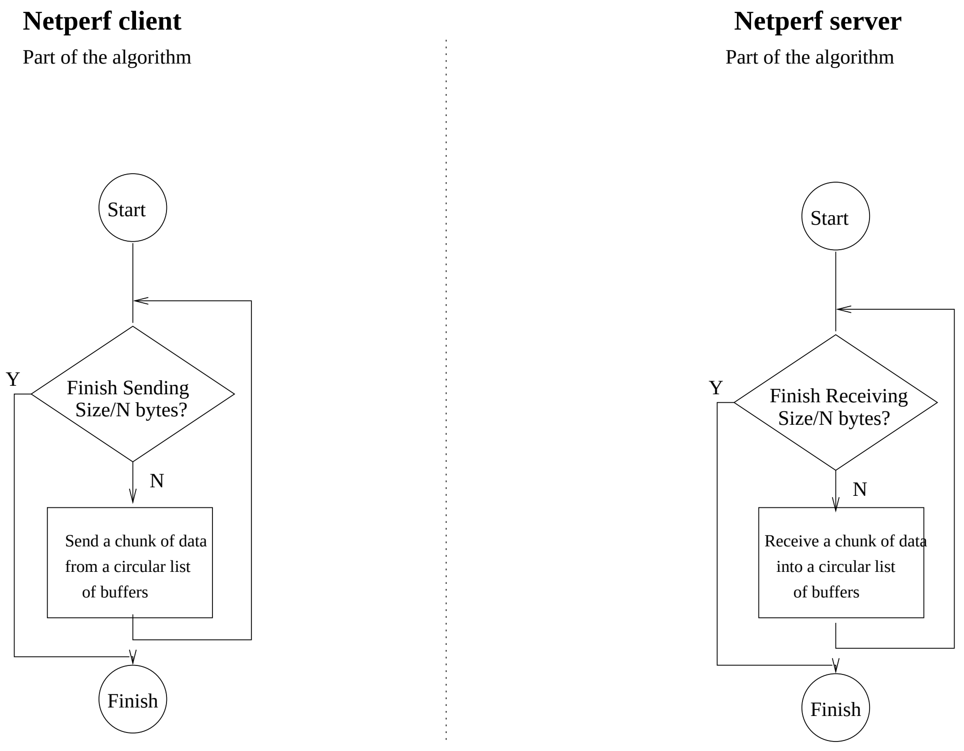

Netperf client
Part of the algorithm
Start
Y
Finish Sending Size/N bytes?
N
Send a chunk of data from a circular list of buffers
Finish
Netperf server
Part of the algorithm
Start
Y
Finish Receiving Size/N bytes?
N
Receive a chunk of data into a circular list of buffers
Finish

Figure 4.10: **The algorithm of *Netperf*.**

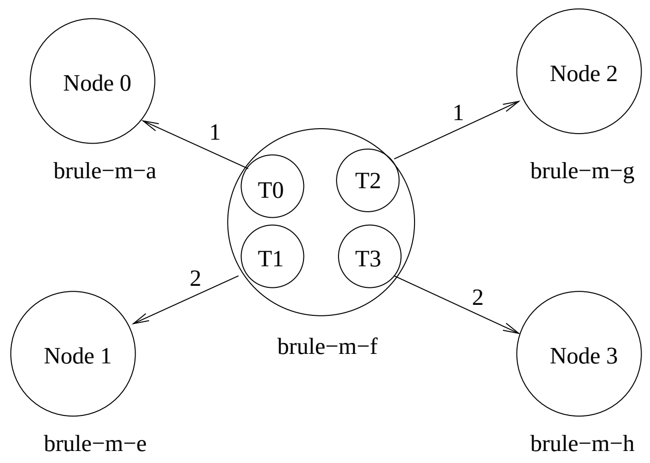


Figure 4.11: **One-to-four connections for** *pdiol_nettest*. *pdiol_nettest_client*, which spawns 4 communication threads, T0, T1, T2 and T3, run on client node. *pdiol_nettest_server* is run on node0, node1, node2, and node3 concurrently. T0 communicates with node 0, T1 with node 1, T2 with node 2, and T3 with node 3.

4.1.3 Justification of PDIOL

From the results shown in Figures 4.3, 4.4, 4.5, 4.6, and 4.9, we can see that when reading a 64KB page from remote agents, the bandwidth can reach nearly 100MB, and the latency is only 300 μs. However, when reading one page from the local disk, the bandwidth is only about 20MB and the waiting time is almost 2,500 μs. Therefore, we can expect that PDIOL, with the proper policies and a reasonable implementation, can bring about a performance improvement. As discussed later (Chapter 4.2), it will likely require more than 2 disks to saturate the network.

4.2 Microbenchmarks

In this section, we test the reading time of PDIOL logical files and compare the performance with that of files on local disks.

The scalability of PDIOL is an important metric, and we test it with different numbers of disks and file sizes. The core technique of PDIOL is prefetch. By varying the size of the prefetch window, we evaluate the effect of the size of the prefetch window in PDIOL.

From the experiments, we can see that PDIOL decreases the I/O waiting time when the number of disks increases. In the following experiments, every benchmark application is run 5 times – the result is shown in both a table, which shows the average, maximal, minimal value and variance, and a figure which shows the performance variation.

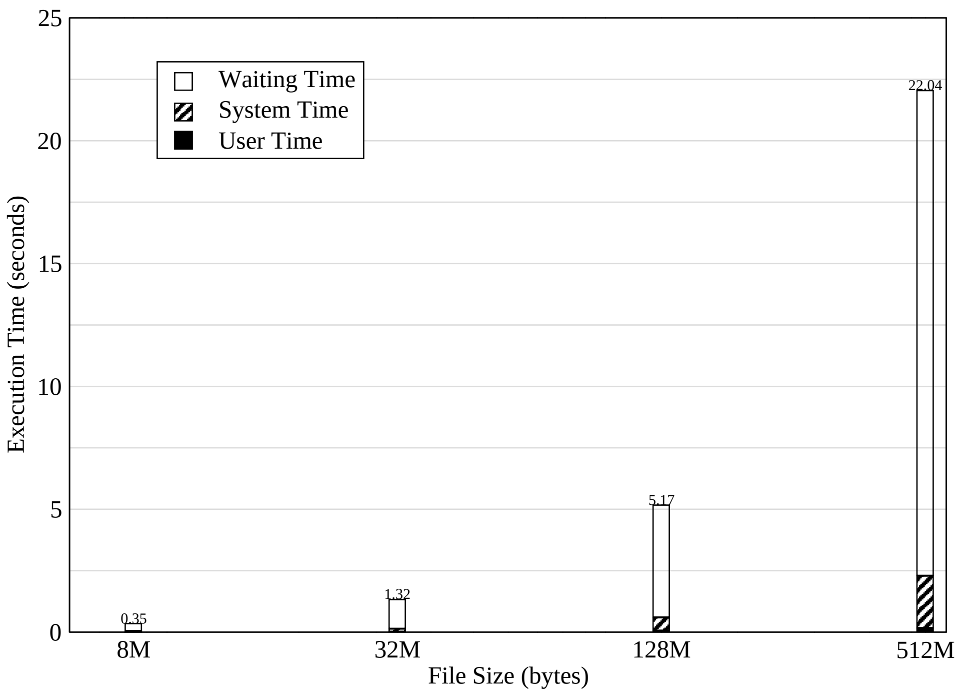


Figure 4.12: **The execution time for reading files** on the local disk (1 disk). The program *mycat* uses a buffer of the same size as those of the PDIOL *mycat*, i.e., 64KB.

File size	Avg	Max	Min	σ^2
8MB	0.35	0.36	0.34	0.000
32MB	1.32	1.54	1.26	0.015
128MB	5.17	5.31	5.06	0.009
512MB	22.04	22.28	21.77	0.050

Table 4.3: **The execution time for reading data** on the local disk (1 disk). The program *mycat* uses a buffer of the same size as those of the PDIOL *mycat*, i.e., 64KB bytes.

4.2.1 *mycat* On The Local Disk

We use a simple program *mycat* to measure the reading time of files on the local disk. The key differences between *mycat* and standard *cat* are that *mycat* has a tunable buffer size. Then by replacing the file I/O functions, it is easy to transform *mycat* to a PDIOL version, and compare the performance.

Figure 4.12 and Table 4.3 show the reading time of files with different file sizes. *mycat* uses a buffer of size 64KB, the same size as PDIOL *mycat*, in order to make the comparison between PDIOL *mycat* and *mycat* fair.

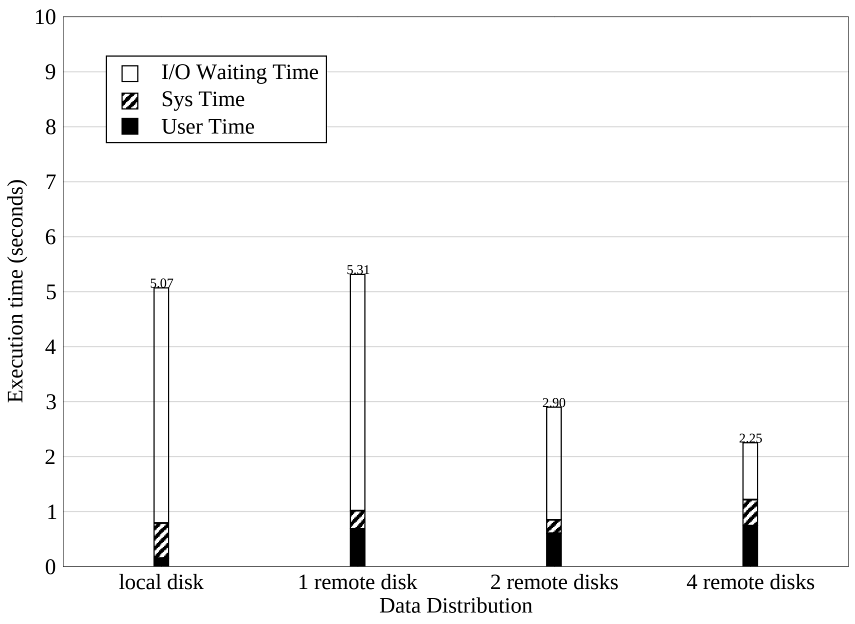


Figure 4.13: **The benchmark of *mycat* on the local disk and PDIOL *mycat* on 1, 2, and 4 disks**. The file size is 128MB. The prefetch window size is 128 pages.

4.2.2 *mycat* vs. PDIOL *mycat*

By replacing the I/O operations in *mycat*, we get the PDIOL *mycat*. We test the performance PDIOL *mycat* with different numbers of disks. The file size is fixed at 128MB and the prefetch window size is 128 pages. The experiments are done in the following steps (Figure 4.14):

1. A file of 128MB is created;

2. The PDIOL utility program *fs2pfs* is used to transform it to a PDIOL logical file, which is striped across the remote disks;

3. The file buffers of all nodes are flushed;

4. PDIOL *mycat* is used to read PDIOL logical files and redirect the output to */dev/null* on the local node.

From Figure 4.13, it can be seen that: using parallel disks is effective in reducing I/O-waiting time and *mycat* gains a speedup of 2.53 with 4 disks. However, the user time increases because of the communication overhead and cache management overhead.

In the following section, we will explore the the effects of other parameters – different file sizes and prefetch window sizes.

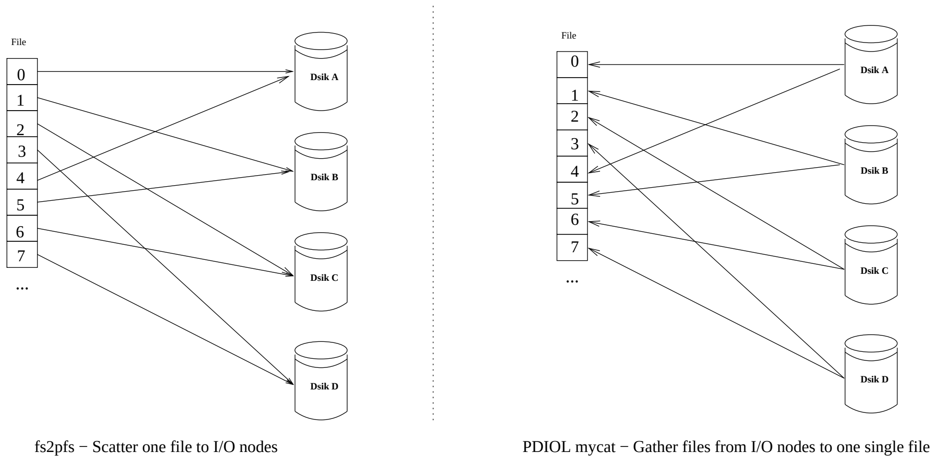

File
0
1
2
3
4
5
6
7
...
Dsik A
Dsik B
Dsik C
Dsik D
fs2pfs − Scatter one file to I/O nodes
PDIOL mycat − Gather files from I/O nodes to one single file

Figure 4.14: **The PDIOL logical file and standard file transformation diagram by** *fs2pfs* **and PDIOL** *mycat* **with 4 remote disks.**

	1 disk				2 disks				4 disks			
File Size	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2
8MB	0.42	0.43	0.41	0.000	0.45	0.93	0.28	0.080	0.26	0.28	0.23	0.000
32MB	1.44	1.58	1.37	0.009	0.94	1.45	0.78	0.083	0.63	0.65	0.62	0.000
128MB	5.31	5.49	5.14	0.016	2.90	3.17	2.82	0.023	2.25	2.57	2.11	0.040
512MB	22.12	22.46	21.83	0.056	12.57	13.39	11.05	0.985	9.08	9.73	8.84	0.141

Table 4.4: **The execution time of PDIOL** *mycat* with different file sizes on 1 (left), 2 (middle), and 4 (right) disks. The size of prefetch window is 128 pages.

4.2.3 PDIOL *mycat* Performance with Different File Sizes

In this section, we test the effect of file sizes on the performance of PDIOL with different numbers of disks and file sizes . The file sizes are 8MB, 32MB, 128MB, and 512MB. The reason we select these numbers is that the PDIOL cache size is 256MB, and we want to examine the performance for files whose sizes are larger and smaller than cache size.

The size of the prefetch window is set at 128 pages. In reality, as the next experiment shows, the size of the prefetch window does not improve the performance beyond a certain point, for I/O-intensive applications. Figure 4.15 and Table 4.4 show the scalability of PDIOL.

Comparing the results of the above two experiments, we can see that the waiting time decreases with the increase in the number of disks. For the 512M file, the waiting time is 15.16s for 1 disk, 7.1s for 2 disks, and 1.35s for 4 disks. The waiting time decreases faster than the increase in the number of disks. We speculate that the reason for the rapid decrease is that with the increase of the number of disks, there is more caching at the hard disk's host workstation, and more I/O waiting time is overlapped with the increased user and system

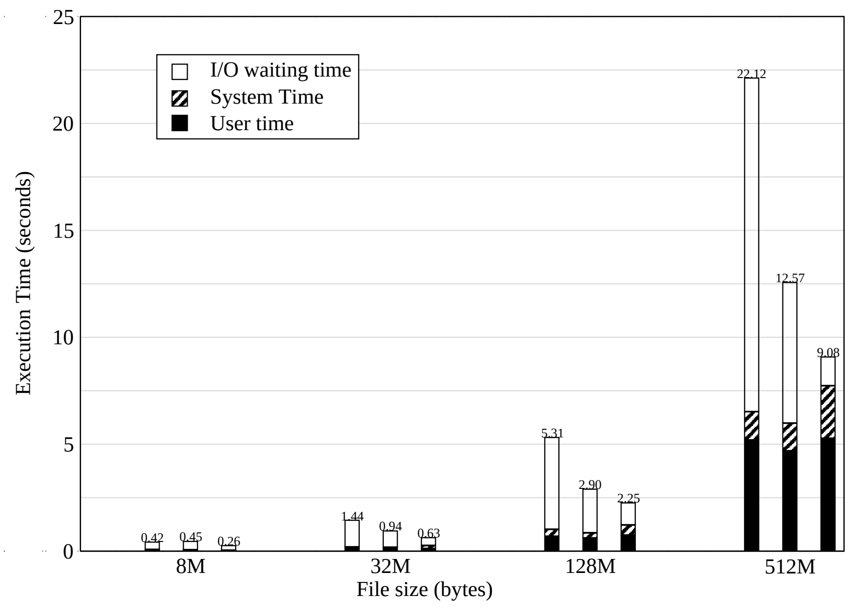


Figure 4.15: **The benchmark of PDIOL** *mycat* with different file sizes on 1 (left), 2 (middle), and 4 (right) disks. The prefetch window size is 128 pages.

	1 disk				2 disks				4 disks			
Window	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2
1	7.82	8.13	7.60	0.052	8.04	8.68	7.26	0.377	7.99	8.10	7.88	0.007
4	5.32	5.64	5.13	0.048	3.24	3.41	3.14	0.014	3.75	4.61	3.43	0.240
16	5.32	5.74	5.10	0.073	2.97	3.52	2.82	0.095	2.42	2.76	2.13	0.093
64	5.39	5.59	5.23	0.022	2.83	2.89	2.80	0.001	2.51	3.95	2.13	0.645
256	5.40	5.58	5.29	0.017	3.34	5.10	2.83	0.977	2.26	2.47	2.12	0.031

Table 4.5: **The execution time of PDIOL** ***mycat*** **with an 128MB PDIOL logical file** with different prefetch window size on 1 (left), 2 (middle), and 4 (right) disks. The experiment is done on a 128MB PDIOL logical file.

time. As shown by the breakdown of execution time, system time increases with the number of remote nodes, while user time is largely unaffected. This shows that the overhead of the PDIOL library remains reasonable, regardless of the increase in the number of disks.

4.2.4 PDIOL *mycat* Performance With Different Prefetch Window Sizes

The purpose of the experiments is to show the effect of prefetch windows on the performance of PDIOL. In this experiment, the file size is fixed at 128MB. The performance of PDIOL *mycat* is shown in Figure 4.16 and Table 4.5.

From these graphs, it can be seen that:

1. Because there is no computation that can be overlapped with reading in *mycat*, the performance cannot improve after the size of the prefetch window increases to a cer-

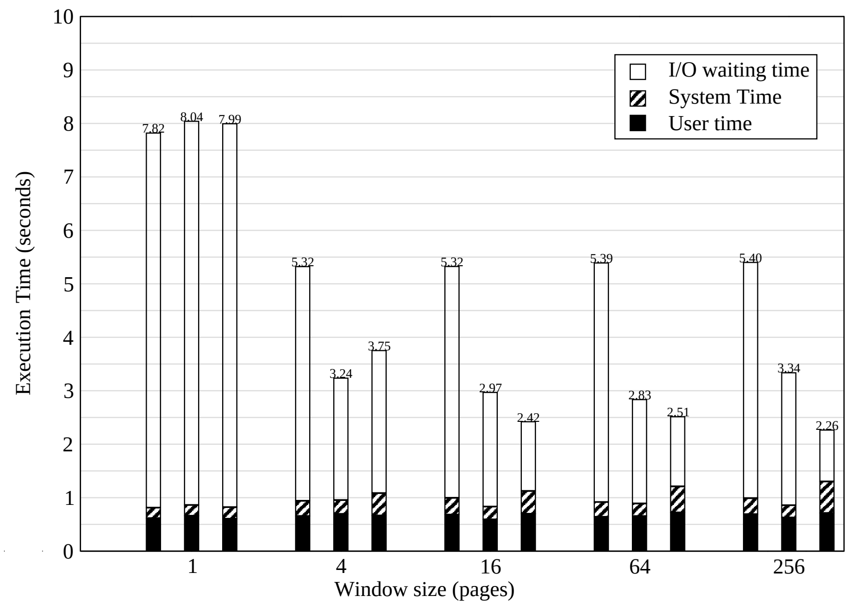


Figure 4.16: **The execution time of PDIOL** *mycat* with different prefetch window sizes on 1 (left), 2 (middle), and 4 (right) disks. The experiment is done on a 128MB PDIOL logical file.

tain point, as all disks are kept busy already. For 4 disks, after the size of the prefetch window increases to 16, no significant improvement can be obtained from increasing the size of the prefetch window – and for 1 disk the threshold is 4. The reason is that with 4 pages prefetched for each disk, the latency is overlapped completely. For *mycat*, the number of disks has a greater impact on performance than does the prefetch window. A larger size of the prefetch window, without an increase in the number of disks, does not result in greater improvement.

2. The waiting time decreases proportionally with the increase in the number of disks, as expected. But the user time and, especially, the system time increase with the number of disks. The user time is spent on data copying in user space and cache management. The system time is spent transferring data between computers and copying data across kernel-space and user-space.

Careful readers may notice that when the size of prefetch window is 4 pages, the performance of 4 disks is lower than 2 disks. We speculate that the reason for this is that, with a prefetch window of only 4 pages, the benefit of more bandwidth is too little and is offset by increased communication overhead.

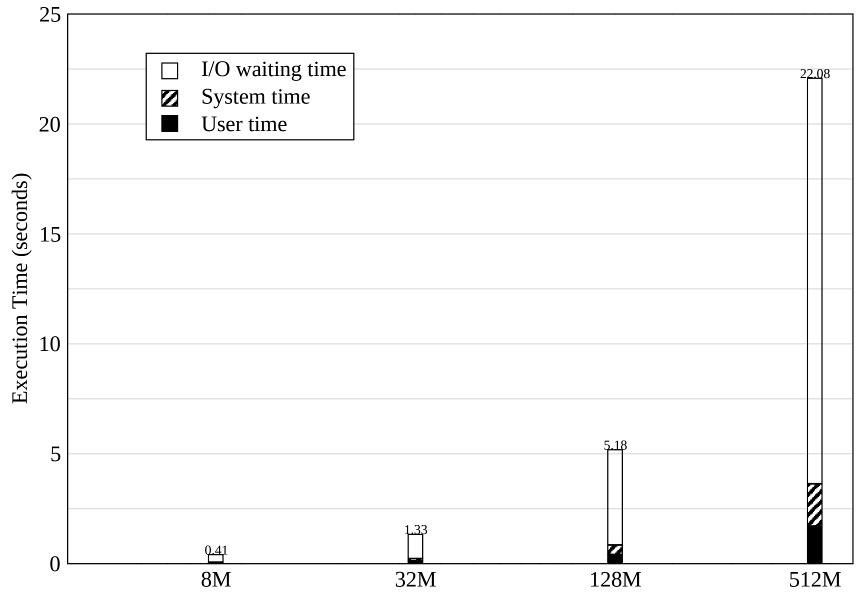


Figure 4.17: **The execution time breakdown of** *grep* on the local disk (1 disk).

	grep				*sort*				*bzip2*			
File Size	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2
8MB	0.41	0.44	0.40	0.000	2.38	2.42	2.32	0.001	2.10	2.29	2.07	0.004
32MB	1.33	1.47	1.26	0.003	9.03	9.07	8.99	0.001	8.06	8.27	8.00	0.006
128MB	5.18	5.65	5.08	0.031	83.18	87.35	66.03	37.859	32.13	32.22	32.00	0.007
512MB	22.1	22.9	21.7	0.115	696.8	703.3	685.3	34.06	129.8	130.2	129.4	0.058

Table 4.6: **The execution time of** *grep* **(left),** *sort* **(middle), and** *bzip2* **(right)** on the local disk (1 disk).

4.3 Application Benchmarks: *grep*, *sort* and *bzip2*

In this section, we evaluate the performance of PDIOL with three common applications. *grep* is a program to retrieve text from text files. *Sort* is used to sort text files. *bzip2* can compress data files.

grep is an I/O-intensive application, while *sort* and *bzip2* are compute-intensive applications. As PDIOL can reduce I/O waiting time through disk parallelism, we expect *grep* to benefit most from PDIOL.

4.3.1 *grep*, *sort* and *bzip2* On The Local Disk

In order to evaluate the performance of PDIOL, it is necessary to know how much time is spent waiting on I/O. The performance of *grep*, *sort*, and *bzip2* is tested, and the results are shown in the following table and figures.

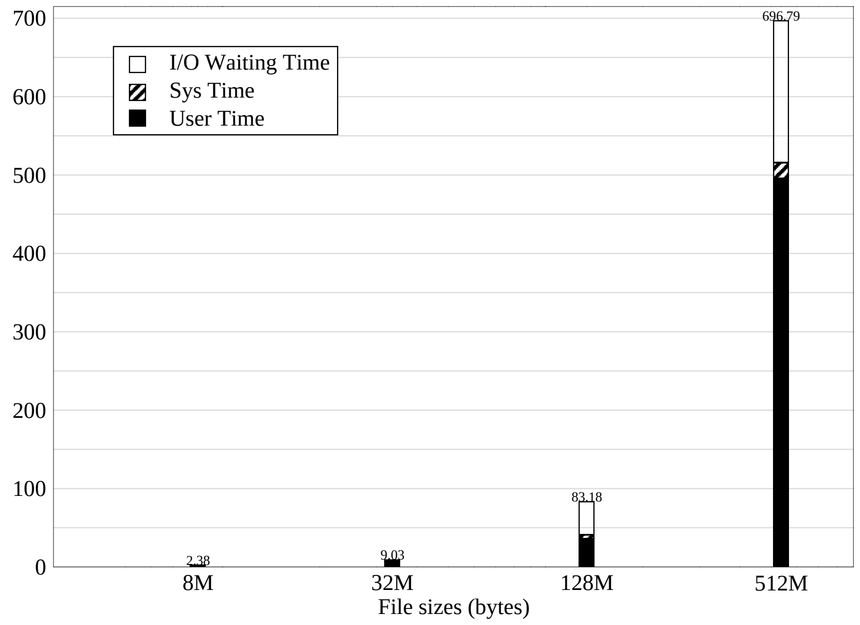


Figure 4.18: **The execution time breakdown of** *sort* on the local disk (1 disk).

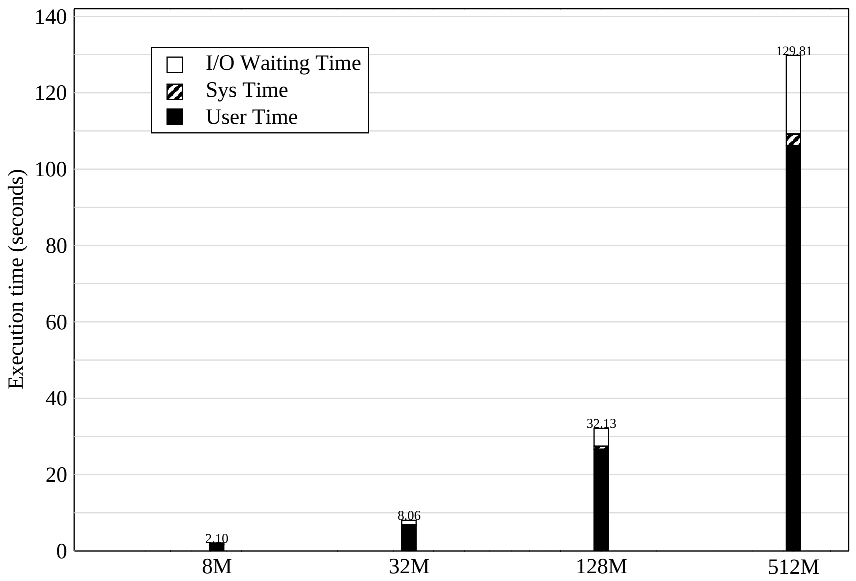


Figure 4.19: **The execution time breakdown of** *bzip2* on the local disk (1 disk).

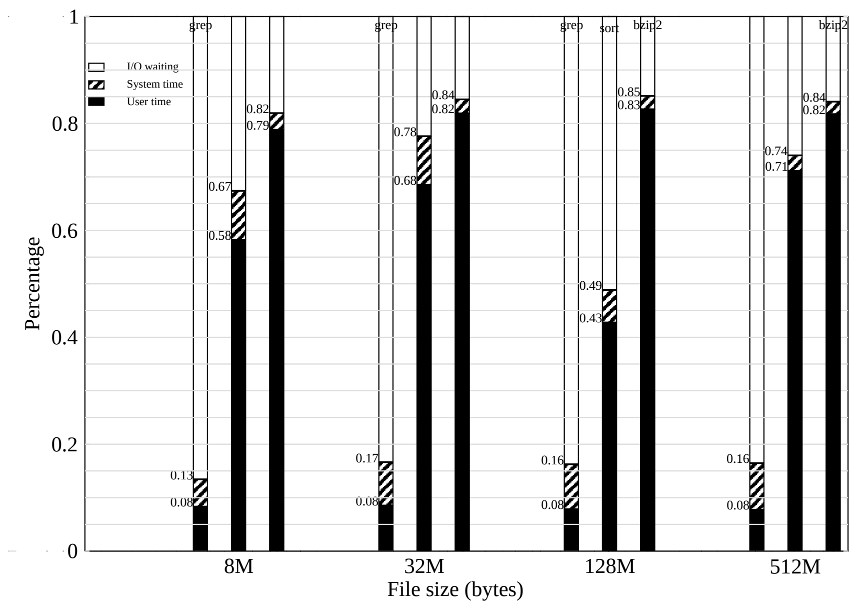


Figure 4.20: **The normalized execution time of** *grep* **(left),** *sort* **(middle), and** *bzip2* **(right)** on the local disk (1 disk).

From the results shown in Figures 4.17, 4.18, 4.19, and 4.20, and Table 4.6, we can see that *grep* spends most of its time waiting, while *sort* and *bzip2* spend most of their time computing. With PDIOL, *grep* should be able to reduce I/O waiting time greatly and obtain the most improvement.

4.3.2 *grep*, *sort* and *bzip2* vs. PDIOL *grep*, PDIOL *sort* and PDIOL *bzip2*

By replacing the I/O functions with the corresponding PDIOL I/O functions in *grep* and compiling with the PDIOL library, we generate a PDIOL version of *grep*. With the same method, we create PDIOL versions of *sort* and *bzip2*.

Figure 4.21, 4.22, and 4.23 show the experiment results of PDIOL *grep*, PDIOL *sort* and PDIOL *bzip2*. The file size is fixed at 128MB and the prefetch window size is 128 pages. From Figure 4.21 it can be seen that PDPIOL improves the performance of *grep* as the I/O-waiting time is reduced greatly. Figure 4.22, unfortunately, shows that PDIOL brings marginal improvement for *sort*. As the I/O-waiting time is a small part in the total execution time of *sort*, there is not too much I/O-waiting time to be overlapped. This result is consistent with our intuition. Figure 4.23 compares the performance of *bzip2* and PDIOL *bzip2*. As little I/O-waiting time can be overlapped, bzip2 gains marginal performance improvement as well.

In the following sections, we experiment with parameters – file sizes, prefetch window

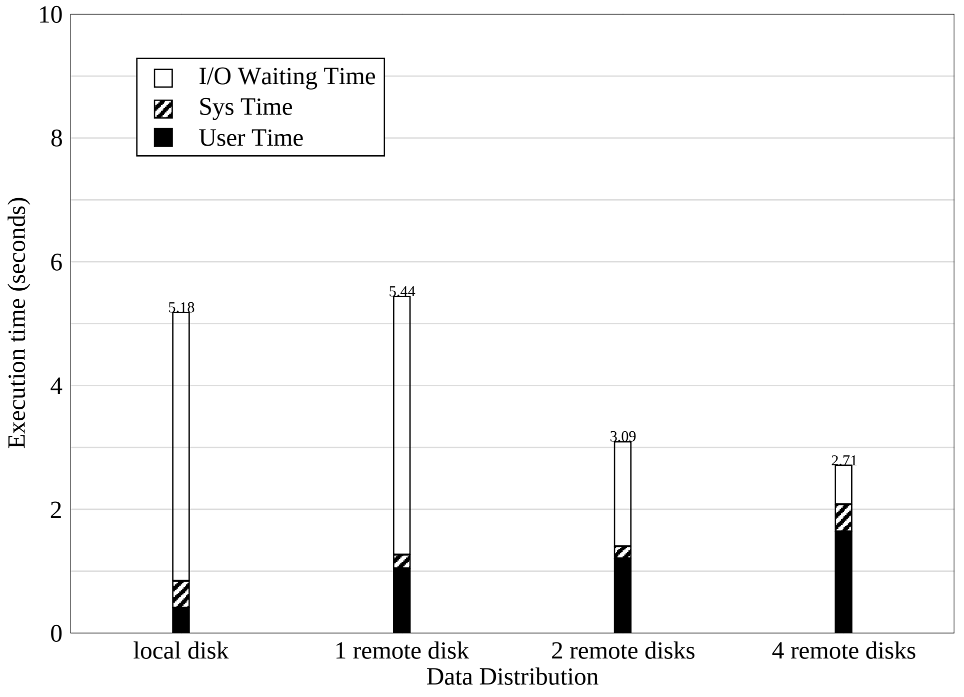


Figure 4.21: **The benchmark of *grep* on the local disk and PDIOL *grep* on 1, 2, and 4 disks**. The file size is 128MB. The prefetch window size is 128 pages.

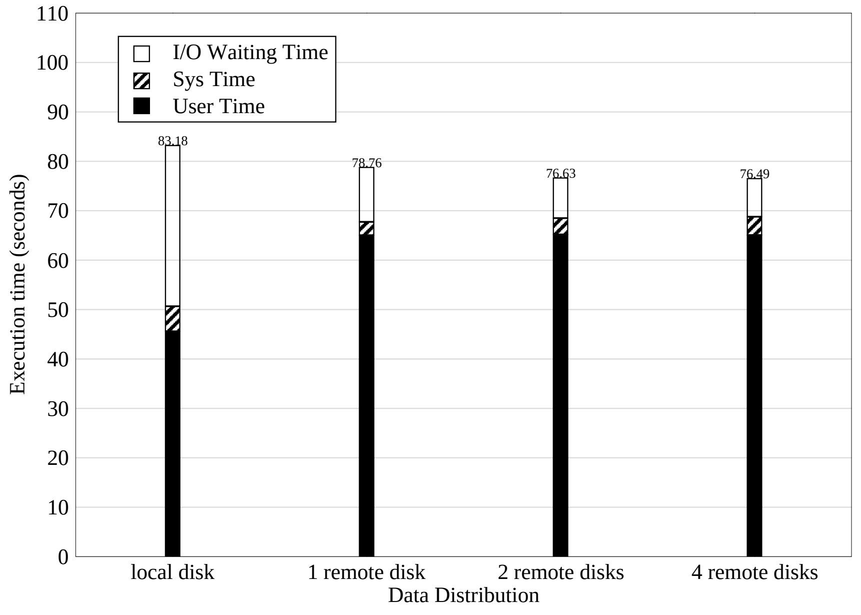


Figure 4.22: **The benchmark of *sort* on the local disk and PDIOL *sort* on 1, 2, and 4 remote disks**. The file size is 128MB. The prefetch window size is 128 pages.

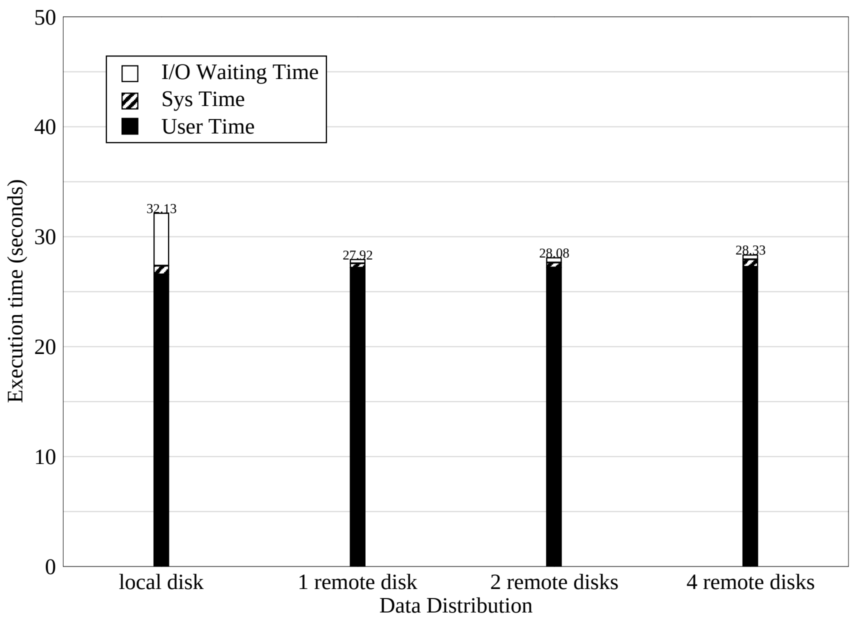


Figure 4.23: **The benchmark of** *bzip2* **on the local disk and PDIOL** *bzip2* **on 1, 2, and 4 remote disks**. The file size is 128MB. The prefetch window size is 128 pages.

	1 disk				2 disks				4 disks			
File Size	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2
8MB	0.52	0.55	0.51	0.000	0.38	0.39	0.36	0.000	0.36	0.37	0.34	0.000
32MB	1.51	1.64	1.46	0.005	0.91	1.02	0.84	0.004	0.73	0.74	0.73	0.000
128MB	5.44	5.59	5.24	0.030	2.99	3.17	2.93	0.010	2.31	2.58	2.22	0.023
512MB	22.21	22.49	21.97	0.044	11.34	11.91	11.06	0.119	8.44	8.87	8.27	0.064

Table 4.7: **The execution time of PDIOL** *grep* with different file sizes on 1 (left), 2 (middle), and 4 (right) disks. The prefetch window size is 128 pages.

sizes, and the number of disk for each of the three applications to explore the effects of these parameters.

4.3.3 PDIOL *grep*

By varying the file size and the number of disks, we obtain the following results, as shown in Figures 4.24 and 4.25, and Tables 4.7 and 4.8. From the results, it can be seen that PDIOL improves the performance of *grep* by reducing I/O waiting time through disk parallelism. Figure 4.25 and Table 4.8 show the execution time of PDIOL *grep* with different prefetch window sizes on 1, 2, and 4 disks. Again, we see that the window size needs to increase with the number of disks, but that there is an effective upper bound on the window size.

As expected, the performance of *grep* improves with an increase in the number of disks. Since there is little computation in *grep*, the overhead is almost the same as that of PDIOL

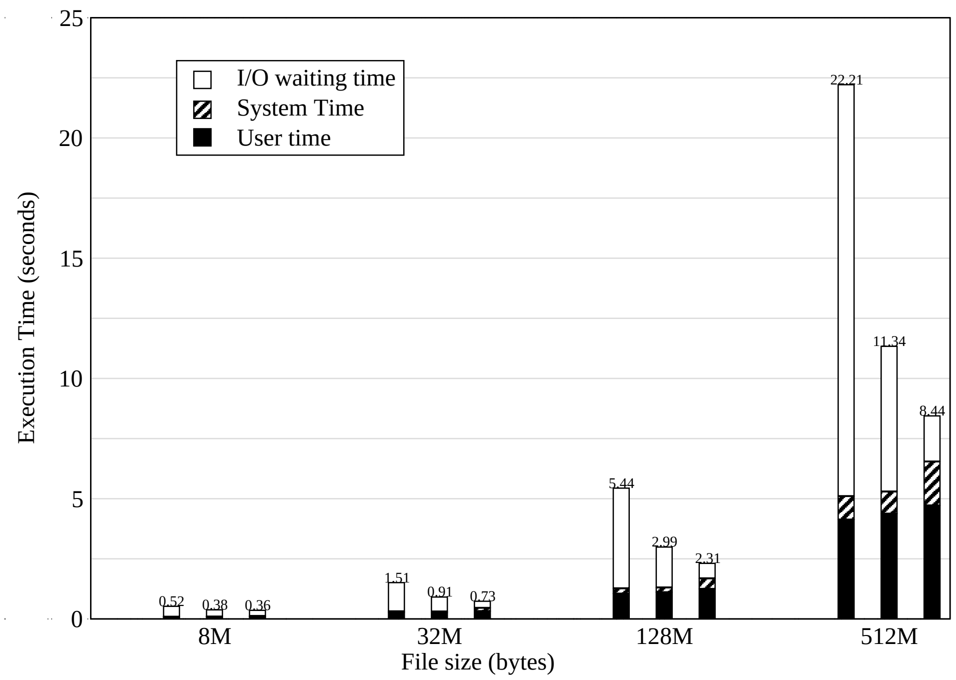


Figure 4.24: **The execution time of PDIOL** *grep* with different file sizes on 1 (left), 2 (middle), and 4 (right) disks. The prefetch window size is 128 pages.

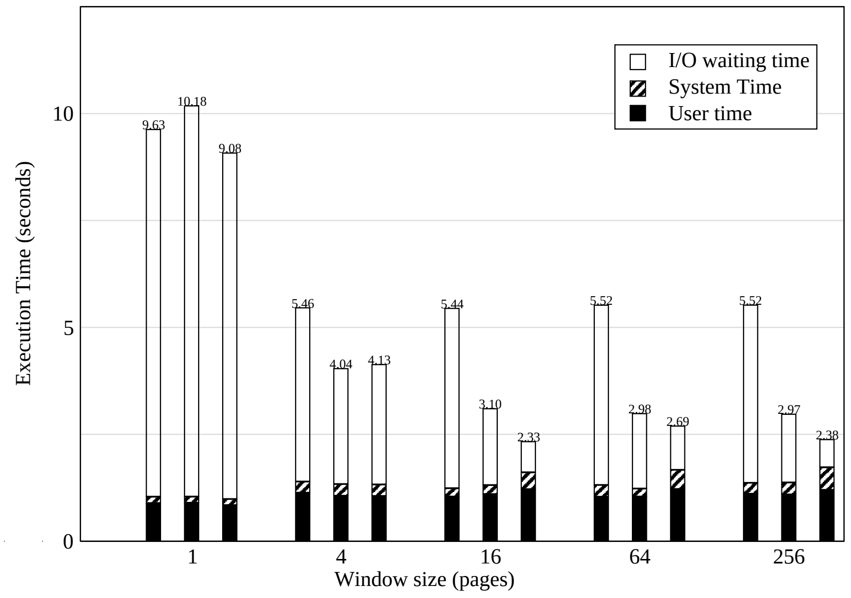


Figure 4.25: **The execution Time of PDIOL** *grep* with different prefetch window sizes on 1 (left), 2 (middle), and 4 (right) disks. The file size is fixed at 128MB.

	1 disk				2 disks				4 disks			
Window	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2
1	9.46	9.59	9.40	0.006	8.65	9.32	7.99	0.250	8.74	9.02	8.59	0.027
4	5.42	5.86	5.25	0.065	3.37	3.55	3.28	0.015	3.88	4.21	3.62	0.064
16	5.43	5.85	5.23	0.063	3.19	3.45	2.94	0.040	2.65	4.18	2.26	0.730
64	5.40	5.82	5.23	0.057	3.01	3.18	2.93	0.012	2.25	2.28	2.22	0.001
256	5.49	5.76	5.27	0.044	3.06	3.29	2.93	0.028	2.24	2.30	2.20	0.002

Table 4.8: **The execution time of PDIOL grep** with different prefetch window sizes on 1, 2, and 4 disks. The file size is fixed at 128MB.

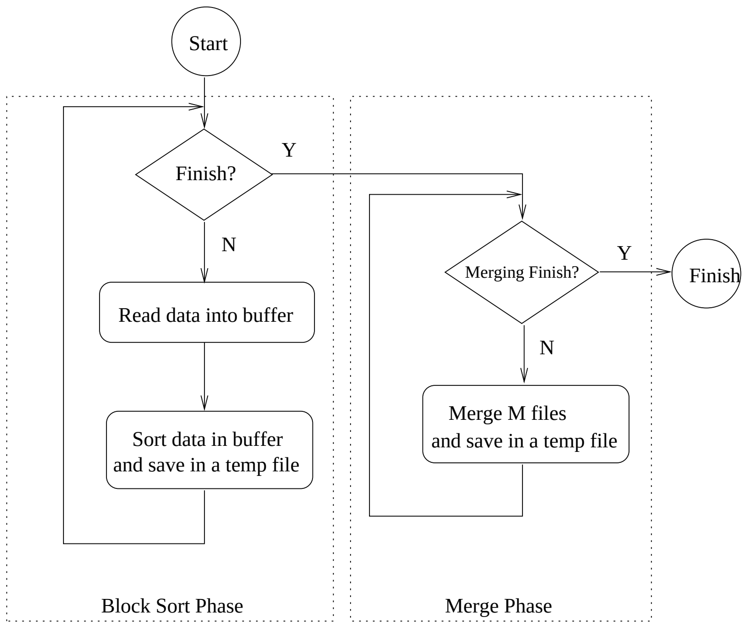


Figure 4.26: *Sort* **Algorithm.**

mycat, as described in the last section.

4.3.4 PDIOL *sort*

This experiment is done to test the performance of PDIOL for compute-intensive applications. The buffer size is set at 8MB, the same size that of 128 PDIOL cache pages.

The *sort* algorithm contains two phases – block sort phase and merge phase (Figure 4.26):

1. Block sort phase – Read a block of data into a buffer, sort it, and store the result into a temporary file until all data in the file are sorted;

2. Merge phase – Merge the data in all temporary files into one result file.

With PDIOL, the performance improvement can be obtained through hiding access la-

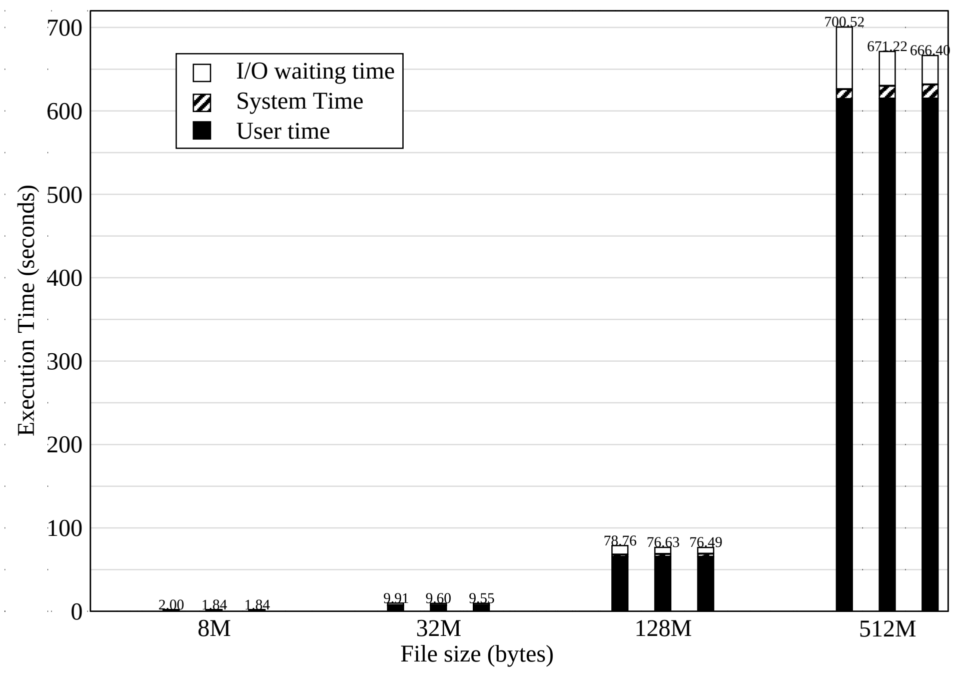


Figure 4.27: **The execution time of PDIOL** *sort* with different file sizes on 1 (left), 2 (middle), and 4 (right) disks. The prefetch window size is fixed at 128 pages.

	1 disk				2 disks				4 disks			
File Size	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2
8MB	2.00	2.02	1.97	0.001	1.84	1.85	1.84	0.000	1.84	1.86	1.81	0.000
32MB	9.91	9.94	9.86	0.001	9.60	9.77	9.54	0.010	9.55	9.59	9.51	0.001
128MB	78.76	79.48	78.35	0.179	76.63	77.27	76.38	0.133	76.49	76.56	76.45	0.002
512MB	700.5	700.8	700.1	0.067	671.2	672.5	670.3	0.745	666.4	667.2	665.7	0.358

Table 4.9: **The execution time of PDIOL** *sort* with different file sizes on 1 (left), 2 (middle), and 4 (right) disks. The prefetch window size is fixed at 128 pages.

tency by sending more than one read request, and overlapping computation with access latency. At the same time, unlike *mycat* and *grep*, PDIOL *sort* writes temporary sorted data and the final results to I/O nodes with PDIOL functions, thus PDIOL *sort* should also benefit from the parallel writing ability of PDIOL. However, because the writing can be overlapped efficiently by OS, no significant improvement can be obtained through PDIOL, for this application. The results are shown in Figures 4.27, 4.28, and 4.29, and Table 4.9.

From the results shown in Figure 4.27 and Table 4.9, it can be seen that with more disks, the performance of PDIOL *sort* improves. The improvement of 2 disks over 1 disk is greater than that of 4 disks over 2 disks.

To analyze the effect of PDIOL in different phases of *sort*, the execution time break-

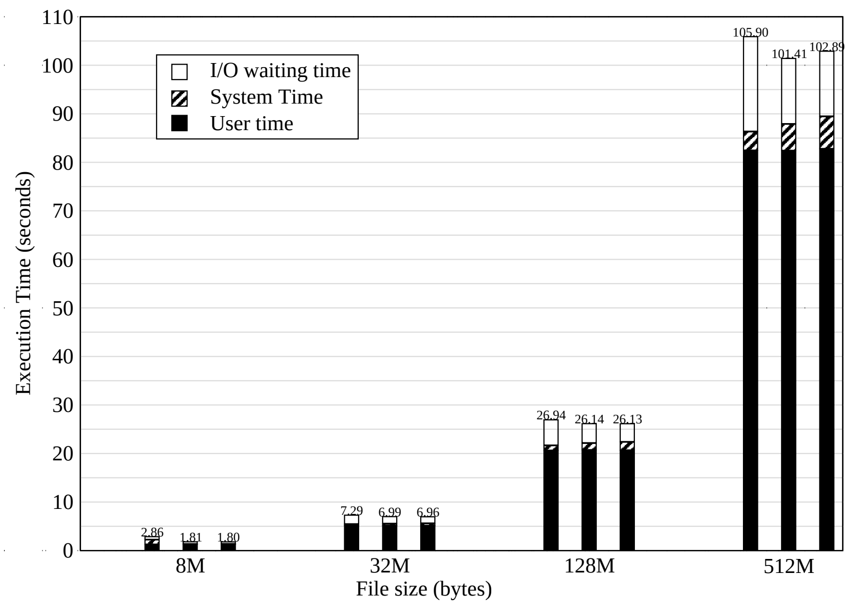


Figure 4.28: **The execution time of block sort phase in PDIOL** *sort* with different file sizes on 1 (left), 2 (middle), and 4 (right) disks. The prefetch window size is fixed at 128 pages.

down is shown in Figures 4.28 and 4.29

Figures 4.18, 4.30, and 4.31 show the performance of *sort* without PDIOL. From Figures 4.30 and 4.28, it can be seen that the I/O waiting time of the block sort phase decreases greatly. Especially for the 512MB file, the size of the file plus the size of the sorted temporary file exceeds the available physical memory, and most pages that are flushed to disk cause the I/O waiting time to increase greatly. For the PDIOL version, because the physical memory on the compute node and remote agent node can be utilized, fewer pages will be flushed to disk and the I/O waiting time decreases. A comparison of Figures 4.29 and 4.31 shows that the I/O waiting time of the merge phase also decreases.

Figures 4.32, 4.33, and 4.34, and Table 4.10 show the effect of the size of the prefetch window on the execution time of *sort*. From Figure 4.32 and Table 4.10, the following can be seen:

1. When the size of the prefetch window is one page, limited improvement can be obtained as there is little overlap.

2. When the size of the prefetch window is 64 pages, most of the reading latency is overlapped with computation, and no significant benefit can be obtained by increasing the number of remote disks from 2 to 4.

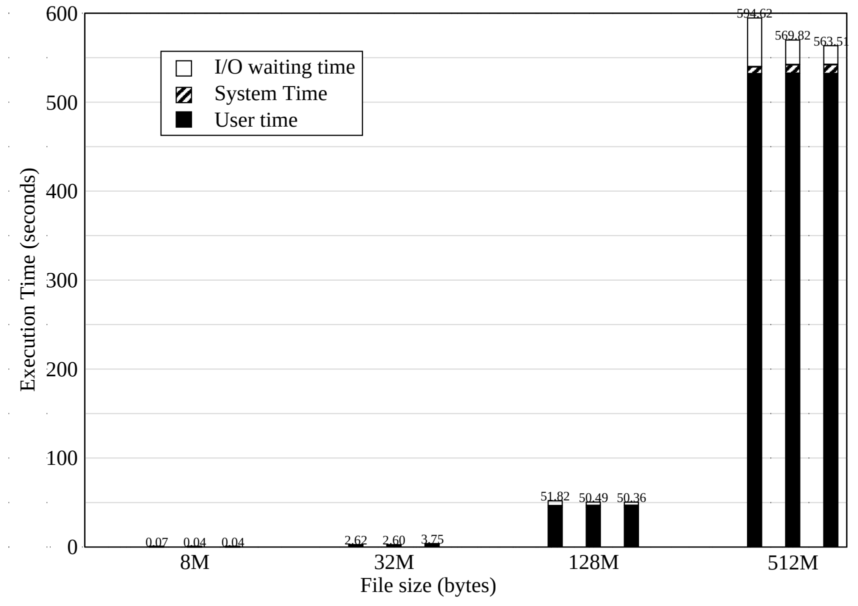


Figure 4.29: **The execution time of merge phase in PDIOL** *sort* with different file sizes on 1 (left), 2 (middle), and 4 (right) disks. The prefetch window size is fixed at 128 pages.

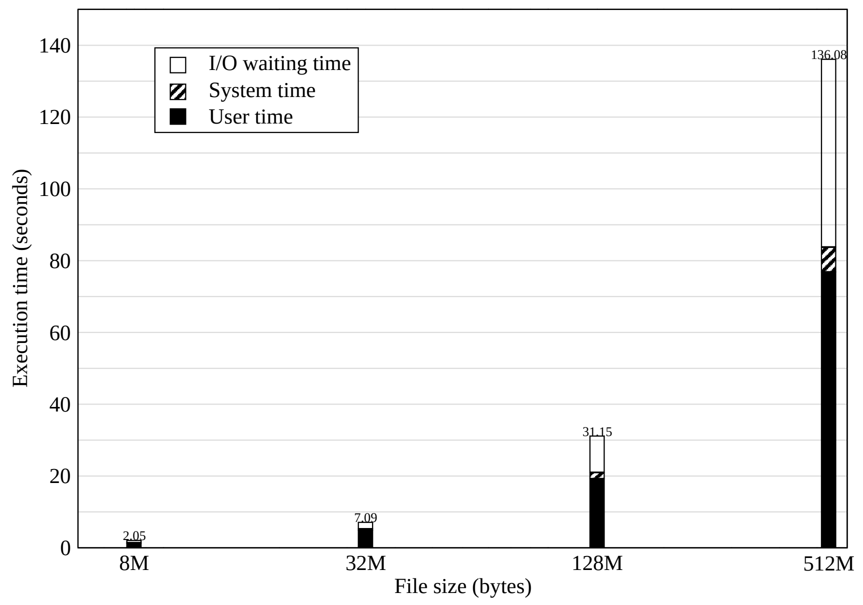


Figure 4.30: **The execution time of block sort phase in** *sort* **algorithm** on the local disk (1 disk).

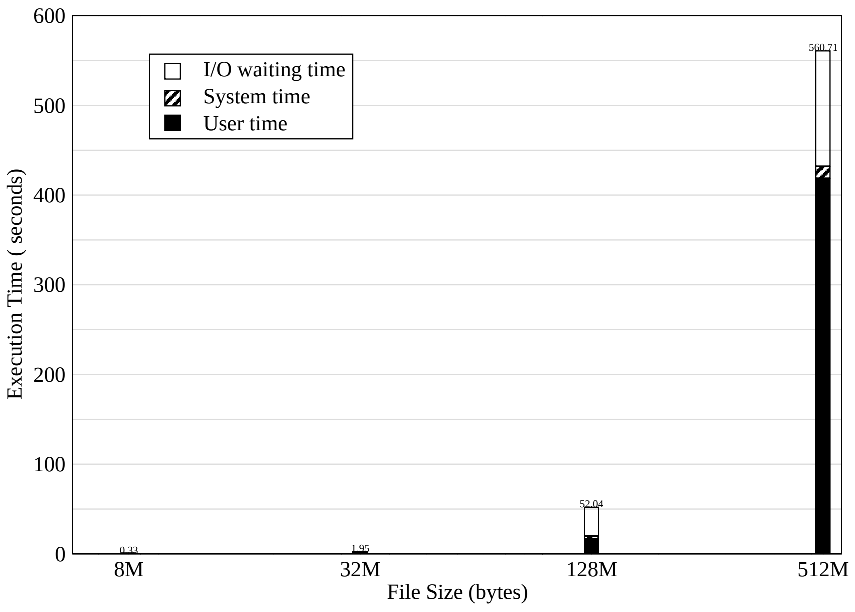


Figure 4.31: **The execution time of merge phase in** ***sort*** **algorithm** on the local disk (1 disk).

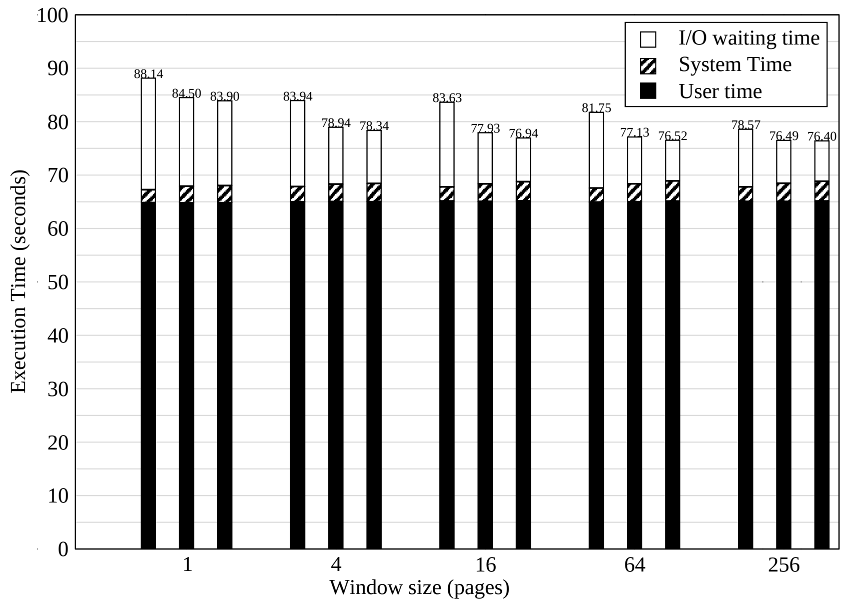


Figure 4.32: **The execution time of PDIOL** *sort* with different prefetch window sizes on 1 (left), 2 (middle), and 4 (right) disks. The file size is fixed at 128MB.

	1 disk				2 disks				4 disks			
Window Size	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2
1	88.14	89.00	87.85	0.238	84.50	85.85	82.49	1.874	83.90	84.24	83.52	0.089
4	83.94	84.02	83.82	0.006	78.94	79.17	78.60	0.066	78.34	78.47	78.20	0.014
16	83.63	83.93	83.42	0.054	77.93	78.40	77.78	0.068	76.94	77.15	76.76	0.023
64	81.75	81.87	81.64	0.012	77.13	77.39	76.89	0.049	76.52	76.62	76.43	0.005
256	78.57	78.72	78.35	0.023	76.49	76.74	76.33	0.024	76.40	76.50	76.30	0.005

Table 4.10: **The execution time of PDIOL** *sort* with different prefetch window size on 1 (left), 2 (middle), and 4 (right) disks. The file size is fixed at 128MB.

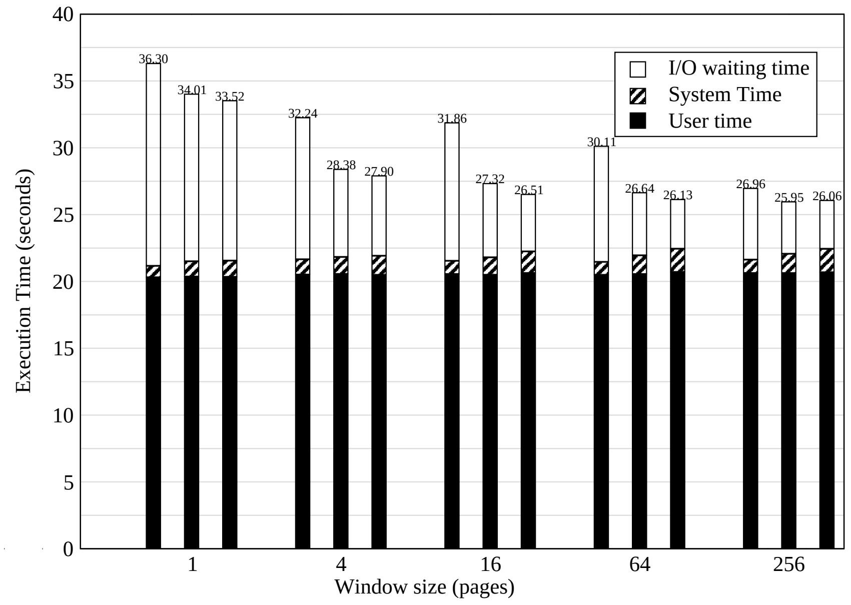


Figure 4.33: **The execution time for block sort phase in PDIOL** *sort* with different sizes of prefetch windows on 1 (left), 2 (middle), and 4 (right) disks. The file size is fixed at 128MB.

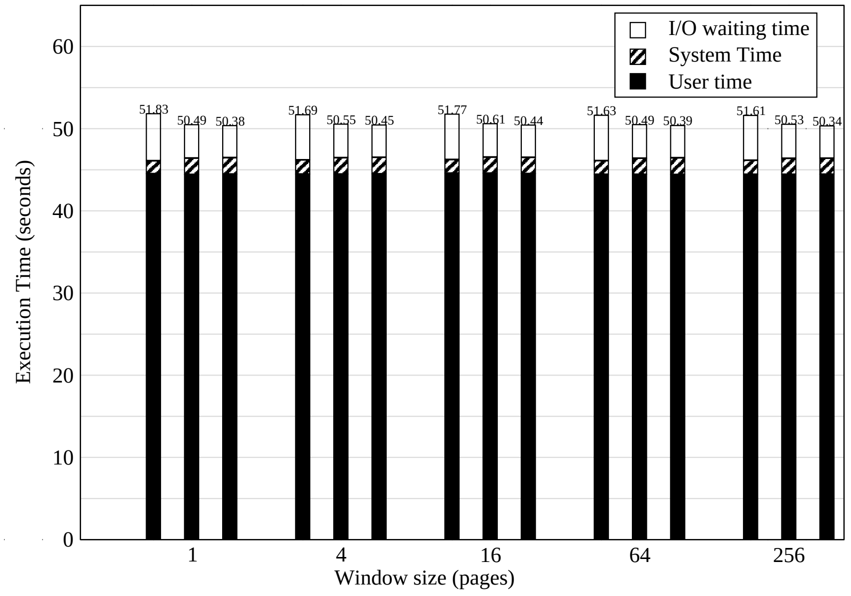


Figure 4.34: **The execution time for merge phase in PDIOL** *sort* with different prefetch window sizes on 1, 2 and 4 disks. The file size is fixed at 128MB.

4.3.5 PDIOL *bzip2*

bzip2 is a freely-available, high-quality data compressor. It compresses data using the algorithm shown in Figure 4.35. The version we used is 1.0.2 [2]. After replacing the standard I/O file functions with PDIOL logical file functions, we generated the PDIOL version of *bzip2*. In order to make a fair comparison between the PDIOL version and the non-PDIOL version, the segment sizes for each compression were both set to 512KB. In the following experiments, we will evaluate the performance of the PDIOL version of *bzip2*.

By varying the number of disks and the size of files to be compressed, we obtain the following results as shown in Figure 4.36 and Table 4.11. In this experiment, we set the size of prefetch windows at 128 pages. From Figure 4.36, we can see that since *bzip2* is a compute-intensive application, I/O waiting time is only a small part of the total execution time. Even with more disks, the execution time does not decrease. It is worth noting that in Figure 4.19 there is more I/O waiting time than there is for the 1-disk case in Figure 4.36. The reason is that for the PDIOL version, the local/remote agents and the PDIOL application work concurrently on two nodes. Before the PDIOL application asks for data to be processed, the data has already been prefetched by the read agent. But for the version without PDIOL, the OS does not prefetch quickly enough, even though *bzip2* is a compute-intensive application.

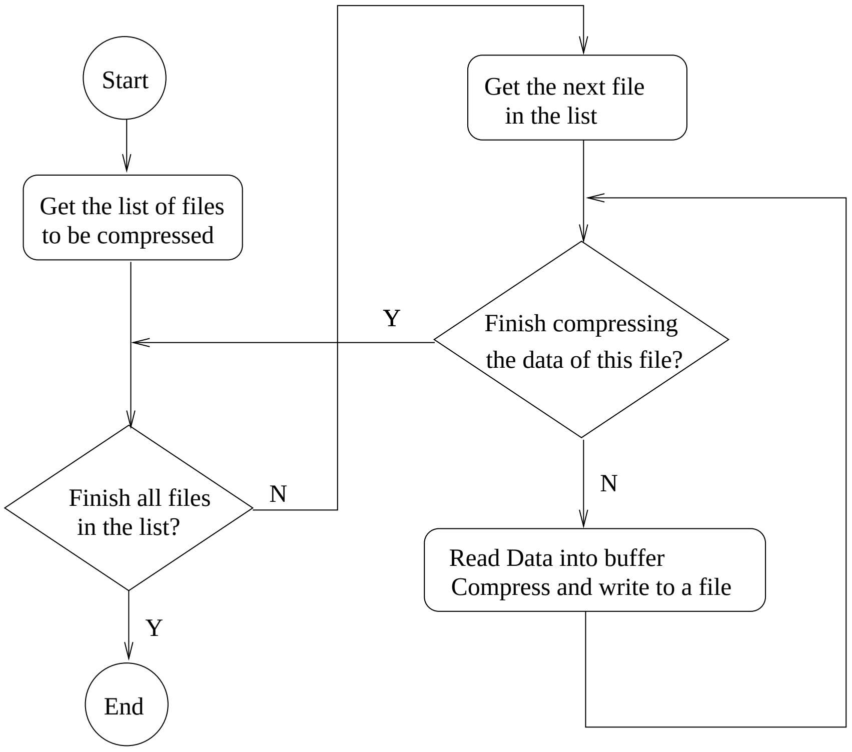

Start
Get the list of files to be compressed
Get the next file in the list
Finish compressing the data of this file?
Y
N
Finish all files in the list?
N
Y
Read Data into buffer Compress and write to a file
End

Figure 4.35: *bzip2* **algorithm**

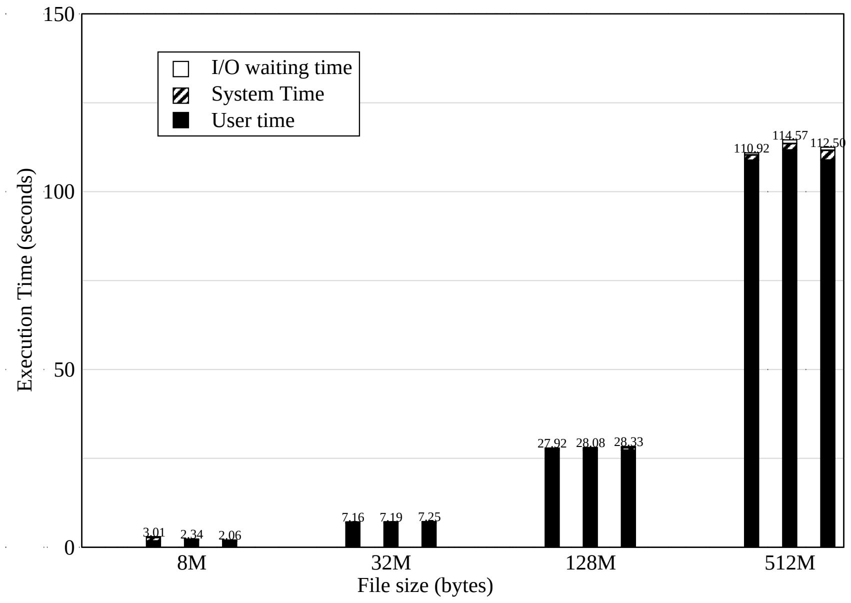


Figure 4.36: **The execution time of PDIOL** *bzip2* with different file sizes on 1 (left), 2 (middle), and 4 (right) disks. The size of the prefetch window is 128 pages.

	1 disk				2 disks				4 disks			
File Size	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2
8MB	2.04	2.07	2.03	0.000	2.02	2.03	2.00	0.000	2.06	2.07	2.04	0.000
32MB	7.16	7.17	7.15	0.000	7.19	7.21	7.18	0.000	7.25	7.27	7.20	0.001
128MB	27.92	27.96	27.88	0.001	28.08	28.15	28.03	0.002	28.33	28.36	28.28	0.001
512MB	110.9	111.2	110.7	0.044	111.6	111.8	111.4	0.033	112.5	112.7	112.4	0.013

Table 4.11: **The execution time of PDIOL** *bzip2* with different file sizes on 1 (left), 2 (midd

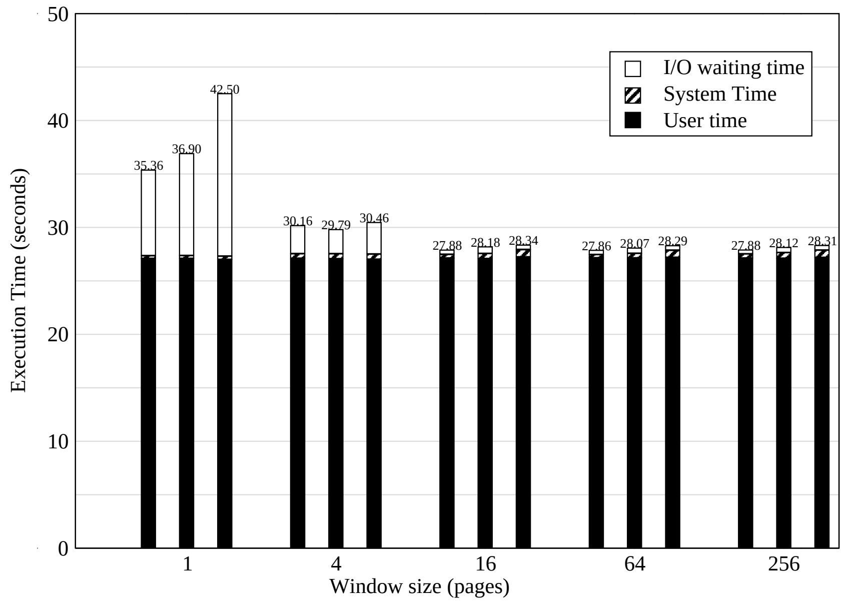


Figure 4.37: **The execution time of PDIOL** *bzip2* with different prefetch window sizes on 1 (left), 2 (middle), and 4 (right) disks. The file size is fixed at 128MB.

By varying the sizes of prefetch windows, we evaluate the PDIOL version of *bzip2* using a 128MB data file. Figure 4.37 and Table 4.12 show the execution time of *bzip2* with different prefetch window sizes.

From the results, we can see that when the size of the prefetch window is one, the performance is much worse; in fact, it is even slower than that of the non-PDIOL version. The reason is that out of every 512KB of data compressed, only 64KB (1 page) of data is prefetched. Furthermore, for every page which is not prefetched, the latency is the sum of the network latency and the disk latency. When the size of the prefetch window increases to 16, the prefetched data size is 16*64KB = 1,024KB. Except for the first segment, other segments can be prefetched before being compressed. Thus, almost all the waiting time for disk I/O is overlapped by the computation. Even with a larger size of the prefetch window, the performance cannot be improved.

	1 disk				2 disks				4 disks			
Window Size	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2	Avg	Max	Min	σ^2
1	35.36	35.43	35.22	0.007	36.90	37.83	36.38	0.300	42.50	42.70	42.28	0.031
4	30.16	30.19	30.12	0.001	29.79	29.91	29.68	0.009	30.46	30.76	30.28	0.036
16	27.88	27.92	27.86	0.001	28.18	28.56	28.05	0.047	28.34	28.39	28.32	0.001
64	27.86	27.89	27.84	0.000	28.07	28.09	28.04	0.001	28.29	28.32	28.24	0.001
256	27.88	27.92	27.84	0.001	28.12	28.15	28.07	0.001	28.31	28.38	28.27	0.002

Table 4.12: **The execution time of PDIOL** *bzip2* with different prefetch window sizes on 1 (left), 2 (middle), and 4 (right) disks. The file size is fixed at 128MB.

4.4 Concluding Remarks

In this chapter, we evaluated the performance of PDIOL with a suite of experiments. First the baseline performance of the network and disk systems were quantified with *Netperf* and *Bonnie*. The higher bandwidth and lower latency of the network, compared with those of disks, support our idea of overcoming the I/O bottleneck with disk parallelism.

The microbenchmarks written using PDIOL evaluate the effect of the number of disks and the size of the prefetch window. The results show that the number of disks and the size of prefetch window should be considered together. Neither of them can bring improvements without the correct configuration of the other factor.

With PDIOL, *grep* reduces much of the I/O waiting time and obtains a speedup of 2.6 with up to 4 remote disks, while *sort* and *bzip2* overlap little I/O waiting time, and gain a marginal speedup. From the results, we can see that I/O-intensive applications provide more benefit, while compute-intensive applications provide less – a result which is consistent with our intuition.

In summary, with PDIOL, multiple disks can be utilized to improve the I/O performance of applications. However, only I/O-intensive applications are able to derive the maximum benefit.

Chapter 5

Concluding Remarks

As new I/O-intensive applications emerge, the need for high-performance I/O increases. One promising approach is through parallel disk systems. In this thesis, the Parallel Disk Input-Output Library (PDIOL) is implemented to utilize parallel disks over a workstation cluster.

The contributions of this thesis include:

1. Design and implementation of PDIOL with a UNIX-like API and the techniques – windows-based prefetching and write-behind (Chapter 3 and Appendices A, B);
2. Evaluation of PDIOL using microbenchmarks and applications (Chapter 4);
3. Evaluation of the impact of parameters, such as the number of disks, the prefetch window sizes, and application characteristics (Chapter 4).

The value of PDIOL can be summarized by the following:

1. PDIOL is implemented completely at the user-level, which makes it unnecessary to recompile the operating system kernel;
2. It is simple to transform an application with normal file access to a PDIOL version with UNIX-like APIs;
3. PDIOL is effective at decreasing I/O waiting time through disk parallelism, prefetch and write-behind which is confirmed by microbenchmarks and application benchmarks.

From the experiments, we draw the following conclusions: PDIOL is especially useful for I/O-intensive applications. For example, PDIOL *grep* gains a speedup of 2.6 with four disks while PDIOL *sort* and *bzip2* gain virtually no speedup. In most cases, larger prefetch windows and a larger number of disks are good for performance. The two factors are correlated, i.e., changing only one factor without changing another, cannot improve performance in certain cases.

5.1 Future Work

Because PDIOL is a user-level library, performance is limited by the overhead of context switches between user-level and kernel-level. The data path of PDIOL shown in Figure 3.2 should be shortened to decrease the contention of memory bus. By implementing the read/write agents at the kernel-level, pinning down memory blocks and transferring data directly to/from user-level area, the overhead of context switch can be decreased. These techniques are well described by Mukherjee *et al.* [29].

In addition to this, the current version of PDIOL supports only sequential applications, which limits its value greatly. It would be ideal to upgrade PDIOL to support parallel applications. However, when different threads can write to the same unit of data, efficiently keeping the data consistent, especially if there are many cache copies, is a challenging problem.

Bibliography

[1] Bonnie Unix filesystem benchmark. `http://www.textuality.com/bonnie/intro.html`, 2003.

[2] bzip2. `ftp://sources.redhat.com/pub/bzip2/v102/bzip2-1.0.2.tar.gz`, 2003.

[3] Intel micro-processors series. `http://www.archivebuilders.com/whitepapers/22016p.pdf`, 2003.

[4] Netperf. `"ftp://ftp.cup.hp.com/dist/networking/benchmarks/`, 2003.

[5] Rotational latency of hard disk, 2003. `http://www.pcguide.com/ref/hdd/op/spinSpeed-c.html`.

[6] D. Anderson, J. Chase, S. Gadde, A. Gallatin, K. Yocum, and M. Feeley. Cheating the i/o bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the Usenix Technical Conference.*, New Orleans, LA, 1998.

[7] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, CO, December 1995.

[8] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[9] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly and Associate4s, Inc, Sebastopol, CA, 2001.

[10] J. Braam and M. Callahan. Lustre: A SAN file system for Linux. `http://www.lustre.org/docs/luswhite.pdf`, 2003.

[11] A. Brown and T. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 31–44, Berkeley, CA, October 2000. `http://citeseer.nj.nec.com/brown00taming.html`.

[12] P. Cao, E. Felten, A. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[13] P. Cao, E. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Operating Systems Design and Implementation*, pages 165–177, November 1994.

[14] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[15] P. Chen and E. Lee. Striping in a RAID level 5 disk array. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 136–145, Ottawa, Canada, May 1995.

[16] P. Chen and D. Patterson. Maximizing performance in a striped disk array. In *Proceedings of 17th Annual International Symposium on Computer Architecture, ACM SIGARCH Computer Architecture News*, page 322, 1990.

[17] P. Chen and D. Patterson. Storage performance—metrics and benchmarks. In *Proceedings of the IEEE 81(8)*, pages 1151–1165, August 1993.

[18] Hewlett-Packard Company. Netperf. `http://www.netperf.org/netperf/NetperfPage.html`.

[19] M. Farley. *Building Storage Networks, Second Edition*. Osborne/McGraw-Hill, 2001.

[20] G. Gibson, D. Nagle, K. Amiri, J. Butler, and F. Chang. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, Portland, OR, 1998. ACM Press and IEEE Computer Society Press.

[21] G. Gibson, D. Nagle, K. Amiri, F. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, and D. Rochberg. A case for network-attached secure disks. Technical Report CMU–CS-96-142, Department of Electrical and Computer Engineering, Carnegie-Mellon University, June 1996.

[22] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *J. Griffioen and R. Appleton. Performance Measurements of Automatic Prefetching. In Paralel and Distributed Computing Systems, pages 165– 170, IEEE, September 1995.*, 1995.

[23] Myricom Inc. Myrinet clusters in the June 2003 TOP500 List. `http://www.myrinet.com/news/03623/index.html`, 2003.

[24] T. Kimbrel, A. Tomkins, R. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 19–34, Seattle, Washington, October 1996. USENIX Association.

[25] N. Kronenberg, H. Levy, and W. Strecker. Vaxcluster: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, pages 130–146, May 1986.

[26] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference*, Anaheim, California, 1997.

[27] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, San Diego, CA, January 1996.

[28] T. Mowry, A. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, 1996.

[29] S. Mukherjee and M. Hill. A survey of user-level network interfaces for system area networks. Technical Report CS-TR-1997-1340, Computer Sciences Department, University of Wisconsin–Madiso, February 1997.

[30] W. Norcott and D. Capps. Iozone filesystem benchmark, 2003. `http://www.iozone.org/docs/IOzone_msword_98.pdf`.

[31] P. Chen and E. Lee and G. Gibson and R. Katz and D. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[32] A. Park and J. Becker. Iostone: A synthetic file system benchmark. In *Computer Architecture News*, pages 45–52, 1990.

[33] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 224–244. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[34] E. Riedel. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[35] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *In Proceedings of the First Conference on File and Storage Technologies*, January 2002. `citeseer.nj.nec.com/540208.html`.

[36] P. Shenoy and H. Vin. Efficient striping techniques for variable bit rate continuous media file servers. Technical Report UM-CS-1998-053, Department of Computer Science, University of Massachusetts at Amherst, December 1998.

[37] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[38] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations*. Prentice Hall, 1999.

[39] C. Yang, T. Mitra, and T. Chiueh. A decoupled architecture for application-specific file prefetching. In *Freenix Track of USENIX 2002 Annal Conference*, Monterey, CA, June 2002.

Appendix A

PDIOL Usage

In this appendix, we demonstrate how to use the PDIOL library. First, the header files and library files of PDIOL will be described. Second, a sample configuration file of PDIOL nodes is given. Third, we will detail the options of PDIOL applications. Finally a sample program which transfers a PDIOL logical file striping over PDIOL nodes to a normal file is given.

A.1 Header Files and Library Files

To use PDIOL, an application needs to include the header file of PDIOL and be linked with the PDIOL library. The header file is *pdiol.h* and the library is *libpdiol.a*. In addition to these two files, *tlv.h* and *utility.h* contain prototypes of some convenient routines for tuning PDIOL programs.

A.2 PDIOL Nodes Configuration File

The configuration file of PDIOL nodes is located in *$HOME/bin/conf*, which contains all the computer names in the cluster. The configuration file for our experiments is shown in Figure A.2.

A.3 Command-line Options

PDIOL provides some options for controlling the behavior of its routines. In order to differentiate between the PDIOL options and the options of an application, the symbol ”–” separator is used in the command line of PDIOL applications. The arguments before ”–” will be regarded as the options for PDIOL, and the rest are regarded as the arguments of the application.

The command line to run a PDIOL application has the following form:

```
brule-m-a
brule-m-b
brule-m-c
brule-m-d
```

Figure A.1: **Configuration file of PDIOL nodes.**

long option	short	argument	meaning	default	environment variable
prefetch	p	0/1	Prefetch enabled	1	PREFETCH
write-behind	w	0/1	Write-behind enabled	0	WRITEBEHIND
disk_number	d	unsigned int	Disk number	1	DISKNUMBER
window	W	unsigned int	Prefetch window size	1	WINDOW_SIZE
ssock	s	unsigned int	send socket buffer size	65,536	SND_BUFF
rsock	r	unsigned int	receive socket buffer size	65,536	RCV_BUFF
LOG	l	0/1	Logging enabled	1	LOG
log	L	char*	Log file name	log.txt	LOG_FILE
Help	h		Help	0	

Table A.1: **PDIOL options.**

pdiol-applicaiton [PDIOL options] – *application_argument*

The PDIOL options are detailed in Table A.1. For example, the following command is used to transfer a pdiol file *sort.txt* striped over 4 disks to a traditional file *sorttxt.bak* with prefetch enabled and write-behind disabled:

```
pfs2fs -disk_number 4 -prefetch 1 -w 0 – sort.txt sorttxt.bak
```

A.4 Sample Program

A sample program named pfs2fs.c is shown in Figure A.4. The purpose of this program is to transfer a PDIOL logical file striped across a workstation cluster into a normal file.

```
/* pfs2fs.c - To transfer a PDPL file to a normal file
 */

#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <sys/time.h>
#include <unistd.h>
#include "pdiol.h"

#define BUFFERSIZE    655360
int main(int argc, char** argv)
{
        PFS_FILE* pf;
        FILE      *outf;
        int size, count, w_count = 0;
        char   pbuffer[BUFFERSIZE+1], *input, *output;

        initialize_pdiol (&argc, &argv); // initialize pdiol

        if(argc < 2 || argc > 3){
         fprintf(stderr, "\n%s file [pdiol_file]\n", argv[0]);
         exit(1);
        }

        input = argv[1];

        if(argc ==3) output = argv[2];
        else output = NULL;

        pf = pdiol_fopen(input, "r"); //open pdiol file
        pdiol_hint(pf, AP_SEQUENTIAL, 0);  // hint

        if(output != NULL) outf = fopen(output, "w");
        else outf = fdopen(1, "w");

        w_count = 0;

        while(!pdiol_feof(pf)) {
                count = pdiol_fread(pbuffer, 1, BUFFERSIZE, pf);
                w_count += fwrite(pbuffer, 1, count, outf);
        }

        fclose(outf);
        pdiol_fclose(pf); //close pdiol file

        finalize_pdiol();  //finalize pdiol
}
```

Figure A.2: **PDIOL sample program – pfs2fs.c**.

Appendix B

PDIOL API Reference

PDIOL provides the following APIs, which are presented in the same format as UNIX man pages.

B.1 initialize_pdiol;

Initialize the PDIOL environment

Synopsis

```
#include <pdiol.h>
int initialize_pdiol(int* pargc, char*** pargv);
```

Description

The `initialize_pdiol` function initializes the PDIOL environment. This function should be called before any other statements. pargc is the pointer of the argc argument of the main entry function. pargv is the pointer of the argv argument of the main entry function. It sets the PDIOL library option in the following sequence: command line arguments; environment variable; hard coded default value;

Return Value

Upon successful completion, 0 is returned. Otherwise, -1 is returned

B.2 finalize_pdiol

Finalize PDIOL environment

Synopsis

```
#include <pdiol.h>
int finalize_pdiol();
```

Description

The function `finalize_pdiol` finalizes the PDIOL environment. This function should be called after any statements related to PDIOL. It closes the connections between PDIOL applications and remote agents, and releases PDIOL resources.

Return Value

Upon successful completion, 0 is returned. Otherwise, -1 is returned

B.3 pdiol_fopen;

pdiol_fopen – PDIOL logical file open function

Synopsis

```
#include <pdiol.h>
PFS_FILE* *pdiol_fopen (const char * const path,
                        const char * const mode);
```

Description

The `pdiol_fopen` function opens the PDIOL logical file whose name is the string pointed to by path. The argument mode is the pointer of a string which is one of the following sequences: ”r”, ”r+”, ”w”, ”w+”, ”a”, ”a+”. The meaning of these strings is the same as the mode in standard fopen. See fopen for detailed description.

Return Value

Upon successful completion, a PFS_FILE pointer is returned. Otherwise, NULL is returned.

B.4 pdiol_fclose;

pdiol_fclose – close a PDIOL logical file

Synopsis

```
#include <pdiol.h>
int pdiol_fclose(PFS_FILE* pFile);
```

Description

The `pdiol_fclose` function closes the PDIOL logical file pointed to by pFile. If the PDIOL logical file was being used for output, any buffered data will be written out when finalize_io is called, or its buffer is reclaimed by the PDIOL library.

Return Value

Upon successful completion, 0 is returned. Otherwise, -1 is returned

B.5 pdiol_fstat;

pdiol_fstat – get the status of a PDIOL logical file.

Synopsis

```
#include <pdiol.h>
int int pdiol_fstat(struct pdiol_file* pf,
                    struct pdiol_state_struct* pstat);
```

Description

The `pdiol_fstat` function obtains the status of a PDIOL logical file pointed to by pf and stores the status information in a pdiol_state_struct structure pointed to by pstat.

Return Value

Upon successful completion, 0 is returned. Otherwise, -1 is returned

B.6 pdiol_stat;

pdiol_stat – get the status of a PDIOL logical file.

Synopsis

```
#include <pdiol.h>
int int pdiol_stat(char* const path,
                   struct pdiol_state_struct* pstat);
```

Description

The `pdiol_stat` function obtains the status of a PDIOL logical file whose path is pointed to by path and stores the status information in a pdiol_state_struct structure pointed to by pstat.

Return Value

Upon successful completion, 0 is returned. Otherwise, -1 is returned

B.7 pdiol_unlink;

pdiol_unlink – unlink a PDIOL logical file.

Synopsis

```
#include <pdiol.h>
int pdiol_unlink(const char* const path);
```

Description

The `pdiol_unlink` function unlinks a PDIOL logical file whose path is pointed to by path. Currently, every PDIOL logical file has only one link and does not support multiple links.

Return Value

Upon successful completion, 0 is returned. Otherwise, -1 is returned

B.8 pdiol_fflush;

pdiol_fflush – flush a PDIOL logical file

Synopsis

```
#include <pdiol.h>
int pdiol_fflush(PFS_FILE *pf);
```

Description

The `pdiol_fflush` function forces a write of all user-space buffered data for the given output of a PDIOL logical file pointed to by pf. The open status of the PDIOL logical file is unaffected.

Return Value

Upon successful completion, 0 is returned. Otherwise, -1 is returned

B.9 pdiol_fread;

pdiol_fread – read data from a PDIOL logical file

Synopsis

```
#include <pdiol.h>
size_t  pdiol_fread(void * ptr, size_t size,
                     size_t nmemb, PFS_FILE * pf);
```

Description

The `pdiol_fread` function reads nmemb elements of data, each size bytes long, from the PDIOL logical file pointed to by pf, storing them at the location given by ptr.

Return Value

Returns the number of items successfully read. If an error occurs, -1 will be returned.

B.10 pdiol_fwrite;

pdiol_fread – write data to a PDIOL logical file

Synopsis

```
#include <pdiol.h>
size_t  pdiol_fwrite(void * ptr, size_t size,
                      size_t nmemb, PFS_FILE * pf);
```

Description

The `pdiol_fwrite` function writes nmemb elements of data, each size bytes long, to the PDIOL logical file pointed to by pf, obtaining them from the location given by ptr.

Return Value

Returns the number of items successfully written. If an error occurs, -1 will be returned.

B.11 pdiol_fseek;

pdiol_fseek – re-position a PDIOL logical file

Synopsis

```
 #include <pdiol.h>
int pdiol_fseek(PFS_FILE * pf, long offset, int whence);
```

Description

The `pdiol_fseek` function sets the file position of the PDIOL logical file pointed to by pf. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence. The semantics of whence is the same as that of standard fseek.

Return Value

Returns 0 if successful, otherwise returns -1.

B.12 pdiol_ftell;

pdiol_ftell – get the file position.

Synopsis

```
#include <pdiol.h>
long pdiol_ftell( const PFS_FILE *const pf);
```

Description

The `pdiol_ftell` function obtains the current value of the file position indicator for the PDIOL logical file pointed to by pf.

Return Value

Returns the current value of the file position

B.13 pdiol_hint;

pdiol_hint – set the access pattern of a PDIOL logical file.

Synopsis

```
#include <pdiol.h>
int pdiol_hint(PFS_FILE *pf, int access_pattern, int hint);
```

Description

The `pdiol_hint` function sets the access pattern and hint of a PDIOL file pointed to by pf. The access pattern is specified by access_pattern. Currently, only `AP_SEQUENTIAL` is supported. hint specifies special characteristics of file access. Currently the hint `FINAL_WRITE`, which means the file won't be accessed in this application after being closed, is supported.

Return Value

Returns 0 upon successful completion, otherwise returns -1.

Appendix C

Thread Log Viewer

C.1 Overview

Thread Log Viewer (TLV) is a tracing tool for debugging/tuning multiple threaded applications. The method is to record the activity of some or all of the threads in one process and visualize the log file in a graphical interface. With TLV, the following information can be obtained:

1. The start and end time of a called routine;
2. The total number, total execution time, and distribution of a certain routine;
3. The concurrency of routines between different threads in one process.

TLV includes a TLV library and a viewer. The TLV library was developed with C language, while the viewer was developed with Java language.
In order to use TLV, four steps are necessary:

1. Include the header file of TLV and insert TLV API calls;
2. Compile the application;
3. Run the application and create log files;
4. Use TLV to show the log file and analyze the execution of the application.

C.2 TLV API

1. TLV_Initialize – Initialize TLV and open TLV log file;
2. TLV_Finalize – Finalize TLV and close TLV log file;
3. TLV_AddState – Add the states to be recorded;
4. TLV_AddThread – Let certain thread be logged;
5. TLV_Record – Record the state of current thread.

C.3 Sample Session

In this sample session, we will show how to use TLV to record and visualize the execution of multiple threaded applications. Figure C.1 and C.2 include the source code for the sample program, which uses a parallel random sampling sort algorithm to sort an array of integers.

Figures C.3, C.4, C.5, C.6, and C.7 are snapshots of of this sample session.

There are three modes of view:

1. Event duration view – Shows the duration of every recorded routine. See Figure C.3;
2. Event sequence view – Shows the sequence of every recorded routine. See Figure C.4;
3. Event distribution view – Shows the execution time distribution of recorded routines. See Figure C.5.

Figure C.6 shows the events and their color representations.

A summary view is provided to offer general information about an event. It is shown in Figure C.7.

Three commands for analyzing the execution of a process include:

1. Total duration – Obtain the total execution of a certain routine;
2. Overlap – Obtain the overlap time between routines;
3. Single duration – Obtain the duration of single called routines.

Besides these commands, TLV also provides commands for zoom-in, zoom-out, and move view, in order to provide a more clear view. TLV can open multiple log files of different processes in different windows, thus making it easy to see the interaction between processes.

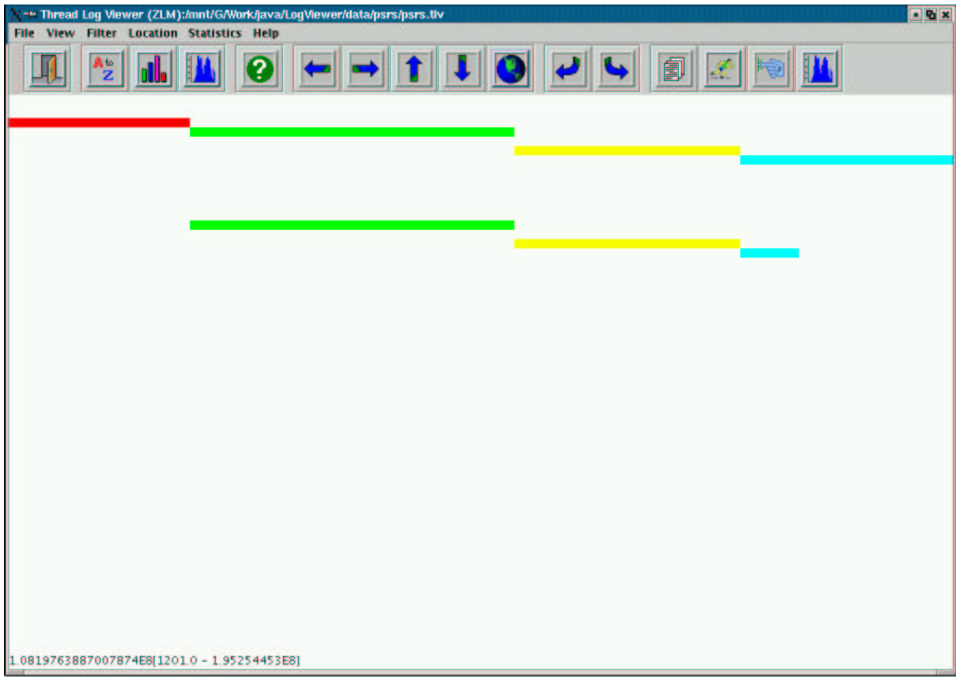
Conclusion TLV is helpful in debugging/tuning multiple-threaded programs. With it, some important bugs/bottlenecks in PDIOL were removed during development. However, TLV has considerable room for improvement. One possibility would be to show the synchronization between threads. Also, TLV cannot detect the context switch information because TLV library is a user-level library. For example, TLV shows that two threads run some routines concurrently, and the reality is that they perhaps use only one processor interchangeably. Thus, the current version of TLV is only suitable for applications in which there is no competion for processors.

```
/* TLV state and the name of the state */
enum tlv_states {INIT, INIT_END, P1, P1_END,
                 P2, P2_END, P3, P3_END, P4, P4_END };

char* tlv_state_names[] = { "INIT", "INIT_END", "Phase1", "P1_END",
      "PHASE2", "P2_END", "PHASE3", "P3_END", "PHASE4", "P4_END" };

main(int argc, char* argv[])
{
 int   length, *arg, index, start, end, count, my_seqid;
 char*  cs;

 over_sampling = atof(argv[3]);
 length = atoi(argv[1]);
 thread_number = atoi(argv[2]);
 my_seqid = 0;

 //TLV initalize
 TLV_Initialize("psrs");

 TLV_AddState(INIT, tlv_state_names[INIT],
              INIT_END, tlv_state_names[INIT_END], TLV_RED);
 TLV_AddState(P1, tlv_state_names[P1],
              P1_END, tlv_state_names[P1_END], TLV_GREEN);
 TLV_AddState(P2, tlv_state_names[P2],
              P2_END, tlv_state_names[P2_END], TLV_BLUE);
 TLV_AddState(P3, tlv_state_names[P3],
              P3_END, tlv_state_names[P3_END], TLV_YELLOW);
 TLV_AddState(P4, tlv_state_names[P4],
              P4_END, tlv_state_names[P4_END], TLV_CYAN);

 TLV_Record(INIT);

 my_array.RandomGenerate(length);
 my_array.divide(args_for_thread);

 TLV_Record(INIT_END);

 for(int i=1;i<thread_number;i++){
    arg = args_for_thread + 3*i;
    *arg = i;
    pthread_create(&(threads[i].thread_id),NULL,slave, (void*)arg);
 }

 arg =  args_for_thread;
 slave(arg);

 for(int i=1;i<thread_number;i++) pthread_join(threads[i].thread_id, NULL);

 index = 0;

 for(int i=0;i<thread_number;i++){
     memcpy(my_array.nArray+index,threads[i].local_array,
                                  threads[i].local_array_len*sizeof(int));
     index = index + threads[i].local_array_len;
  }

 TLV_Finalize(); // Finalize TLV
}
```

Figure C.1: **Parallel Random Sampling Sort – main Program.**

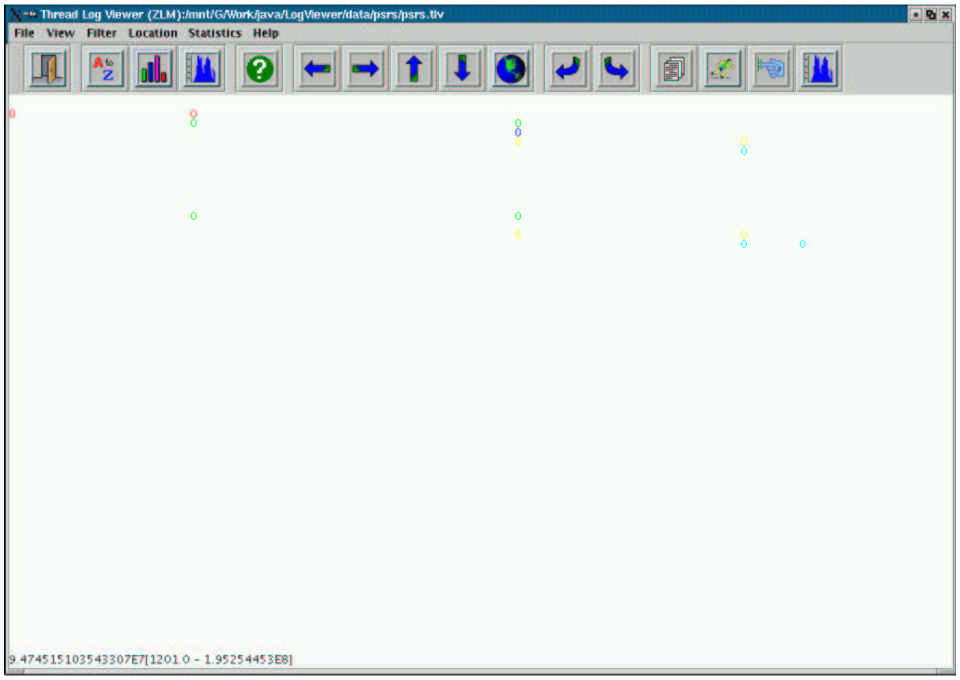
```
void* slave(void* args)
{
  int* my_arg = (int*)args;
  int  seq = *my_arg;
  int start = *(my_arg+1);
  int end   = *(my_arg+2);
  int index,i,count;

  pthread_t my_tid = pthread_self();

  // only the child thread need to add thread to record as
  // the main thread has done in TLV_Intialize
  if(args !=  args_for_thread) TLV_AddThread();

  TLV_Record(PHASE1_START);

  myarr.quicksort(start, end); /* sort */
  myarr.GetSampling(my_arg);  /* sampling */
  barrier(thread_number, AFTER_PHASE1);

  TLV_Record(PHASE1_END);

  if(seq==0){
    TLV_Record(PHASE2_START);
    sampling();
    TLV_Record(PHASE2_END);
  }

  barrier(thread_number, AFTER_PHASE2);

  TLV_Record(PHASE3_START);

  index = start;
  for(i=0;i<thread_number;i++){
    int pivot;
    if(i<thread_number-1) pivot = myarr.nPivots[i];
    else pivot = MAX_VALUE;

    count = 0;
    while(myarr.nArr[index+count]<=pivot&&index+count<=end) count++;

    if(count>0){
     if((thrds[i].parts[seq]=(int*)malloc(sizeof(int)*count))==NULL){
        printf("failure to allocate memory\n");
        exit(1);
      }
      thrds[i].parts_len[seq] = count;
      memcpy(thrds[i].parts[seq], myarr.nArr+index, count*sizeof(int));
      index = index+count;
    }
  }

  /* partition & distribute parts */
  barrier(thread_number, AFTER_PHASE3);

  TLV_Record(PHASE3_END);
  TLV_Record(PHASE4_START);
  myarr.mergePartitions(seq);
  TLV_Record(PHASE4_END);

  return NULL;
}
```

Figure C.2: **Parallel Random Sampling Sort – slave Program.**


Thread Log Viewer (ZLM):/mnt/G/Work/java/LogViewer/data/psrs/psrs.tlv
File
View
Filter
Location
Statistics
Help

```
1.0819763887007874E8[1201.0 - 1.95254453E8]
```

Figure C.3: **Snapshot of Thread Log Viewer – Event Duration View.**


Thread Log Viewer (ZLM):/mnt/G/Work/java/LogViewer/data/psrs/psrs.tlv
File
View
Filter
Location
Statistics
Help

```
9.474515103543307E7[1201.0 - 1.95254453E8]
```

Figure C.4: **Snapshot of Thread Log Viewer – Event Sequence View.**

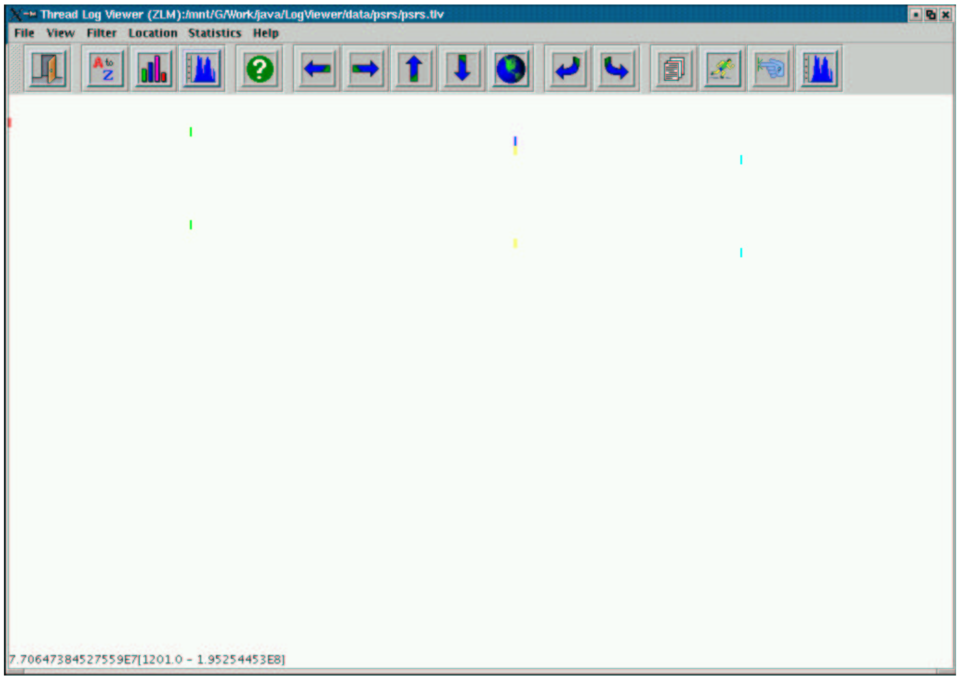

Thread Log Viewer (ZLM):/mnt/G/Work/java/LogViewer/data/psrs/psrs.tlv
File View Filter Location Statistics Help
7.70647384527559E7[1201.0 - 1.95254453E8]

Figure C.5: **Snapshot of Thread Log Viewer – Event Distribution View.**

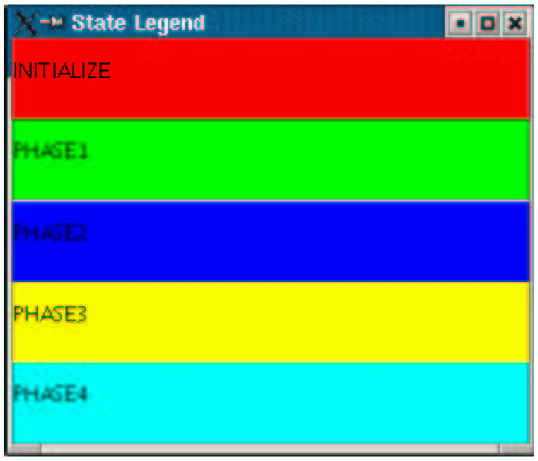
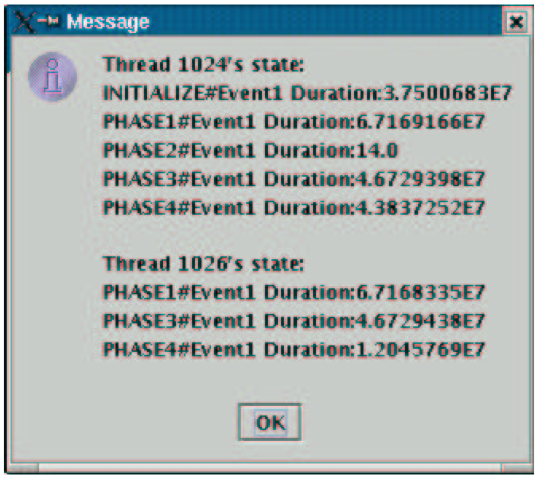


Figure C.6: **Snapshot of Thread Log Viewer – Event Legend.**



```
Thread 1024's state:
INITIALIZE#Event1 Duration:3.7500683E7
PHASE1#Event1 Duration:6.7169166E7
PHASE2#Event1 Duration:14.0
PHASE3#Event1 Duration:4.6729398E7
PHASE4#Event1 Duration:4.3837252E7

Thread 1026's state:
PHASE1#Event1 Duration:6.7168335E7
PHASE3#Event1 Duration:4.6729438E7
PHASE4#Event1 Duration:1.2045769E7
```

Figure C.7: **Snapshot of Thread Log Viewer – Summary View.**