PRINCIPAL DESIGN CRITERIA INFLUENCING THE PERFORMANCE OF A PORTABLE, HIGH PERFORMANCE PARALLEL I/O IMPLEMENTATION

By
Kumaran Rajaram

A Thesis
Submitted to the Faculty of Mississippi State University in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science in the Department of Computer Science

Mississippi State, Mississippi
May 2002
Copyright by
Kumaran Rajaram
2002
PRINCIPAL DESIGN CRITERIA INFLUENCING THE PERFORMANCE OF A PORTABLE, HIGH PERFORMANCE PARALLEL I/O IMPLEMENTATION

By

Kumaran Rajaram

Approved:

Anthony Skjellum  
Associate Professor of  
Computer Science  
(Major Professor)

Donna S. Reese  
Associate Professor of  
Computer Science  
(Committee Member)

Tomasz Haupt  
Research Associate Professor of  
Computer Science  
(Committee Member)

Julian E. Boggess  
Associate Professor of  
Computer Science and Graduate Coordinator of the Department of Computer Science

A. Wayne Bennett  
Dean of the College of Engineering
Name: Kumaran Rajaram
Date of Degree: May 11, 2002
Institution: Mississippi State University
Major Field: Computer Science
Major Professor: Dr. Anthony Skjellum

Title of Study: PRINCIPAL DESIGN CRITERIA INFLUENCING THE PERFORMANCE OF A PORTABLE, HIGH PERFORMANCE PARALLEL I/O IMPLEMENTATION

Pages in Study: 84
Candidate for Degree of Master of Science

MPI-IO, the parallel I/O functionality of MPI-2, is a portable interface designed specifically to achieve high-performance. This thesis proposes fundamental design criteria influencing the performance of a portable high performance I/O middleware. This thesis hypothesizes that overlap of I/O and computation and agglomeration of I/O requests based on an application’s access pattern improve the performance of a portable parallel I/O implementation. The work included the development of MercutIO, a complete, portable, high performance MPI-IO implementation. MercutIO achieves portability through the Bulldog Abstract File System, a portable, efficient non-collective I/O interface, also developed in this thesis work. A new data access model based on non-blocking semantics is presented here. Two new I/O metrics (degree of overlapping and degree of non-contiguity) as well as parallel I/O benchmarks essential in the performance appraisal of a parallel I/O implementation are introduced in this thesis.
DEDICATION

To my brother and my parents.
ACKNOWLEDGMENTS

The author would like to express his sincere gratitude towards his major advisor, Dr. Anthony Skjellum, for his support and guidance in the course of this research work. The author would also like to thank his committee members – Dr. Donna Reese and Dr. Tomasz Haupt. Also, special thanks to Dr. Rossen Dimitrov, Mr. Purushotham Bangalore, the author’s colleagues at MPI Software Technology, Inc. and at the High Performance Computing Laboratory for their invaluable suggestions and constant moral support. Finally, the author would like to thank Ms. Neli Fairfield for helping him in the copy-editing of this thesis.
# TABLE OF CONTENTS

| DEDICATION | ......................................................... | ii |
| ACKNOWLEDGMENTS | ........................................ | iii |
| LIST OF TABLES | ........................................ | vii |
| LIST OF FIGURES | ........................................ | viii |

## CHAPTER

### I. INTRODUCTION ................................. 1

1.1 Background .................................. 1
1.2 Hypothesis .................................. 2
1.3 Motivation .................................. 2
1.4 Contributions ............................. 3
1.5 Organization ............................. 4

### II. LITERATURE REVIEW ......................... 5

2.1 The CHARISMA Project ............................ 6
2.2 File System ................................. 7
2.2.1 Sequential File System .................. 7
2.2.2 Distributed File System ................. 8
2.2.3 Parallel File System ................... 8
2.3 Parallel I/O Library .......................... 9
2.3.1 The Panda Project ....................... 10
2.3.2 The Parallel and Scalable Software for I/O (PASSION) Project 10
2.4 MPI-IO ...................................... 11
2.4.1 Support for MPI-IO Features on Existing File Systems .... 13
2.5 Existing MPI-IO Implementations ............ 17
2.5.1 ROMIO .................................. 18
2.5.2 MPI-IO over GPFS ....................... 19
2.6 Summary ................................... 20

iv
## III. OVERLAPPING OF I/O AND COMPUTATION AND AGGLOMERATION OF I/O REQUESTS: A QUANTITATIVE ANALYSIS

3.1 Overlapping of I/O and Computation .................................................. 22  
  3.1.1 Theoretical Description ......................................................... 22  
  3.1.2 Degree of Overlapping: Definition and Evaluation Strategy .......... 24  
  3.1.3 Practical Use of Overlapping .................................................. 25  
3.2 Agglomeration of Smaller I/O Requests to Optimal Sized Chunks .......... 26  
  3.2.1 Theoretical Description ....................................................... 27  
    3.2.1.1 Non-Collective I/O Involving Non-contiguous File Accesses .......... 27  
    3.2.1.2 Collective I/O Involving Non-contiguous File Accesses .......... 28  
    3.2.1.3 Non-contiguous File Accesses Involving Large Holes with Considerably Smaller Non-contiguous Chunks .......... 29  
    3.2.1.4 Non-contiguous File Accesses Involving Large Non-contiguous Chunks .......... 30  
  3.2.2 Degree of Non-contiguity ..................................................... 31  
  3.2.3 Procedure for Determining Degree of Non-contiguity .............. 31  
3.3 Summary ......................................................................................... 32

## IV. MERCUTIO: PORTABLE, HIGH-PERFORMANCE MPI-IO IMPLEMENTATION

4.1 MercutIO Architecture .................................................................... 33  
4.2 BAFS ......................................................................................... 36  
  4.2.1 BAFS Design ............................................................................ 37  
  4.2.2 BAFS Implementation ............................................................... 40  
4.3 The MPI-IO Component of MercutIO ........................................... 41  
  4.3.1 Design .................................................................................... 42  
  4.3.2 Implementation ....................................................................... 44  
4.4 Summary ......................................................................................... 44

## V. EXPERIMENTS, RESULTS, AND ANALYSIS

5.1 Hardware and Software Configuration .......................................... 46  
5.2 Latency Measurements ..................................................................... 48  
  5.2.1 Latency Program Description .................................................... 48  
  5.2.2 Experimental Results and Analysis ........................................... 48  
5.3 Parallel-IO Benchmark for Non-contiguous access (PIOBN) ........... 49  
  5.3.1 PIOBN Description .................................................................... 49  
  5.3.2 Expected Results ...................................................................... 50
LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>MPI-IO Support on Existing File Systems</td>
</tr>
<tr>
<td>2.2</td>
<td>Asynchronous MPI-IO Implementation Approaches</td>
</tr>
<tr>
<td>5.1</td>
<td>Latency of Non-Collective and Collective MPI-IO Implementation</td>
</tr>
<tr>
<td>5.2</td>
<td>Read Degree of Non-contiguity on MercutIO-UFS</td>
</tr>
<tr>
<td>5.3</td>
<td>Write Degree of Non-contiguity on MercutIO-UFS</td>
</tr>
<tr>
<td>5.4</td>
<td>Read Degree of Non-contiguity on MercutIO-NFS</td>
</tr>
<tr>
<td>5.5</td>
<td>Write Degree of Non-contiguity on MercutIO-NFS</td>
</tr>
<tr>
<td>5.6</td>
<td>Degree of Overlapping on MercutIO-UFS</td>
</tr>
<tr>
<td>5.7</td>
<td>Degree of Overlapping on MercutIO-NFS</td>
</tr>
<tr>
<td>5.8</td>
<td>Degree of Overlapping on MercutIO-PVFS</td>
</tr>
<tr>
<td>5.9</td>
<td>Portability Overhead of MercutIO</td>
</tr>
<tr>
<td>5.10</td>
<td>Non-Blocking Semantics Overhead of MercutIO</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 MPI-IO Model</td>
<td>12</td>
</tr>
<tr>
<td>2.2 MPI-IO Organization</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Non-contiguous I/O Access Pattern</td>
<td>17</td>
</tr>
<tr>
<td>4.1 MercutIO Architecture</td>
<td>34</td>
</tr>
<tr>
<td>4.2 Non-Collective BAFS</td>
<td>35</td>
</tr>
<tr>
<td>4.3 Multiple I/O Targets</td>
<td>36</td>
</tr>
<tr>
<td>4.4 BAFS Model for Non-Blocking I/O</td>
<td>37</td>
</tr>
<tr>
<td>4.5 Data-Sieving: Non-collective, Non-contiguous Optimization</td>
<td>39</td>
</tr>
<tr>
<td>4.6 Two Phase Approach: Collective Non-contiguous Optimization</td>
<td>43</td>
</tr>
<tr>
<td>4.7 MPI-IO over BAFS</td>
<td>45</td>
</tr>
<tr>
<td>5.1 Pseudocode to Measure I/O Latency of the MPI-IO implementation</td>
<td>48</td>
</tr>
<tr>
<td>5.2 Pseudocode of PIOBN</td>
<td>50</td>
</tr>
<tr>
<td>5.3 Pseudocode of PIOBA</td>
<td>54</td>
</tr>
<tr>
<td>5.4 Pseudocode for Measuring the Non-Blocking Semantics Overhead</td>
<td>59</td>
</tr>
<tr>
<td>5.5 File Access Time using MPI-IO APIs and Native File System APIs</td>
<td>61</td>
</tr>
<tr>
<td>5.6 Non-Collective Degree of Overlapping on MercutIO-UFS</td>
<td>62</td>
</tr>
<tr>
<td>5.7 Collective Degree of Overlapping on MercutIO-UFS</td>
<td>63</td>
</tr>
<tr>
<td>5.8 Non-Collective Degree of Overlapping on MercutIO-NFS</td>
<td>64</td>
</tr>
<tr>
<td>5.9 Collective Degree of Overlapping on MercutIO-NFS</td>
<td>65</td>
</tr>
<tr>
<td>5.10 Non-Collective Degree of Overlapping on MercutIO-PVFS</td>
<td>66</td>
</tr>
<tr>
<td>5.11 Collective Degree of Overlapping on MercutIO-PVFS</td>
<td>67</td>
</tr>
</tbody>
</table>
5.12 File Access Time Using Blocking and Non-Blocking Semantics  68
CHAPTER I
INTRODUCTION

1.1 Background

While Central Processing Unit (CPU) performance has increased more than 55% per year during the past decade, disk performance has only grown at the speed of 4% to 6% per year (Hennessey and Patterson 1998). This widening gap between the CPU and I/O performance is undesirable, as the CPU may remain idle most of the time while waiting for the completion of I/O. In order to improve the performance of the I/O subsystem, researchers have recently focused on parallel disk schemes that use several independent disks in parallel to aggregate disk bandwidth (Thakur, Gropp and Lusk 1999b). A robust parallel file system and a well-defined parallel file interface are both needed in order to exploit the I/O performance of such a parallel disk scheme.

Existing parallel file systems, such as the Parallel Virtual File System (PVFS), Galley, Parallel File System (PFS), and Portable Parallel File System (PPFS), enable fast retrieval and storage of data in storage systems; however, each of these has its own parallel file interfaces and thus all are non-portable. Furthermore, the Unix Application Programming Interface (API) is not an appropriate API for parallel I/O; it lacks some essential features needed to express access patterns common in parallel programs, such as noncontiguous accesses and collective I/O, consequently resulting in poor performance (Thakur, Lusk and Gropp 1998). In order to overcome these and other limitations, the Message Passing Interface (MPI) Forum defined a new API
for parallel I/O (commonly referred to as MPI-IO) as a part for the MPI-2 standard (Thakur, Gropp and Lusk 1999b). MPI-IO supports features that promise increased I/O parallelism, such as non-contiguous access, collective I/O, asynchronous I/O, file preallocation, shared file pointers and portable data representation. This thesis involves portable, high-performance design and implementation of MPI-IO.

1.2 Hypothesis

This thesis hypothesizes that overlap of I/O and computation and agglomeration of I/O requests based on an application’s access pattern improve the performance of a portable parallel I/O implementation. The work addresses the quantitative analysis of overlapping and agglomeration that aids in the design of the parallel I/O implementation. The hypothesis is supported through the design and implementation of MercutIO, a portable, high-performance middleware library for parallel I/O. This implementation achieves portability with the support of BullDog Abstract File System (BAFS), an abstract file interface for I/O, primarily developed to provide an efficient, portable, point-to-point I/O interface.

1.3 Motivation

The motivation for this thesis arises from the lack of a portable, high-performance MPI-IO implementation. Existing parallel file systems address only a subset of issues related to parallel I/O. Most of the parallel file systems have no support for collective I/O and non-contiguous file accesses (Thakur, Gropp and Lusk 1999a). Moreover, these file systems have their own parallel-IO interfaces and are thus non-portable. MPI-IO implementations that are currently available are either non-portable, or, if portable, do not deliver high-performance parallel I/O. Support for asynchronous file I/O in these implementations is rudimentary and is based on
blocking semantics or the polling model. In either approach, the CPU is forced to idle and is prevented from doing useful work. Existing implementations do not evidently provide a flexible approach to handling non-contiguous file access that can impair the performance of the implementation for certain I/O access patterns. This thesis addresses these performance issues and provides a quantitative description of the fundamental sources of parallel I/O performance. This study aids in efficient design and implementation of a portable, high-performance I/O middleware library.

1.4 Contributions

The following are the key contributions of this thesis:

1. This thesis proposes the fundamental design criteria influencing the performance of a portable high-performance I/O middleware.

2. The work developed MercutIO, a portable, high-performance MPI-IO implementation.

3. This thesis developed the Bulldog Abstract File System (BAFS), a portable, efficient non-collective I/O interface. BAFS can be used to portably implement any parallel I/O API over a parallel file system. It supports MercutIO.

4. A new data access model based on non-blocking semantics is presented here.

5. This thesis introduces two I/O metrics, namely degree of overlapping and degree of non-contiguity. These two metrics are found in this work to be essential in the performance appraisal of a parallel I/O implementation.

6. The work developed new parallel I/O benchmarks to measure the performance of an MPI-IO implementation.
1.5 Organization

The organization of this thesis is as follows: the following chapter discusses the related work going in the parallel-IO field. Chapter 3 presents the quantitative analysis of the principal design factors contributing to MPI-IO implementation’s performance. New I/O metrics are introduced in this chapter also. Chapter 4 describes the design and implementation of MercutIO, while the experimental setup and performance analysis to validate the hypothesis are presented in Chapter 5. Chapter 6 concludes the thesis and suggests some future research directions.
CHAPTER II

LITERATURE REVIEW

In recent years, there has been tremendous improvement in CPU performance. Furthermore, parallel computers and clusters of workstations have been widely employed to solve scientific and commercial problems. Most of these applications perform I/O for a number of reasons, such as reading initial data, writing intermediate and/or final results, checkpointing, processing out-of-core data sets, using scratch files for temporary storage, and visualization (Thakur, Lusk and Gropp 1998). Since the I/O is evidently slow, the performance of such applications is apparently bounded by the performance of the I/O subsystems. This chapter discusses the recent trends in the field of I/O in order to solve the I/O bottleneck.

The outline of this chapter is as follows: the first section gives an overview of the file access characteristics of sequential and parallel applications. Section 2.2 discusses the different types of file systems available and their respective purposes. The drawbacks of the Unix API for parallel I/O and the emergence of parallel file systems to alleviate these problems are discussed further in this section. Section 2.3 gives a brief explanation of existing parallel I/O libraries. Section 2.4 describes the need for MPI-IO and the support for MPI-IO features on existing parallel file systems. Section 2.5 gives a brief description of some of the existing MPI-IO implementations and identifies the flaws in these implementations that provide certain motivation for this work.
2.1 The CHARISMA Project

The goal of the CHARacterize I/O in Scientific Multiprocessor Applications (CHARISMA) project was to identify the file access characteristics of a parallel application that enables in the design of a high-performance parallel file system (Nieuwejaar and Kotz 1996). The results from the CHARISMA project describe the workload observed on an Intel iPSC/860 and a Thinking Machines CM-5. Based on the work in this project, the I/O workload characterization of sequential and parallel applications is summarized below:

- File access characteristics of applications running on uniprocessors and vector supercomputers:
  
  - Files are huge - hundreds of megabytes, gigabytes, or larger.
  
  - Files are accessed in large pieces - hundreds of kilobytes or megabytes at a time.
  
  - Files are accessed sequentially. That is, every byte in the file is accessed, in order, from beginning to end.

- File access characteristics of parallel applications:
  
  - Many parallel applications access files in small, non-contiguous pieces.
  
  - Within a single file, these pieces tend to be regularly sized and spaced.
  
  - Many parallel applications use many different files in a single run.
  
  - There is a great deal of interprocessor sharing of files.
  
  - There is little interjob sharing of files.

Based on the results of this project, it can be inferred that sequential and parallel application have different file access characteristics. While sequential applications
access files in large contiguous chunks, parallel applications tend to access files in small, non-contiguous pieces. A file system can apparently service a single large, contiguous I/O request more efficiently than numerous smaller, non-contiguous I/O requests. The following section discusses the responsibilities of a file system and describes the role of a file system in different computing environments.

2.2 File System

File systems provide a high-level interface for the application to store or retrieve data from the storage medium. The responsibilities of a file system (May 2001) include the following:

- Moving data efficiently between the memory and storage devices.
- Coordinating concurrent access by multiple processes to the same file.
- Allocating data blocks on storage devices to specific files and reclaiming those blocks when files are deleted.
- Recovering as much data as possible if the file system becomes corrupted.

Most existing file systems handle the tasks listed above in sequential and parallel environments. The difference in the way these tasks are handled segregates file systems into sequential, distributed, and parallel file systems.

2.2.1 Sequential File System

Most file systems, both sequential and parallel, define themselves in relation to the Unix file system (Nieuwejaar and Kotz 1996). This section describes the Unix File System (UFS). Under Unix, each file is represented as a linear, addressable stream of bytes (Thompson 1978). In other words, files are collections of bytes arranged
in a one-dimensional structure. Each byte within the file may be addressed using a
single integer representing that byte's offset, or distance from the beginning of the
file. This file system provides basic file access operations such as open, close, seek,
delete, contiguous read, and contiguous write. These interfaces enable contiguous
access of data on local disks. These operations, though adequate for sequential file
accesses, are clearly inadequate for the I/O requirements of distributed and parallel
applications. The following subsections describe the characteristics of distributed
and parallel file systems.

2.2.2 Distributed File System

Distributed file systems let processes on multiple computers access a common
set of files. The Network File System (NFS) (Gould and Xinu 1986), Andrew File
System (AFS) (Howard 1988) and Distributed File System (DFS) (Kazar et al.
1990) are well-known distributed file systems. These file systems are based on
the client-server model, wherein the server hosts the files and the clients mount
a collection of files at a particular location in their own directory hierarchy. AFS
and DFS have a different architecture and have more sophisticated techniques for
controlling user access to files. In general, a distributed file system guarantees file
access transparency in a distributed environment. File replication can be done offline
and fault tolerance can be achieved with minimal effort. However, these file systems
are not designed to provide multiple processes with efficient, concurrent access to
the same file, suggesting the need for parallel file systems.

2.2.3 Parallel File System

Large computational problems generally require proportionately large datasets
for the solution. It is impossible to store huge volumes of data in a single disk and
simultaneously guarantee high-performance access to solve these problems. In order
to improve the performance of the I/O subsystems, researchers have recently focused
on parallel disk schemes that use several independent disks in parallel to aggregate
disk bandwidth (Tsai et al. 1998). A robust file system and well-defined parallel
I/O interface are needed to exploit the performance of these parallel disk schemes.
In a parallel file system, the file is partitioned into multiple disjoint sequences so that
they can be accessed in parallel. The file data are striped across multiple disks in
different nodes in a cluster enabling concurrent file accesses.

Many commercial and research parallel file systems have been developed to
facilitate parallel I/O and reduce the I/O bottleneck. Intel’s Parallel File System
(PFS) (Intel Corporation 1996), IBM’s Parallel Input/Output File System (PIOFS)
(IBM Corporation 1995), NEC’s Supercomputing File System (SFS) (Mazieres 1998)
and SGI’s Serverless Network File Service (XFS) (Ellis and Levine 1998) are some
of the well-known commercial parallel file systems. The Parallel Virtual File System
(PVFS) (P.H.Carns et al. 2000) and Galley (Nieuwejaar and Kotz 1996) are two of
the best known research parallel file systems.

2.3 Parallel I/O Library

In addition to parallel file systems, parallel I/O libraries have also been
developed to address parallel I/O issues. Parallel And Scalable Software for I/O
(PASSION) (Thakur et al. 1994) and Panda (Winslett et al. 1996) are examples of
existing I/O libraries with special APIs for parallel I/O. The following subsections
discuss these libraries in detail.
2.3.1 The Panda Project

The goal of the Panda project was to produce new data management techniques for I/O-intensive applications that would be useful in high-performance scientific computing (Winslett et al. 1996). Panda is a database-style I/O library designed to support SPMD-style application programs running on ordinary workstations, distributed memory parallel architectures, and networks of workstations. Panda consists of clients and servers, one per compute node and I/O node, respectively. In this library, the application communicates with its local Panda client via a high-level interface. The clients pass a high-level description of the I/O request, describing the distribution in memory and on disk, to the servers. The servers then sequentially read data from files, scattering the data to clients as data arrive from disk. In case of a write operation, the server gathers data from all clients and sequentially writes to a disk. This library enables choosing an efficient data organization among the servers and thus optimizes file accesses from client machines.

2.3.2 The Parallel and Scalable Software for I/O (PASSION) Project

The PASSION project focused on considering the I/O problem from a language, compiler and runtime support point of view (Thakur and Choudhary 1996). This project paved the way for techniques to optimize non-contiguous accesses. The optimizations include data sieving (Thakur et al. 1994) and a two-phase approach (Thakur and Choudhary 1996). In data sieving, when the application generates non-contiguous I/O requests, the requests are merged and a single large I/O request is made to the file system. Then, the data that is really needed is extracted from the memory. The two-phase approach enables the parallel I/O implementation to analyze and merge the I/O requests of different processes (in a parallel environment). In many cases, the merged requests may be large and contiguous, although the
individual requests of each process might be non-contiguous. The merged request(s) can therefore be serviced efficiently. The agglomeration techniques discussed in this subsection will be employed to implement non-contiguous and collective I/O in MercutIO.

2.4 MPI-IO

The Message-Passing Interface (MPI) has become the de-facto standard for programming multicomputers and multiprocessor architectures as well as clusters of workstations. MPI is based on the message-passing model and is an efficient way of expressing parallelism. The MPI standard aims at providing a range of efficient functionality without compromising portability (Gropp, Lusk and Skjellum 1999).

MPI-IO began as a research project at IBM in 1994, and researchers at other sites subsequently contributed to it. The design began with the observation that parallel I/O optimizations require two basic abstractions that the MPI (Message Passing Interface) already possesses: the ability to define sets of processes (MPI communicators) and the ability to specify complex access patterns (MPI datatypes) (May 2001). Using communicators and datatypes as a foundation, the MPI-IO designers created an interface that supports many parallel I/O operations and optimizations. In 1997, this interface was included into the MPI-2 standard and became known as the I/O chapter of that standard.

File access and message passing involve moving data from one address space to another. File access is similar to one-sided communication, wherein only one process has an active role in the message initiation and transfer. In general, MPI-IO is a portable parallel-I/O interface implemented over a file system in order to guarantee high-performance parallel file access. The MPI-IO model is depicted in Figure 2.1.
MPI-IO supports features that promise increased I/O parallelism, such as non-contiguous file access, collective I/O, asynchronous I/O, file preallocation, shared file pointers, and portable data representation. The amount of support for each of these advanced I/O features (indicated by the number of APIs in parentheses) is shown in Figure 2.2 and enumerated in Appendix A. I/O problems in parallel environments discussed in Section 2.1 can be easily addressed in MPI-IO using the notion of the file-view. The non-contiguous I/O access pattern of the parallel application can be specified with a single MPI-IO read/write call by setting appropriate file views. In addition, MPI-IO allows the application to specify the I/O requests of all
the processes collectively, thereby providing the implementation with even greater access to information and a greater scope for optimization (Thakur, Gropp and Lusk 1999a). MPI-IO supports nonblocking versions of all independent read/write functions. The collective I/O can be performed asynchronously using the MPI-IO split-collective APIs and thus I/O can ostensibly be performed in the background. File I/O can be achieved using shared file pointers in addition to using individual file pointer and explicit offsets. MPI-IO consistency and atomic semantics define the degree of consistency in a file when multiple processes perform concurrent file accesses. File access can be optimized by the MPI-IO implementation using the user-defined hints defined in the MPI-IO standard.

2.4.1 Support for MPI-IO Features on Existing File Systems

The previous section presented the I/O features supported by MPI-IO that promise portable, high-performance parallel I/O. This section outlines the support extended by existing commercial and research parallel file systems towards an MPI-IO implementation. The file systems considered for the study include PVFS, NFS, PIOFS, PFS, XFS, and SFS. The basic file access mechanisms – *open*, *seek*, contiguous *read/write* and *close* – are supported by all these file systems. Non-contiguous file access and collective I/O are not supported by any of these file systems. Asynchronous I/O is implemented only in PFS, while shared file pointers are only supported by Intel PFS. Only PFS and XFS provide mechanisms for file pre-allocation. The details regarding the support for advanced MPI-IO features by these file systems are tabulated in Table 2.1. *NS* in Table 2.1 signifies the corresponding feature is “not supported” by that file system.

In summary, existing parallel file systems have little or no support for advanced parallel I/O features such as non-blocking I/O, non-contiguous file access, and
collective I/O. Therefore, any proposed MPI-IO implementation over these file systems requires considerable effort. This thesis is concerned with an efficient implementation of asynchronous I/O, non-contiguous file access and collective I/O. The existing approaches to implement these features are discussed in the following paragraphs.
Table 2.1: MPI-IO Support on Existing File Systems

<table>
<thead>
<tr>
<th>Feature</th>
<th>PVFS</th>
<th>NFS</th>
<th>PIOFS</th>
<th>PFS</th>
<th>XFS</th>
<th>SFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-blocking I/O</td>
<td>NS</td>
<td>iread/ iwrite</td>
<td>NS</td>
<td>iread/ iwrite</td>
<td>NS</td>
<td>NS</td>
</tr>
<tr>
<td>Non-contiguous file access</td>
<td>Supports</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
</tr>
<tr>
<td>Collective I/O</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
</tr>
<tr>
<td>Split Collective I/O</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
</tr>
<tr>
<td>Shared File Pointer</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
<td>Supports</td>
<td>NS</td>
<td>NS</td>
</tr>
<tr>
<td>File Preallocation</td>
<td>NS</td>
<td>NS</td>
<td>NS</td>
<td>_size()</td>
<td>fcntl</td>
<td>NS</td>
</tr>
</tbody>
</table>

Asynchronous I/O in MPI-IO can be implemented through two approaches (illustrated in Table 2.2). On file systems that do not support asynchronous I/O, asynchrony in MPI-IO can be implemented using the native file system’s blocking I/O interface. This approach prevents overlap of I/O with computation or communication. The CPU is forced to idle when I/O is being performed. On file systems that support asynchronous I/O, asynchrony in MPI-IO can be achieved using the polling model. In the polling model, the implementation keeps polling the status register in regular intervals of time for I/O completion. The drawback of this approach is that though I/O asynchrony is achievable, the CPU is underutilized by constantly polling the status register. Nowadays, the CPU operates in the nanoseconds range yet the storage disks still perform at millisecond frequencies. The
Table 2.2: Asynchronous MPI-IO Implementation Approaches

<table>
<thead>
<tr>
<th>Program Flow</th>
<th>Blocking Semantics</th>
<th>Polling Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU State</td>
<td>I/O State</td>
</tr>
<tr>
<td>Compute Cycles</td>
<td>Busy</td>
<td>Idle</td>
</tr>
<tr>
<td>I/O Start</td>
<td>Idle</td>
<td>Busy</td>
</tr>
<tr>
<td>Compute Cycles</td>
<td>Busy</td>
<td>Idle</td>
</tr>
<tr>
<td>I/O Wait</td>
<td>Busy</td>
<td>Idle</td>
</tr>
</tbody>
</table>

CPU underutilization becomes more pronounced when the overlapping computation activity finishes quickly compared to I/O. Thus, the polling model forces the CPU to constantly poll the status registers and prevents the CPU from doing useful work. In summary, none of the asynchronous I/O approaches discussed above are efficient.

As discussed in Section 2.1, when many parallel applications access files in small, non-contiguous pieces and within a single file, these pieces tend to be regularly sized and spaced. An example of such non-contiguous I/O access pattern is illustrated in Figure 2.3. The data blocks are stored in row-major order in the file, yet each process accesses its data in column-major order. This data storage pattern leads to non-contiguity of file accesses in each process. A simple approach to satisfy this access pattern is to perform a separate seek to access each non-contiguous chunk, an approach most commonly referred to as the Unix-style approach. This approach is not recommended as many disk seeks have to be performed (to access each non-contiguous chunk) and disk seeks are slow (on the order of milliseconds). As shown in Table 2.1, most of the file systems do not support non-contiguous file access. Hence, on file systems that do not provide non-contiguous APIs, the MPI-IO implementation has to adopt the Unix-style approach in order to satisfy the non-contiguous access pattern, which proves inefficient. Another approach to optimize non-contiguous file accesses is to implement agglomeration techniques – data sieving and the two-phase
approach, discussed in Section 2.3.2 in the MPI-IO layer. However, these approaches require extra memory for the special buffers used in the algorithm. Moreover, these approaches perform worse when the size of the holes separating the non-contiguous chunks is large.

![Diagram of non-contiguous I/O Access Pattern]

Figure 2.3: Non-contiguous I/O Access Pattern

### 2.5 Existing MPI-IO Implementations

This section describes the two predominant approaches to implementing MPI-IO over a file system. These approaches can be categorized into the portable approach and the point-to-point approach. In the portable approach, MPI-IO is implemented over a portable I/O interface, which is, in effect, implemented directly over the native file system. This approach ensures portability with an associated overhead. The second approach involves implementing MPI-IO directly over the native file system. This approach, though lower in overhead, is not portable. The following subsections briefly discuss existing MPI-IO implementations using both of these approaches.
2.5.1 ROMIO

ROMIO (Thakur, Gropp and Lusk 1999b) is a portable MPI-IO implementation
developed by researchers at Argonne National Laboratory. The key component of
ROMIO that enables such a portable MPI-IO implementation is an internal layer
called the Abstract Device interface for parallel I/O (ADIO) (Thakur, Gropp and
Lusk 1996). ADIO consists of a small set of basic functions for parallel I/O. The
MPI-IO interface is implemented portably on top of ADIO, and only ADIO is
implemented separately for different file systems. ROMIO performs data sieving
in order to satisfy the non-contiguous requests from one process. The collective
I/O implementation is based on either the two-phase approach or the data-sieving
technique depending on the degree of adjacency between the process’s non-contiguous
requests. The performance of the non-contiguous implementation degrades when
the hole size between the data segments is large (Thakur, Gropp and Lusk 1999b).
ROMIO uses the native file system’s non-blocking I/O functions to implement the
MPI-IO asynchronous APIs. On file systems that do not support asynchronous
I/O, the implementation is based on the native file system’s blocking I/O routines.
Either of these approaches blocks the CPU from doing useful work, and the value
of overlapping is lost. The split-collective I/O implementation in ROMIO performs
the entire collective-I/O operation in the main thread during the begin function
(Thakur, Gropp and Lusk 1999b), but this approach prevents overlapping. Shared
file pointers are implemented using the file-lock mechanism. In summary, ROMIO
has the following limitations:

- Sub-optimal implementation of collective I/O operations.
- Sub-optimal implementation of asynchronous I/O.
• Split-collective I/O implementation based on blocking semantics.

• Polling method for I/O-completion notification.

• ADIO approach enables portability with an overhead (Thakur, Gropp and Lusk 1996).

• Differs significantly from MPI-2 standard in some APIs.

2.5.2 MPI-IO over GPFS

This section deals with the point-to-point MPI-IO implementation and discusses the MPI-IO implementation over the General Parallel File System (GPFS). GPFS (IBM 2000) is used as the underlying file system in IBM SP systems. The GPFS interface is similar to the POSIX interface. Non-contiguous access can be achieved by passing hints to GPFS prior to accessing the data from the file. MPI-IO/GPFS uses data shipping to prevent concurrent access of GPFS file blocks by multiple tasks, possibly residing on separate nodes. Data shipping binds each GPFS file block to a single I/O agent, which will be responsible for all accesses to this block. Thus, whenever a task needs to perform I/O, it passes the request to the corresponding I/O agent. The I/O agent performs I/O and then ships back the data to the requesting task. The MPI-IO/GPFS implementation employs a prefetching policy in each of its I/O agents to prefetch data based on the file-view. Double-buffering techniques are employed to optimize large data accesses. The limitations of MPI-IO/GPFS implementation (Prost et al. 2001) include the following:

• The write performance degrades when the data shipping option is enabled in GPFS.

• Prefetching ensures only limited benefits for write operations.
• Asynchronous I/O implementation is based on the polling model.

• The implementation is not portable.

2.6 Summary

This chapter began with the file access characteristics of sequential and parallel applications. Different types of file systems and the purposes of each were discussed. The Unix I/O API was noted to be suitable only for contiguous file access and was unsuited to parallel I/O. In order to satisfy a parallel application’s I/O needs, parallel file systems and parallel I/O libraries were developed. However, most of the parallel file systems were designed only for a particular machine architecture. Also, the inability of parallel file systems to support the essential functionalities at the application level required for parallel file access resulted in the emergence of the MPI-IO standard. The support for different MPI-IO features on some important file systems was discussed in Section 2.4.1. Most file systems did not support collective I/O. Non-contiguous file access was supported only in PVFS. Asynchronous I/O was not supported in most of the file systems and, if supported, was based on the inefficient polling model. Thus, it was concluded that the MPI-IO implementation on these file systems require new design approaches to guarantee maximal performance on the I/O subsystems. A brief description of some of the existing MPI-IO implementations – ROMIO and MPI-IO over GPFS – was given in Section 2.5. Asynchronous I/O on these implementations was based either on blocking semantics or on the polling model, neither of which efficiently utilizes the CPU. Further, non-contiguous file access was based on agglomeration techniques, with poor heuristics to assess the non-contiguity in the file.
CHAPTER III
OVERLAPPING OF I/O AND COMPUTATION AND
AGGLOMERATION OF I/O REQUESTS: A QUANTITATIVE
ANALYSIS

This chapter presents a quantitative analysis of the two important design factors influencing the performance in a MPI-IO implementation. The design factors influencing the performance of a portable MPI-IO implementation include asynchronous I/O, which facilitates overlapping of I/O and computation, collective I/O, non-contiguous file accesses, prefetching, I/O using shared file pointers, file-preallocation, and data distribution across storage media.

Performing I/O using shared file pointers serializes the I/O access and consequently limits parallelism. File preallocation, prefetching, and data distribution across the storage medium are often characteristics of a file system. These issues, when being addressed from the middleware level, add additional overhead and do not contribute to significant performance gains. Asynchronous I/O facilitates overlapping of computation, communication, and I/O if properly implemented. Non-contiguous file access helps to specify the access pattern prior to the implementation, which can be efficiently used to optimize the I/O accesses. As discussed briefly in Section 2.4.1, asynchronous I/O, non-contiguous file access, and collective I/O are not supported in most file systems. The I/O middleware plays a significant role in emulating these features with the underlying file systems. Non-contiguous file accesses and collective I/O can be efficiently addressed based on agglomeration.
of the I/O requests. Hence, overlap of I/O and computation using asynchronous I/O and agglomeration of I/O requests can be presumed to contribute significantly to the performance of a portable MPI-IO implementation. The following sections offer a quantitative description of these two principal design factors and lay the foundation for efficient MPI-IO design. Two new relevant parallel I/O metrics, degree of overlapping and degree of non-contiguity, are introduced in this chapter.

3.1 Overlapping of I/O and Computation

Overlapping of I/O and computation enables independent I/O and computation activities to be scheduled concurrently. This approach ensures maximal utilization of the CPU when I/O and computation are scheduled efficiently by the algorithm. The overhead of using asynchronous MPI-IO APIs to access data is minimal compared to that of the synchronous MPI-IO APIs. In effect, this technique helps to reduce the overall application time by utilizing the hardware and software architectures that offer concurrent progress of I/O and computation. In order to effectively utilize this technique, the data access routines in MercutIO are based on non-blocking semantics. The objective of this section is to validate that data access mechanisms based on the blocking semantics and data access based on non-blocking semantics have the same asymptotic complexity.

3.1.1 Theoretical Description

The time \( T \) taken to access \( n \) chunks of data in blocking I/O mode, each of size \( C \) bytes with I/O throughput (read or write) of \( P \) bytes/sec is as follows:

\[
T = n\left(\frac{C}{P}\right). \tag{3.1}
\]
The non-blocking I/O implementation uses the thread-queue model, which is described in detail in Section 4.2.1. The proposed non-blocking I/O strategy consists of three stages: Initializing (I) the data structure (early binding), posted I/O requests waiting in the work queue to get serviced (W: average wait time) and, finally, blocking I/O being performed asynchronously (A) and signaling the completion of the request. Therefore, the total time taken \( T' \) to access \( n \) chunks of data, each of size \( C \) bytes with I/O throughput (read or write) of \( P \) bytes/sec is:

\[
T' = n(I + W + A).
\]  

(3.2)

where \( A \) is \( \frac{C}{P} \). Therefore Equation 3.2 becomes:

\[
T' = n(I + W + \frac{C}{P}).
\]  

(3.3)

For sufficiently large values of the I/O request size,

\[
\frac{C}{P} \gg nI + nW.
\]

(3.4)

and, thus, the file access time compensates the overhead introduced by the work queue model. The performance of the non-blocking I/O can be further enhanced by making use of the early binding technique discussed in Section 4.2.1. Therefore, Equation 3.2 becomes:

\[
T' = I + n(W + \frac{C}{P}).
\]

(3.5)
Agglomeration techniques (discussed in the next section) combine many smaller I/O requests to a single large I/O request. Therefore, for large values of $C$ and small values of $n$:

$$T = T'. \quad (3.6)$$

Complexity $= \Theta(n)$.

Therefore, it can be seen that the asymptotic complexity of the proposed non-blocking strategy is same as that of the blocking I/O. However, the foremost advantage is the ability to overlap I/O and computation. Based on this premise, the synchronous MPI-IO APIs will be implemented based on the proposed non-blocking strategy, thereby freeing the CPU to do useful tasks while I/O is being performed.

3.1.2 Degree of Overlapping: Definition and Evaluation Strategy

This metric gives a measure of the amount of overlap between I/O and CPU-bound tasks and thus aids in determining the overall application’s performance gain. The degree of overlapping between computation and communication (Dimitrov 2001) gives a measure of amount of overlapping achieved between a communication and computation activity scheduled concurrently. The degree of overlapping between I/O and computation is adapted from degree of overlapping between communication and computation since I/O and communication overlap are related and are primarily dependent on efficient software and hardware architectures, rather than on high speed components.

The procedure for determining the degree of overlapping involves determining the computation time ($T_c$), I/O time ($T_i$), and the overlap time ($T_o$). First, the read/write time ($T_i$) for a given block size is measured using the non-blocking API.
Then, the time taken for the completion of a given computation problem is measured \( (T_c) \). Subsequently, the given computation problem, which is independent of I/O, is scheduled concurrently with the I/O and the time taken for the completion of the computation and I/O (using non-blocking API) is measured \( (T_o) \). The Degree of Overlapping \( (\alpha) \) can be computed using the following formula:

\[
\alpha = \frac{T_i + T_c - T_o}{T_i + T_c}.
\]

When the implementation does not support overlapping of I/O and computation, then

\[
T_o = T_i + T_c,
\]  
(3.7)

and \( \alpha = 0 \). Values of \( \alpha \to 1 \) indicates maximum overlapping has been achieved and can be obtained theoretically. In practice, \( \alpha \) can attain a value of one in Symmetric Multi-Processor (SMP) environments with a truly parallel I/O subsystem and the operating system supporting kernel-level threading. When processing super-scalar-computation-I/O-intensive problems, with the computation efficiently scheduled among different processors and parallel I/O being performed simultaneously, the overlapping strategy will result in the maximal utilization of CPU and the I/O subsystem with \( \alpha \) tending to one. Negative values of \( \alpha \) indicates poor implementation of asynchronous I/O.

### 3.1.3 Practical Use of Overlapping

The advantages of overlapping are as follows:
• Relieves CPU and Network Interface Card (NIC) when I/O is being performed and thus facilitates efficient utilization of CPU and NIC.

• Improves the overall performance of the application.

Sufficient memory bandwidth is, of course, needed to allow simultaneous activities; but this is expected to exist in practical systems.

3.2 Agglomeration of Smaller I/O Requests to Optimal Sized Chunks

Sequential I/O interfaces, for example the POSIX interface, do not permit non-contiguous file access or collective I/O. Therefore, many individual disk seeks and reads/writes have to be performed in order to satisfy the non-contiguous I/O requests. This access mechanism significantly undermines the I/O performance. MPI-IO supports non-contiguous file access and collective I/O, and it is the responsibility of the MPI-IO implementation to implement these features efficiently. Agglomeration of smaller requests based on the application’s access pattern to a single large I/O request prior to performing the disk access is one of the key factors to the I/O middleware’s performance gain. The single large I/O request can be satisfied with a single disk access. The data can then be scattered back to the smaller individual buffers from the single contiguous buffer. File systems can generally service large requests more efficiently than smaller I/O requests. The agglomeration approach avoids repeated accesses to disks, which are slower, and thus improves the performance. This technique is used at the individual process level (data sieving) as well as at the communicator level (two-phase technique) in the MPI-IO implementation. The non-collective MPI-IO APIs are implemented using the data-sieving technique and the collective MPI-IO APIs are implemented using the two-phase approach.
3.2.1 Theoretical Description

This subsection gives a theoretical explanation for the performance that can be obtained for different non-contiguous access patterns using different file access strategies. The access techniques considered in this study include Unix-style mechanism, data-sieving and the two-phase approach.

3.2.1.1 Non-Collective I/O Involving Non-contiguous File Accesses

When non-collective I/O involving non-contiguous file accesses is being performed, if the requests possess temporal or spatial locality, then instead of performing $n$ individual seeks and $n$ read/writes, a single seek and read/write is performed. The I/O requests are agglomerated and the data is read/written into a large buffer, which is allocated only once when the middleware is initialized. Then, $n$ memory copies `memcpy` are made to copy the relevant data from the data sieving buffer into the user buffer.

For a total of $n$ non-contiguous I/O requests using sequential I/O APIs, the total time taken would be:

$$
T = n(S + \frac{C}{P}).
$$

(3.8)

where $S$ is the average seek time, $C$ is the size of each contiguous chunk, and $P$ is the I/O bandwidth guaranteed by the file system. Since disk seeks and I/O are generally slow, for large values of $n$, $T$ would be significantly large.

Using the data sieving technique, the total time taken would be:

$$
T' = S + \frac{C'}{P} + nM.
$$

(3.9)
where $C'$ is the extent of the entire request and $M$ is the time taken for `memcpy`. If `memcpy` throughput is $Q$ bytes/sec, then Equation 3.9 reduces to:

$$T' = S + \frac{C'}{P} + n\left(\frac{C}{Q}\right). \tag{3.10}$$

It is known that a disk read is expensive compared to `memcpy` and that I/O bandwidth is higher if file read/write is done in larger chunks instead of smaller chunks. According to the CHARISMA study (Section 2.1), many parallel applications access files in small, non-contiguous pieces. Therefore, for higher values of $n$, from Equations 3.8 and 3.10:

$$T' \ll T. \tag{3.11}$$

$Complexity = \Theta(n)$.

3.2.1.2 Collective I/O Involving Non-contiguous File Accesses

Collective I/O provides a good opportunity to exploit I/O parallelism. Collective I/O enables the MPI-IO implementation to analyze and merge the I/O requests of different processes in a parallel environment. When all the processes make I/O requests using collective I/O, the simple approach would be to satisfy each individual process’s requests in sequence. If there are $p$ processes and each process makes $n$ I/O requests, using sequential I/O APIs the total time taken would be:

$$T = p \times n(S + \frac{C}{P}). \tag{3.12}$$

$Complexity = \Theta(pn)$. 
In the collective read implementation using the two-phase algorithm, the first phase involves inter-process communication of each other’s I/O request and performing data sieving in each process’s file domain and the second phase involves communication of the collective buffer contents to the appropriate destination process. In the worst case, each process generates $n$ I/O requests. The time complexity using this approach would be:

$$T'' = n\left(\frac{R}{B}\right) + T' + n\left(\frac{M}{B}\right),$$  \hspace{1cm} (3.13)

where $R$ is the I/O request size, $M$ is the maximum message size that has to be communicated from each process’s file domain to appropriate destination process, $B$ is the bandwidth of underlying communication media, and $T'$ is the data sieving algorithm complexity (from Equation 3.10). Extensive use of Collective MPI calls with derived datatypes in the collective I/O implementation helps to minimize the communication overhead. Thus, for high values of $n$ and $p$

$$T'' \ll T.$$  \hspace{1cm} (3.14)

3.2.1.3 Non-contiguous File Accesses Involving Large Holes with Considerably Smaller Non-contiguous Chunks

When the I/O access pattern possesses non-contiguity with larger holes, the agglomeration techniques discussed above do not perform well compared to the Unix style approach. The time taken to access a single large I/O request (extent of non-contiguous requests) would be higher compared to many small disk accesses using the
Unix-style approach because of the extra costs involved in fetching the holes along with the data. Quantitatively, this access pattern can be explained using Equations 3.8 and 3.10. The agglomeration costs in Equation 3.10 supersedes the additional seek costs involved with the Unix style approach (in Equation 3.8).

\[
\frac{C'}{P} \gg n(S + \frac{C}{P}).
\]

Therefore, \(T_{agg} \gg T_{Unix}\). (3.15)

### 3.2.1.4 Non-contiguous File Accesses Involving Large Non-contiguous Chunks

Consider the case when the size of each non-contiguous chunk \((C')\) is equivalent to or larger than the data sieving buffer size \((C')\). The agglomeration technique is not sufficiently efficient to satisfy this access pattern since redundant memory copies have to be performed between the data-sieving buffer and the user buffer. The Unix style access mechanism is suitable for this access pattern as it helps to overcome the redundant memory copies and thus improve the performance. For this access pattern, the agglomeration techniques would perform worse in sequential and distributed file systems as huge volumes of data are stored in single storage device, preventing parallel disk reads/writes. For this access pattern, Equation 3.10 reduces to:

\[
T' = n(S + \frac{C'}{P} + \frac{C}{Q}).
\] (3.16)

and \(C \equiv C'\). Comparing Equations 3.8 and 3.16, it can be concluded that Unix-style approach is a more appropriate access mechanism for this kind of non-contiguous access pattern.
Based on the CHARISMA study (Section 2.1), the non-contiguous I/O access patterns discussed in Sections 3.2.1.3 and 3.2.1.4 occur seldom in parallel applications. However, they should be taken into consideration in parallel I/O middleware design.

3.2.2 Degree of Non-contiguity

Non-contiguous file accesses can be achieved using Unix-style mechanism, data-sieving technique and the two-phase approach. This metric helps to determine the appropriate technique to be used for file I/O, depending on the nature of the non-contiguous data layout in the file. The efficiency of non-contiguous file access implementation in a file system or an MPI-IO implementation can be evaluated using this metric.

3.2.3 Procedure for Determining Degree of Non-contiguity

The procedure for determining the degree of non-contiguity involves measuring the read and write time for non-contiguous file access using sequential I/O APIs ($T_s$), and both non-collective ($T_n$) and collective ($T_c$) MPI-IO APIs. The I/O time ($T$) is a function of the hole size ($H$), amount of data to be accessed ($D$), and the number of non-contiguous chunks ($N$) as shown below.

$$T = f(N, H, D).$$  \hspace{1cm} (3.17)

The Degree of Non-contiguity ($\beta$) can be calculated as follows:

$$\beta = 1 - \frac{\min(T_n, T_c)}{T_s}.$$
For values of $\beta < 0$, the Unix-style access is recommended; it minimizes the I/O time and the memory usage. Values of $\beta \to 1$ indicates that maximum optimization with respect to non-contiguous access has been achieved.

### 3.3 Summary

This chapter presented a quantitative analysis of the fundamental sources of performance in an MPI-IO implementation. Overlap of I/O and computation and agglomeration of I/O requests were identified as important design factors influencing the performance of a portable parallel I/O implementation. The quantitative analysis introduced the optimization strategies to be employed in the MPI-IO implementation. The use of early binding concept in the asynchronous I/O implementation facilitates creation and initialization of the necessary data structure once, thereby minimizing the overhead. The flexible approach to service the non-contiguous I/O requests based on the nature of the application’s access pattern theoretically promises significant performance gains. Finally, two new parallel I/O metrics – degree of overlapping and degree of non-contiguity – that help in the performance appraisal of the parallel I/O implementation were introduced in this chapter.
CHAPTER IV

MERCUTIO: PORTABLE, HIGH-PERFORMANCE MPI-IO

IMPLEMENTATION

MPI-IO, the I/O part of the MPI-2 standard, is a parallel I/O interface designed specifically for portable, high-performance parallel I/O (Gropp, Lusk and Thakur 1999). MPI-IO supports features that promise increased I/O parallelism and guarantee I/O portability, such as non-contiguous file access, collective I/O, asynchronous I/O, file preallocation, shared file pointers and portable data representation. This chapter briefly describes the architecture and design of MercutIO, a portable high-performance MPI-IO implementation. MercutIO achieves portability through BAFS, a portable non-collective I/O interface. The following section describes the MercutIO (MPI-IO-BAFS) architecture, Section 4.2 discusses the design and implementation of BAFS, and Section 4.3 describes the design and implementation of the MPI-IO component of MercutIO.

4.1 MercutIO Architecture

The architecture of MercutIO is depicted in Figure 4.1. The primary objective of MercutIO is to provide a portable, parallel I/O interface to the user application. Portability and parallelism are achieved efficiently by demarcating these functionalities into two components: MPI-IO and BAFS. I/O Parallelism is achieved in the MPI-IO layer while portability is taken care by BAFS layer. The MPI-IO APIs are implemented in the MPI-IO layer. BAFS consists of a set of APIs that provides
the necessary functionality for parallel I/O. BAFS is layered between the MPI-IO layer above and an underlying parallel file system. Apart from MPI-IO, other parallel I/O APIs can easily be implemented across disparate file systems by implementing them over BAFS. This architecture ensures ease of parallel I/O deployment over different vendor-specific file systems.

BAFS is relatively a thin abstract I/O interface (denoted by dotted lines in Figure 4.2 between MPI-IO and the underlying file systems. It provides a non-collective I/O interface that functions independently for each MPI process created. MPI-IO, on the other hand, operates on the communicator level for a group of processes defined by the parallel environment. Communication between processes
takes place in this layer. The MPI-IO layer manages collective I/O and shared file pointer operations. This layer maintains data consistency across different processes and delivers file atomicity semantics. The non-collective BAFS interface is illustrated in Figure 4.2.

The architecture of MercutIO enables parallel processes to simultaneously perform I/O with different file systems. This architecture enables a parallel application to transparently perform I/O with different file systems as depicted in Figure 4.3. Thus, a group of processes constituting the parallel environment can perform I/O on a file system while another set of processes can operate on a different
MPI-IO (or) Parallel I/O Library

BAFS

BAFS

BAFS

BAFS

NFS

UFS

PFS

PVFS

SD

SD

SD

SD

SD: Storage Device

Figure 4.3: Multiple I/O Targets

file system. The compute nodes in the emerging Grid (Foster, Kesselman and Tuecke 2001) environments could, for example, concurrently perform I/O with different file systems using this architecture and thus could significantly benefit from this real-time heterogeneous file system support. This architecture also helps to portably access data in a storage device through multiple file systems with minimal effort.

4.2 BAFS

This section discusses the design and implementation of BAFS. BAFS provides a portable high-performance point-to-point parallel I/O interface. The optimization
techniques used in the implementation will be briefly discussed in the following subsections.

4.2.1 BAFS Design

![Diagram of BAFS Model for Non-Blocking I/O]

Figure 4.4: BAFS Model for Non-Blocking I/O

The BAFS data access routines are implemented based on non-blocking semantics in order to overlap I/O and computation. The blocking APIs are implemented using the non-blocking APIs in order to take advantage of the overlapping concept. Smaller I/O requests are agglomerated to larger chunks before the actual disk access is performed. BAFS attempts to exploit the underlying file
system features for higher I/O performance and simulates the functionalities not supported by the file system essential for the MPI-IO layer above it.

In BAFS, the data access routines are implemented based on non-blocking semantics. The current implementation of BAFS has two provisions for non-blocking I/O. The first approach makes use of the non-blocking APIs supported by the underlying file system. Support for non-blocking I/O by various parallel file systems is tabulated in Table 2.1. As discussed in Section 2.4.1, most of the existing file systems do not support asynchronous I/O and, if supported, it is based on the polling model.

The second approach is based on the producer-consumer model. Threads are used to transfer data between the user thread and the BAFS system thread using the work queue model as depicted in Figure 4.4. In this approach, the user thread asynchronously posts the I/O requests (denoted by ‘R’ in Figure 4.4) in the work queue. The system thread operates in a First In First Out (FIFO) strategy and upon the completion of an I/O request asynchronously notifies the user thread. Use of the thread model to process I/O requests facilitates simultaneous execution of other threads, if kernel-level threads are supported by the system. The CPU is utilized efficiently. I/O and communication and computation tasks, if independent of each other, can be processed simultaneously using this approach. On a multiprocessor machine, many different threads can run in parallel (Lewis and Berg 1998). Using this data access approach, the application can significantly take advantage of Symmetric Multiple Processors (SMP) systems.

The thread work queue model is the default approach used for the non-blocking I/O implementation. The first approach keeps polling the I/O request for completion and subsequently prevents the CPU from doing useful work. The blocking I/O APIs
are layered on top of the non-blocking I/O APIs and thus inherently overlapping of I/O with computation can be achieved.

The concept of early binding is made use of in the BAFS implementation. With this approach, whenever the user does repeated file access using the same data structure, the data structure is initialized only once, along with the associated memory allocation. The user can then call the non-blocking data access routine to fetch the data repeatedly making use of the same data structure. This approach significantly reduces the initialization overheads. The BAFS implementation over emerging file systems like the Direct Access File System (DAFS) can significantly benefit from this concept and a considerable performance gain can be expected.

Another I/O optimization in BAFS is the implementation of the data-sieving technique (Thakur, Gropp and Lusk 1999a) to optimize non-contiguous file accesses.
This technique involves agglomeration of smaller I/O requests to optimal chunk size a priori to performing disk access as illustrated in Figure 4.5. Whenever the application makes several smaller I/O requests instead of performing several separate disk seeks that are time consuming, BAFS finds the extent of all the requests and performs a single disk read/write into a single large contiguous data-sieving buffer. It then scatters back the data to the individual buffers from the data-sieving buffer. Memory operations are faster compared to disk accesses. Hence this technique promises significant performance gains compared to the traditional Unix approach of accessing non-contiguous chunks in a file. Read-modify-write (Thakur, Gropp and Lusk 1999a) is performed before the agglomerated data is written into the file. This is done to maintain data consistency in the file and prevent destruction of the data already present in the holes between contiguous data segments. The portion of the file that is being accessed is locked during the read-modify-write to prevent concurrent updates by other processes. This might affect the performance, but is needed to maintain data consistency in the file. BAFS also attempts to find a compromise between I/O performance and the resources needed to achieve the desired results. In the case of smaller non-contiguous chunks separated by relatively larger holes or large non-contiguous chunks separated by relatively smaller holes, BAFS can be directed to employ the Unix-style access mechanism through user-defined hints passed to the implementation; user-defined hints are also supported in the MPI-IO standard. Data-sieving is the default approach used to handle non-contiguous I/O requests in MercutIO.

4.2.2 BAFS Implementation

BAFS is designed and implemented using the object-oriented approach. This approach facilitates data encapsulation, polymorphism, code reusability and
other features essential for project development. The concepts of polymorphism, inheritance and data encapsulation are extensively used in the BAFS design. “BAFS_File” is the name of the base class, and each file system is represented by a derived class, “BAFS_FS_File” where FS is the file system name. For instance, BAFS_NFS_File represents the NFS file system. Polymorphism in BAFS helps an object of a specific file system (derived class) act as an object of the base class, thereby facilitating code reusability.

Dynamic class loading (Norton 2001) is a means of providing extensibility to object-oriented design without compromising robustness. This technique is used to implement the BAFS framework. This technique enables the BAFS interface of newly developed file system to be dynamically plugged into the existing framework facilitating a greater amount of flexibility to the developer.

4.3 The MPI-IO Component of MercutIO

MercutIO achieves portability by implementing the MPI-IO component over BAFS. The primary design features of BAFS - file access based on the thread queue model, early binding and agglomeration of smaller I/O requests to optimally sized chunks - ensure high-performance with a low overhead. Basic MPI-IO file manipulation functions and the non-collective data access functions are implemented using appropriate BAFS APIs (APPENDIX B). However, collective I/O and shared file pointer implementation, and provision for consistency and atomic semantics require significant communication among the parallel processes. These features are efficiently implemented in the MPI-IO component of MercutIO using BAFS APIs and the MPI library for inter-process communication. The following subsections discuss the design and implementation of the MPI-IO component of MercutIO.
4.3.1 Design

All MPI-IO data access functions are based on non-blocking semantics, although it is possible to switch to blocking semantics using a compile-time option. Non-blocking semantics help to overlap I/O, computation, and communication effectively and thus significantly improve the overall performance of the application. Non-contiguous file access is achieved using file views. Depending upon the file view and the nature of the I/O access pattern in the non-collective I/O request, MercutIO makes use of the BAFS data-sieving routines or the BAFS basic-file-access (Unix-style-file-access) mechanism. When the hole size between the data segments is comparatively larger than the data segment or the size of each non-contiguous chunk is comparatively larger than the hole size, the BAFS basic file access is adopted. This approach significantly minimizes the memory costs and I/O latency. On the other hand, when the hole size is smaller than that of the data segment, BAFS data-sieving routines are invoked to perform the non-contiguous file accesses.

The collective I/O implementation is based on the data-sieving technique, the two-phase approach, or Unix-style file access depending on the individual process’s access list. When the access list of individual processes are close to each other, the collective I/O implementation uses the two-phase algorithm for performing I/O. Otherwise, depending on the hole size to data segment ratio, the BAFS basic file access or the BAFS data-sieving routines are invoked. The two-phase approach is shown in Figure 4.6. In this approach, the first phase involves each process exchanging its access lists, determining individual file domains and performing I/O into a single, large contiguous buffer within their file domains. The second phase involves communicating the data to appropriate processes. MPI-IO collective calls, along with suitable derived datatypes, are used in the communication
of I/O requests within processes to reduce communication costs. In the split-collective I/O implementation, the collective I/O is commenced asynchronously in the MPI_File_XXX_begin() call and is blocked by the MPI_File_XXX_end() call until I/O completion. The split-collective I/O uses the BAFS thread model to achieve asynchrony and uses data agglomeration techniques to optimize non-contiguous accesses. The collective I/O implementation in MercutIO is implemented over split collective APIs to facilitate overlapping of computation and I/O.

Data access using the shared file pointer is achieved using the Unix file lock mechanism. In this approach, the shared file pointer value is stored in a file and the value is updated atomically using the file lock. The MPI-IO consistency and atomic semantics APIs are ensured using pthread_semaphores and fcntl() system calls.
4.3.2 Implementation

The I/O chapter of the MPI-2 Standard (MPI-IO Component of MercutIO) is developed over BAFS. The actual process involved in accessing the storage system is hidden from this layer. The current implementation of MPI-IO, illustrated in Figure 4.7, supports C binding only; there is no for support C++ bindings. The abstract file system over which the MPI-IO interface is developed, however, uses C++ bindings. A transparent C++ interface that translates the C++ bindings in the abstract file system to the C bindings of the MPI-IO interface is developed as a convenient solution. This transparent layer provides C interfaces to the various objects in the abstract file system and their associated member functions and is not expected to affect the performance of the system. The MPI-IO layer verifies the validity of the corresponding I/O operation and passes control onto the abstract file system. As discussed in the design subsection, collective I/O and operations involving shared file pointers are implemented in the MPI-IO layer.

4.4 Summary

This chapter presented the design and implementation of MercutIO. The architecture of MercutIO outlined the role of BAFS and the MPI-IO component. I/O parallelism is achieved in the MPI-IO layer, while portability is provided by BAFS. The design features of BAFS, namely file access based on the thread-queue model, early binding, and agglomeration of smaller I/O requests to optimally sized chunks highlighted the non-collective optimizations in MercutIO. The collective I/O optimizations and the shared file pointer implementation that requires inter-process communication are implemented in the MPI-IO layer. This chapter also discussed the methodology used in the implementation of MercutIO.
Figure 4.7: MPI-IO over BAFS
CHAPTER V
EXPERIMENTS, RESULTS, AND ANALYSIS

This chapter presents the experimental methodology and analyzes the results that validate the hypothesis. The experiments involve implementing a performance test suite, which applies several metrics to a range of non-collective and collective functions in MPI-IO. New benchmarks have been designed and implemented using MPI-IO APIs to measure portability overhead, non-blocking semantics overhead, degree of overlapping, and degree of non-contiguity. The metrics for performance testing includes the following:

- Degree of Overlapping,
- Degree of Non-contiguity,
- Portability Overhead of MPI-IO over the native file system calls,
- Non-Blocking Semantics Overhead, and
- Read & Write Latency.

The subsequent sections describe the experimental methodology used to measure each of the metrics described above and to analyze the results obtained.

5.1 Hardware and Software Configuration

MercutIO has been implemented portably over three different file systems and on two different platforms. The file systems include the Unix File System (UFS),
Network File System version 3 (NFSv3) and Parallel Virtual File System (PVFS). UFS is a sequential file system, NFS is an instance of a distributed file system, and PVFS is an instance of a parallel file system.

The experiments were carried out on Sun and Linux clusters. The Sun cluster includes four nodes interconnected by 100Mbps Fast Ethernet. Each node is a Sun Ultra-5 workstation operating at 333 MHz. NFS version 3, with 4GByte disk space, is mounted on one of the nodes with the noac (no attribute caching) option enabled. Enabling noac option prevents client-side caching in NFS. Disabling the attribute caching reduces the performance but is necessary to guarantee file consistency across the nodes. The Linux cluster also includes four nodes interconnected by 100Mbps Fast Ethernet. Each node consists of a Pentium III processor operating at 745 MHz. PVFS is installed on all four nodes of the Linux cluster, with one node serving as the metadata server. All nodes are configured to serve as I/O servers. The stripe size in PVFS was set to 64KByte. The data-sieving read buffer size is 4MBytes and the write buffer size is 1MByte. Similarly, the collective read and write buffer sizes are set to 4MBytes and 1MByte respectively. NFS version 3 on the Solaris platform and PVFS on the Linux platform were used for testing purposes. Only one node in the Sun or the Linux cluster was used to gauge MercutIO’s performance on UFS because the scope of data in this file system is restricted to a single machine. The benchmark discussed in Section 5.5.2 was tested on UFS on a single node of the Sun and the Linux clusters. The benchmark discussed in Section 5.3 and 5.4 was tested on UFS in the Solaris platform while rest of the benchmarks were tested on UFS in the Linux platform.
5.2 Latency Measurements

5.2.1 Latency Program Description

This test measures the read/write latency of MPI-IO non-collective and collective routines. The time taken to read/write one byte of data gives the value of read/write latency. MPI_File_sync() is called after the write operation to ensure that the data in the buffer is actually being written into the file. The pseudocode of the synchronous, non-collective I/O subtest is shown in Figure 5.1.

```c
BlockSize = 1; /* 1 Byte */
StartTime = MPI_WTime();
for (i=0; i<MAXITER; i++)
    MPI_File_<read/write>(BlockSize);
IOTime = MPI_WTime() - StartTime;
MPI_Reduce(IOTime, MPI_MAX, Root);
IOLat = IOTime/MAXITER;
```

Figure 5.1: Pseudocode to Measure I/O Latency of the MPI-IO implementation

5.2.2 Experimental Results and Analysis

The results are tabulated in Table 5.1. From the results it can be inferred that the write latency is generally greater than the read latency. This is because of the MPI_File_sync(), which forces and flushes the data in the file system buffer onto the external storage medium, disabling the write-behind strategy used in the most file systems. In addition, the latency of the collective calls is greater than that of the non-collective calls because of the extra inter-process communication and synchronization costs involved in the collective I/O implementation.
Table 5.1: Latency of Non-Collective and Collective MPI-IO Implementation

<table>
<thead>
<tr>
<th>File System</th>
<th>Non-Collective I/O</th>
<th>Collective I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read (s)</td>
<td>Write (s)</td>
</tr>
<tr>
<td>UFS</td>
<td>0.0004</td>
<td>0.10857</td>
</tr>
<tr>
<td>NFS</td>
<td>0.02709</td>
<td>0.09889</td>
</tr>
<tr>
<td>PVFS</td>
<td>0.02568</td>
<td>0.13147</td>
</tr>
</tbody>
</table>

5.3 Parallel-IO Benchmark for Non-contiguous access (PIOBN)

5.3.1 PIOBN Description

PIOBN tests an MPI-IO implementation for its non-contiguous file access implementation. This benchmark measures the efficiency of the non-contiguous implementation by employing a simple Unix-style access mechanism and invoking MPI-IO APIs after setting efficient file views. MercutIO uses data agglomeration techniques (discussed in Section 4.2.1 and 4.3.1) for non-contiguous file access. The data agglomeration techniques include data-sieving for non-collective I/O and the two-phase approach for collective I/O. The pseudocode of the synchronous collective I/O subtest is shown in Figure 5.2. The MPI-IO accesses involve calling appropriate synchronous non-collective and collective MPI-IO routines after setting efficient file-views. The test involves several iterations, the hole size varying with each iteration. The test involves accessing 1,000 elements, each element of size four bytes, striped across four non-contiguous chunks. The times taken to access all the elements using the Unix-style and agglomeration techniques are measured in this benchmark. The maximum time (worst case) taken by each individual process participating in parallel I/O is noted to compute the degree of non-contiguity that is calculated using the procedure listed in Section 3.2.3. This benchmark is not tested on PVFS because of the lack of support for a file lock mechanism in PVFS. File lock support is essential to maintain file consistency in non-contiguous read/write implementations.
for(i=0; i<NumRuns; i++)
{
    StartTime = MPI_WTime();
    MPI_File_set_view(); /* hole size and block size */
    /* varied in each iteration */
    MPI_File_<collective>();
    IOTime = MPI_WTime() - StartTime;
    MPI_Reduce(IOTime, MPI_MAX, Root);
}

Figure 5.2: Pseudocode of PIOBN

5.3.2 Expected Results

Both data sieving and the two-phase approach perform better than the Unix-style access mechanism to access non-contiguous I/O chunks when the intervening hole size is small. The theoretical explanation was discussed in Section 3.2.

5.3.3 Experimental Results and Analysis

The experimental results are tabulated in Tables 5.2, 5.3, 5.4, and 5.5. It can be seen that I/O performance improves significantly when agglomeration techniques are used to access non-contiguous data in the file. The amount of performance gained is evident from the values of degree of non-contiguity almost tending to one. As the hole size is increased, the performance of agglomeration techniques decreases because of extra overheads involved in fetching the holes. MercutIO uses Unix-style access mechanism to satisfy non-contiguous request with large holes. The write performance of agglomeration techniques on NFS degrades compared to its peer read performance, evidently because of the noc attribute enabled in NFSv3.

The write performance is high in UFS compared to the read performance because of the write-behind-strategy optimization in the file system. In the write-
behind-strategy optimization, whenever a file write operation is invoked, the file system writes the data into the file-system cache and returns to the user immediately (without actually writing the data into the disk). On the other hand, for read operations, even for smaller reads the data has to be fetched from the disks for the first time. When large quantities of data are written to a file, the write-behind-strategy does not help as the amount of data to be written supersedes the file system cache limits. This causes data to be written to external storage media. Thus, for large file accesses, the read and write performance of agglomeration techniques in UFS are comparable.

Table 5.2: Read Degree of Non-contiguity on MercutIO-UFS

<table>
<thead>
<tr>
<th>Hole Size (bytes)</th>
<th>Unix Collective Read Time (s)</th>
<th>Non Collective Read Time (s)</th>
<th>Collective Non-contiguity Degree of Read Time (s)</th>
<th>Read Degree of Non-contiguity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3192</td>
<td>0.79896</td>
<td>0.03314</td>
<td>0.07294</td>
<td>0.95852</td>
</tr>
<tr>
<td>3768</td>
<td>1.08338</td>
<td>0.03245</td>
<td>0.07289</td>
<td>0.97005</td>
</tr>
<tr>
<td>6072</td>
<td>1.15826</td>
<td>0.03291</td>
<td>0.07319</td>
<td>0.97159</td>
</tr>
<tr>
<td>15288</td>
<td>1.15184</td>
<td>0.03411</td>
<td>0.07453</td>
<td>0.97038</td>
</tr>
<tr>
<td>52152</td>
<td>1.10779</td>
<td>0.05549</td>
<td>0.07988</td>
<td>0.94991</td>
</tr>
<tr>
<td>199608</td>
<td>1.09259</td>
<td>0.07536</td>
<td>0.10932</td>
<td>0.93102</td>
</tr>
</tbody>
</table>
Table 5.3: Write Degree of Non-contiguity on MercutIO-UFS

<table>
<thead>
<tr>
<th>Hole Size (bytes)</th>
<th>Unix Write Time (s)</th>
<th>Non Collective Write Time (s)</th>
<th>Collective Write Time (s)</th>
<th>Write Degree of Non-contiguity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3192</td>
<td>0.71791</td>
<td>0.01317</td>
<td>0.03553</td>
<td>0.98165</td>
</tr>
<tr>
<td>3768</td>
<td>1.13275</td>
<td>0.01018</td>
<td>0.03597</td>
<td>0.99101</td>
</tr>
<tr>
<td>6072</td>
<td>1.12006</td>
<td>0.01017</td>
<td>0.03715</td>
<td>0.99092</td>
</tr>
<tr>
<td>15288</td>
<td>1.10366</td>
<td>0.01255</td>
<td>0.03989</td>
<td>0.98863</td>
</tr>
<tr>
<td>52152</td>
<td>1.09026</td>
<td>0.01729</td>
<td>0.05127</td>
<td>0.98414</td>
</tr>
<tr>
<td>199608</td>
<td>1.25634</td>
<td>0.07692</td>
<td>0.09244</td>
<td>0.93877</td>
</tr>
</tbody>
</table>

Table 5.4: Read Degree of Non-contiguity on MercutIO-NFS

<table>
<thead>
<tr>
<th>Hole Size (bytes)</th>
<th>Unix Read Time (s)</th>
<th>Non Collective Read Time (s)</th>
<th>Collective Read Time (s)</th>
<th>Read Degree of Non-contiguity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3192</td>
<td>1.18344</td>
<td>0.03184</td>
<td>0.03422</td>
<td>0.9731</td>
</tr>
<tr>
<td>3768</td>
<td>1.16845</td>
<td>0.04687</td>
<td>0.03555</td>
<td>0.96958</td>
</tr>
<tr>
<td>6072</td>
<td>1.23759</td>
<td>0.03869</td>
<td>0.03948</td>
<td>0.96874</td>
</tr>
<tr>
<td>15288</td>
<td>1.20014</td>
<td>0.06368</td>
<td>0.06093</td>
<td>0.94923</td>
</tr>
<tr>
<td>52152</td>
<td>1.25298</td>
<td>0.15071</td>
<td>0.13558</td>
<td>0.89179</td>
</tr>
<tr>
<td>199608</td>
<td>1.26188</td>
<td>0.54968</td>
<td>0.45047</td>
<td>0.64302</td>
</tr>
</tbody>
</table>
Table 5.5: Write Degree of Non-contiguity on MercutIO-NFS

<table>
<thead>
<tr>
<th>Hole Size (bytes)</th>
<th>Unix Write Time (s)</th>
<th>Non Collective Write Time (s)</th>
<th>Collective Write Time (s)</th>
<th>Write Degree of Non-contiguity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3192</td>
<td>83.48726</td>
<td>0.32915</td>
<td>0.2551</td>
<td>0.99694</td>
</tr>
<tr>
<td>3768</td>
<td>100.64567</td>
<td>0.33767</td>
<td>0.28357</td>
<td>0.99718</td>
</tr>
<tr>
<td>6072</td>
<td>101.08365</td>
<td>0.5201</td>
<td>0.42229</td>
<td>0.99582</td>
</tr>
<tr>
<td>15288</td>
<td>100.6339</td>
<td>1.03305</td>
<td>0.81475</td>
<td>0.9919</td>
</tr>
<tr>
<td>52152</td>
<td>115.6262</td>
<td>3.87482</td>
<td>2.73985</td>
<td>0.9763</td>
</tr>
<tr>
<td>199608</td>
<td>114.87027</td>
<td>14.27263</td>
<td>10.52783</td>
<td>0.90835</td>
</tr>
</tbody>
</table>

5.4 Parallel-IO Benchmark for Asynchronous IO (PIOBA)

5.4.1 PIOBA Description

This benchmark evaluates the performance of non-collective and collective MPI-IO routines with and without overlapping with a fixed-size computation problem. The pseudocode of the asynchronous, non-collective I/O subtest is shown in Figure 5.3. In every iteration, data blocks of different sizes (1KB-64KB) are accessed contiguously from the file. First, the read/write time for a given block size is measured using the synchronous MPI-IO API. Then, the time taken for the completion of a given computation problem, independent of I/O, is measured. Subsequently, the given computation problem is scheduled concurrently with the I/O using asynchronous MPI-IO APIs. The time taken for the completion of the computation and I/O is measured. This benchmark generates performance numbers for the write operations with MPI_File_sync(). The degree of overlapping is computed
using the procedure listed in Section 3.1.2. This benchmark is tested on UFS, NFS, and PVFS.

```c
for(i=0; i<NumRuns; i++)
{
    BlockSize = pow(2,i);
    StartTime = MPI_WTime();
    MPI_File_<iread/iwrite>();
    Compute_Cycles();
    MPI_Wait();
    IOTime = MPI_WTime() - StartTime;
    MPI_Reduce(IOTime,MPI_MAX,Root);
}
```

Figure 5.3: Pseudocode of PIOBA

### 5.4.2 Expected Results

The amount of overlapping that can be achieved is influenced by the nature of the computation problem, the amount of data being accessed, the architecture of the machine, and the type of threads supported by the operating system. The experimental testbed consists of single CPU nodes. The operating systems Solaris and Linux both support kernel-level threads. The computation problem does not perform file access and also lacks data dependency with I/O performed asynchronously. Hence, on this testbed, employing asynchronous I/O based on the BAFS thread model efficiently overlaps I/O and computation. The overhead introduced by the BAFS thread-queue-data-access model is minimal compared to using the blocking native file system’s interface. The theoretical explanation was discussed in Section 3.1.1.
Table 5.6: Degree of Overlapping on MercutIO-UFS

<table>
<thead>
<tr>
<th>Block Size (Bytes)</th>
<th>Non-Collective I/O</th>
<th>Collective I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RDO</td>
<td>WDO</td>
</tr>
<tr>
<td>4096</td>
<td>0.06933</td>
<td>0.18195</td>
</tr>
<tr>
<td>8192</td>
<td>0.09779</td>
<td>0.14431</td>
</tr>
<tr>
<td>16384</td>
<td>0.12389</td>
<td>0.18971</td>
</tr>
<tr>
<td>32768</td>
<td>0.16792</td>
<td>0.21294</td>
</tr>
<tr>
<td>65536</td>
<td>0.25614</td>
<td>0.39476</td>
</tr>
<tr>
<td>131072</td>
<td>0.38097</td>
<td>0.3579</td>
</tr>
</tbody>
</table>

5.4.3 Experimental Results and Analysis

The Read Degree of Overlapping (RDO) and Write Degree of Overlapping (WDO) are tabulated in Tables 5.6, 5.7, and 5.8. It can be inferred from these tables that I/O access based on the non-blocking semantics using the threaded model helps to overlap computation and I/O (positive values of degree of overlapping). The degree of overlapping can also be understood from the graphical illustrations in Figures 5.6, 5.7, 5.8, 5.9, 5.10 and 5.11. In all the graphs, the overlap time (compute cycles with asynchronous APIs) is plotted against non-overlapping time (compute cycles with synchronous APIs). The overlap time is less than the non-overlapping time in all these graphs. The amount of separation between these two plots (overlap and non-overlapping time) indicates the magnitude of the degree of overlapping attained.

5.5 Overhead Testing

5.5.1 Performance Test for Portability Overhead

In this subsection, the portability overhead benchmark is described, then applied to MercutIO.
Table 5.7: Degree of Overlapping on MercutIO-NFS

<table>
<thead>
<tr>
<th>Block Size (Bytes)</th>
<th>Non-Coll I/O</th>
<th></th>
<th>Collective I/O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RDO</td>
<td>WDO</td>
<td>RDO</td>
<td>WDO</td>
</tr>
<tr>
<td>4096</td>
<td>0.14489</td>
<td>0.33099</td>
<td>0.32726</td>
<td>0.36757</td>
</tr>
<tr>
<td>8192</td>
<td>0.16516</td>
<td>0.39037</td>
<td>0.37194</td>
<td>0.42501</td>
</tr>
<tr>
<td>16384</td>
<td>0.20253</td>
<td>0.34966</td>
<td>0.01992</td>
<td>0.35084</td>
</tr>
<tr>
<td>32768</td>
<td>0.34748</td>
<td>0.19346</td>
<td>0.01561</td>
<td>0.19076</td>
</tr>
<tr>
<td>65536</td>
<td>0.23368</td>
<td>0.07123</td>
<td>0.04072</td>
<td>0.09483</td>
</tr>
<tr>
<td>131072</td>
<td>0.13519</td>
<td>0.03614</td>
<td>0.02587</td>
<td>0.06567</td>
</tr>
</tbody>
</table>

Table 5.8: Degree of Overlapping on MercutIO-PVFS

<table>
<thead>
<tr>
<th>Block Size (Bytes)</th>
<th>Non-Coll I/O</th>
<th></th>
<th>Collective I/O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RDO</td>
<td>WDO</td>
<td>RDO</td>
<td>WDO</td>
</tr>
<tr>
<td>4096</td>
<td>0.42823</td>
<td>0.02985</td>
<td>0.01381</td>
<td>0.02021</td>
</tr>
<tr>
<td>8192</td>
<td>0.23984</td>
<td>0.09743</td>
<td>0.13975</td>
<td>0.0661</td>
</tr>
<tr>
<td>16384</td>
<td>0.22011</td>
<td>0.14477</td>
<td>0.15418</td>
<td>0.10624</td>
</tr>
<tr>
<td>32768</td>
<td>0.25805</td>
<td>0.10692</td>
<td>0.05251</td>
<td>0.04883</td>
</tr>
<tr>
<td>65536</td>
<td>0.18061</td>
<td>0.0174</td>
<td>0.16127</td>
<td>0.10647</td>
</tr>
<tr>
<td>131072</td>
<td>0.02918</td>
<td>0.21626</td>
<td>0.06792</td>
<td>0.09096</td>
</tr>
</tbody>
</table>
5.5.1.1 Description

Portability overhead test measures the latency and overhead associated with the portable MPI-IO implementation over the native file system calls. This test involves reading and writing data blocks of different size (64KB-1MB) contiguously to and from a file for several (ten) iterations and calculating the average I/O access time. This test measures the read and write access time at the MPI level ($T_m$) and at the low level ($T_p$) using the native file system calls. The low-level implementations will yield a baseline for the I/O access time that will be used for comparison against MPI-IO. The portability overhead ($\gamma$) is computed using the following formula:

$$\gamma = \frac{T_m - T_p}{T_p}.$$

The portability overhead of MercutIO on UFS, NFS, and PVFS will also be calculated using this test.

Table 5.9: Portability Overhead of MercutIO

<table>
<thead>
<tr>
<th>Block Size (bytes)</th>
<th>UFS</th>
<th>NFS</th>
<th>PVFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>65536</td>
<td>32.749%</td>
<td>3.721%</td>
<td>37.340%</td>
</tr>
<tr>
<td>262144</td>
<td>27.460%</td>
<td>1.470%</td>
<td>9.073%</td>
</tr>
<tr>
<td>1048576</td>
<td>11.352%</td>
<td>1.295%</td>
<td>10.325%</td>
</tr>
</tbody>
</table>

5.5.1.2 Experimental Results and Analysis

It is evident from Table 5.9 that performing file access using MercutIO has a low overhead compared to using native file system APIs. The portability overhead decreases as the block size is increased. This is because of improved file systems
performance and a decrease in non-blocking data access model overhead (discussed in Section 3.1.1) for large data accesses. Also from the graph shown in Figure 5.5, it can be inferred that MercutIO APIs have the same order of complexity as the native file system interfaces. The amount of portability overhead is indicated by the amount of separation between the native file system and MPI-IO-native file system plots (in Figure 5.5), which is evidently small.

5.5.2 Performance Test for Non-Blocking Semantics Overhead

In this subsection, the non-blocking semantics overhead benchmark is described, then applied to MercutIO.

5.5.2.1 Description

The Non-blocking semantics overhead test measures the latency and overhead associated with the portable MPI-IO implementation based on the non-blocking semantics against blocking semantics. The test involves reading and writing data blocks of different size (64KB-1MB) contiguously to and from a file for several (ten) iterations and calculating the average I/O access time. This test measures the read and write access time of an MPI-IO implementation based on the blocking ($T_b$) and non-blocking semantics ($T_{nb}$). The non-blocking semantics are based on the thread-queue model and the blocking semantics use the blocking APIs of the native file system. The overhead is determined using the following formula:

$$\lambda = \frac{T_{nb} - T_b}{T_b}.$$  

The pseudocode of this test is shown in 5.4. This test is performed on UFS in Solaris and Linux platform.
5.5.2.2 Experimental Results and Analysis

The overhead introduced by the non-blocking semantics data access model in MercutIO is tabulated in Table 5.10 and shown in Figure 5.12. The non-blocking semantics data access model has minimal overhead compared to the blocking semantic data access model. This overhead becomes smaller as the amount of data being accessed increases because the context switching overhead (in BAFS thread model) becomes minimal compared to the read/write time in the case of large I/O requests. On systems that do not support kernel-level threads, the data access mechanism can be switched to blocking semantics to reduce the non-blocking semantics overhead.

```c
for(i=0; i<NumRuns; i++)
{
    MPI_Info_set(info,"blocking/non-blocking");
    MPI_File_open(..,info,..);
    StartTime = MPI_WTime();
    BlockSize = pow(2,i);
    MPI_File_write(2,i);
    ElapsedTime = MPI_WTime() - StartTime;
    MPI_Reduce(ElapsedTime,MPI_MAX,Root);
}
```

Figure 5.4: Pseudocode for Measuring the Non-Blocking Semantics Overhead

5.6 Summary of Results

All the performance tests yielded the expected results, validating of the hypothesis. The agglomeration optimizations implemented in MercutIO significantly improved the performance compared to the Unix-style access mechanism when the intermediate hole size between non-contiguous data chunks was small. PIOBA
revealed that I/O access based on the non-blocking semantics using the threaded model helps to overlap computation and I/O and subsequently lowers the parallel I/O application’s overall execution time. It was also shown that employing MercutIO to perform file access introduced minimal overhead compared to native file system APIs. Furthermore, it was experimentally proven that the default data access model in MercutIO based on non-blocking semantics introduces minimal overhead compared to the blocking semantic data access model.

Table 5.10: Non-Blocking Semantics Overhead of MercutIO

<table>
<thead>
<tr>
<th>Block Size (bytes)</th>
<th>SOLARIS-UFS</th>
<th>LINUX-UFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>65536</td>
<td>33.829%</td>
<td>12.694%</td>
</tr>
<tr>
<td>262144</td>
<td>11.31%</td>
<td>8.896%</td>
</tr>
<tr>
<td>1048576</td>
<td>4.73%</td>
<td>4.771%</td>
</tr>
</tbody>
</table>
Figure 5.5: File Access Time using MPI-IO APIs and Native File System APIs
Figure 5.6: Non-Collective Degree of Overlapping on MercutIO-UFS
Figure 5.7: Collective Degree of Overlapping on MercutIO-UFS
Figure 5.8: Non-Collective Degree of Overlapping on MercutIO-NFS
Figure 5.9: Collective Degree of Overlapping on MercutIO-NFS
Figure 5.10: Non-Collective Degree of Overlapping on MercutIO-PVFS
Figure 5.11: Collective Degree of Overlapping on MercutIO-PVFS
Figure 5.12: File Access Time Using Blocking and Non-Blocking Semantics
CHAPTER VI

CONCLUSIONS

6.1 Summary

This thesis hypothesized that overlap of I/O and computation and agglomeration of I/O requests based on an application’s access pattern improve the performance of a portable parallel I/O implementation.

The hypothesis, motivation, and presumptive contributions of this thesis were presented in the first chapter. The primary motivation for this thesis arose from the lack of a portable, high-performance MPI-IO implementation. Overlap of I/O and computation and agglomeration of I/O requests were identified as major sources of performance in a portable MPI-IO implementation. The hypothesis was supported through the design and implementation of MercutIO, a portable, high-performance MPI-IO implementation. Portability in MercutIO was ensured through BAFS, an abstract file interface for I/O primarily developed to provide an efficient, portable, point-to-point I/O interface.

The literature review revealed the support for parallel I/O in existing parallel file systems and MPI-IO implementations. It was shown that most file systems did not support collective I/O. Non-contiguous file access was supported only in PVFS. Asynchronous I/O was not supported in most of the file systems and, if supported, was based on the inefficient polling model. Therefore, the MPI-IO implementation on these file systems required newer design approaches to guarantee maximal performance on the I/O subsystems. A brief description of some of the
existing MPI-IO implementations – ROMIO and MPI-IO over GPFS – was given in this chapter. Asynchronous I/O on these implementations was based either on blocking semantics or on the polling model, neither of which efficiently utilizes the CPU. Further, non-contiguous file accesses were based on agglomeration techniques, with poor heuristics to assess the non-contiguity in the file.

The third chapter presented a quantitative analysis of the fundamental sources of performance in an MPI-IO implementation. Overlap of I/O and computation and agglomeration of I/O requests were identified as important design criteria influencing the performance of a portable parallel I/O implementation. The quantitative analysis introduced the optimization strategies to be employed in the MPI-IO implementation. The use of the early binding concept in the asynchronous I/O implementation facilitates the creation and initialization of the necessary data structure once, thereby minimizing overhead. The flexible approach to service the non-contiguous I/O requests based on the nature of the application’s access pattern theoretically promises significant performance gains. Finally, two new parallel I/O metrics – degree of overlapping and degree of non-contiguity – that help in the performance appraisal of the parallel I/O implementation were introduced in this chapter.

The fourth chapter presented the design and implementation of MercutIO. The architecture of MercutIO outlined the role of BAFS and the MPI-IO component. I/O parallelism was achieved in the MPI-IO layer, while portability was taken care of by the BAFS layer. The design features of BAFS, namely file-access-based on the thread-queue model, early binding, and agglomeration of smaller I/O requests to optimally sized chunks highlighted the non-collective optimizations in MercutIO. The collective I/O optimizations and the shared file pointer implementation that
required inter-process communication were implemented in the MPI-IO layer. This chapter also discussed the methodologies used in the implementation of MercutIO.

The experiments included design and implementation of new and revised benchmarks to measure the parallel-I/O metrics. All the performance tests yielded expected results, thereby validating the hypothesis. The agglomeration optimizations implemented in MercutIO significantly improved the performance compared to the Unix-style access mechanism when the hole size was small. PIOBA revealed that I/O access based on non-blocking semantics using the threaded model helps to overlap computation and I/O and subsequently lowers the parallel I/O application’s overall execution time. It was also shown that employing MercutIO to perform file access introduced minimal overhead compared to native file system APIs. Furthermore, it was experimentally proven that the default data access model in MercutIO based on non-blocking semantics introduces minimal overhead compared to the blocking semantic data access model.

6.2 Future Work

This thesis has validated theoretically and experimentally that overlap of I/O and computation and agglomeration of I/O requests based on an application’s access pattern improves the performance of a portable MPI-IO implementation. The file systems considered for the study were all based on the traditional file access protocol. It would be interesting to investigate the performance characteristics of MercutIO on the emerging Direct File Access Protocol (DAFS). DAFS is being designed primarily to take advantage of emerging interconnect technologies such as Virtual Interface (VI) architecture and Infiniband in System Area Networks (SAN).
The MPI-IO implementation described in this thesis provides a portable file interface across file systems present locally through BAFS. It would be interesting to employ BAFS for remote file transfers using the File Transfer Protocol (RFC 959). The application can provide the file name along with other information about the remote machine to BAFS. BAFS can then establish a transparent session with the remote machine and perform file transfers. BAFS can also be implemented over Remote I/O (RIO) library (Foster et al. 1997) for high-performance access to data located in remote, potentially parallel file systems.
REFERENCES


Thakur, R., and A. Choudhary. 1996. An extended two-phase method for accessing


APPENDIX A

CLASSIFICATION OF MPI-IO APIS
• Basic File Manipulation

1. MPI File open
2. MPI File close
3. MPI File seek
4. MPI File delete
5. MPI File set size
6. MPI File get size
7. MPI File preallocate

• Data Access Functions

- Non-Collective Data Access Functions

* Local Pointer

1. MPI File read
2. MPI File write
3. MPI File read
4. MPI File write

* Explicit Offset

1. MPI File read at
2. MPI File write at
3. MPI File read at
4. MPI File write at

* Shared File Pointer

1. MPI File read shared
2. MPI_File_write_shared
3. MPI_File_read_shared
4. MPI_File_write_shared

- Collective Data Access Functions

* Local Pointer
  1. MPI_File_read_all
  2. MPI_File_write_all
  3. MPI_File_read_all_begin
  4. MPI_File_write_all_begin
  5. MPI_File_read_all_end
  6. MPI_File_write_all_end

* Explicit Offset
  1. MPI_File_read_at_all
  2. MPI_File_write_at_all
  3. MPI_File_read_at_all_begin
  4. MPI_File_write_at_all_begin
  5. MPI_File_read_at_all_end
  6. MPI_File_write_at_all_end

* Shared File Pointer
  1. MPI_File_read_ordered
  2. MPI_File_write_ordered
  3. MPI_File_read_ordered_begin
  4. MPI_File_write_ordered_begin
5. MPI_File_read_ordered_end
6. MPI_File_write_ordered_end

- Non-contiguous file access using Views
  1. MPI_File_set_view
  2. MPI_File_get_view

- Consistency and Semantics
  1. MPI_File_set_atomicity
  2. MPI_File_get_atomicity
  3. MPI_File_sync

- Miscellaneous Functions
  1. MPI_File_set_info
  2. MPI_File_get_info
  3. MPI_File_get_amode
  4. MPI_File_get_group
  5. MPI_File_get_offset
  6. MPI_File_get_position
  7. MPI_File_byte_offset
  8. MPI_File_get_position_shared
  9. MPI_File_set_errhandler
  10. MPI_File_get_errhandler
  11. MPI_File_C2F
12. MPI_File_F2C
13. MPI_File_get_type_extent
14. MPI_File_seek_shared
15. MPI_File_call_errhandler
16. MPI_File_create_errhandler
APPENDIX B

BAFS APIs
1. BAFS_FILE::Open
   Opens or creates a file.

2. BAFS_FILE::Close
   Breaks the connection between a file descriptor and an open file, and frees the
   file descriptor for use with some other file.

3. BAFS_FILE::Delete
   Removes the file from the file system.

4. BAFS_FILE::Seek_Individual
   Positions the file pointer in a file.

5. BAFS_FILE::Read_Contig
   Read contiguous chunks of data from a file into contiguous memory.

6. BAFS_FILE::Write_Contig
   Write data contiguously from a contiguous buffer into the file.

7. BAFS_FILE::Read_Contigv
   Read contiguous chunks of data from a file into non-contiguous memory
   locations.

8. BAFS_FILE::Write_Contigv
   Write data contiguously from non-contiguous memory locations into the file.

9. BAFS_FILE::Read_Noncontig
   Read non-contiguous chunks of data from a file into contiguous memory.

10. BAFS_FILE::Write_noncontig
    Write data non-contiguously from a contiguous buffer into the file.
11. BAFS_File::Read_Init
   Specify the read parameters a priori to enable early binding.

12. BAFS_File::Write_Init
   Specify the write parameter a priori to enable early binding.

13. BAFS_File::Start
   Start the actual read or write operation in early binding.

14. BAFS_File::Test
   Test the completion of asynchronous file operation.

15. BAFS_File::Wait
   Wait for the completion of the asynchronous file operation.

16. BAFS_File::Read_Init_Pth
    Specify the read parameters a priori to enable early binding in thread data access model introduced in this thesis.

17. BAFS_File::Write_Init_Pth
    Specify the write parameters a priori to enable early binding in thread data access model introduced in this thesis.

18. BAFS_File::Start_Pth
    Start the actual read or write operation in the thread data access model introduced in this thesis.

19. BAFS_File::Test_Pth
    Test for the completion of asynchronous file operation in thread data access model.
20. BAFS_File::Wait_Pth
   Wait for the completion of asynchronous file operation in thread data access model.

21. BAFS_File::Read_Noncontig_Init_Pth
   Specify the read parameters a priori to enable early binding in thread data access model introduced in this thesis. The non-contiguous locations in the file are specified using this API.

22. BAFS_File::Write_Noncontig_Init_Pth
   Specify the write parameters a priori to enable early binding in thread data access model introduced in this thesis. The non-contiguous locations in the file are specified using this API.

23. BAFS_File::Fcntlfunc
   Provides file control functionalities.