A FRAMEWORK ARCHITECTURE FOR SHARED FILE POINTER OPERATIONS IN OPEN MPI

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Carlos R. Vanegas

May 2013

A FRAMEWORK ARCHITECTURE FOR SHARED FILE POINTER OPERATIONS IN OPEN MPI

Carlos R. Vanegas

APPROVED:

Dr. Edgar Gabriel
Dept. of Computer Science, University of Houston

Dr. Jaspal Subhlok
Dept. of Computer Science, University of Houston

Dr. Mohamad Chaarawi
The HDF Group, Champaign, IL

Dean, College of Natural Sciences and Mathematics

A FRAMEWORK ARCHITECTURE FOR SHARED FILE POINTER OPERATIONS IN OPEN MPI

An Abstract of a Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Carlos R. Vanegas
May 2013

Abstract

MPI is a message passing interface that provides a distributed memory programming model for parallel computation. MPI-I/O is a parallel I/O library that is part of the MPI-2 specification. As an intermediate layer between the application and the file system, MPI-I/O is able to support features not directly enabled in the underlying file system. One such feature is a shared file pointer. A shared file pointer is a file pointer to an open file that is shared between the processes that opened the file.

The objective of this thesis is to develop a framework for shared file pointer operations in OMPIO for MPI-I/O, implement and evaluate various existing and new algorithms for shared file pointers, and to develop and evaluate a selection logic to decide which algorithm to use. Four algorithms are implemented in this thesis: the locked file, the shared memory, the individual file, and the additional process algorithms.

The results show that the shared memory algorithm is the fastest. Unfortunately, it can only be used if all of the processes are executing on a single node. The individual algorithm provides a good option when running on multiple nodes, but only supports write operations. The additional process algorithm can be run from multiple nodes and can be used on all file systems, but may not be supported on all environments due to the requirement of spawning an additional process.

Contents

1 Introduction **1**

2 Background **5**

2.1 Parallelism 6

2.2 Approach to parallel computing 8

2.2.1 Instruction level 8

2.2.2 Distributed memory computing 9

2.2.3 Shared memory computing 10

2.2.4 Comparison 10

2.3 Problem in parallel computing 11

2.3.1 The I/O problem 12

2.3.2 The I/O solution 12

2.4 I/O levels 13

2.4.1 Storage device I/O level 13

2.4.2 File system I/O level 15

2.4.3 I/O library and application level 19

2.5 MPI and MPI-I/O 20

2.5.1 MPI communication 21
2.5.2 MPI derived datatype 21
2.5.3 MPI-I/O communication groups and file view 22
2.5.4 MPI-I/O file access functions 23
2.5.5 MPI-I/O hints 24
2.6 MPI-I/O shared file pointers 25

3 Approach **27**
3.1 Locked file algorithm 28
3.2 Shared memory algorithm 29
3.3 Individual file algorithm 30
3.4 Additional process algorithm 31

4 Implementation **32**
4.1 Open MPI 32
4.2 OMPIO 34
4.3 OMPIO sharedfp framework 35
4.3.1 Functions in sharedfp framework 35
4.4 Locked file component 39
4.4.1 Locking the file 39
4.4.2 General steps 39
4.4.3 Selection logic implementation 40
4.4.4 Component interface implementation 41
4.5 Shared memory component 41
4.5.1 Shared memory segment 41

4.5.2 General algorithm steps 43
4.5.3 Selection logic implementation 44
4.5.4 Component interface implementation 44
4.6 Individual file component 45
4.6.1 The metadata record 45
4.6.2 General algorithm steps 45
4.6.3 Merging 46
4.6.4 Selection logic implementation 48
4.6.5 Component interface implementation 48
4.7 Additional process component 49
4.7.1 The additional process 49
4.7.2 General algorithm steps 50
4.7.3 Selection logic implementation 51
4.7.4 Component interface implementation 51

5 Measurements and Evaluation 52

5.1 Benchmark description 53
5.2 Test environment 54
5.3 Test Series 1 : Fixed number of processes 55
5.3.1 Writing to PVFS2 from a single node 56
5.3.2 Writing to PVFS2 from multiple nodes 60
5.3.3 Writing to ext4 62
5.4 Test Series 2: Increasing number of processes 65
5.4.1 Writing to PVFS2 from a single node 66

5.4.2 Writing to PVFS2 from multiple nodes 68

5.4.3 Writing to ext4 70

5.5 Test Series 3: Varying I/O requests 72

5.5.1 Writing to PVFS2 from a single node 73

5.5.2 Writing to PVFS2 from multiple nodes 76

5.5.3 Writing to ext4 77

5.6 Test Series 4: Constant write size per process 79

5.6.1 Writing to PVFS2 from a single node 80

5.6.2 Writing to PVFS2 from multiple nodes 82

5.6.3 Writing to ext4 84

6 Summary **87**

Bibliography **90**

List of Figures

5.1 Test Series 1: On PVFS2 from single node - Comparison for blocking, non-collective write operation . . . 57

5.2 Test Series 1: On PVFS2 from single node - Comparison for blocking, non-collective write operation (close up) . . . 58

5.3 Test Series 1: On PVFS2 from single node - Comparison for blocking, collective write operation . . . 59

5.4 Test Series 1: On PVFS2 from multiple nodes - Comparison for blocking, non-collective write operation . . . 61

5.5 Test Series 1: On PVFS2 from multiple nodes - Comparison for blocking, collective write operation . . . 62

5.6 Test Series 1: On Ext4 - Comparison for blocking, non-collective write operation . . . 63

5.7 Test Series 1: On Ext4 - Comparison for blocking, collective write operation . . . 64

5.8 Test Series 2: On PVFS2 from single node - Comparison for blocking, non-collective write operation . . . 66

5.9 Test Series2: On PVFS2 from single node - Comparison for blocking, collective write operation . . . 68
5.10 Test Series 2: On PVFS2 from multiple nodes - Comparison for blocking, non-collective write operation . . . 69
5.11 Test Series 2: On PVFS2 from multiple nodes - Comparison for blocking, collective operation . . . 70
5.12 Test Series 2: On Ext4 - Comparison for blocking, non-collective write operation . . . 71
5.13 Test Series 2: On Ext4 - Comparison for blocking, collective write operation . . . 72
5.14 Test Series 3: On PVFS2 from single node - Comparison for blocking, non-collective write operation . . . 74
5.15 Test Series 3: On PVFS2 from single node - Comparison for blocking, non-collective write operation (close up) . . . 75
5.16 Test Series 3: On PVFS2 from single node - Comparison for blocking, collective write operation . . . 75
5.17 Test Series 3: On PVFS2 from multiple nodes - Comparison for blocking, non-collective write operation . . . 76
5.18 Test Series 3: On PVFS2 from multiple nodes - Comparison for blocking, collective operation . . . 77
5.19 Test Series 3: On Ext4 - Comparison for blocking, non-collective write operation . . . 78

5.20 Test Series 3: On Ext4 - Comparison for blocking, collective write operation . . . 79
5.21 Test Series 4: On PVFS2 from single node - Comparison for blocking, non-collective write operation . . . 80
5.22 Test Series 4: On PVFS2 from single node - Comparison for blocking, collective write operation . . . 82
5.23 Test Series 4: On PVFS2 from multiple nodes - Comparison for blocking, non-collective write operation . . . 83
5.24 Test Series 4: On PVFS2 from multiple nodes - Comparison for blocking, collective operation . . . 84
5.25 Test Series 4: On Ext4 - Comparison for blocking, non-collective write operation . . . 85
5.26 Test Series 4: On Ext4 - Comparison for blocking, collective write operation . . . 86

List of Tables

4.1 sharedfp framework: General Operations 37

4.2 sharedfp framework: Write Operations 38

4.3 sharedfp framework: Read Operations 38

4.4 Individual File : Metadata Record 45

5.1 Test Series 1: Fixed number of processes, increasing data output . . . 56

5.2 Test Series 2: Increasing number of processes 65

5.3 Test Series 3: Varying I/O requests 73

5.4 Test Series 4: Constant write size per process 80

Chapter 1

Introduction

A shared file pointer is a file pointer to a file that was opened by a group of processes. Each process in the group has access to the shared file pointer. An access request by one process updates the shared file pointer and the system ensures that each process gets the updated shared file pointer position. Shared file pointers facilitate dynamic access to a file by a group of processes. Without a shared file pointer, processes must predetermine the part of the file that each processes is allowed to access.

Shared file pointers are currently not supported natively in most mainstream file systems, but are defined in the MPI-I/O [1] standard. ROMIO [2] is an implementation of MPI-I/O; it supports shared file pointer operations by using file locking to synchronize access to the shared file pointer. Unfortunately, file locking is not supported on all file systems. This means that ROMIO's solution does not work on every file system.

As a result application programmers needing the shared file pointer functionality in their programs must restrict their application to only run on file systems where

ROMIO shared file pointers work correctly. Alternatively the application programmer can take the time to implement partial shared file pointer logic in their program. Having a portable implementation would free the application programmer of these burdens.

Due to the necessary synchronization, shared file pointer implementations tend not to provide good performance. Application programmers will often shy away from using shared file pointers and instead opt for other less elegant solutions that provide faster performance during the program's execution. Often the hardware and application combination meet certain criteria that would allow a specialized shared file pointer implementation to be used instead of a general portable one. For example, an application only needing write access could use an algorithm that was optimized for writing but not for reading. If the I/O layer could provide a good collection of algorithms and smart enough selection logic to choose the fastest algorithm, then application programmers may be more willing to use shared file pointers.

The goal of this thesis is to provide a collection of shared file pointer algorithms and give the end user the possibility to select between those algorithms using appropriate hints. Some researchers proposed a portable solution for synchronizing access to the the file pointer using MPI's one-sided communication [1]. This solution [3] has not yet been merged into ROMIO.

This thesis extends the work done by Ketan Kulkarni in shared file pointer implementation [4]. The flexibility of being able to choose between a collection of algorithms is based on an MPI-I/O implementation called OMPIO [5]. OMPIO is a modular software architecture that provides a highly modular approach to parallel

IO for OpenMPI [6], an open source MPI implementation.

Ideally, the OMPIO I/O library will select a specialized algorithm which will provide the best performance for the current hardware and application combination. In the worst case scenario at least a portable algorithm will be selected which will at least make sure that the shared file pointer functionality is available for the application. This removes the current restriction of only running on file systems with file locking capabilities. Alternative implementations provide better performance if certain criteria is met.

In order to meet this goal the following will be done in this thesis:

1. Develop a framework for shared file pointer operations in OMPIO,
2. Implement and evaluate various existing and new algorithms for shared file pointers,
3. Develop and evaluate a selection logic to decide which algorithm to use.

The work of this thesis will be evaluated by measuring the performance on several hardware/software configurations. We will be able to set up a test case and run measurements for each algorithm. We can then run the selection algorithm, to verify that the fastest algorithm was indeed chosen by the selection logic.

The remainder of the document is organized as follows. Chapter 2 gives a basic introduction to parallel computation and parallel I/O. It includes a brief description of MPI and various MPI features, including the shared file pointer definition in MPI-I/O. Chapter 3 describes the four algorithms without giving details of the implementation. Chapter 4 describes how shared file pointer framework and the four

algorithms were implemented in OMPIO. Chapter 5 provides a description of the performance measurements obtained by running benchmarks on various file systems. Chapter 6 summarizes the thesis and outlines the scope for future work.

Chapter 2

Background

This chapter introduces the notions of parallelism within the context of parallel computation and then within the context of parallel I/O. In a parallel application the focus is usually on speeding up the computation. It is also important to focus on ways to speed up I/O since I/O is also part of the overall execution time of a parallel application.

Optimizing sequential I/O is complex because different hardware and system configurations respond differently to the same optimizations. Parallel I/O faces these same problems and is complicated further by the challenges faced in parallelization. I/O is abstracted into different levels. Each of these levels needs to be parallelized in order to achieve the optimal speedup. Further speed up can be obtained by making optimizations at the higher level I/O that address limitations of the lower levels of I/O.

This chapter focuses on the I/O library level as the level that can bring the most portable benefits to parallel I/O. MPI-I/O, a current parallel I/O library receives

special attention as it is one of the most widely used parallel I/O libraries.

The section ends by discussing the importance of shared file pointers within the context of parallel computation.

2.1 Parallelism

The idea behind parallel computing is to take a given problem and divide it into smaller sub-problems [7]. Each of these sub-problems can then be solved "concurrently". The main motivation for parallelization is to speedup the execution of the problem. Another reason is to execute problems that are too large to fit into a single computer's memory.

Unfortunately, parallelizing an application is not always automatic or straight forward. An obvious question is whether or not it is worth the required resources. Several metrics have been developed in order to measure and estimate the performance of a parallelized application.

Speedup The "speedup" metric [8] tells us how many times faster the parallelized application is than the sequential version. Dividing the execution time of one process by the execution time of p processes gives the achieved speedup.

```
S(p) = total_time(1) / total_time(p)
```

Efficiency The speedup measure tells us how much faster a parallel application is but does not tell us how efficiently it is achieved. For example, a speedup of 2

with 10 processes is less efficient than the same speedup with only 3 processes The efficiency of a parallel application [8] can be calculated with the following equation:

```
E(p) = S(p) / p
```

This measure normalizes the speedup by dividing it by the number of processes. The ideal efficiency is E(p)=1. If 4 processes are used then the speedup should be 4 in order to have the optimal efficiency of 1.

Maximum expected speedup The efficiency measure tells us how close our solution is to achieving ideal efficiency on p processes. If our solution does not achieve an efficiency value of 1, should we get back to work at improving our solution? Actually, it is possible that is the best we can do. There are several factors that place a maximum limit on the potential speedup.

Inherently sequential - Amdahl's law A computer architect by the name of Gene Amdahl derived an equation [8], in order to find the maximum expected speedup when part of a solution can not be parallelized.

```
S(p) = 1 / (f + (1-f)/p)
```

In the equation, f is the fraction of the problem which can only be executed sequentially and p is the number of processes used. The speed up is achieved from the part of the computation that is parallelizable, *1-f*. The implication of Amdahl's law is that the speedup is inherently limited by the time needed for the sequential fraction of the problem.

Parallel application overhead Many problems which are broken down into sub-problems require information from each sub-problem to be shared during the computation. The communication needed in order to share information takes time that was not needed by the sequential solution. This additional overhead work, effectively increases the problem size. Therefore, in this case, it is impossible to achieve the ideal efficiency since the parallel version is doing more work than the sequential version.

Another form of overhead that adds to the parallel application's execution time is the time that it takes to break up the problem into sub-problems and the time to merge the results. Due to the splitting and merging overhead, the speedup obtained will be less than the ideal.

2.2 Approach to parallel computing

There are several different ways in which parallelism has been achieved in computing. These methods include parallelizing at the hardware instruction level as well as at the abstract software level [9].

2.2.1 Instruction level

The CPU (Central Processing Unit) processes the individual instructions that make up a computer program. In addition to increasing the raw rate at which a CPU can process one instruction, there are other enhancements that have been made that increase the rate at which the CPU can execute multiple instructions [10].

The processing of one instruction has been broken up into multiple stages, allowing instructions to be pipelined while going through the CPU. This means that the

processor does not have to wait for one instruction to complete before starting the next one.

CPU designers also realized that during the execute stage, instructions used different hardware units at that stage. By duplicating the hardware in the other stages, multiple instructions could go into the pipeline as an instruction bundle to be processed in parallel.

These techniques have been crucial in CPU development, but they do have their limitations. Increasing the number of stages speeds up the execution time, but having too many stages increases the complexity of the CPU making it not cost-effective to manufacture. Instruction bundles require the instructions to be independent of each other. Therefore, it is not always possible to fill up the instruction bundle and maintain the peek speed.

2.2.2 Distributed memory computing

While instruction level parallelization brought about immense speed improvements, there was still a need for a more scalable solution. This need led to the development of distributed memory computing systems [7]. In a distributed memory computing system, computers, also referred to as nodes, are networked together. A problem can be decomposed and the sub-problems can be processed by different nodes. The nodes communicate with each other by passing messages and data back and forth through a message passing system. A distributed memory computing system does not face the same limitation that instruction level parallelization faces. The idea is that if a bigger problem arises all one needs to do is add another node.

2.2.3 Shared memory computing

The desire to speed up the instruction level, by duplicating parts of the CPU, eventually led to having a completely separate, additional CPU on the same computer [10]. Computers with more than one processor are known as multi-core computers. On a multi-core computer the hardware does not automatically split up the instructions of a single program between the available CPUs.

The operating system aims to automatically take advantage of the multiple CPUS by running multiple programs (processes) at the same time. For example, Process A can run on CPU 1 and Process B can run on CPU 2.

In order to give a program the ability to execute on more than one CPU at the same time, light weight processes, called a threads, have been developed. After the program starts, the main thread can create additional threads. All threads in a program can communicate with each other by reading and writing to a shared computer memory location. The work originally done by the single thread application can now be divided and executed in parallel. This type of programming is called shared memory programming.

As of this writing 4 core and 8 core computers are available for the general consumer market to buy [11]. 24 and 48 core systems are available but tend to be used for research and industrial purposes [12].

2.2.4 Comparison

Instruction level parallelism is not implemented by an application developer, it is done automatically by the compiler and computer hardware. Application developers

have two options when parallelizing their programs, the distributed message passing model or the shared memory model.

The distributed message passing model requires more changes to the sequential program than the shared memory model, but allows it to scale indefinitely. Shared memory programming on the other hand is limited by the amount of cores in the computer. It is much easier to add another node to your network, than to add another processor to the motherboard. Another limiting factor of shared memory programming is that shared resources, such as memory and the network interface, can become a bottleneck.

Because the shared memory programming model is easier to implement, many programmers prefer to use. The limiting factors of the shared memory model have become less of a concern since 24 and 48 core node computers are now available. The important thing to recognize is that the size of the problem matters. For problems that can not fit on one node the distributed message passing model is still required.

2.3 Problem in parallel computing

One way or another the benefits of parallelism are reaching all computer users. Now, even cell phones have dual and quad core processors [13]. Since parallel computing is an integral part of computer technology, it is worth the effort to make improvements by continuing research in this area.

In this thesis, we are interested in an issue that has not gotten enough attention, I/O in parallel computing [14].

2.3.1 The I/O problem

When an application starts up, it often needs to retrieve data from a storage device. Retrieving and saving the data is often set up as a sequential procedure. As a result, many parallel applications do not see the speedup they anticipated [14]. According to Amdahl's law, the more time the application spends reading and writing sequentially to a storage device, the less parallel efficiency it can hope to achieve.

The problem with the I/O bottleneck is currently even worse. While the CPU has been making drastic improvements in speed, I/O has been lagging behind. Disk drives are extremely slow compared to CPU speeds. Recently, big improvements have been made in access speeds, but even these improvements can not keep up with a parallel program's needs as it scales in size [14].

2.3.2 The I/O solution

Since parallelizing the computation can speed up execution time, it makes sense to also parallelize I/O. Unfortunately, only a few parallel programming models actually have any notion of parallel I/O.

Parallelizing I/O is not trivial. The challenge is to have a single logical view of the data, but to have multiple I/O channels to allow the processes to read and write data at the same time. To better understand the challenges in parallel I/O it is helpful to examine the different levels in the I/O stack. The next section provides an overview of each of the levels.

2.4 I/O levels

The following subsections examine the four main levels of the I/O stack from bottom to top. The storage device is examined first, followed by the file system, then the I/O library and finally the application level. Each section reviews the issues and the optimizations that can be implemented at each level in order to increase I/O performance.

2.4.1 Storage device I/O level

Current computers need to maintain a data transfer rate of up to 200 megabytes per second (MB/s) in order to support an I/O intensive application [14]. An I/O intensive parallel application may need many times higher transfer rates. Unfortunately, current disk drives can only transfer data at up to 100 MB/s.

Since the rate at which data can be retrieved from a storage device is a lot slower than the rate needed by the CPU, a faster intermediate storage device is used. Data is first copied from the normal storage device, secondary storage, to the intermediate device, primary storage. The CPU then accesses the data from the primary storage. Current primary storage, also known as main memory, has maximum transfer rates in the range of 3,200MB/s to 12,800 MB/s [15]

If the data are in primary storage, then there is no need to access secondary storage. The application would then perceive the data transfer rates of primary storage. Due to cost, primary storage is a lot smaller than secondary storage. This means, that the data may not be in primary storage when the application requests it. As a result the perceived data transfer rate by the application can potentially be

just as slow as if there were no main memory.

Speeding up secondary storage devices

Since data access performance is still very dependent on the secondary storage, speeding up the transfer rates that can be achieved by secondary storage devices is still very important.

The maximum transfer rate normally refers to the amount of data that a device can read or write once it is transferring data continuously. But, there is actually a delay before the first byte is transferred called the access time. The effective transfer rate, includes the access time and the data transfer time. For disk drives, avoiding the access time on a data request can increase the effective data transfer rate. One way to avoid the repeated access cost is to read a larger chunk of data at once.

Disk drives take advantage of reading larger chunks of contiguous data, by reading more data than is requested. The extra data are kept in a disk buffer. If the application then requests the next sequential set of bytes that are already in the buffer, the disk will be able to serve the data much faster. If the application does not have this type of data access pattern, then this optimization will not help.

One revolutionary solution that has been used to speed up disk drive access is called RAID (redundant array of independent disks)[14]. A group of disk drives are put together so that there are multiple channels for reading the data. Data can be accessed from each of the disks in parallel. The RAID configuration allows the operating system to access it as if it were one single drive. The file data is split into chunks and distributed (striped) across the disks in the RAID configuration. In order to provide high transfer rates all disks should transfer data at the same time for each

request. Unfortunately, the application's access pattern does not always match the way the data were stripped across the disks.

Solid state drives

Solid state drives (SSD), also known as flash drives, are a newer technology that has appeared in recent years. Unlike disk drives, solid state drives do not have any moving parts. As a result, they have access times that are a lot lower than disk drives. Additionally, an SSD is not limited by the spinning disks when accessing data. Solid state drives can transfer data at consistent rates of 100MB/s to 500MB/s [16].

Even though solid state drives provide better performance than disk drives, disk drives are still more commonly used due to cost and reliability [16].

2.4.2 File system I/O level

A file system provides a higher level view of the storage device. It organizes the underlying bytes into files and folders.

A traditional file system [14] offers a local file access model. It manages data access from the local host computer to a storage device which is directly attached to it. Information needed to locate the data blocks that have been allocated for a file are maintained in a list in a special place on the file system.

Distributed file systems [14] use a client-server file access model which allows files to be shared between a group of computers. The storage device is attached to a server. Client computers communicate with the server computer over the network. The server processes the access requests locally and responds to the client node with

the requested data.

The single server in a distributed file system results in a performance bottleneck as the load on the system increases. Parallel file systems [14] attempt to distribute the load over a group of data servers while still providing a view of a single unified file system. Client nodes send access requests to a metadata server. The metadata server tells the client node which data servers and what data blocks contain the requested data. The client node then interacts directly with the data servers.

Caching and Buffering in file systems

Similar to storage devices, file systems use caching and buffering to speed up data access [14].

Local file systems use local main memory to cache and buffer data. File data are cached not only when the application accesses it, but also when the file system prefetches data. Data buffering occurs when the file system delays writing to secondary storage. This can improve performance since writing large chunks of data is much more efficient than reading and writing many small chunks.

Caching and buffering in non-local file systems is much more complex and makes it difficult to provide local file system semantics. For example, local file system semantics dictate that every process should have a consistent view of a file. Extra communication overhead is required in order to maintain a consistent view of the data on every client. In order to avoid such a big performance loss, one solution is a relaxed consistency model which only informs the clients of changes when the client requests the file to be synchronized.

Additional speed up in parallel file systems

In addition to caching and buffering, parallel file systems provide faster data access by spreading file data across each of it's data servers. The concept is much like the one used in RAID at the storage device level. Similar to RAID, the striping configuration must match the data access for it to provide optimal performance.

Concurrent file access in parallel file systems

When a parallel application opens a file, the data that is read is divided between each process in the parallel program. The results produced by each process need to be written back to secondary storage. Below are two ways in which the data access can be set up for a parallel application [8]:

1. In this scenario, only one process is responsible for I/O. It must read and distribute the data to the other processes. It must then gather and write the data back to storage.

2. In this scenario, each process writes its data to a separate file. This scenario attempts to avoid data shuffling between the processes and hopes to achieve an I/O speedup. Unfortunately, the data often need to be merged through a time consuming post-processing step.

Option one is still sequential I/O. Option 2 does parallelize the reading and writing of the data but it introduces pre- and post-processing steps. The ideal solution is a parallel I/O access model which provides the parallel program an API

to access the same file in parallel without the extra work required in the options above.

Predefined and dynamic access In the parallel I/O access model, there are two ways in which the data access can be coordinated: a predefined way based on the number of processes, and a dynamic way implicitly decided while the application is running. In predefined access, each process is assigned a portion of the file which it can access. In dynamic access, it is decided during run time which process will access the next element in the file. Coordinating this type of file access requires additional overhead communication.

Individual and shared file pointers A parallel filesystem can allow an application to write to a file by allowing each of the processes to keep an individual file pointer to the open file. Each process updates it's individual file pointer but does not have any knowledge of the state of the other processes' file pointers. Individual file pointers are best suited for predefined file access.

In order to facilitate dynamic access to a file, a file system can provide a shared file pointer. A shared file pointer is a file pointer to the open file that is shared between the processes that opened the file. Keeping the shared file pointer updated on each process can incur a lot of communication overhead and is therefore not easy to implement efficiently. As a result, file systems often do not provide shared file pointer functionality.

Collective I/O

Collective I/O [8] [14] is an optimization that has proven to greatly increase performance. A collective I/O request is made by all processes at the same time. Collective I/O allows the I/O system to optimize by combining the access requests of many processes simultaneously.

2.4.3 I/O library and application level

A file system provides an API (application programming interface) in order to do routine I/O as well as to access the special features provided by the file system. Without a general I/O library an application programmer would need to write specialized code to support different file systems. A general I/O library can act as an intermediate layer that hides the details of individual file systems and presents the application with a common way to access files.

Operating systems provide a common I/O API for sequential I/O, but a common I/O library that provides parallel I/O semantics still does not exist. Attempts have been made to come up with a standard API for parallel I/O, but none have been adopted as the standard [14].

One benefit of having a common parallel I/O library is that the application logic can stay the same for whatever I/O configuration the system may have. The idea is that the I/O library will be able to receive I/O requests from the processes in a parallel application and then coordinate the access requests. For example, if the underlying file system is not parallel, the I/O library can still give the application the illusion that multiple processes are writing concurrently to the same file, but

in actuality it would collect all of the I/O requests and use only one process to do the actual I/O access. Of course, the performance will not be the same as having an underlying parallel file system, but at least the application logic would not have to change. Another benefit of having a generic parallel I/O library is that it can support features not directly enabled in the underlying file system.

Hints

Additional information about the parallel application's access patterns can aid the I/O library in making optimizations in order to improve performance. Many optimizations, such as collective I/O, require the programmer to change the code in order to give the I/O library an opportunity to make optimizations. Other optimizations do not require the code to change but require the programmer to tell the I/O library of the intended access or details of the I/O system configuration.

2.5 MPI and MPI-I/O

The MPI-I/O library is part of MPI. MPI is a message passing interface that provides a distributed memory programming model for parallel computation. MPI-I/O was added when it became obvious that there was also a need for parallel I/O [14] [8].

This thesis focuses on the MPI-I/O library because it is the most widely used and because it provides a solid environment for implementing the shared file pointer algorithms for this thesis.

This section begins by giving a brief overview of the components of MPI that are part of or heavily influenced the design of MPI-I/O.

2.5.1 MPI communication

Process/Communication group

A process/communication group in MPI is a group of processes that are grouped together with the intention of communicating with each other. Each process in the group is identified by it's rank. The main process is given rank 0 and the rest of the processes are given ranks from *1* to *group_size-1*.

Point-to-point communication

Once a communication group is formed, any two processes in the group can send messages to each other by using point-to-point communication.

Collective communication

Collective communication allows all of the processes in a group to participate in the communication operation. For example, a broadcast operation allows one process to send a message to all of the other processes in the group. A scatter operation allows one process to distribute data evenly among the other processes in the group. A gather operation allows data from all processes to be collected at one process.

2.5.2 MPI derived datatype

Datatypes provide a higher level view of the data in memory. The default datatype in MPI is a MPI_BYTE. Basic MPI datatypes also include MPI_INT, MPI_DOUBLE, MPI_CHAR.

More complex datatypes, similar to structs in C, can also be defined. These custom datatypes are called derived datatypes. For example, an element in memory can be defined by a derived datatype that is composed of a CHAR and a DOUBLE. When the application requests the next element in the data buffer, MPI will automatically read the correct number of bytes based on the derived datatype definition.

2.5.3 MPI-I/O communication groups and file view

Process/Communication group

MPI-I/O uses process groups to define which processes will access to the same file. When a file is opened in MPI-I/O, a communication group must be provided. Each open file knows which group it is associated with.

File view

A file view [8] extends the MPI derived datatype. It defines how the data elements are laid out in the file. The file view also designates which part of the file can be accessed.

It is possible for each process to have a different file view. For example, the file view for one process may allow access to the first block of the file and the file view of the second process may allow access to the second block of the file. File views do not have to be specify contiguous data segments. For example, the file view may define a strided data element layout in which a process is given access to every nth element of data. If a process does not specify a file view, MPI-I/O uses the default file view which is simply a long array of bytes, and each process gets access to the

entire file.

By defining the layout of the file, the application gives the MPI-I/O library a hint on how to best optimize data access requests. For example, the MPI-I/O library could implement proper prefetching and buffering that matches the needs of the application.

2.5.4 MPI-I/O file access functions

File pointers

MPI-I/O defines individual file pointer, shared file pointer, and explicit offset operations. The file pointer location references the elements in the file view. Reading the first element of the file returns a block of data the size of the derived datatype defined in the fileview.

If the portion of the file that each process will access is known ahead of time, a different file view can be provided for each process. The logic for reading the file is greatly simplified since each process can simply read through all of the elements and not worry about accessing data belonging to another process.

If the predefined portion of the file that the processes will access is not known, the processes can all use the same file view. In this case the application has the option of using a shared file pointer or to implement additional logic to determine which part of the file each process should access.

Non-collective (individual) vs collective requests

Non-collective (individual) I/O requests allow the processes to independently access the file.

A collective I/O request requires that all of the processes make the request at the same time. Collective I/O, is possible because of the communication group that has been associated with the open file. The collective file request allows the I/O library to further optimize and speed up file access by combining I/O requests.

MPI-I/O defines collective file operations with all three of the file pointer types. Modifying the application to do collective I/O is sometimes as simple as changing the file operation to the corresponding collective one.

Blocking and non-blocking operations

Another optimization that MPI-I/O provides is non-blocking file operations. A normal file operation will block and make the application wait until the I/O request is complete. A non-blocking operation allows the application to make the file request and then immediately continue with it's computation. The only restriction is that the data buffer can not be reused until the I/O operation completes. MPI-I/O provides a mechanism for the application to check the status of the non-blocking I/O request.

2.5.5 MPI-I/O hints

Optimization hints are passed to the application from the command line or as an MPI_Info object when the file is opened. The MPI-I/O specification defines a list of

standard hints. The MPI-I/O standard does not require an implementation to accept these hints. Additionally, MPI-I/O implementations can define implementation specific hints.

An implementation of MPI-I/O can also automatically set optimization parameters if it knows how to detect different system configurations and set the corresponding optimization parameters.

2.6 MPI-I/O shared file pointers

MPI-I/O [17] creates a shared file pointer for each file open call from a process group. A call to one of the shared file pointer I/O routines updates the shared file pointer. Multiple shared file pointer I/O requests are serialized. In order for the file pointer to be valid for all processes, MPI-I/O requires that all processes have the same file view.

Non-collective I/O calls issued at the same time are ordered in a non-deterministic way. Collective calls are ordered by rank id.

It is not easy to implement shared file pointers efficiently due to the communication that must occur between the processes in order to keep the shared file pointer updated on each process. As a result, existing MPI-I/O implementations have given little attention to shared file pointers.

There is clearly a need for shared file pointer functionality to be available to parallel applications. Two specific usage scenarios are:

1. Worker processes read new data items or tasks from a file as they are ready to process them. Each process works at its own speed. The locations where each

process reads could not have been precomputed. The file pointer needs to be shared between the processes as it changes.

2. If a group of processes need to write ordered messages to a common log file, they would need to know the position of the shared file pointer. In this case the application could not have precomputed where each process should write to.

The most widely used MPI-I/O implementation is ROMIO. The ROMIO implementation supports shared file pointer operations by using a hidden file to store the current file pointer value. A process must read and update the hidden file during a shared file pointer I/O request. In order to synchronize access to the shared file pointer each process must first lock the file containing the shared file pointer. Unfortunately, file locking and consequently ROMIO's shared file pointer functionality is not supported on all file systems [18].

In [3], Latham et.al. present an approach that uses one-sided communication operations, such as MPI_Win_lock and MPI_Win_unlock. If the shared file pointer is not in use, it can be accessed from the root process. Otherwise the process must wait for a message from the process that is currently using the shared file pointer. This approach has not yet been implemented in ROMIO due to needed functionality that has not yet been implemented in ROMIO.

In [19], Cole et.al. propose to integrate shared file support into an I/O forwarding middle layer that would have access to all of the I/O requests from all of the processes. This approach would require the I/O system to have I/O forwarding middle-ware, such as the I/O Forwarding Scalability Layer (IOFSL) [20], in place in order to work.

Chapter 3

Approach

This thesis explores different algorithms that can be used in order for shared file pointers to work efficiently. I/O is a complex problem because different solutions work best in different configurations. Therefore, the goal is not to find one shared file pointer algorithm that will perform the best. Instead, the hope is to develop a group of algorithms that will work best in different situations. Selection logic will also be developed in order to automatically select the most appropriate algorithm.

This section presents a high level description of each algorithm. Four different algorithms are explored:

1. Locked file algorithm
2. Shared memory algorithm
3. Individual file algorithm
4. Additional process algorithm

3.1 Locked file algorithm

The general idea behind the locked file algorithm is for the shared file pointer offset to be kept in a separate file that can be accessed by all of the processes in the process group that opened the file [21]. In order to control access to the file pointer, this algorithm depends on the file systems ability to lock the file. A process that intends to perform a shared file pointer I/O operation must first acquire a file system lock on the file. The process can then read the value of the shared file pointer offset and update it based on the I/O operation being executed. Finally, the process releases the file system lock so that other processes can perform shared file pointer operations.

The biggest problem with implementing shared file pointers is that the shared file pointer needs to be kept in sync across all processes. In order to do this, some form of communication between the processes needs to take place. The overhead of this communication is what results in poor I/O performance for shared file pointer operations. One benefit of the locked file algorithm on a local file system is that it reduces the amount of communication between the processes while keeping the shared file pointer consistent on all of the processes.

It is important for this algorithm to correctly determine if the file system supports file locking. If the file systems file locking functionality is faulty or non-existent then this algorithm can not guarantee strict ordering of the data. Additionally, it can not guarantee that processes will not write over data written by other processes.

It has already been mentioned that ROMIO [18] uses file locking in order to implement shared file pointer operations and that it does not work on all file systems. Despite this issue, we have decided to also implement a shared file pointer module

which uses file locks. It is possible for the locked file algorithm to provide the best performance in some cases.

3.2 Shared memory algorithm

The shared memory algorithm is similar to the locked file algorithm in the way that it manages the shared file pointer. The shared file pointer offset is kept in a shared memory segment instead of a hidden locked file. The shared memory segment can be accessed by all of the processes in the process group. In order to control access to the file pointer, this algorithm depends on the system's semaphore implementation. A process that intends to perform a shared file pointer I/O operation must first acquire the semaphore. The process then reads the value of the shared file pointer offset and updates it based on the I/O operation being executed. Finally the process releases the semaphore lock so that other processes can perform shared file pointer operations.

In order for this algorithm to work, all of the processes must be on the same node. Requiring all of the processes to run on the same node is a restriction, but it is also an advantage. It is expected that access to the shared memory segment will be faster than using a locked file. This is an example of how identifying a special case scenario may provide better performance.

Since the general management of the shared file pointer is similar to the locked file algorithm, some of the advantages and disadvantages also apply. The shared memory algorithm also reduces the communication that needs to happen between the process group in order to keep the file pointer synchronized. But reduced performance is

also expected as the number of processes increases due to the sequential access that is imposed when updating the file pointer.

3.3 Individual file algorithm

The individual file algorithm [4] attempts to reduce the amount of communication between the processes by not updating a shared file pointer. Instead, all write requests are delayed. Write requests along with a timestamp are first written to an individual file kept by each processes. The timestamp allows the I/O library to correctly order the write requests when the data is finally merged.

The idea in the individual file algorithm is to wait until a collective I/O operation is called. The data cannot be merged into the main file unless all of the data from each process is collected. This is necessary because the data can not be properly ordered without having knowledge of all of the pending write operations.

The individual file can be written to local storage or to the targe file system. For this algorithm we have chosen to write the individual file to the target storage. In this manner, the individual files are available for completing the data merge at a later time in a post-processing step.

One issue with this approach is that the algorithm cannot guarantee consistent ordering if the clock is not properly synchronized across the processes in the group. Ideally, effort will be taken by the system administrator to synchronize the time. The user must be willing to accept this possibility, since this algorithm does not have a way to ensure a synchronized clock.

The bigger and more important restriction with this algorithm is that it does

not support read operations. This restriction is due to the fact that the algorithm does not attempt to keep the file data consistent between collective I/O calls. If one process performs a non-collective write to the file, the data is essentially buffered in the individual files. As a result, any other process performing a read operation on the same file location will not be aware of the changes. In order for this algorithm to be selected, the file must be opened in write only mode.

3.4 Additional process algorithm

The additional process algorithm [4] spawns another process whenever a file is opened. The additional process acts like a server which listens for requests to retrieve and update the shared file pointer.

The requests to update the shared file pointer are processed sequentially. This ensures that all shared file pointer operations are synchronized.

A process that intends to perform a shared file pointer I/O operation must first send a message to the additional process and then wait for a response. The additional process reads the value of the shared file pointer offset and updates it based on the I/O operation being executed. It then sends a message with the updated shared file pointer value to the calling process. The additional process can then process the next request.

The additional process may get congested as the number of shared file pointer operations go up. But, the benefit of this algorithm is that it can support both read and write shared file pointer operations on any file system.

Chapter 4

Implementation

The shared file pointer algorithms were implemented in a framework that integrates with OMPIO [5]. OMPIO is an MPI-I/O implementation developed by the University of Houston. OMPIO itself is implemented in Open MPI [6] [22], an open source MPI implementation. Below is a quick overview of Open MPI, OMPIO, and the shared file pointer framework developed in this thesis. Details of the implementation of each of the algorithms are also included.

4.1 Open MPI

Open MPI is an open source implementation of the MPI-2 [17] specification that takes a modular approach to implementing MPI functionality. The modular approach allows different implementations of the same functionality, known as components, to be easily used in the same system. For example, when an MPI application is run, Open MPI can load and use a specialized network communication component without

needing to recompile Open MPI. The specialized network component is loaded during runtime instead of the default one.

The following terms from the Open MPI FAQ [6] explain important parts of the Open MPI architecture.

- **MCA:** The Modular Component Architecture (MCA) is the foundation upon which the entire Open MPI project is built. It provides all the component architecture services that the rest of the system use.

- **Framework:** An MCA framework is a construct that provides a common public interface for a group of components. An MCA framework uses the MCA's services to find and load components at run time.

- **Component:** An MCA component is an implementation of a framework's interface.

- **Module:** An MCA module is an instance of a component An MCA component is analogous to a C++ class and an MCA module is analogous to an ïnstanceöf that class.

There are different ways that components can be selected and loaded. The primary way is through the selection logic system implemented by the MCA framework. Each component is queried for a priority value. The queried component can run different tests to determine how well it can run in the current environment. The component with the highest priority value is selected. The priority values need to be carefully chosen by the developers to ensure that values make sense within the context of the group of components being queried. Another way to select a component

is by setting an environment value which tells the system which component to load for a specified framework.

4.2 OMPIO

Open MPI's modular approach makes it possible to have more than one MPI-I/O implementation within the same Open MPI installation. While this is a great step in the right direction, it is still not enough. I/O itself is very complex and can benefit from having the ability to modularize it's own components.

An Open MPI-I/O component would need to be modified and recompiled in order to support more than one file system. It would need to be modified and recompiled again in order to support a different communication protocol. In order to support all combinations of the two file systems and the two communication protocols, four different versions of the I/O component would need to be installed.

ROMIO, uses an abstracted file system interface, ADIO [23], to easily support files on different file systems. Nonetheless, it does not account for the need for other components to be easily switched out.

OMPIO [5] has sub-components which can be selected and loaded during runtime. In order to accomplish this, OMPIO reuses Open MPI's framework architecture. The following I/O sub-component categories have been identified and implemented as MCA frameworks [5]:

1. fs - The file system framework

2. fbtl - The file byte-transfer layer framework

3. fcoll - The collective I/O framework

4. sharedfp - The shared file pointer framework

The OMPIO sub-frameworks reside in the same directory as the Open MPI I/O framework. OMPIO sub-frameworks adhere to the Open MPI's MCA framework guidelines.

4.3 OMPIO sharedfp framework

The sharedfp framework is implemented as one of OMPIO's sub-frameworks. When a file is opened, Open MPI queries the available I/O components. If OMPIO is selected, Open MPI uses OMPIO's implementation of MPI_File_open. In OMPIO's file open, it initializes the set of OMPIO sub-frameworks that it uses. Among these is the sharedfp framework.

Each of the shared file pointer algorithms are implemented as components under the sharedfp framework. During initialization the sharedfp framework attempts to query each of the installed sharedfp components for their priority number. The component with the highest priority is selected.

4.3.1 Functions in sharedfp framework

OMPIO allows the sharedfp components to independently implement the shared file pointer read and write operations. As a result, the corresponding shared file pointer read and write operations in OMPIO are essentially wrapper functions around the functions implemented in the sharedfp components. When a user calls a shared file

pointer operation such as MPI_File_write_shared, Open MPI forwards the call to the OMPIO module. The OMPIO module then forwards the call to the selected sharedfp module.

In order to implement a new sharedfp component the write operations listed in table 4.2, the read operations listed in table 4.3, and the general operations listed in table 4.1 must be defined. Even though the functions must be defined, components are not required to have a working implementation for all of the functions. For example, the individual file algorithm does not support read operations. The individual file component's implementations of the read operations return an error.

The shared file pointer components implemented in this thesis all make use of the OMPIO read_at and write_at functions. The current version of OMPIO does not support non-blocking read and write operations. As a result, the sharedfp components implemented in this thesis also do not support non-blocking shared file pointer read and write operations. The non-blocking functions have been added to the sharedfp framework to make it easier to enable this functionality when OMPIO implements non-blocking read_at and write_at functions.

Table 4.1: sharedfp framework: General Operations

file_open	This sharedfp function is called by OMPIO when a file is opened. It allows the sharedfp component to initialize itself. File open is a collective operation, therefore this is the time to perform any operations that need to be synchronized between all of the processes. For example, the shared file pointer value is initialized in this function.
file_close	This function is called by OMPIO when the file is closed. This function allows the sharedfp component to clean up after itself and to do some last minute processing before the file is actually closed.
file_seek	This function is called by OMPIO when MPI_File_seek_shared is called. A shared file pointer seek operation updates the shared file pointer according to the parameters offset and whence. Whence can have the following possible values: • MPI_SEEK_SET: the pointer is set to offset • MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset • MPI_SEEK_END: the pointer is set to the end of file plus offset Seek shared is collective. As a result, all the processes in the communicator group associated with the open file must call MPI_FILE_seek_shared with the same values for offset and whence.
file_get_position	Returns he current position of the shared file pointer.

Table 4.2: sharedfp framework: Write Operations

file_write_shared	non-collective blocking shared file pointer write operation
file_iwrite_shared	non-collective non-blocking shared file pointer write operation
file_write_ordered	collective blocking shared file pointer write operation
file_write_ordered_begin	function that initializes the collective non-blocking shared file pointer write operation
file_write_ordered_end	function that finalizes the collective non-blocking shared file pointer read operation

Table 4.3: sharedfp framework: Read Operations

file_read_shared	non-collective blocking shared file pointer write operation
file_iread_shared	non-collective non-blocking shared file pointer write operation
file_read_ordered	collective blocking shared file pointer read operation
file_read_ordered_begin	function that initializes the collective non-blocking shared file pointer read operation
file_read_ordered_end	function that finalizes the collective non-blocking shared file pointer read operation

4.4 Locked file component

4.4.1 Locking the file

Several kinds of file-locking mechanisms [24] are available in Unix-like [25] systems. The two most common mechanisms are fcntl and flock. File locks will not work properly if the target filesystem does not support them.

The locked file algorithm uses the fcntl implementation to lock the file containing the shared file pointer value.

4.4.2 General steps

Below are the general steps executed while running the locked file algorithm.

1. All processes in a process group collectively open a data file with the MPI_File_open operation. An additional file, the lock file, is opened in the same directory as the data file and is shared by all of the processes. The lock file is named the same as the data file but has ".locked" appended to the end.

2. The root process initializes and stores the shared file pointer in the lock file.

3. During a non-collective shared read or write operation:

    1. The process performing the I/O operation, first attempts to lock the hidden file.

    2. The process waits until the filesystem grants the file lock.

    3. When the lock is granted, the process reads the current value of the shared file pointer and increases the value by the amount of data it needs to

read or write.

4. The process releases the file lock.

5. The process reads or writes data in the data file at the offset it retrieved from the lock file. The OMPIO non-collective explicit offset I/O operations are used.

4. During a collective write operation the root process sums the total amount of data that all of the processes will write. The root process updates the shared file pointer in the lock file. The processes collectively write the data using the OMPIO collective explicit offset I/O operations.

5. During a file close operation, the lock file is closed and removed.

4.4.3 Selection logic implementation

To determine whether or not the locked file algorithm can run, a test file is created by each process. Each process attempts to lock the file. If the test fails this component reports that it can not run. Otherwise it returns a medium priority value.

This test can be potentially time consuming and taxing on the system if a parallel application is run on hundreds of nodes. For the purposes of this thesis this is not an issue. In order to run this component in a production environment we plan to revisit this issue in order to find a more efficient test.

This is a simple test and has at least two limitations:

1. Some implementations of NFS are known to return success for setting a file lock when in fact no lock has been set. This test will not detect such erroneous

implementations of NFS.

2. Some implementations will wait indefinitely within the fcntl call and not return. This test will also hang in that case. Under normal conditions, this test should only take a few seconds to run.

4.4.4 Component interface implementation

The locked file algorithm implements the sharedfp framework's write operations as well as the read operations. The read operations access and update the shared file pointer through the locked file mechanism. After the shared file pointer has been updated the processes then proceed to read from the file using OMPIO's explicit offset read operations.

4.5 Shared memory component

4.5.1 Shared memory segment

The shared memory algorithm needs to share a segment of memory between a group of processes that are running on the same node. To accomplish this, it makes use of the shared memory functionality implemented by the operating system. This shared memory can make one copy of data available to groups of related or unrelated processes.

Several kinds of shared memory mechanisms are available in Unix/Linux. The two most common mechanisms are shmget [26] and mmap [27]. Open MPI uses mmap to establish shared memory [6]. The Open MPI website states that in future

releases there may be an option to use other types of shared memory. To avoid any possible issues with shmget and to avoid any conflict with the current implementation of Open MPI we also use mmap to implement the shared memory algorithm for this thesis.

Managing access to the shared file pointer

Semaphores and mutexes are normally used to control access to shared memory resources. They are created and managed in the operating system. The semaphore or mutex itself does not protect the shared resource. In order for the shared resource to be protected all of the threads or processes that access the shared resource must use the semaphore/mutex's locking mechanism.

Setting up the shared memory segment

To set up the shared memory segment and the semaphore we followed the steps proposed in the book UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications [27].

A data structure, composed of an offset and a semaphore, is allocated and initialized in the shared memory segment. The following steps are taken to setup the shared memory segment:

1. Every process in the MPI process group opens the same file.

2. Every process maps the file to memory in order to have a pointer to the shared memory location.

3. The root process initializes the data structure in the shared memory segment

4. The root process initializes the POSIX memory-based semaphore to 1 (unlocked).

4.5.2 General algorithm steps

Below are the general steps executed while running the shared memory algorithm.

1. A shared memory segment is allocated and shared by all of the processes during the MPI_File_open operation.

2. The root process initializes and stores the shared file pointer in the shared memory segment.

3. During any individual shared file pointer operation:

   1. The processing performing the shared file operation first attempts to lock the semaphore which is stored in the shared memory segment.

   2. The process waits until it acquires the semaphore.

   3. The process retrieves the current value of the shared file pointer and increases the value by the amount of data it needs to read or write.

   4. The process releases the semaphore lock.

   5. The process writes data to the data file at the offset it retrieved using the non-collective MPI_File_write_at or MPI_File_iwrite_at I/O operations.

4. During a collective shared file pointer operation the root process sums the total amount of data that all of the processes will access. The root process updates the shared file pointer. The root process calculates the offset at which other

processes should access the file and scatters the calculated offsets to them. For write operations, the processes collectively write the data using the OMPIO collective MPI_File_write_at_all or MPI_File_write_at_all_begin I/O operations.

5. During a file close operation, the shared memory segment is released.

4.5.3 Selection logic implementation

Each process uses the Open MPI process list utility functions to ensure that all of the processes in the process group are on the same node. It is anticipated that this algorithm will perform the best, since shared memory is known for providing the fastest inter-processes communication [26]. As a result, if this component can run, it returns a very high priority during selection.

4.5.4 Component interface implementation

The shared memory component also implements the shared file pointer read operations. The read and write operations in the shared memory component behave similar to the read and write operations in the locked file component.

4.6 Individual file component

4.6.1 The metadata record

The individual file algorithm uses an individual data file per process to delay writing to the main file. Each process also keeps a metadata record for each write request in order to store the timestamp, the position in the individual file, and the length of each request. A full description of the metadata record is shown in table 4.4.

When a file is opened, each process opens a metadata file and an individual data file. New metadata records are first stored in the main memory of each process using a linked list. Once a limit to the number of records in the linked list is reached, the linked list is written into the metadata file. Therefore, in order to read the full list of metadata records for a process, the records in memory must be combined with the records in the metadata file.

4.6.2 General algorithm steps

Below are the general steps executed while running the individual file algorithm.

1. During the MPI_File_open operation each process opens an individual data file and a metadata file.

Table 4.4: Individual File : Metadata Record

Record Id	Id indicating the write operation of this record
Timestamp	Time at which data is being written to the data file
Local Position	Offset in the data file
Record Length	Number of bytes written to the data file

2. During any individual write operation:

    1. Each process writes the data into its individual data file instead of the main file using the OMPIO MPI_File_write_at or MPI_File_iwrite_at I/O operations.

    2. A metadata record is created and appended to the metadata record list in the main memory. If the metadata record list size has reached it's limit, it is flushed to the metadata file.

3. During a collective write operation, the data in the individual data files is first ordered and merged into the main file using the non-collective explicit offset OMPIO write operations. Each process then writes the current write request directly to the main file using the OMPIO explicit offset collective write routines.

4. During a file close operation, which is a collective operation, any independent data that has not yet been merged into the main file is ordered and merged into the main file.

4.6.3 Merging

The merge step can only be performed during MPI-I/O's collective shared file pointer operations because merging needs to know about all of the pending write requests before it can properly order them. The collective shared file pointer operations are MPI_File_write_ordered, MPI_File_write_ordered_begin and MPI_File_close.

Below are the general steps executed in order to merge the data into the main

file.

1. **Compute total number of the metadata records:** Each process generates a list of the unprocessed metadata records. A pointer is kept that indicates the last position to which the metadata file was read. Thus, during each merging operation the metadata file is not processed again from the beginning. If unprocessed metadata records exist in the metadata file, they are processed first followed by the records stored in memory.

   The processes use an MPI_Allgather operation to combine the metadata record count from all processes in order to compute the total number of metadata records. This count is needed in the next step in order to allocate a buffer of the correct size.

2. **Gather timestamps and record lengths:** Each process collects the timestamps and record lengths corresponding to all unprocessed metadata records from the other processes using an MPI Allgatherv operation.

3. **Sorting of the metadata nodes:** Once each process receives all the timestamps and record lengths, it sorts them based on the timestamps in an ascending order.

4. **Assign global offsets:** Each process assigns a global offset to the sorted list of metadata records. The global offset corresponds to the position at which the data needs to be written in the main file.

5. **Write to the main file:** Once each process has global offsets assigned to each metadata record, it processes the metadata records corresponding to the data

in it's local data file. It reads data from the local data file using the local file offset. It then writes the data into the main file using the computed global offset.

4.6.4 Selection logic implementation

This algorithm needs to ensure that the shared file pointer will not be used for read operations. When an MPI file is opened the user must indicate that the file will be write only be setting the write-only access mode flag. This implementation checks the access mode flags to make sure that the file was opened write-only. Read operations performed while in this mode return an error.

4.6.5 Component interface implementation

The read operations defined in the sharedfp framework were not implemented since this algorithm does not support read operations. The seek and get position functions are also not supported. The non-blocking write operations were implemented but do not currently work since they make use of the non-blocking write operations that have not yet been implemented in OMPIO.

4.7 Additional process component

4.7.1 The additional process

The additional process needed to manage the shared file pointer value is created by using the dynamic process management feature provided in the MPI-2 specification. It is spawned during the file open operation by using MPI_Comm_spawn. MPI_Comm_spawn returns a new inter-communicator that contains the already existing processes and the newly spawned process. This new inter-communicator is used to communicate with the additional process.

The additional process can process six types of requests.

1. **Request offset:** A calling process requests the shared file pointer value before performing an I/O operation. The request also contains the number of elements which the calling process intends to read or write. The additional process returns the current shared file pointer value. It also updates the shared file pointer by the offset value provided in the request.

2. **End additional process:** A calling process, usually the root process, indicates to the additional process that the file is being closed. The additional process stops maintaining the shared file pointer for this file and terminates.

3. **Seek End:** A calling process requests that the value of the shared file pointer be set to the end of the file plus the offset value provided in the request.

4. **Seek Current:** A calling process requests that the value of the shared file pointer be set to the current value of the file pointer plus the offset value

provided in the request.

5. **Seek Set:** A calling process requests that the value of the shared file pointer be set to the value provided in the request.

6. **Get current position:** A calling process requests the current shared file pointer value. The additional process returns the value to the calling process.

4.7.2 General algorithm steps

Below are the general steps executed while running the additional process algorithm.

1. A new process is spawned by all of the processes during the MPI_File_open operation.

2. The new process initializes and stores the shared file pointer in memory.

3. During any individual shared file pointer operation:

   1. The process performing the shared file operation sends a message to the additional process with the number of elements that it intends to read or write.

   2. The additional process returns the current shared file pointer value and increases the current value by the offset indicated by the calling process.

   3. The calling process writes data to the data file at the offset it retrieved. It uses the corresponding explicit offset non-collective MPI_File_write_at, MPI_File_iwrite_at, MPI_File_read_at, or MPI_File_iread_at I/O operation.

4. During a collective shared file pointer operation the root process sums the total amount of data that all of the processes will read or write. The root process

retrieves and updates the shared file pointer by sending a request to the additional process. The root process calculates the offset at which other processes should access the file and scatters the calculated offsets to them. For write operations, the processes collectively write the data using the OMPIO explicit offset collective MPI_File_write_at_all or MPI_File_write_at_all_begin I/O operations. For read operations MPI_File_read_at_all or MPI_File_read_at_all_begin are used.

5. During the file close operation, the root process signals the additional process that the file is closed, and frees the inter-communicator. The additional process stop listening for further requests and terminates.

4.7.3 Selection logic implementation

Since this algorithm can be used with files on any file system, it always returns a medium priority value during selection. It is anticipated that other algorithms may perform better in certain cases.

4.7.4 Component interface implementation

The additional process component also implements the shared file pointer read operations. The read and write operations in the additional process component behave similar to the read and write operations in the shared memory and locked file components.

Chapter 5

Measurements and Evaluation

This chapter provides a description of the performance measurements that were run to evaluate the four shared file pointer algorithms.

The intention was to benchmark all four of the shared file pointer write operations and measure the attained write bandwidth. Since the underlying OMPIO implementation does not yet support non-blocking write operations, we were unable to test the two non-blocking write operations.

A read benchmark was not developed due to the time constraint on completing the thesis. Furthermore, reading access patterns tend to match the way that the data is written. Therefore the first task is to verify that shared file pointers can be used to efficiently write computed results to a file.

Whenever possible all four of the algorithms participated in each test. All tests were repeated at least 3 times. The minimum execution time is used in the performance plots.

One of the uses of the results is to determine which algorithm is best suited for

certain environments. The priority values given to each algorithm for the selection logic were derived from the relative performance of the algorithms in these tests. It is a difficult challenge to test all possible scenarios that affect I/O. Therefore these tests focus on only a few, but important components.

One important component of the tests is the type of file system used. These tests were run on a local file system, ext4, and a parallel file system, PVFS2. Another important component is weather the processes are running on a shared memory (one node) or distributed memory (multiple nodes) environment. Lastly, since the purpose of shared file pointers is to share a file pointer between a group of processes, these tests also try to focus on how the number of processes affects the performance of the shared file pointer algorithms.

5.1 Benchmark description

The write benchmark program writes data to a single file using one of the four shared file pointer write operations defined in MPI-2. The benchmark program takes five input parameters at run time.

Filename: Name of the output file to be written by the program,

Iteration Data Size: Data size (in bytes) to be written in one write iteration

Loops: Number of times the iteration data size is written.

Mode: Write operation to be used for writing the output file. Accepted values are:

1 - MPI File write shared

2 - MPI File write ordered

3 - MPI File iwrite shared

4 - MPI File write ordered begin

Variability factor: This factor introduces a variability in the amount of the data written during a single write operation by each process. For example, if the variability factor is set to 20%, and the total data to be written is specified as 1024 bytes, then each process would write the data in the range of 820 to 1228 of 1024 bytes during each write operation.

The usage command to run the benchmark program is:

```
mpirun -np 8 ./benchmark -f <filename > -d <iteration data size>
-i <loops> -m <file write mode> -v <variability>
```

The benchmark measures the time it takes to execute. In the plots showing the performance of the algorithms the time is converted to bandwidth in order to more easily compare the performance of different kinds of tests.

5.2 Test environment

The four algorithms have been evaluated on the crill cluster in the Parallel Software Technologies Lab at the University of Houston.

The crill cluster consists of 16 compute nodes and one front end node. Each node consists of four 2.2 GHz 12-core AMD Opteron processors (48 cores total) with 64GB of main memory.

The nodes are connected by a 4xInfiniBand network. The cluster has a parallel file system (PVFS2) mounted as /pvfs2, which utilizes 8 hard drives, and each hard drive is located on a separate compute node. The PVFS2 file system internally uses the Gigabit Infiniband network to communicate with the PVFS2-server. The metadata server of the PVFS2 file system runs on each node of the cluster. The home file system on crill is NFS. However, we do not perform any tests over NFS. Each node machine has a local disk storage as its main file system using the Ext4 file system.

5.3 Test Series 1 : Fixed number of processes

This test series measures the write performance of the algorithms when writing with a fixed number of processes. The tests in this series include writing to a parallel file system from a single node, writing to a parallel file system from multiple nodes, and writing to a local file system from a single node. The number of processes is fixed at 8 for all tests in this series.

The tests are run multiple times, each time writing different amounts of data. The total data is written in increments of 2MB. As a result, as the total amount of data written increases, the number of shared file pointer write operations also increases. When running the benchmark, the iteration_data_size is set to 2MB and the number of loops are computed to match the target total data output size.

The benchmark is run for each blocking write operation, each time increasing the amount of data written to the file. The output data sizes used are: 40000MB, 50000MB, 65000MB, 80000MB, 100000MB, and 128000MB.

Table 5.1: Test Series 1: Fixed number of processes, increasing data output

1	Fixed number of processes set to 8.
2	For each algorithm run each of the blocking write operations. 1-write_shared, 2-write_ordered
3	Fixed iteration_data_size per loop: 2MB
4	Run with each of the following total data output: 40000MB, 50000MB, 65000MB, 80000MB, 100000MB, 128000MB

5.3.1 Writing to PVFS2 from a single node

The first test writes to the PVFS2 parallel file system from a single node. The node used is a 48-core computer from the crill cluster.

The individual file algorithm and the shared memory algorithm are used in this test. The PVFS2 parallel file system does not support file locks, therefore the locked file algorithm and ROMIO are not able to run.

Results : write_shared - blocking, non-collective write

Figure 5.1 gives the performance of the blocking, non-collective write operation. Two lines are shown for the individual file algorithm. One for the performance before the merge step and another to show the performance after the merge step. The reason for this is that the merge operation can potentially be done in a post-processing step.

Since the shared memory (sm) and additional process (addproc) algorithms do not have to execute a merge step it was expected that they would always outperform the individual algorithm. But the individual algorithm's performance without the merge process is still slower than the sm algorithm's performance. It is possible that this is due to the individual file algorithm having to write many files instead

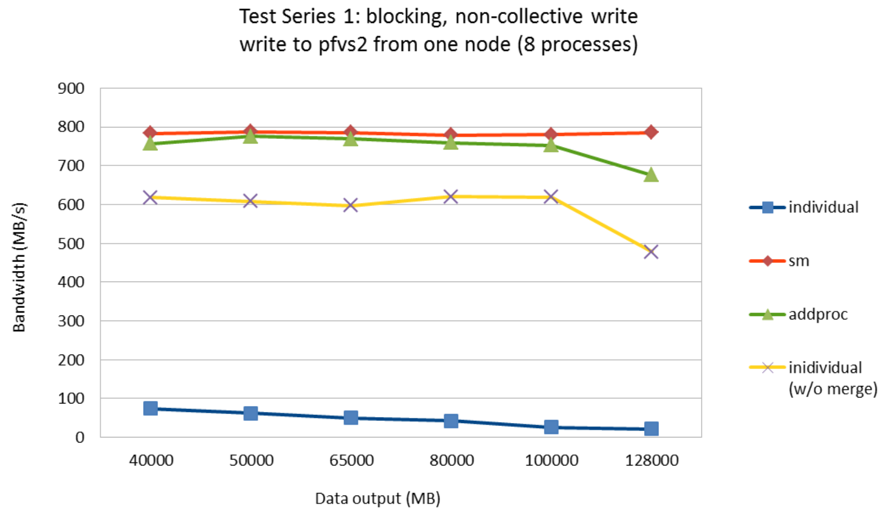

Figure 5.1: Test Series 1: On PVFS2 from single node - Comparison for blocking, non-collective write operation

of only one. The individual file algorithm's before merge performance also drops when writing 128G. It is not clear what is the cause for the drop in performance. One possibility is that multiple individual data files were mapped to the same data servers.

The sm algorithm shows steady performance. This was expected since the amount of data written during each loop is 2MB. The sm algorithm's primary execution steps are updating the shared file pointer and writing directly to the main file. The addproc's execution steps are similar to the sm algorithm. Since the addproc's performance is consistently slightly lower, it shows that the it is slower than the sm algorithm at updating the shared file pointer. The addproc algorithm's performance drops when writing 128G. This can be due to the additional process getting congested with shared file pointer update requests.

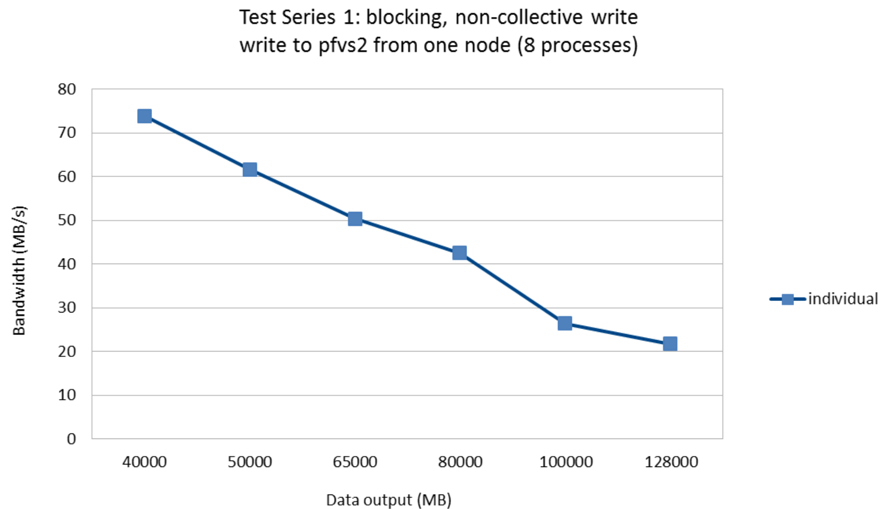

Figure 5.2: Test Series 1: On PVFS2 from single node - Comparison for blocking, non-collective write operation (close up)

Since the bandwidth attained by the shared memory algorithm is much higher than the individual file algorithm it is helpful to zoom in and create a chart, figure 5.2, that only includes the performance of the individual file algorithm. The individual file algorithm first writes the individual files. Then, during the merge step, it must read the those files and write the contents to the main file. The extra steps taken by the individual file algorithm explain why the individual file algorithm is much slower. This tests shows that performance of the merge step decreases as the output file size increases.

Results: write_ordered - blocking, collective write

Figure 5.3 gives the performance of the blocking, collective write operation. Note that in the collective write case, the individual file algorithm writes directly to the

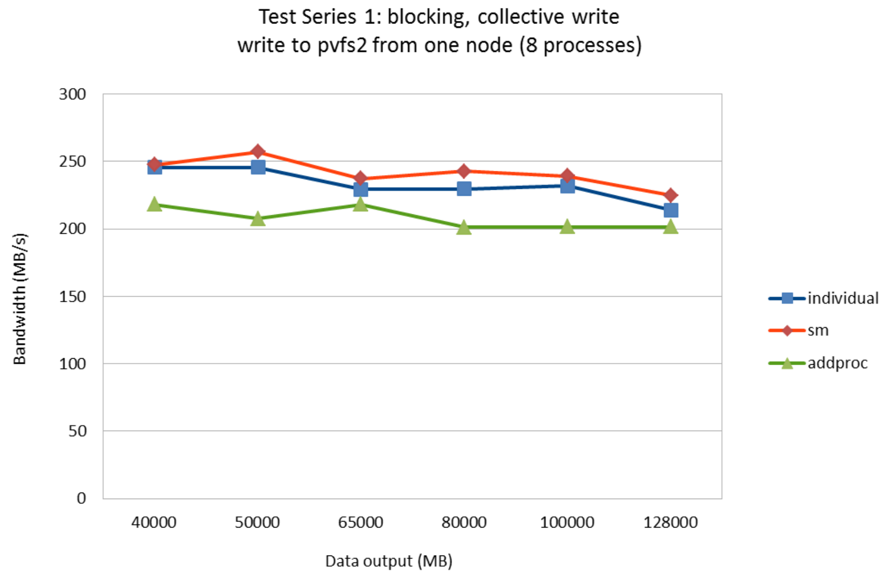

Figure 5.3: Test Series 1: On PVFS2 from single node - Comparison for blocking, collective write operation

main file instead of the individual files. As a result the blocking, collective write resulted in similar performance for the individual, sm, and addproc algorithms.

At first glance, it could have been expected that the individual algorithm would perform better than the sm algorithm in this collective write test. The individual file algorithm does not need to update a file pointer while the shared memory algorithm does. The reason for this is that the individual file algorithm needs to communicate with all of the processes to check if there is any data in the individual files that needs to be merged. Since this test only executes collective writes it never has to perform the merge operation, but the communication step to check is enough to make the algorithm slower.

The addproc algorithm was the slowest of the three algorithms. This tells us that

the shared memory approach is faster than the additional process in updating the shared file pointer.

The non-collective writes for the sm and addproc algorithms performed better than the collective writes. The benefits of collective I/O are not seen in this test. A collective write can get a performance boost by combining many discontiguous writes into one contiguous write. The nature of the shared file pointer access is already contiguous. The collective write may actually be wasting execution time since it still needs to go through the collective write's data shuffling process.

On further investigation, it was also discovered that the collective algorithm does not use all of the processes to write to the file. It chooses a set number of aggregators to collect the data and then write it to the file. In this scenario, the 2MB per loop cause the number of aggregators to default to only one.

5.3.2 Writing to PVFS2 from multiple nodes

This test uses a combination of 8 processes from 8 nodes to write to the PVFS2 parallel file system. The Open MPI command line parameter bynode is used in order to force mpirun to distribute the 8 processes in a round robin fashion among the 8 nodes. Without this option, all 8 processes are mapped to the same node.

Since this test is run from multiple nodes, only the individual file and addproc algorithms can run. The shared memory algorithm can not run because it requires all of the processes to be on the same node. The locked file algorithm and ROMIO are not able to run because the PVFS2 parallel file system does not support file locks.

Figure 5.4 gives the performance of the blocking, non-collective write operation.

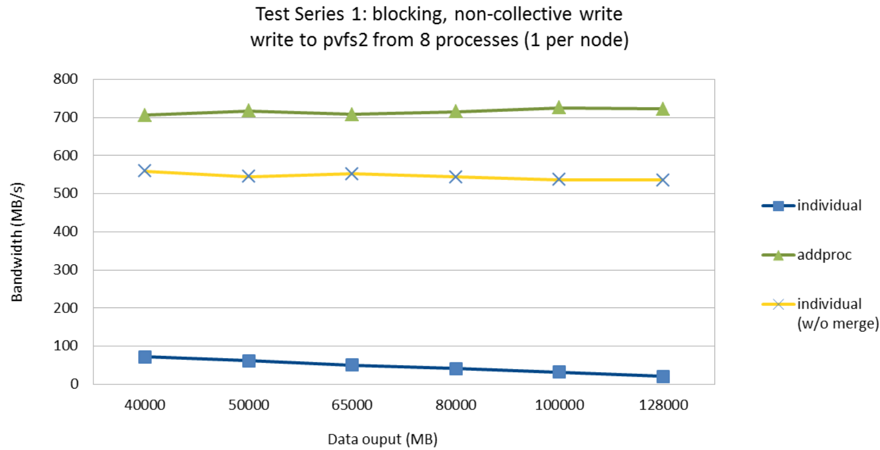

Figure 5.4: Test Series 1: On PVFS2 from multiple nodes - Comparison for blocking, non-collective write operation

The performance on 8 nodes was similar to the performance on the one node test. For the individual file before merge and the addproc performance there is actually a slight decrease in performance. It is not clear what caused this.

Figure 5.5 gives the performance of the blocking, collective write operation. Since the amount of data being written and the access pattern is the same as the single node test, better performance was not expected by having more nodes. Actually, the collective write, like the non-collective write, was also slower than the single node test. Open MPI tries to use shared memory communication when all of the processes are on the same node. In the multi-node case it uses the next available communication protocol. Slower communication between the processes might be the factor producing the slower performance for the algorithms.

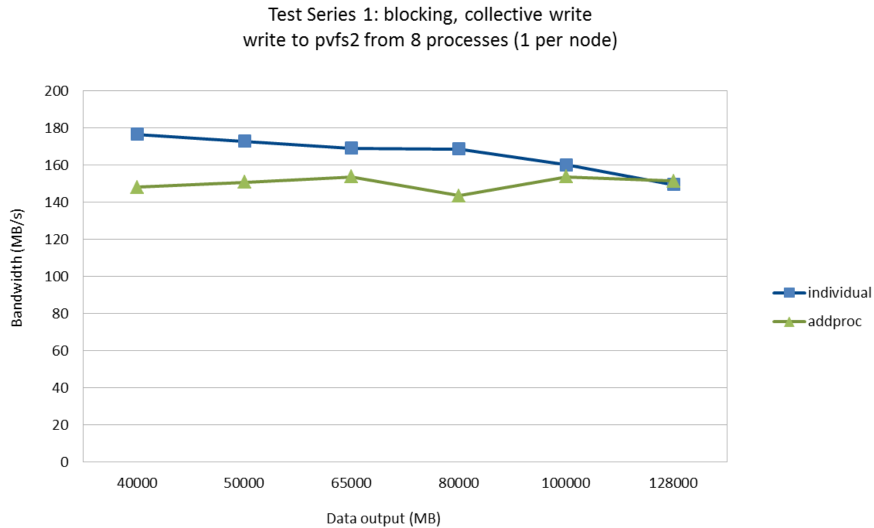

Figure 5.5: Test Series 1: On PVFS2 from multiple nodes - Comparison for blocking, collective write operation

5.3.3 Writing to ext4

This test measures the write performance of the algorithms when writing to a local ext4 file system. This test uses 8 processes from a 48-core computer from the crill cluster to write to the local ext4 file system.

The individual file algorithm was run in this test. Since this test is run on a single node, the sm algorithm is also able to run. The local file system ext4 supports locked files, therefore both this thesis' version of the locked file algorithm as well as ROMIO are also able to run.

The addproc algorithm was added nearing the completion of this thesis because it was thought it would enhance the research. As a result of the time constraint, the ext4 test was not run. Based on the results of the PVFS2 test, results comparable

to the sm algorithm are expected.

Results : write_shared - blocking, non-collective

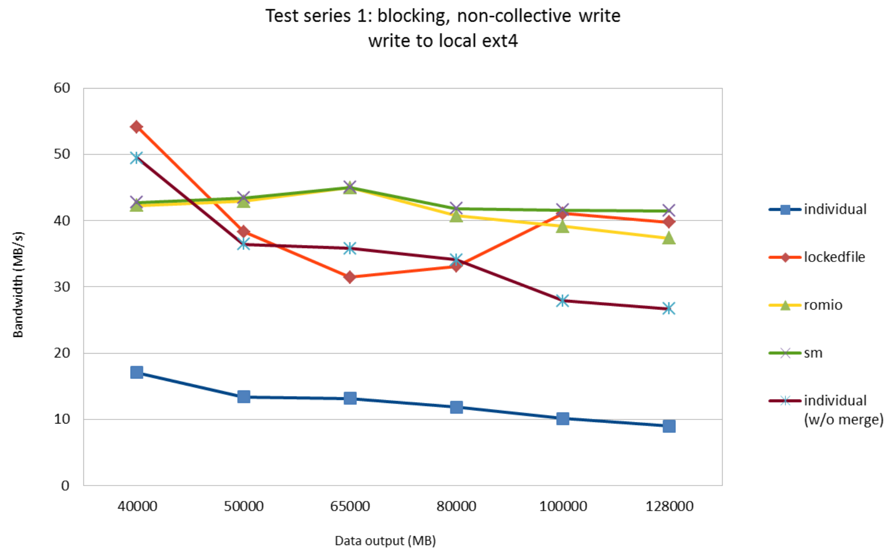

Figure 5.6: Test Series 1: On Ext4 - Comparison for blocking, non-collective write operation

Figure 5.6 gives the performance of the blocking, non-collective write operation. In this test, all of the algorithms, except the individual algorithm, produced comparable results. The sm algorithm and ROMIO produced more consistent results.

Since the only difference between the locked file and shared memory algorithm is in the way that the shared file pointer is updated, it indicates that the shared memory access mechanism is more stable than the implementation of the locked file algorithm. The more consistent ROMIO performance indicates that our implementation of the lockedfile algorithm has the potential to be optimized.

The individual file algorithm's before merge bandwidth dropped as the amount of output data increased. The drop may be due to writing to multiple individual data files and potentially the extra I/O requests to maintain the metadata file.

The individual file and sm algorithms show greatly reduced performance when compared to the test in which they wrote to the PVFS2 filesystem. This indicates the limiting factor is the speed of the local disk. It would be beneficial to setup a local raid storage for scenarios writing to the local file system.

Results: write_ordered - blocking, collective write

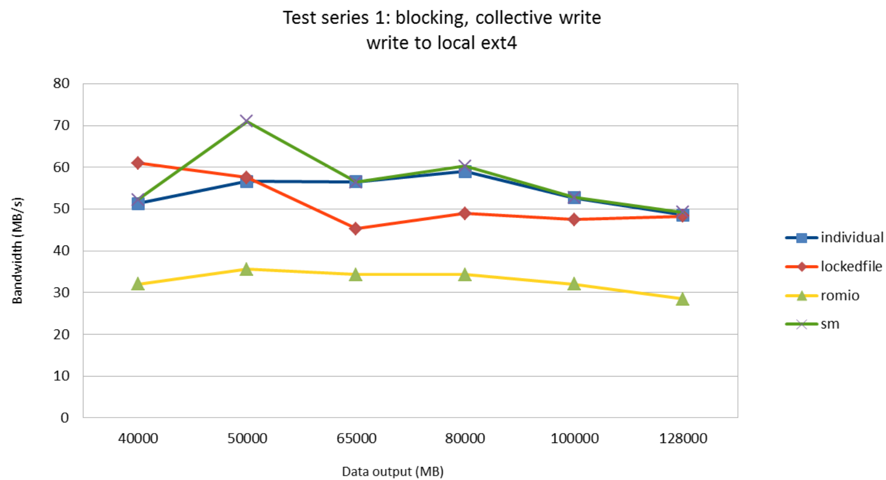

Figure 5.7: Test Series 1: On Ext4 - Comparison for blocking, collective write operation

Figure 5.7 gives the performance of the blocking, collective write operation. The performance of the collective write was uniformly a little faster for all of the algorithms implemented in this thesis. On the other hand, the ROMIO collective

write performed somewhat slower than in the non-collective write. The algorithms implemented for this thesis use OMPIO's two phase collective I/O sub-component. This indicates that OMPIO's two phase implementation is better optimized than ROMIO's default collective write implementation.

5.4 Test Series 2: Increasing number of processes

The tests in this series measure the performance as the number of processes are increased. The number of processes are increased from 2 to 32 (2,4,8,16,32). As the number of processes increase, the number of simultaneous shared file pointer updates also increases. The iteration_data_size is fixed at 2MB and the total data output is fixed at 65G.

The tests include writing to a parallel file system from a single node, writing to a parallel file system from multiple nodes, and writing to a local file system from a single node.

Table 5.2: Test Series 2: Increasing number of processes

1	Run with each of the following number of processes: 2, 4, 8, 16, 32
2	For each algorithm run each of the blocking write operations. 1-write_shared, 2-write_ordered
3	Fixed chunk size per loop: 2MB
4	Fixed amount of data output: 65G

5.4.1 Writing to PVFS2 from a single node

This test writes to the PVFS2 parallel file system from a single 48-core computer. The individual file, sm, and addproc algorithms are used in this test.

Results : write_shared - blocking, non-collective write

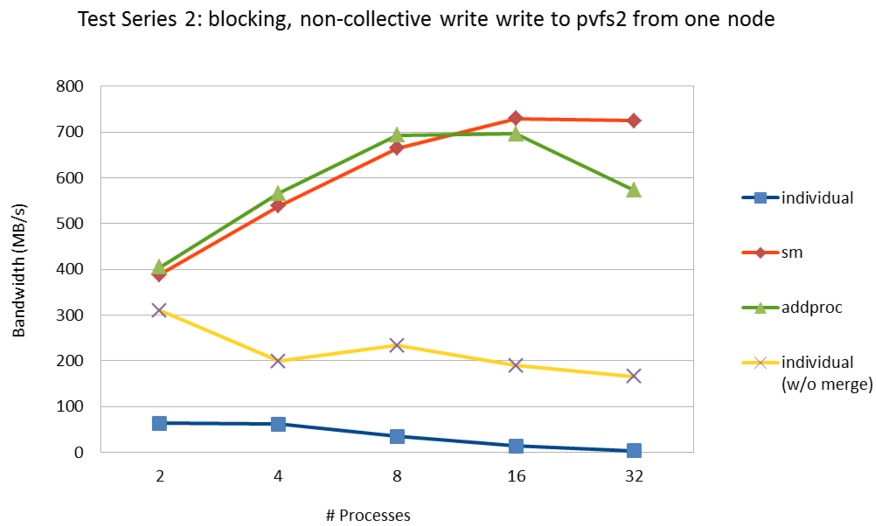

Figure 5.8: Test Series 2: On PVFS2 from single node - Comparison for blocking, non-collective write operation

Figure 5.8 gives the performance of the blocking, non-collective write operation. As in test series 1, the sm and addproc algorithms showed much better performance than the individual file algorithm.

The performance of the sm and addproc algorithms increases steadily from 2 to 16 processes. With 32 processes the sm algorithm leveled off and the addproc algorithm's performance went down. One possible cause is congestion caused by the

increased number of shared file pointer accesses as the number of processes increases.

Another reason that may explain the performance degradation with 32 processes is that as the number of processes increase, the data chunk size written by each process decreases. For an 8 process run, the 2MB write request by the application is converted into 8 partial writes. For a 32 process run, the 2MB write request by the applicatioin is converted into 32 partial writes. The application is not only increasing the number of writes, it is also writing in smaller less efficient data chunk sizes.

The after merge performance of the individual algorithm showed a steady decline as the number of processes went up. As the number of processes goes up, more communication is needed between the processes in order to perform the merge process. Additionally, congestion on the file system may be produced by the two-fold number of writes as the individual algorithm first writes to the individual files before writing to the main file.

Results: write_ordered - blocking, collective write

Figure 5.9 gives the performance of the blocking, collective write operation. As in test series 1, the sm and addproc algorithm show much lower performance due to the number of aggregators used by the collective algorithm. Also, as in test series 1, the individual file algorithm's performance is comparable to the sm algorithm during the collective write operation.

The performance of all three algorithms decreases as the number of processes increases. This is may be due to the increased data communication and increased data shuffling needed by the collective algorithm. Also, recall that the collective write is limited in how much data can be combined because the application always

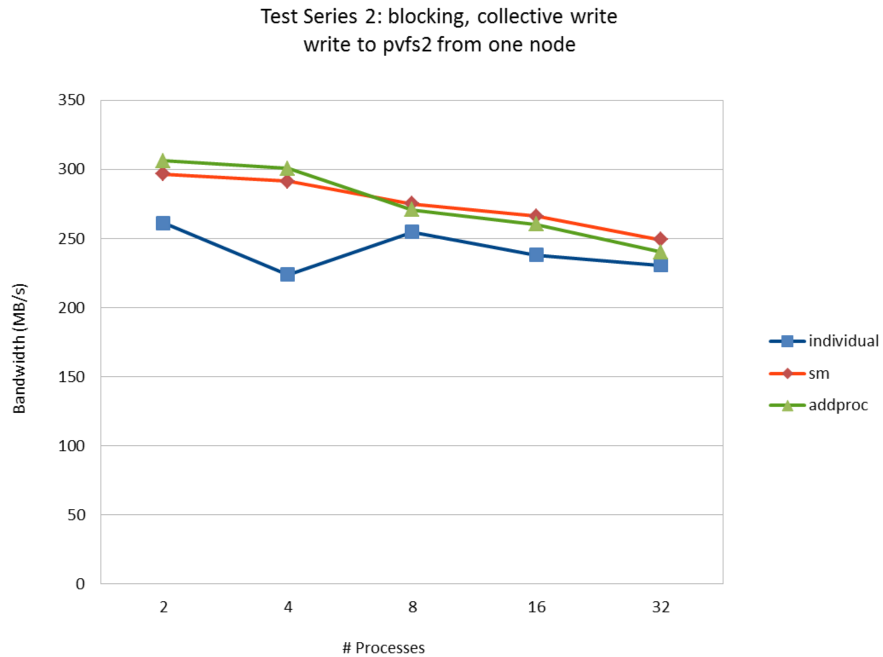

Figure 5.9: Test Series2: On PVFS2 from single node - Comparison for blocking, collective write operation

writes in 2MB data chunks regardless of how many processes are spawned. As a result, the largest combined write that can be produced is 2MB.

5.4.2 Writing to PVFS2 from multiple nodes

This test spreads the increasing number of processes across 8 nodes and writes to the PVFS2 parallel file system. The Open MPI command line parameter bynode is used in order to force mpirun to distribute the processes in a round robin fashion among the 8 nodes.

Since this test is run from multiple nodes, only the individual and addproc algorithms can run.

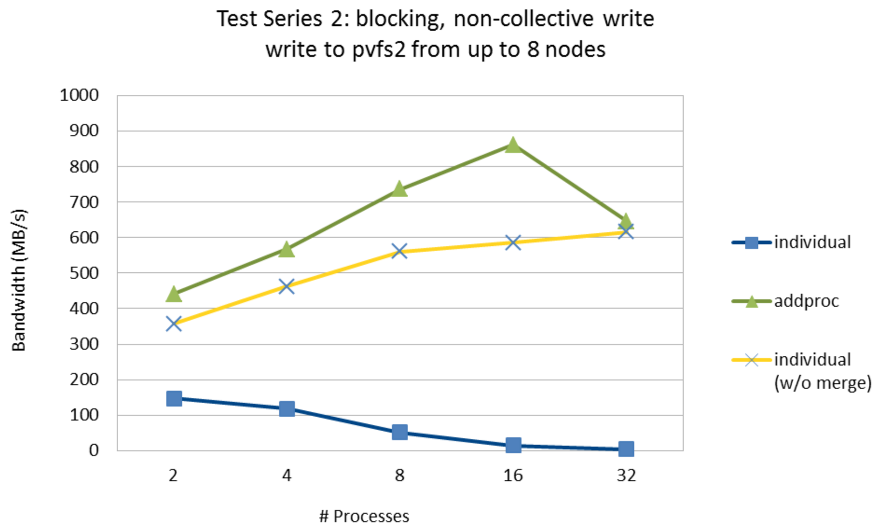

Figure 5.10: Test Series 2: On PVFS2 from multiple nodes - Comparison for blocking, non-collective write operation

Figure 5.10 gives the performance of the blocking, non-collective write shared operation and figure 5.11 gives the performance of the blocking, collective write operation. This test gave similar results as the single node test.

When compared to the single node test, the individual file algorithm non-collective write after merge does show slightly better performance when using 2,4 and 8 processes. With 16 and 32 processes the performance drops just as drastically as in the single node test.

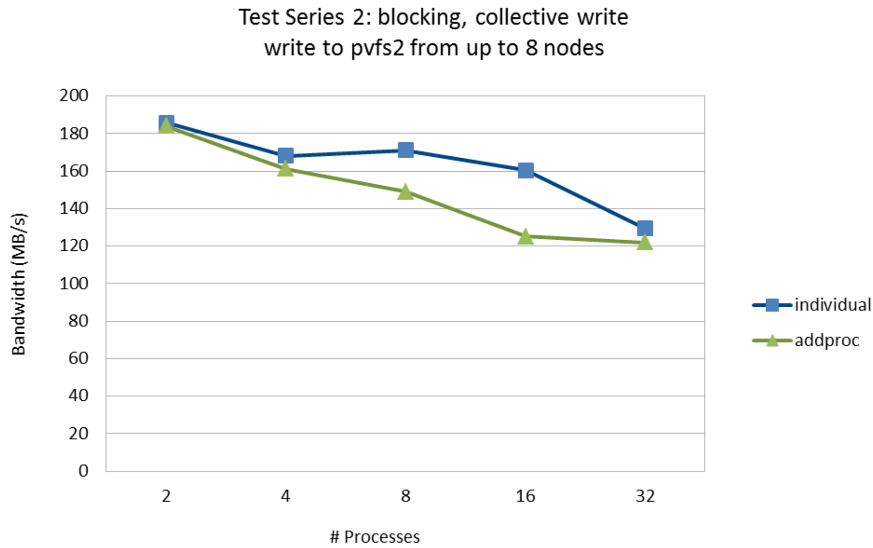

Figure 5.11: Test Series 2: On PVFS2 from multiple nodes - Comparison for blocking, collective operation

5.4.3 Writing to ext4

This test writes to a local ext4 file system on a 48-core computer. All of the algorithms, including ROMIO, are able to participate in this test.

Figure 5.12 gives the performance of the blocking, non-collective write operation. All of the algorithms performed similar to the ext4 test in test series 1. As the number of processes increased the bandwidth stayed steady for all but the individual file algorithm with the merge step.

Figure 5.13 gives the performance of the blocking, collective write operation.

The collective write had a slightly higher bandwidth than the non-collective write. But the bandwidth does decrease slowly as the number of processes increases. This is probably due to the increased communication and data shuffling that occurs as

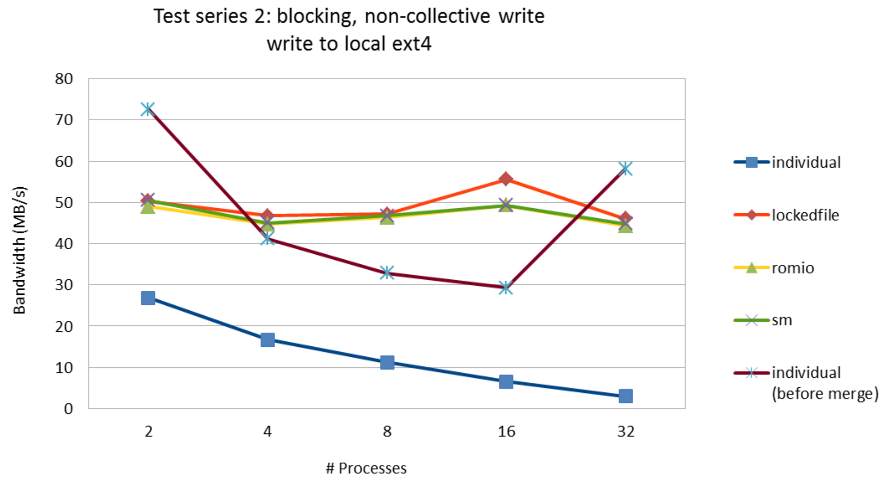

Figure 5.12: Test Series 2: On Ext4 - Comparison for blocking, non-collective write operation

the number of processes goes up.

As in the ext test in test series 1, the ROMIO collective write has a lower performance than OMPIO.

The main conclusion of this test is that the shared file pointer access mechanism is not much of a factor when writing to a single local disk.

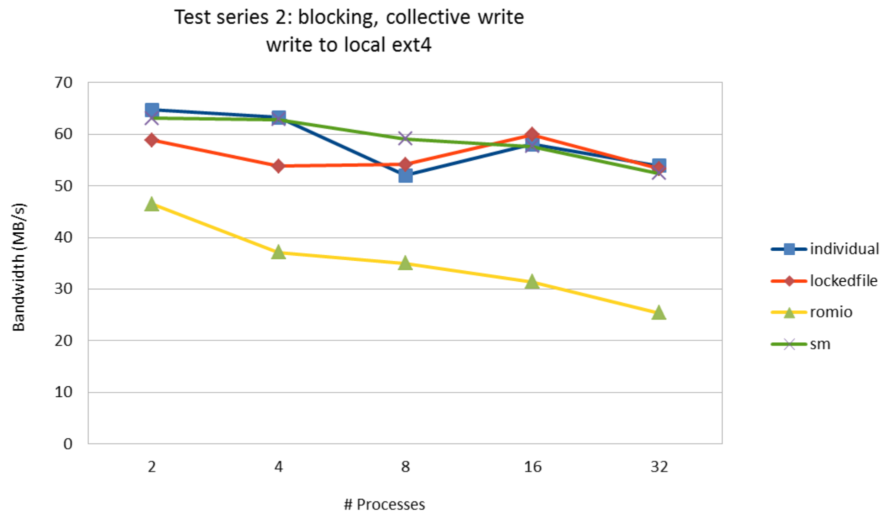

Figure 5.13: Test Series 2: On Ext4 - Comparison for blocking, collective write operation

5.5 Test Series 3: Varying I/O requests

This test series measures the performance of the algorithms by varying the number of I/O requests but keeping the total data written constant at 65G. As a result, as the number of I/O requests increase, the data chunk written decreases in size. The number of I/O requests are set by changing the benchmark's loop parameter. In order to make the results easier to compare to the results of other test series the number of processes is kept at 8.

The number of loops are varied from 32 to 65536 in multiples of two (32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536). The data written per loop vary in order to keep the total data written constant. For example, with 32,768 loops the benchmark must write 2MB per loop iteration. 2MB is the data chunk size that

Table 5.3: Test Series 3: Varying I/O requests

1	Run with 8 processes.
2	For each algorithm run each of the blocking write operations. 1-write_shared, 2-write_ordered
3	Fixed amount of data output: 65G
4	Vary number of loops from 32 to 65536 in multiples of two (32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536)

is used in test series 1 and 2.

5.5.1 Writing to PVFS2 from a single node

This test writes to the PVFS2 parallel file system from a single 48-core computer. The individual file, sm and addproc algorithms are used in this test.

Results : write_shared - blocking, non-collective write

Figure 5.14 gives the performance of the blocking, non-collective write operation. The performance of the individual without merge, sm, and addproc algorithms produced similar performance. Performance begins to drop after 8192 loops. The drop in performance can be due to the work that each algorithm must do in order to process each I/O request. Another possibility is that the performance drop is due to the increased number of smaller write requests that the file system must process.

Figure 5.15 gives a close up of the performance of the individual file algorithm. The individual file after merge performance begins it's decline at 8192 loops. At 4096 loops the performance is 300 MB/s and at 32768 it has dropped to 50MB/s. This shows that setting a bigger chunk size can result in drastically improved bandwidth.

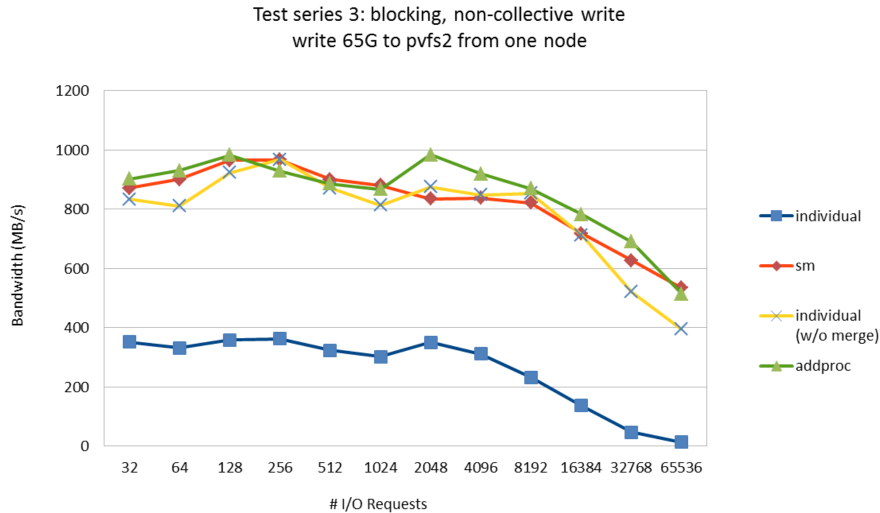

Figure 5.14: Test Series 3: On PVFS2 from single node - Comparison for blocking, non-collective write operation

Results: write_ordered - blocking, collective write

Figure 5.16 gives the performance of the blocking, collective write operation. The individual file, sm, and addproc produce similar performance. The individual algorithm seems to have a slight advantage on the sm algorithm and the sm algorithm seems to have a slight advantage on the addproc algorithm.

As the data chunk written per loop decreases, there are less data that each collective call can combine and write. Also, there is a big drop in performance when going from 1024 loops to 2048 loops. This the point at which the collective write algorithm reduces the number of aggregators.

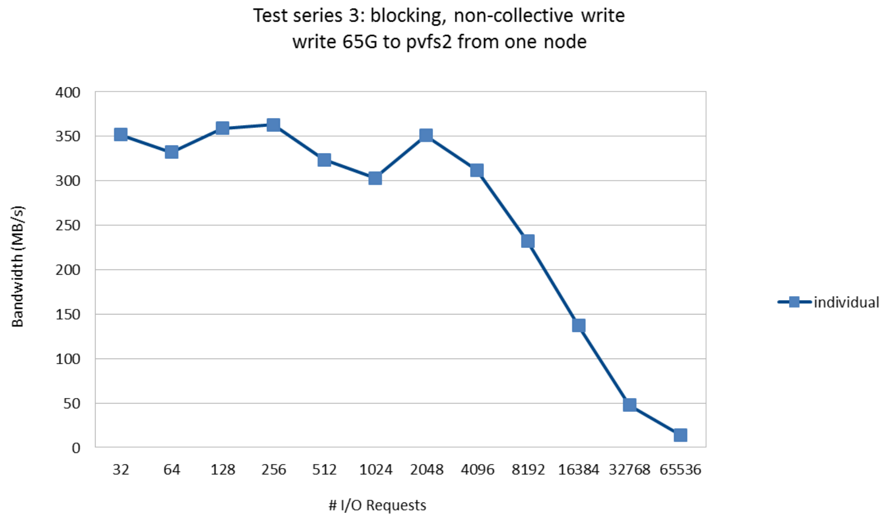

Figure 5.15: Test Series 3: On PVFS2 from single node - Comparison for blocking, non-collective write operation (close up)

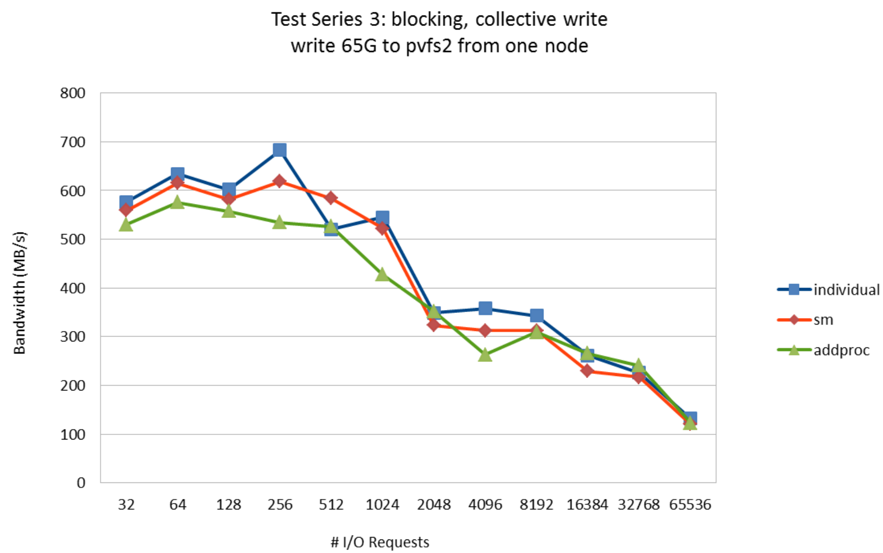

Figure 5.16: Test Series 3: On PVFS2 from single node - Comparison for blocking, collective write operation

5.5.2 Writing to PVFS2 from multiple nodes

Since this test is run from multiple nodes, only the individual and additional process algorithms can run.

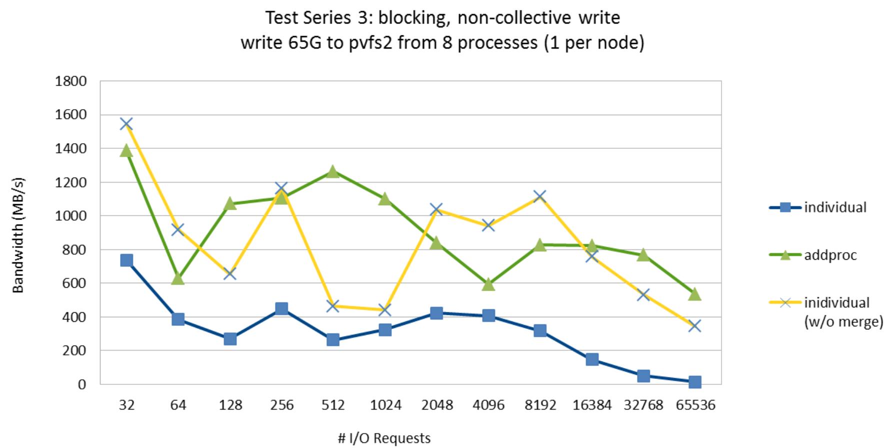

Figure 5.17: Test Series 3: On PVFS2 from multiple nodes - Comparison for blocking, non-collective write operation

Figure 5.17 gives the performance of the blocking, non-collective write shared operation. The performance for addproc and individual file before merge were similar to the single node test as the number of loops increased. With fewer loops the larger writes were able to produce much higher bandwidth. It is believed that speeds higher than 1000 MB/s are the result of caching effects.

Figure 5.18 gives the performance of the blocking, collective write operation. The results were also similar to the single node test. The higher speeds with fewer loops could be attributed to the file system's ability to write larger chunk sizes more efficiently as well as the caching effects observed in the non-collective write case.

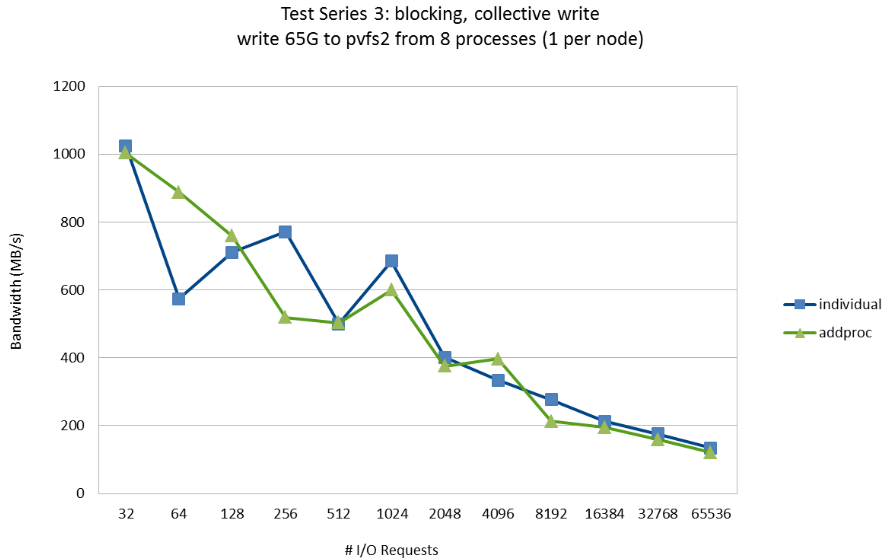

Figure 5.18: Test Series 3: On PVFS2 from multiple nodes - Comparison for blocking, collective operation

5.5.3 Writing to ext4

This test writes to a local ext4 file system on a single 48-core computer. All of the algorithms, including ROMIO, are able to participate in this test.

Figure 5.19 gives the performance of the blocking, non-collective write operation. As expected, all of the algorithms show better performance when the data chunks are bigger. As the number of loops increase and the shared file pointer accesses increase, the results show a slight separation between the sm, lockedfile and ROMIO algorithms. The lockedfile algorithm achieved lower bandwidth than the sm algorithm.

The individual algorithm, before merge and after merge, is slower than the other

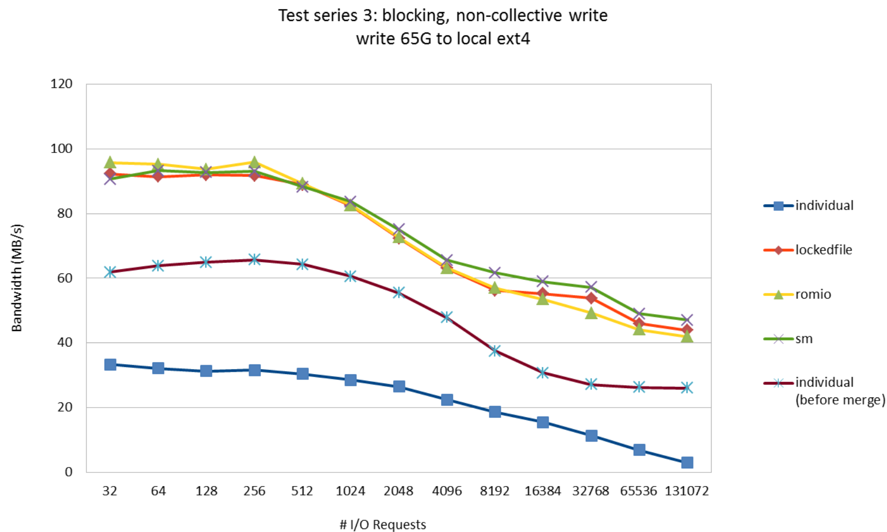

Figure 5.19: Test Series 3: On Ext4 - Comparison for blocking, non-collective write operation

algorithms due to all of the extra communication needed for merge check and the merge itself.

Figure 5.20 gives the performance of the blocking, collective write operation.

The collective write performance went down as the size of the write request decreased. The performance of the sm, lockedfile and ROMIO algorithms was comparable until 1024 loops (or 64MB data chunk sizes). Since the lockedfile algorithm performed better than the ROMIO, it is more likely that the slower performance is due to the collective algorithm used in ROMIO.

As in the non-collective test, the sm algorithm performed slightly better than the lockedfile algorithm.The individual algorithm was able to perform better than the other algorithms when there were less loops.

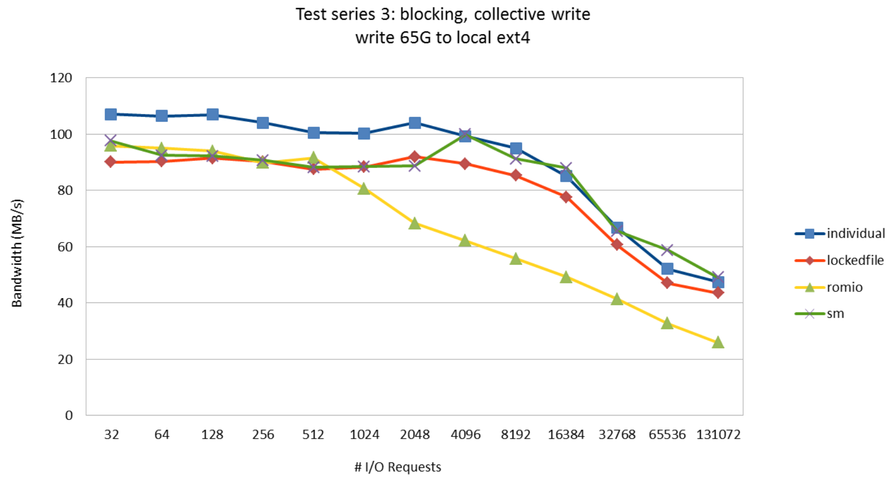

Figure 5.20: Test Series 3: On Ext4 - Comparison for blocking, collective write operation

5.6 Test Series 4: Constant write size per process

This test series is a variation of test series 2. Test series 2 writes 2MB per group of processes per loop. In this test series each process writes 2MB per loop. The tests in this series are run multiple times, each time increasing the number of processes. The total data output is fixed at 65G. As the number of processes increase, the number of simultaneous shared file pointer updates also increases.

The tests include writing to the PVFS2 parallel file system from a single node, writing to PVFS2 from multiple nodes, and writing to a local ext4 file system from a single node.

Table 5.4: Test Series 4: Constant write size per process

1	Run with each of the following number of processes: 2, 4, 8, 16, 32
2	For each algorithm run each of the blocking write operations. 1-write_shared, 2-write_ordered
3	Fixed chunk size per process: 2MB
4	Fixed amount of data output: 65G

5.6.1 Writing to PVFS2 from a single node

This test writes to the PVFS2 parallel file system from a single 48-core computer. The individual file, sm, and addproc algorithms are used in this test.

Results : write_shared - blocking, non-collective write

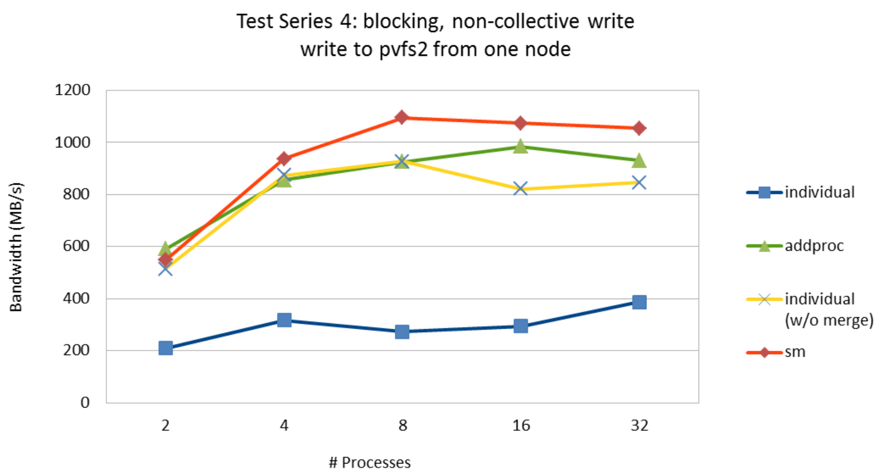

Figure 5.21: Test Series 4: On PVFS2 from single node - Comparison for blocking, non-collective write operation

Figure 5.21 gives the performance of the blocking, non-collective write operation.

Performance increases from 2 to 8 processes. From 8 to 32 processes the performance levels off. One reason for this may be that the system is saturated with write requests. Another reason for this behavior may be because the PVFS2 filesystem is configured to have 8 data servers.

The sm algorithm performed the best, followed by the addproc algorithm, and lastly the individual algorithm without the merge step.

As expected the performance of the individual file algorithm after merge was dramatically lower than the other algorithms. But, the performance was much higher than in test series 2 which wrote the file in smaller data increments. The performance actually went up with 32 processes instead of going down as it did in test series 2.

Results: write_ordered - blocking, collective write

Figure 5.22 gives the performance of the blocking, collective write operation. The performance of the collective write was steady up until 16 processes. This performance is better than the single node test in test series 2. In test series two the performance slowly dropped as the number of processes went up. In this test, there is more data to write collectively as the number of processes goes up.

The bigger surprise is the big jump from 350MB/s at 16 processes to 500 MB/s at 32 processes. The reason for this improvement in performance is that at 32 processes the total data written during one collective call are enough to cause the collective algorithm to use two aggregators instead of only one.

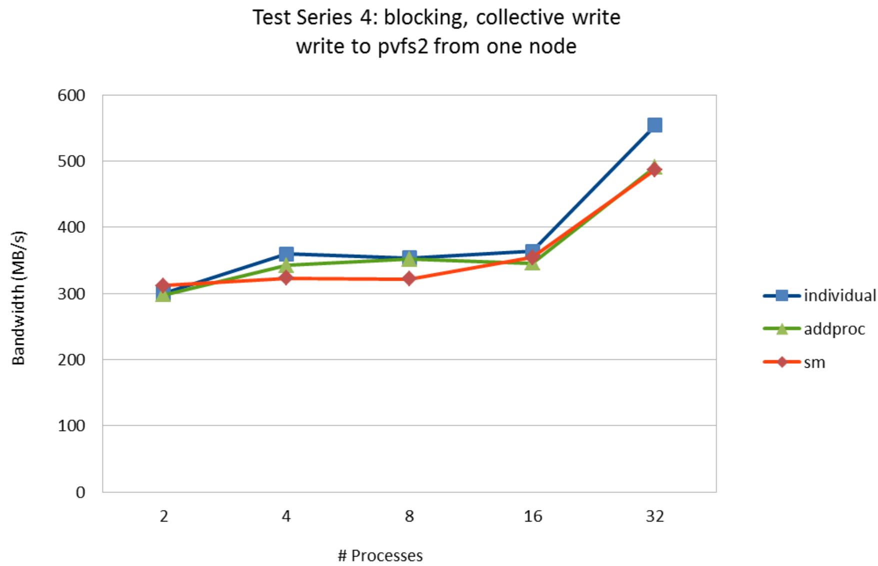

Figure 5.22: Test Series 4: On PVFS2 from single node - Comparison for blocking, collective write operation

5.6.2 Writing to PVFS2 from multiple nodes

This test spreads the processes across 8 nodes to write to the PVFS2 parallel file system. The Open MPI command line parameter bynode is used in order to force mpirun to distribute the processes in a round robin fashion among the 8 nodes.

Since this test is run from multiple nodes, only the individual algorithm and the additional process algorithm can run.

Results : write_shared - blocking, non-collective write

Figure 5.23 gives the performance of the blocking, non-collective write shared operation. Using multiple nodes produced higher performance than using a single node.

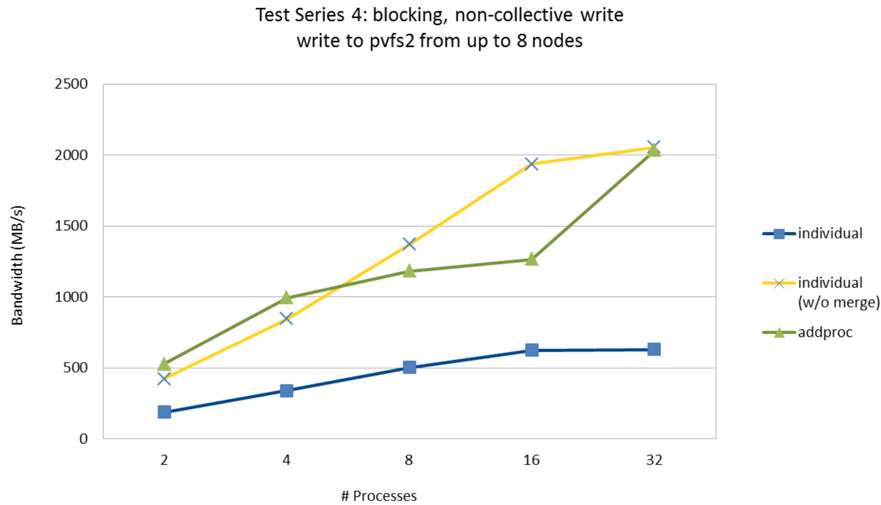

Figure 5.23: Test Series 4: On PVFS2 from multiple nodes - Comparison for blocking, non-collective write operation

The individual file without merge and additional process were able to easily surpass 1000MB/s. These especially high speeds are probably due to caching effects caused by having 65G of memory on each node.

The individual file algorithm after merge performance was also higher. One reason may be due to the caching effects seen in the before merge performance. The other reason is that the number of I/O requests decrease in number but increase in size as the number of processes increase.

Results : write_ordered - blocking, collective write

Figure 5.24 gives the performance of the blocking, collective write operation. The performance of the multiple node test was similar to the single node test. This is due to the fact that the collective algorithm is using the same number of aggregators

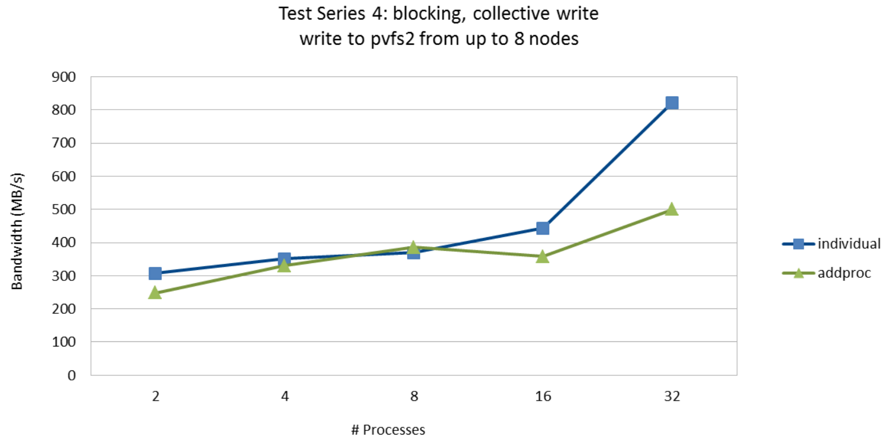

Figure 5.24: Test Series 4: On PVFS2 from multiple nodes - Comparison for blocking, collective operation

as in the single node test. The performance increases by a large amount when going from 16 to 32 processes due to the increase in the number of aggregators.

The addproc algorithm did not outperform the individual file algorithm. It is possible that the additional processes was slowed down by the higher number of simultaneous shared file pointer requests. Also, the additional process may have been competing for communication bandwidth with the other processes.

5.6.3 Writing to ext4

This test writes to a local ext4 file system on a 48-core computer. All of the algorithms, including ROMIO, are able to participate in this test.

Figure 5.25 gives the performance of the blocking, non-collective write operation. By not decreasing the chunk size as the number of processes increased, the individual

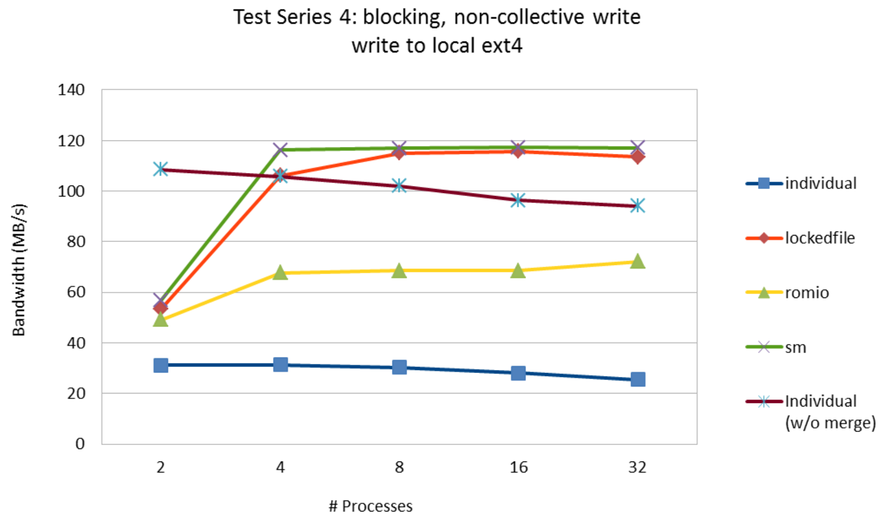

Figure 5.25: Test Series 4: On Ext4 - Comparison for blocking, non-collective write operation

algorithm was able to maintain a relatively steady bandwidth from 2 to 32 processes. The performance was still lower than the other algorithms due to the merge process.

The other algorithms were able to keep a high write speed from 4 to 32 processes. It is not clear why ROMIO was not able to reach the same speeds as the other algorithms.

Figure 5.26 gives the performance of the blocking, collective write operation. The collective write performed similar to the test in test series 2. All of the algorithms showed similar performance, except for ROMIO which was consistently lower. The performance for the lockedfile dropped slightly with 32 processes. It is unlikely that it was due to the locking mechanism, since the non-collective write did not have the same drop in performance.

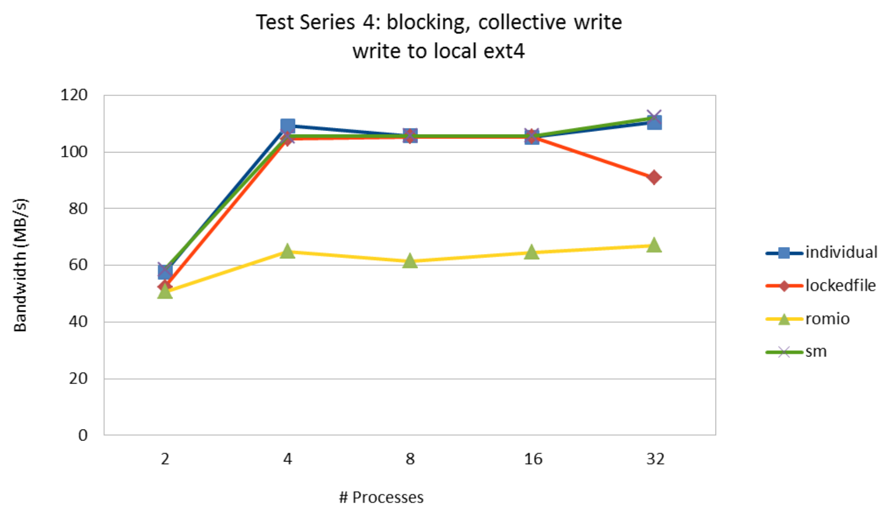

Figure 5.26: Test Series 4: On Ext4 - Comparison for blocking, collective write operation

Chapter 6

Summary

This thesis aims at designing portable and optimized algorithms that can allow the shared file pointer operations to work in more systems. The goal is not to find one algorithm that will work on every system. Instead, the hope is to develop a suite of specialized algorithms that will work in different situations.

Picking an algorithm to use from a suite of different algorithms is not an easy or desirable task. To aid with this task a selection logic was also developed that automatically selects the most appropriate algorithm. The selection logic can be controlled with user hints and system configuration parameters.

This thesis explores the following four algorithms:

Locked file algorithm : The locked file algorithm takes advantage of file systems that implement file locks. The shared file pointer is kept in a file that can be accessed by every process. Each process must acquire the file lock before reading or updating the shared file pointer value.

Shared memory algorithm : The shared memory algorithm is meant to run on a shared memory computer and uses a shared memory section to keep track of the shared file pointer value.

Individual file algorithm : Each process writes data to a local file instead of the main file. The contents of the local files are merged into the main file during collective file operations such as a collective write or a file close. The individual file algorithm can only be used for write operations.

Additional process algorithm : An additional process is spawned by a group of processes participating in the file open operation. This process maintains the shared file pointer value. Each process must communicate with this additional process in order to read or update the shared file pointer value.

The performance of these algorithms was evaluated on ext4, a local file system, and on PVFS2, a parallel file system. Tests were run from a single node shared memory environment as well as a multi-node distributed memory environment.

The results show that the performance of the locked file, the shared memory and the additional process algorithms is comparable to each other. The shared memory algorithm is the fastest but can only be used if all of the process are on the same node. The individual algorithm provides a good option when running on multiple nodes, but is much slower and only supports write operations. The additional process algorithm can be run from multiple nodes and can be used on all file systems, but a specialized algorithm may yield better performance.

The selection logic was updated to choose the shared memory algorithm if all of

the processes are running on the same node. If the processes are running on multiple nodes the additional process algorithm is chosen. The individual file algorithm performs better than the additional process algorithm when the application only performs collective writes. Unfortunately, the selection logic cannot determine this. The user has the option of manually selecting the individual file algorithm in this scenario.

This work can be expanded in the following directions:

- The additional process algorithm was implemented as part of the OMPIO shared file pointer framework when this thesis was well under way. Further debugging and testing maybe needed. Having this algorithm available provides shared file pointer read operation functionality on a multi-node environment.
- The available algorithms can be further optimized and tested.
- The selection logic can be fine tuned as more test data is collected and analyzed.
- Measurements need to be collected on the shared file pointer read performance.

Bibliography

[1] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, June 1995. URL: http://www.mpi-forum.org/ (Retrieved: 12/01/2012).

[2] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, page 182, Washington, DC, USA, 1999. IEEE Computer Society.

[3] Robert Latham, Robert Ross, and Rajeev Thakur. Implementing MPI-IO Atomic Mode and Shared File Pointers Using MPI One-Sided Communication. *Int. J. High Perform. Comput. Appl.*, 21(2):132–143, 2007.

[4] Ketan Kulkarni and Edgar Gabriel. Evaluating Algorithms for Shared File Pointer Operations in MPI I/O. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science: Part I*, volume 5544, pages 280–289. Springer Berlin Heidelberg, 2009.

[5] Mohamad Chaarawi, Edgar Gabriel, Rainer Keller, Richard L. Graham, George Bosilca, and Jack J. Dongarra. Ompio: a modular software architecture for

mpi i/o. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 81–89, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] Open MPI: Open Source High Performance Computing. URL: http://www.open-mpi.org/ (Retrieved: 12/01/2012).

[7] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing*. Addison Wesley, second edition, January 2003.

[8] Edgar Gabriel. COSC6374 Parallel Computation class slides. URL: http://www2.cs.uh.edu/g̃abriel/courses/cosc6374_s09 (Retrieved: 04/01/2009).

[9] Blaise Barney. Introduction to Parallel Computing, July 2006. http://www.llnl.gov/computing/tutorials/parallel_comp/ (Retrieved: 12/01/2012).

[10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition, September 2006.

[11] AMD. Official Press Release - AMD FX Series Processor. URL: http://www.amd.com/us/press-releases/Pages/unlock-your-record-setting-2011oct12.aspx (Retrieved: 12/01/2012).

[12] Ryan Shrout. Intel Shows 48-core x86 Processor as Single-chip Cloud Computer. URL: http://www.pcper.com/reviews/Processors/Intel-Shows-48-core-x86-Processor-Single-chip-Cloud-Computer (Retrieved 12/01/2012).

[13] Chris Davies. Samsung Exynos 4 Quad confirmed for Galaxy S3. URL: http://www.slashgear.com/samsung-exynos-4-quad-confirmed-for-galaxy-s3-26224798 (Retrieved: 12/01/2012).

[14] John May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2003.

[15] L.P Hewlett-Packard Development Company. Memory Technology Evolution: An Overview of System Memory Technologies, Technology Brief, 2010.

[16] The Free Encyclopedia Wikipedia. Solid-state drive. URL: http://en.wikipedia.org/wiki/Solid-state_drive (Retrieved: 03/24/2013.

[17] Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, July 1997. URL: http://www.mpi-forum.org/ (Retrieved: 12/01/2012).

[18] Rajeev Thakur, Robert Ross, Ewing Lusk, and William Gropp. *Users Guide for ROMIO: A High Performance, Portable MPI-IO Implementation*. Argonne National Laboratory, May 2004. OpenMPI 1.2.3 documentation.

[19] Jason Cope, Kamil Iskra, Dries Kimpe, and Robert B. Ross. Portable and Scalable MPI Shared File Pointers. In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors, *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 312–314. Springer, 2011.

[20] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa. Optimization Techniques at the I/O Forwarding Layer. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 312 –321, sept. 2010.

[21] Rajeev Thakur, William Gropp, and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, New York, NY, USA, 1999. ACM.

[22] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[23] Rajeev Thakur, William Gropp, and Ewing Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[24] The Free Encyclopedia Wikipedia. File locking. URL: http://en.wikipedia.org/wiki/File_locking (Retrieved: 12/01/2012).

[25] The Free Encyclopedia Wikipedia. Unix-like. URL: http://en.wikipedia.org/wiki/Unix-like (Retrieved: 12/01/2012).

[26] A. Silberschatz, G. Gagne, and P.B. Galvin. *Operating System Concepts 8th Edition Binder Ready Version*. John Wiley & Sons, 2008.

[27] W.R. Stevens. *Interprocess Communications: V. 2: Interprocess Communications*. Unix Network Programming. Prentice Hall, 2012.