2009

Measurement of PVFS2 performance on InfiniBand

Sudhindra Prasad Tirupati Nagaraj
Iowa State University

Measurement of PVFS2 performance on InfiniBand

by

Sudhindra Prasad Tirupati Nagaraj

A thesis submitted to the graduate faculty

in partial fulfillment of requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Brett Bode, Co-Major Professor
Robyn R. Lutz, Co-Major Professor
Soma Chaudhuri

Iowa State University

Ames, Iowa

2009

TABLE OF CONTENTS

LIST OF FIGURES iii

LIST OF TABLES iv

ACKNOWLEDGEMENTS v

ABSTRACT vii

CHAPTER 1. INTRODUCTION 1

CHAPTER 2. BACKGROUND 5
2.1 InfiniBand Architecture 5
2.2 Parallel Virtual File System (PVFS) 7
2.2.1 PVFS/InfiniBand 7
2.3 MPI I/O 9
2.4 IOR Parallel I/O Benchmarking Tool 10
2.5 Overall I/O path used for benchmarking 11

CHAPTER 3. SYSTEM CONFIGURATION 13
3.1 Hardware configuration 13
3.2 Software configuration 15

CHAPTER 4. BENCHMARKING METHODOLOGY 16
4.1 Outline of the methodology used 16
4.2 Experimental Setup 17
4.3 Test Strategy 17

CHAPTER 5. BENCHMARKING RESULTS 19
5.1 Graph description 19
5.2 Results 19
5.3 Analysis of the Results 24

CHAPTER 6. CONCLUSION 32

CHAPTER 7. FUTURE WORK 33

BIBLIOGRAPHY 34

APPENDIX 36
Makefile changes 36
IOR.c changes 37
aiori.h changes 37
aiori-PVFS2.c 38
benchmark script 50
perfquery/vmstat script 54

LIST OF FIGURES

Figure 1. OpenIB software stack 6
Figure 2. PVFS2 architecture 8
Figure 3. PVFS2/InfiniBand 9
Figure 4. I/O path 12
Figure 5. ibmcluster in Ames Lab 14
Figure 6. 2-client write bandwidth performance 20
Figure 7. 2-client read bandwidth performance 20
Figure 8. 4-client write bandwidth performance 21
Figure 9. 4-client read bandwidth performance 21
Figure 10. 6-client write bandwidth performance 22
Figure 11. 6-client read bandwidth performance 22
Figure 12. MPI I/O write bandwidth comparison 30

LIST OF TABLES

Table 1. Hardware configuration of test system 13
Table 2. Software configuration of test system 15
Table 3. Standard deviation for writes on 6 clients 23
Table 4. Standard deviation for reads on 6 clients 24
Table 5. Context Switch counts across interfaces for writes 27
Table 6. Mean CPU time 28

ACKNOWLEDGEMENTS

The work was supported in part by Iowa State University under the contract DE-AC02-07CH11358 with the U.S. Department of Energy.

I would like to take this opportunity to express my deep felt gratitude and thanks to those who were instrumental in helping me with my research and the writing of my thesis. Without their support, I would not have been successful in my endeavor. To start with, I would like to thank Dr. Brett Bode for his outstanding support and the freedom he offered me to pursue my research. His patience and insistence have inspired me a lot to complete my thesis. Without his favorable support, I would not have been able to complete my thesis early enough. I would like to thank the contribution of Troy Benjegerdes who gave valuable inputs towards my thesis and helped me in constantly troubleshooting and resolving the ever-failing hardware infrastructure in Ames Lab. I would like to thank Dr. Robyn R. Lutz for her kind encouragement and timely advices to help me in writing my thesis. Her smiling support and her research experience motivated me to great heights. Her course on Software Safety helped me understand the intricacies of writing a research article.

I would like to thank my committee member Dr. Soma Chaudhuri for her encouragement. Her gentle guidance and the course on Distributed Algorithms I took with her provided me a solid foundation for the way research is conducted. I also would like to thank the Ames Lab staff for providing me a good research environment to work in.

On a personal level, I am grateful to Dr. Kasthurirangan Gopalakrishnan for his constant suggestions to help me align my research focus. I am thankful to Sandeep Krishnan, Ankit Agrawal and Ganesh Ram Santhanam for helping me in writing my thesis. I am most grateful to my parents Mr. T. S. Nagaraj and Mrs. Sarala Nagaraj and my sister Ms. Deepti for their love and moral support all through my graduate studies.

ABSTRACT

InfiniBand is becoming increasingly popular as a fast interconnect technology between servers and storage. It has far better price/performance ratio compared to both Gigabit Ethernet and 10 Gigabit Ethernet, and hence is being increasingly used for high-performance computing applications. PVFS2, the second generation Parallel Virtual File System (PVFS), is a distributed file system for parallel data access that is being increasingly used in clustered applications. As previous studies have shown, in general, PVFS2 over InfiniBand offers enhanced I/O rates compared to PVFS2 over TCP and Gigabit Ethernet. Apart from the hardware technology, the application programming interface into the file system also makes a difference. To get better parallel performance, the choice of a file system interface is important. Our study is to benchmark and compare the performance of PVFS2 running over InfiniBand using different file system interfaces. IOR is a popular I/O benchmarking tool that supports the POSIX and MPI I/O file system interfaces. In addition to testing these already supported interfaces, we have written a PVFS2 module extension for IOR to support native PVFS2 interfaces into the PVFS2 file system. As we shall see in this study, using native PVFS2 interface offers significant performance benefit compared to other file system interfaces on the PVFS2 file system. Our benchmarking effort also involves studying the effect of a multi-client environment on the I/O performance of different file system interfaces. Based on the benchmarking results we obtain, we determine the most efficient application programming interface for parallel I/O on PVFS2 in a typical multi-client parallel application scenario.

CHAPTER 1. INTRODUCTION

As the size of data grows exponentially [15], there is a need for serving the data more efficiently in a cost-effective way. This is especially true of scientific applications that work with terabytes, or even petabytes, of data and hence need an advanced file system to allow for efficient storage and retrieval of data. A parallel file system like PVFS addresses the issue of facilitating efficient data processing by the high-performance computing (HPC) applications. PVFS allows for parallel access to data striped across multiple file servers. Thus, parallel applications, where concurrent, large I/O and many file access are common [2], benefit from the dynamic distribution of I/O data and metadata on a PVFS file system. We are now into the second generation of PVFS file system, referred to from here on as PVFS2. The PVFS2 clients communicate with the PVFS2 servers over various interconnect technologies. Traditionally, Gigabit Ethernet was the popular interconnect technology for PVFS client/server communication. With InfiniBand [1] offering high data bandwidth rates, it is fast replacing Gigabit Ethernet as the client/server interconnect technology. In addition InfiniBand has better price/performance ratio. For example, a 24-port DDR switch is under $5000, that is $200/port, whereas a 10 Gigabit Ethernet costs about $400/port, say an Arastra 10 gig switch, thus offering only half the bandwidth. We therefore chose InfiniBand for our benchmarking purposes because of its high potential impact on the world of parallel scientific applications.

Currently, as far as we know, there has not been much effort in benchmarking of PVFS2 on the InfiniBand interconnect. There has been no effort to the best knowledge to study deeply the effect of file system interfaces in a PVFS2/InfiniBand environment for a multiple client parallel I/O scenario. The tools that allow benchmarking of parallel I/O with different APIs into PVFS2 either do not allow for effective testing of a multi-client environment, or do not support all the APIs for an effective analysis. J. Wu et al. [7] offer a I/O performance comparison of PVFS between InfiniBand and TCP/IP communication protocols. They only analyze the impact of different number of compute nodes on the I/O bandwidth rates, and their study was not aimed at comparing the effect of different file system interfaces into the PVFS file system. Also, they test the original PVFS version, and not the second generation PVFS (PVFS2) which offers better performance. The study of L. Chai et al. [8] aims to compare pNFS and PVFS2 I/O performance in an InfiniBand cluster environment. Their study shows the I/O performance improvement as compared to a Gigabit Ethernet implementation. However, their work does not study the effect of different file system interfaces into the PVFS2 file system. Also, they do not study the effect of different I/O transaction unit sizes on the I/O performance. Furthermore, their study does not use IOR as the benchmarking tool that is most suitable for testing multiple client parallel I/O pattern that is most typical of parallel applications.

In order to explore the single tool that allows us to test different file system interfaces in a multi-client environment, we looked at different benchmarking options. IOZone [9] is one popular I/O benchmarking tool, but it is more ideally suited for single client throughput experiments. Another tool b_eff_io [10] examines first write, rewrite and read access, strided and segmented collective patterns on one file per application and non-collective access to one file per process. But it supports only MPI I/O interface.

Furthermore, it does not capture the parallel I/O patterns most typical in parallel applications.

IOR is a parallel I/O microbenchmark that is suited for multi-client experiments. It also allows the characterization of the I/O pattern most typical in parallel applications. In addition, it supports different file system interfaces, including POSIX and MPI I/O. IOR does not support native PVFS2 interface into the PVFS2 file system though. Hence, we wrote a PVFS2 module extension for IOR. Thus we now have a single tool that can give us all the necessary statistics required to benchmark PVFS2 on InfiniBand as per our requirements.

The first key contribution of this thesis is to extend native PVFS2 support to the IOR benchmarking tool for performing I/O into a PVFS2 file system. We believe that this support can help one use the advantages offered by IOR in terms of performing multi-client benchmarking, along with other features like computing mean I/O, standard deviation of reads and writes over multiple runs, performing non-overlapping parallel sequential I/O on a single file by multiple clients etc., and at the same time benchmark native PVFS2 interface I/O performance more effectively in comparison to other interfaces using the same tool.

The second key contribution is to benchmark the I/O bandwidths offered by different file system interfaces into the PVFS2 file system over the InfiniBand client/server interconnect using the IOR tool. This study includes testing the performance of POSIX, MPI I/O over PVFS, MPI I/O over POSIX and native PVFS2 interfaces into the PVFS2 file system. We also test with different number of PVFS2 clients and different

I/O message sizes and capture the results. We analyze the performance results so obtained from IOR to try to identify the factors determining the difference in performance of reads and writes.

The rest of the thesis is structured in the following way. Chapter 2 provides the background information on InfiniBand, PVFS, MPI I/O and IOR. Chapter 3 provides information on the test bed used for our benchmarking effort. It talks about the hardware and software configurations used while running our benchmark. Chapter 4 outlines the benchmarking methodology we used. Chapter 5 consists of the performance results obtained from IOR, a demonstration of the advantages of using native PVFS2 interface and an analysis of the results. Chapter 6 talks about the conclusion and Chapter 7 about the future work.

CHAPTER 2. BACKGROUND

In this chapter, we discuss the background of InfiniBand, Parallel Virtual File System (PVFS), MPI I/O and the IOR I/O benchmarking tool.

2.1 InfiniBand Architecture

The InfiniBand architecture (IBA) [1] provides a point-to-point linking technology used as a base for an I/O fabric that aims to increase the aggregate data rate between servers and the storage devices. In our implementation, we use InfiniBand technology to interconnect the PVFS2 clients (the processing nodes) and the PVFS2 servers (the I/O nodes), where the clients send I/O requests and servers respond to the I/O requests via InfiniBand interfaces. PVFS2 has support for the InfiniBand network fabric between the clients and the servers. The PVFS2 clients send PVFS requests using the native InfiniBand protocol stack by bypassing the traditional TCP/IP protocol stack since the InfiniBand protocol allows client/server communication via RDMA (Remote Direct Memory Access). In order to deploy the InfiniBand hardware as a communication medium between clients and servers, the clients and the servers must have the necessary software to enable InfiniBand access. OpenIB [14] is an open-source software stack developed by the OpenFabrics Alliance (OFA) that enables communication on a RDMA-capable fabric like InfiniBand. Following figure (Figure 1) shows the OpenIB protocol stack used for RDMA data transfers over InfiniBand. It shows a RDMA-based application bypassing the TCP/IP stack to talk to the hardware directly. In an OpenIB implementation, the RDMA application uses the OpenIB Verbs API in the user-space to communicate to the InfiniBand hardware. OpenIB then copies the data from the application memory to the hardware directly to perform RDMA to the remote application.

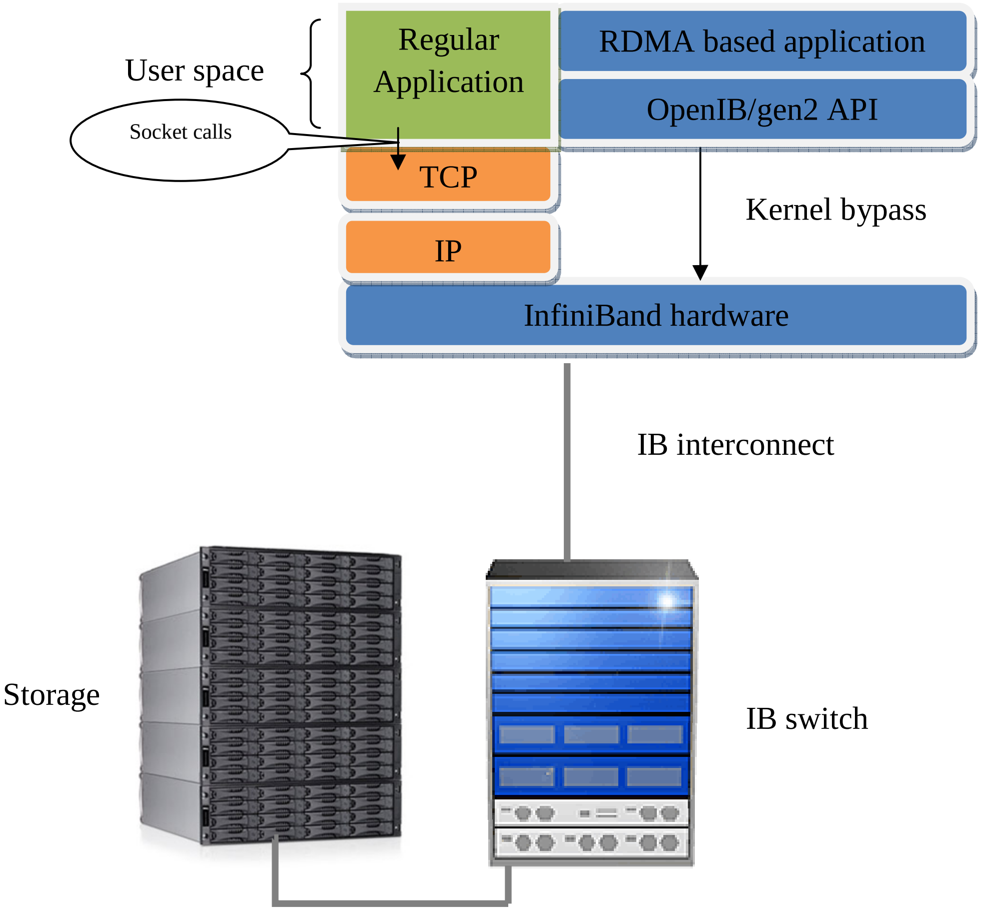

Figure 1. OpenIB software stack

2.2 Parallel Virtual File System (PVFS)

PVFS is a parallel file system that supports high performance I/O of the kind typical in High Performance Computing (HPC) clusters. The primary goal of PVFS is to provide high-speed access to file data for parallel applications [3]. PVFS is a client-server file system, with potentially multiple servers and clients. These servers act as I/O nodes, responsible for serving data, while the clients demand data from the servers. One or more nodes can act as metadata servers, responsible for metadata operations like open, close and remove operations (refer Figure 2). There need not be dedicated PVFS servers, clients and metadata servers. The same node can act as all three, though for better performances, typically deployments have dedicated nodes acting as either I/O nodes, metadata node or clients. In the PVFS file system, each PVFS file is striped across the disks on the PVFS servers. PVFS is a user-space implementation that needs no kernel modifications. PVFS is an upper layer parallel file system that sits on top of traditional native file systems like ext2, ext3, xfs etc.. So actual file data still resides on the native file system. The second generation PVFS (PVFS2) retains the design if the original version, and also provides additional advantages of higher performance and better metadata managements.

2.2.1 PVFS/InfiniBand

PVFS supports the InfiniBand interface between the clients and servers through a Buffered Messaging Interface (BMI) implementation for InfiniBand that uses either Mellanox VAPI or OpenIB APIs (see Figure 3). Since we used the OpenIB software stack in our test setup, PVFS builds over the OpenIB verbs layer to establish communication between the clients and servers via the IB channel. PVFS reads and writes happen via RDMA (Remote Direct Memory Access) operations between the clients and the servers, thus allowing the data transfer to bypass the TCP/IP stack. However, there is an OpenIB component called IPoIB which allows IB communication to happen over TCP/IP, but its discussion is beyond the scope of this thesis. In order to enable IB support with PVFS, we must build PVFS specifically for IB support.

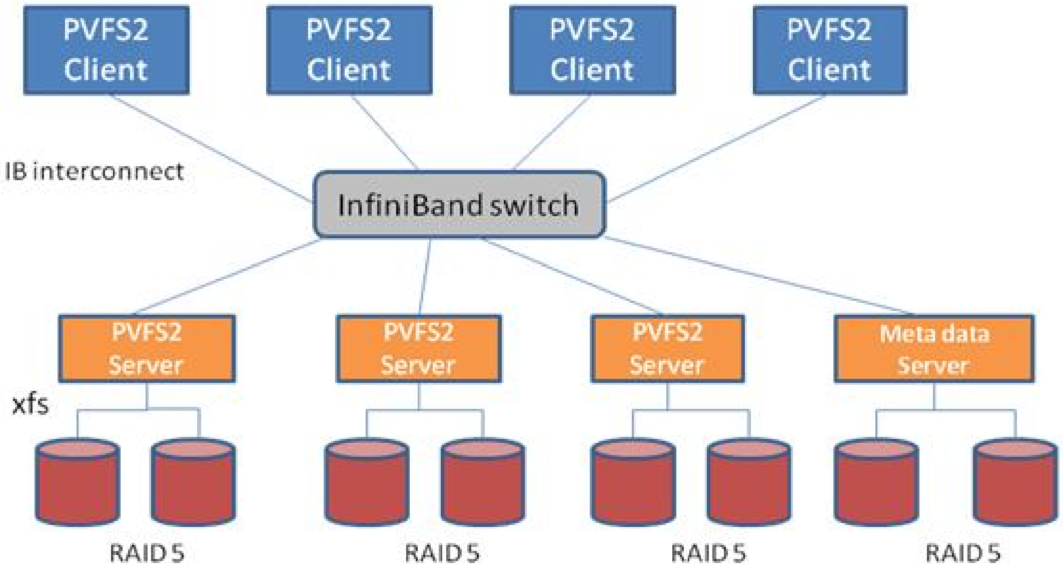

Figure 2. PVFS2 architecture

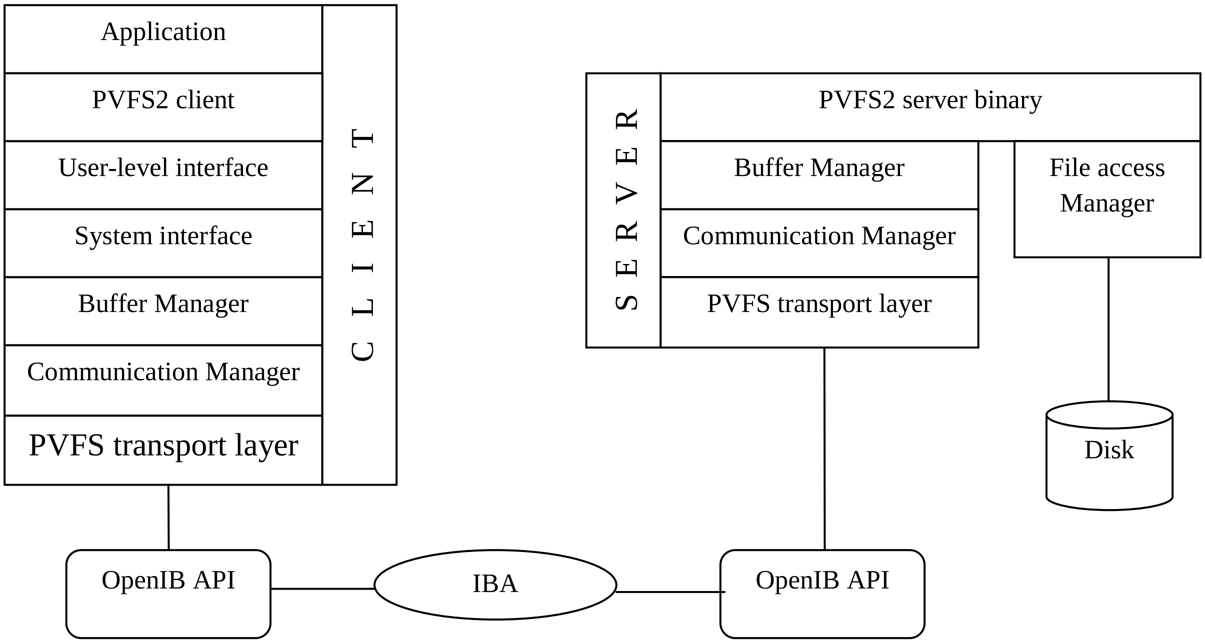

Figure 3. PVFS2/InfiniBand

2.3 MPI I/O

MPI I/O [12] is a standard application programming interface for parallel I/O defined by the MPI Forum. MPI I/O provides for concurrent I/O access to a single file by many processes. The parallelism attained thus enhances the read and write performances significantly with multiple processes performing I/O on a file in parallel, instead of a single process I/O. Since we are testing the effect of multiple PVFS2 clients on the read/write performance of PVFS2, MPI I/O becomes a natural choice for achieving concurrent I/O. The benchmarking tool we have chosen, IOR, supports the MPI I/O interface into the PVFS2 file system. MPI I/O is a process synchronization layer and uses underlying I/O function like PVFS2 I/O or Unix (POSIX) I/O to perform the actual file system I/O [4]. Since the specific choice of the I/O function has an impact on the overall MPI I/O performance, the tests in this paper cover both MPI I/O using POSIX and MPI I/O using PVFS2 stacks for analyzing the I/O performance.

ROMIO [5] is one of the main implementations of MPI I/O that provides high-performance and portability. We used the MPICH2 package that includes ROMIO as the MPI I/O implementation on our test systems. ROMIO provides a portable MPI I/O implementation through the use of an internal abstract I/O device layer called ADIO. The ADIO layer interfaces between MPI I/O and the underlying file system, in our case the PVFS2 file system [6]. ROMIO can be used on top of PVFS2 file system through 2 mechanisms. The first mechanism is where ROMIO interfaces into the POSIX compliant VFS layer to access the PVFS2 file system. In this case, ROMIO is ignorant of the underlying PVFS2 file system. The second mechanism is where ROMIO uses PVFS2 interfaces directly, instead of POSIX semantics. In this case, ROMIO needs to be built with PVFS2 support. In this paper we test with both ROMIO on VFS and ROMIO on PVFS2.

2.4 IOR Parallel I/O Benchmarking Tool

IOR (Interleaved or Random) [13] is used for testing parallel file systems using various interfaces and access patterns. It is particularly used for testing the sequential I/O pattern typical in parallel applications. IOR is especially useful for testing a multiple client parallel I/O environment. IOR needs MPI software to be installed on all the client nodes to achieve process synchronization between multiple clients issuing I/O in parallel. IOR allows configuration of I/O in terms of the transfer size (size of each I/O transfer unit used by the client), block size (size of the total chunk of data written by each client), number of clients, number of iterations (number of times the same test is run to get a more accurate average I/O performance value), the type of interface (MPIIO, POSIX) among others. IOR also allows the user to specify whether the clients should write to the same file or different files, one per client. In our setup, all the clients write to the same file hosted on a PVFS2 file system.

2.5 Overall I/O path used for benchmarking

The following figure shows the I/O path as the reads and writes are issued by IOR on the PVFS2 client nodes. The MPI I/O calls go through the ROMIO interface. Here, the MPI I/O calls can either go via the PVFS2 library (MPI I/O using PVFS) or via the Kernel VFS layer (MPI I/O using POSIX). The calls that go via the Kernel VFS layer follow the POSIX semantics, wherein the applications issue POSIX calls to the PVFS2 file system mounted as a traditional Unix File System on the client nodes. The native PVFS2 and POSIX calls bypass the ROMIO interface. I/O calls through the kernel VFS layer go to a user-space pvfs2-client process (running on each client node) that converts the calls into low level system interface calls to communicate to the pvfs2-server. In contrast, I/O calls via PVFS2 library go via the *libpvfs* library which converts the native PVFS calls again into low-level system interface calls before communicating to the pvfs2-server. In both cases, the PVFS2 software uses RDMA for communication between the PVFS2 clients and servers. To use RDMA, PVFS2 uses the OpenIB Verbs API. The OpenIB layer sits on top of the InfiniBand hardware, acting as an interface between PVFS2 and the underlying InfiniBand hardware.

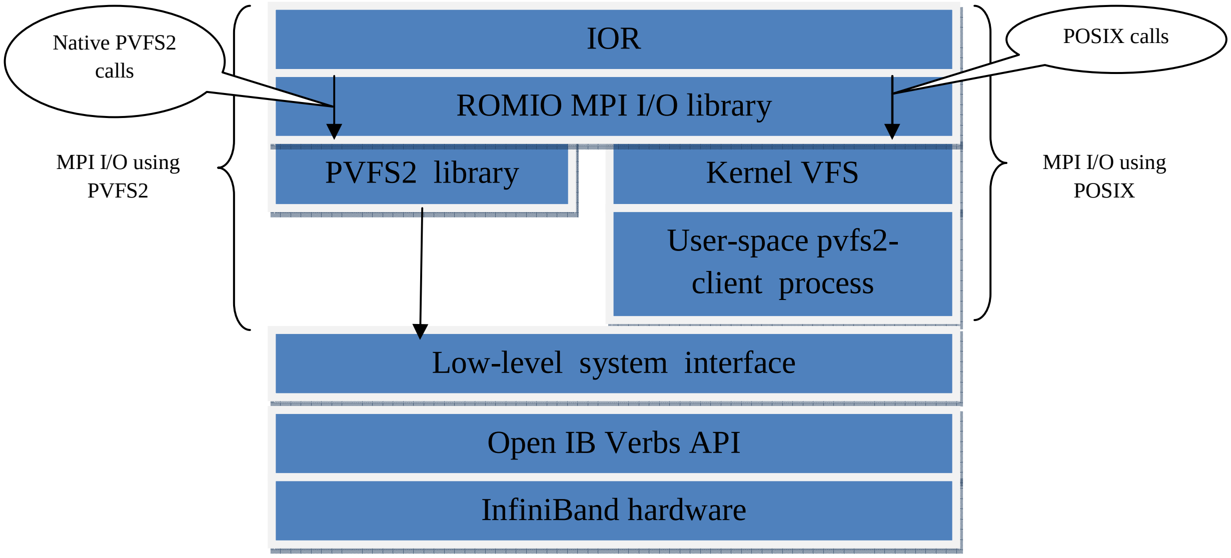

Figure 4. I/O path

CHAPTER 3. SYSTEM CONFIGURATION

3.1 Hardware configuration

The following table shows the hardware configuration of the test cluster called “ibmcluster” at the Scalable Computing Lab, Ames Laboratory, US Department of Energy. The cluster is a non-homogeneous cluster consisting of five AMD Opteron nodes and one Intel Xeon node. The nodes are connected to each other and to the storage disks by InfiniBand channels via 2 InfiniBand switches. All the nodes have a common root file system provided by Andrew File System (AFS). The AFS volume resides on a shared disk.

Table 1. Hardware configuration of test system

Main Memory	4 GB RAM
Processor	AMD Opteron node – Dual processor, 2.4 GHz Intel Xeon node – Dual processor, 2 GHz
Host Channel Adapter card	Mellanox 4X DDR PCI-Express InfiniBand adapter (16 Gbps)
Storage disks hosting the PVFS2 file system	500 GB partition for each node on a 8 disk RAID Set of Seagate SATA HDs
RAID controller for each storage node	2 Areca PCI-X SATA RAID controllers (only one is used for I/O)
Client/Server Interconnect	2 Mellanox 24 port (4X SDR/DDR) switches connected with a 12X DDR interlink (48 Gbps max data payload)

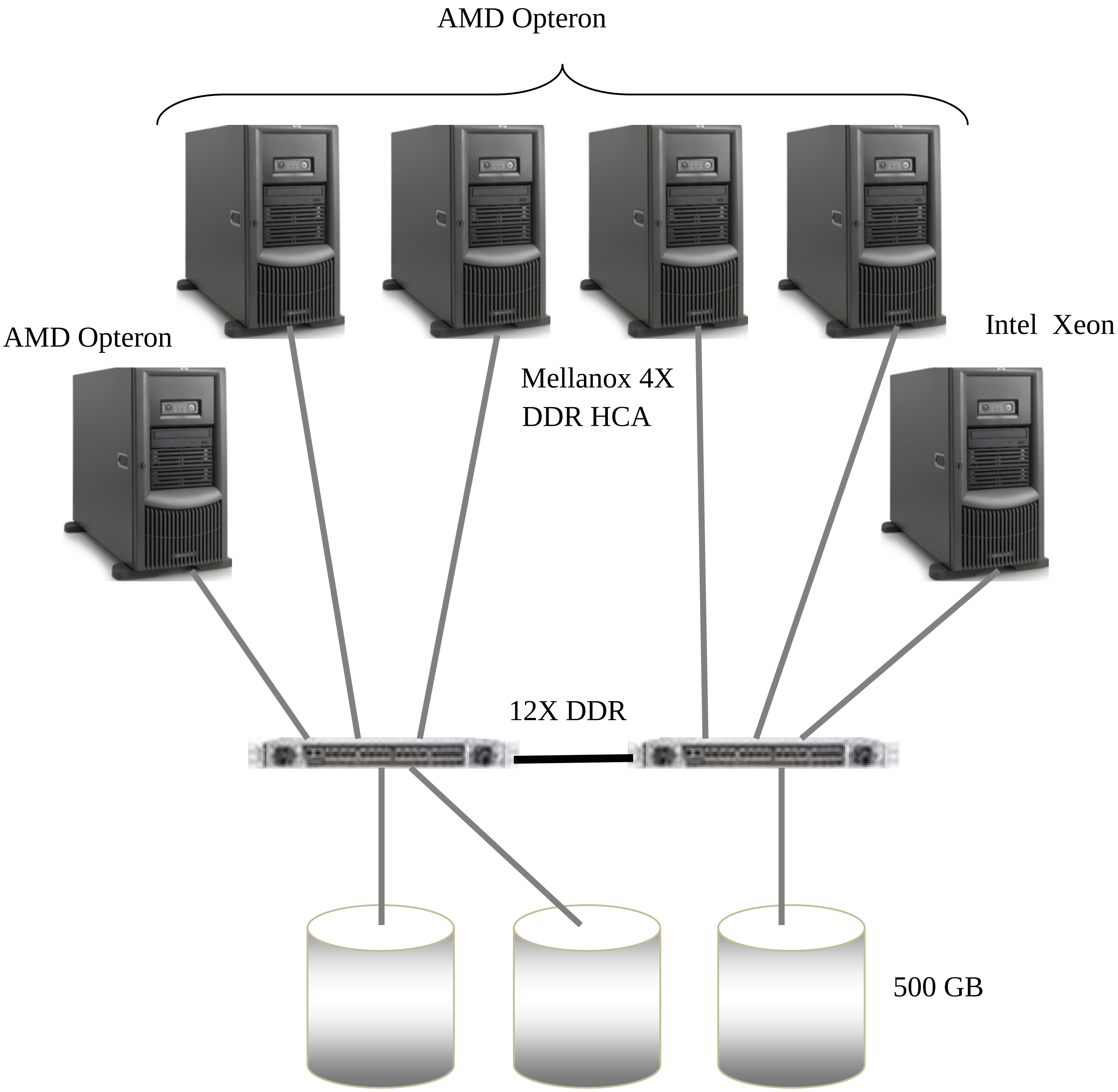

Figure 5. ibmcluster in Ames Lab

3.2 Software configuration

Each node in the PVFS cluster has the following software installed on it:

Table 2. Software configuration of test system

Operating system	Debian GNU/Linux 4.0
PVFS2 version	2.7.1
MPI I/O software	MVAPICH2 1.2
OpenIB version	OFED 1.3
IOR version	2.10.2

For running PVFS2 with InfiniBand, the PVFS2 sources have to be compiled with InfiniBand support. Before compiling PVFS2 with IB, OpenIB must already have been installed in the test systems. PVFS2 binaries link with OpenIB libraries to perform RDMA over the IB fabric. For multiple clients to perform parallel I/O using IOR, an mpd ring must be setup in the cluster. The mpd ring essentially consists of a group of mpd daemons, each of which is running on a node in the PVFS2 cluster. The mpd binary is shipped as part of the MVAPICH2 bundle. To create the mpd ring, we use the “mpdboot” utility. A hostfile listing the names of all the nodes being used as PVFS2 clients is provided as an argument to the mpdboot utility. A run of the “mpdtrace” utility will then list all the client nodes that form the mpd ring. Furthermore, IOR also needs to be built with MVAPICH2. This is done by setting the PATH environment variable to the directory where MVAPICH2 binaries are located, and running the Makefile. In order for IOR to use the POSIX and MPI I/O using POSIX file system interfaces, the PVFS2 file system must be mounted as a Unix file system on the client machines. The PVFS2 kernel module [2] allows mounting the PVFS2 file system as a traditional Unix file system.

CHAPTER 4. BENCHMARKING METHODOLOGY

4.1 Outline of the methodology used

In the test setup, the PVFS2 clients communicate with the PVFS2 servers using the InfiniBand protocol stack, bypassing the TCP/IP stack. To read/write data from/to the PVFS2 servers, applications can use different types of file system interfaces, including POSIX and MPI I/O. As already mentioned, IOR is a popular benchmarking tool for testing parallel filesystems using different interfaces. It is especially used for analyzing multiple client performance doing parallel I/O on a single file. Since our experiments aim to test multiple client I/O performance on PVFS2, we chose IOR to do our benchmarking of PVFS2 on InfiniBand. Presently, IOR only supports MPI I/O and POSIX file system interfaces. We wrote a PVFS2 extension module for IOR to include native PVFS2 support for IOR. This allows IOR to read/write data into the PVFS2 file system using native PVFS2 interfaces. We analyze these different interfaces by running IOR with different I/O message size used for reading or writing PVFS2 data to and from PVFS2 servers, and different number of PVFS2 clients. This allows us to determine the optimum message sizes for different interfaces that would give the maximum read/write performance. We finally make observations on the benchmarking statistics so obtained to determine the best file system interface in terms of offering good read/write bandwidths for application data.

4.2 Experimental Setup

Our test ring consists of 6 servers that double up as both PVFS2 servers and clients. To test PVFS2 with multiple clients, we configure a fixed set of 6 PVFS2 I/O servers and 6 PVFS2 metadata servers. We then vary the number of PVFS2 clients (2, 4 and 6) and run the IOR benchmark.

4.3 Test Strategy

The IOR benchmarking tool uses the sequential access pattern for measuring the read and write bandwidths. In the world of HPC applications, sequential I/O patterns dominate among other access patterns [11]. Using IOR, the PVFS2 clients write data to a single file in parallel, using independent I/O. The different types of I/O access vary from serial (all I/O happens via a single processor), multi-file parallel (each processor does I/O to a separate file) to single-file parallel I/O (multiple processors do I/O to a single file in parallel). The limitation of serial I/O is that it leads to performance bottleneck since all I/O gets routed through a single processor. Also, since the size of the file to be written might exceed the memory capacities of the single processor, I/O cannot take advantage of memory buffers. Similarly, multi-file I/O approach has problems associated with piecing together multiple files into a single file, high metadata overhead and the inherent difficulty in measuring I/O performance. Due to these limitations, single file parallel I/O is the most popular choice in the parallel-programming paradigm [11]. We assumed the total file size written to by the PVFS clients to be 24 GB. This file size was so chosen to offset any impact of buffering either on the client size or at the I/O servers. Each node in our PVFS cluster has a RAM of 4 GB. Since each node acts as both client and server, we assumed a total file size of (6 PVFS servers/clients) * 4 GB = 24 GB. This file size assures us that the I/O is actually hitting the disk, allowing for the measurement of I/O performance more accurately. Since the access is sequential, and each client has its own chunk of file data, it does not make sense to test with collective I/O. For each configuration, we test with multiple message sizes, and see how the read/write performance is affected. The size of I/O transactions used by HPC applications vary across KB to tens of MB. We run our tests on message sizes from 1 MB to 1 GB to capture the effect of message size on read/write performance more effectively, and to determine the optimum message size for peak I/O performance. We restrict the message size at 1 GB since the peak read/write performance is reached before this. We test to see if the ROMIO interface into the PVFS2 file system has any impact on the I/O performance. To this end, we test with MPI I/O using POSIX interfaces into the PVFS2 file system, and MPI I/O using native PVFS interfaces into the PVFS2 file system.

CHAPTER 5. BENCHMARKING RESULTS

5.1 Graph description

The results are shown as line graphs plotted with the read/write bandwidth (in Megabytes per second units) against the I/O message size (in Megabytes). The maximum read/write bandwidth is fixed at 800 MB on the y-axis of each graph. This makes performance comparison easier across different measurements. On the x-axis of each graph, we consider message sizes of 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, 512 MB and 1 GB. Each graph shows plotted lines pertaining to POSIX, MPI I/O using POSIX, MPI I/O using PVFS and native PVFS interfaces. There are different sets of graphs for different client numbers. The plotted lines have different colors and plot points to differentiate between themselves. Figures 6-7 show 2 client read and write bandwidths, figures 8-9 show 4 client read and write bandwidths and figures 10-11 show 6 client read and write bandwidths. For plotting the graphs we used Microsoft Excel software.

5.2 Results

Following are the results captured by our testing:

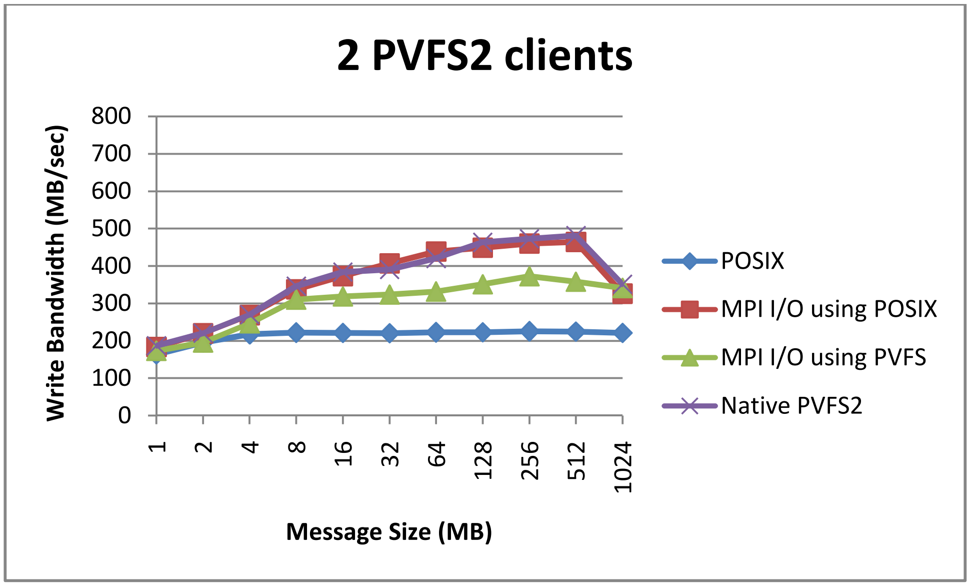

Figure 6. 2-client write bandwidth performance

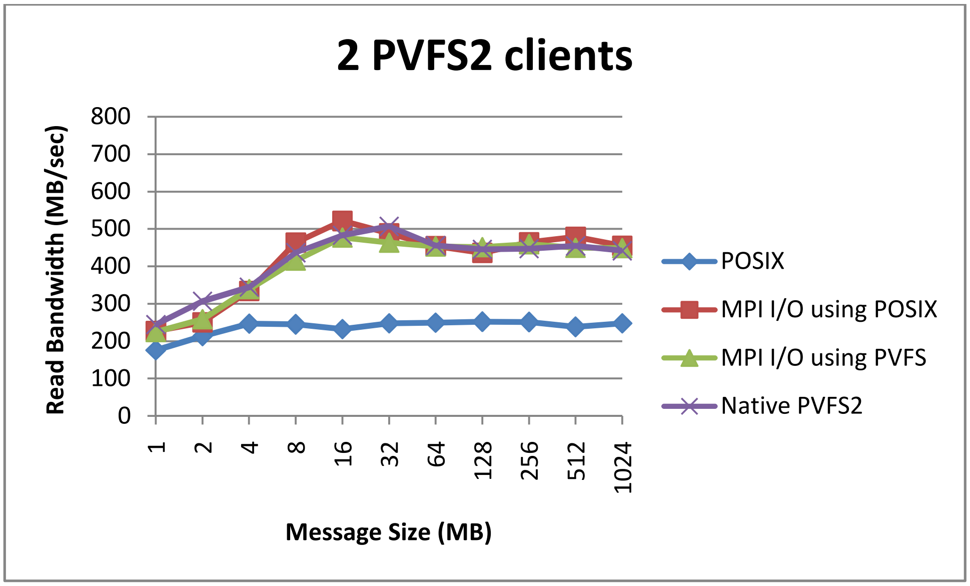

Figure 7. 2-client read bandwidth performance

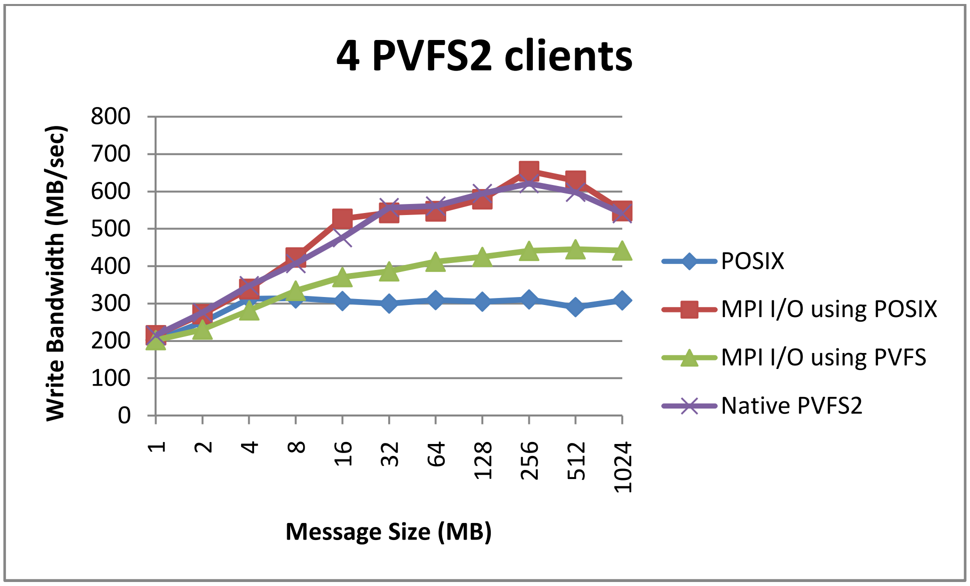

Figure 8. 4-client write bandwidth performance

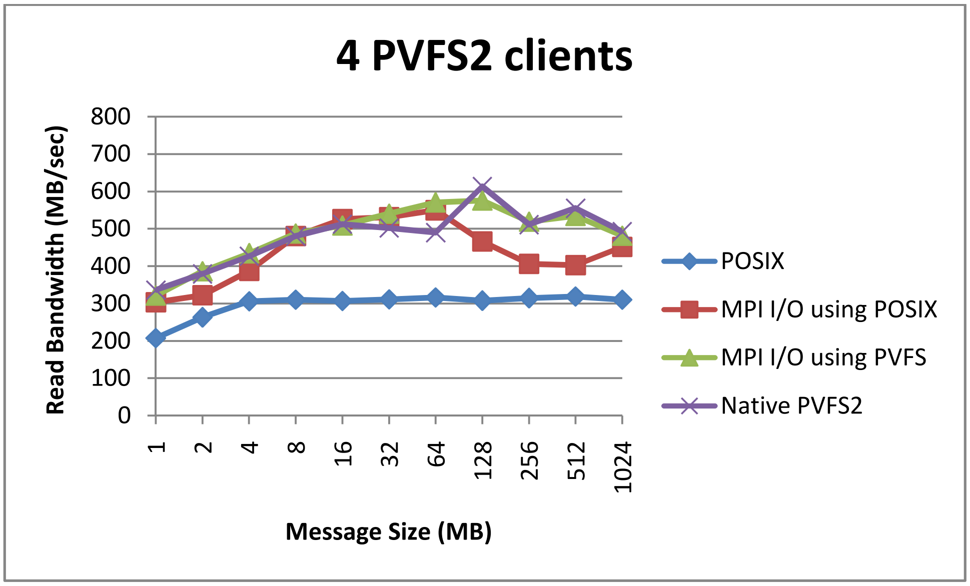

Figure 9. 4-client read bandwidth performance

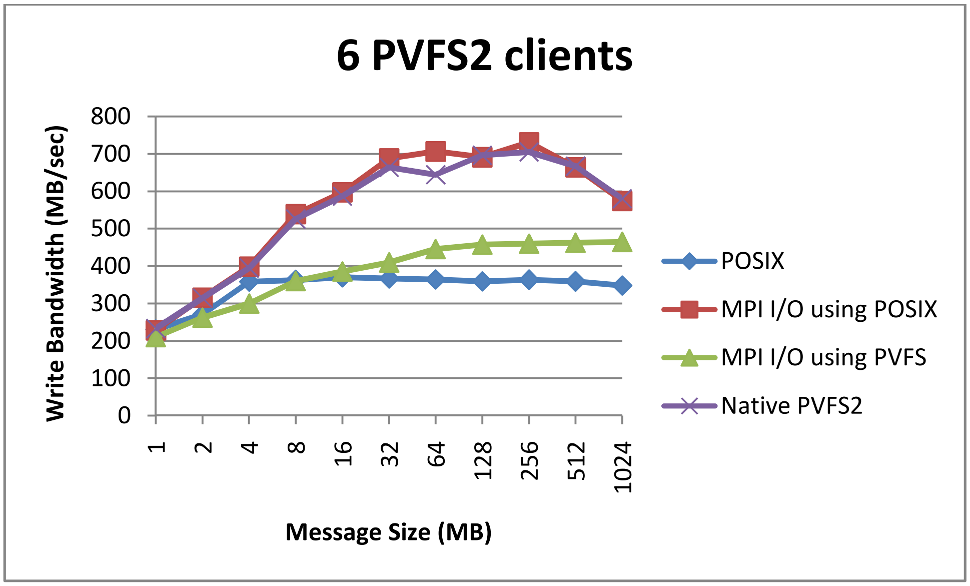

Figure 10. 6-client write bandwidth performance

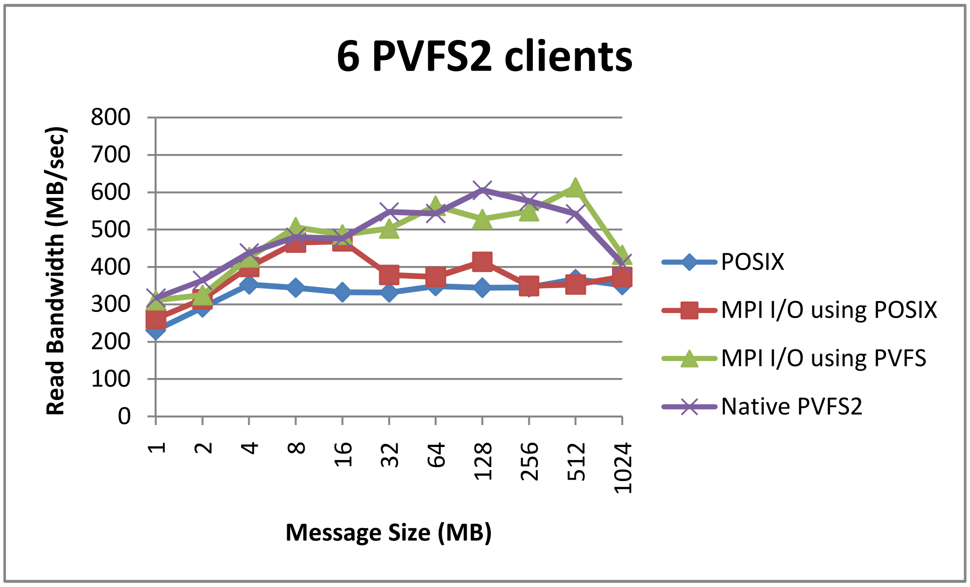

Figure 11. 6-client read bandwidth performance

The following tables (Table 3 and Table 4) show the standard deviation for measuring read and write bandwidth over 3 runs, on each of the file system interfaces. For writes, the standard deviation is very less, on the order of 12 MB/sec, indicating that the bandwidth rates are reliable and predictable. For the reads, the standard deviation is a bit more, especially for the native PVFS interface (this may be because we captured the error margins after the original results were obtained. This may have resulted in a changed system configuration).

Table 3. Standard deviation for writes on 6 clients

Interface	Message size (MB)	Write Bandwidth (MB/sec)			Standard deviation(MB/sec)
		Run #1	Run #2	Run #3	
MPI I/O over PVFS	64	451.35	465.42	471.82	8.38
	128	457.54	456.89	469.66	4.03
	256	460.02	469.66	466.36	3.74
	512	462.33	470.13	470.54	3.77
MPI I/O over POSIX	64	706.23	709.44	681.15	12.55
	128	690.59	695.74	680.59	6.24
	256	730.58	720.03	735.02	6.24
	512	663.73	670.02	659.32	4.55
POSIX	64	364.25	349.77	353.24	6.34
	128	358.79	353.94	349.54	3.68
	256	362.94	355.84	350.75	4.92
	512	358.62	352.85	345.17	5.31
Native PVFS2	64	644.06	637.42	666.94	12.35
	128	695.91	698.15	699.77	1.69
	256	705.05	700.30	697.49	3.30
	512	666.61	661.91	659.51	2.94

Table 4. Standard deviation for reads on 6 clients

Interface	Message size (MB)	Read Bandwidth (MB/sec)			Standard deviation(MB/sec)
		Run #1	Run #2	Run #3	
MPI I/O over PVFS	64	563.30	533.44	532.41	14.38
	128	528.72	564.82	556.91	15.40
	256	548.57	572.47	549.41	11.08
	512	613.42	561.32	554.86	26.31
MPI I/O over POSIX	64	373.69	394.46	400.27	11.57
	128	414.00	408.85	393.68	8.83
	256	349.07	357.95	363.91	5.73
	512	353.18	339.45	363.10	9.84
POSIX	64	348.52	332.72	335.04	6.94
	128	344.18	340.34	333.52	4.54
	256	345.40	337.05	338.29	3.56
	512	367.60	359.02	346.35	8.65
Native PVFS2	64	543.10	520.77	518.14	11.34
	128	605.16	496.51	480.10	55.53
	256	575.90	473.70	482.42	46.10
	512	542.04	478.41	462.71	34.5

5.3 Analysis of the Results

As can be seen in the above results, as far as read performance is concerned, for small message sizes, the low-level interface of MPI I/O to the PVFS file system does not seem to make a difference. Performance is comparable with both MPI I/O over POSIX and MPI I/O over native PVFS2. As the size of the messages increases, the performance of MPI I/O using PVFS starts to improve significantly compared to MPI I/O using POSIX. With 6 PVFS clients, MPI I/O using PVFS hits a peak read bandwidth of ~614 MB/sec at a message size of 512MB. However, MPI I/O using POSIX peaks out at ~469 MB/sec at a message size of 16 MB; for bigger message sizes, its read performance is considerably lower. When IOR uses native PVFS2 interfaces, the read performance is mostly similar to the read performance of MPI I/O using PVFS. For all client numbers, the read performance of MPI I/O over PVFS and native PVFS are almost always similar. With 6 PVFS clients, the native PVFS interface gives a peak read performance of ~606 MB/sec for a message size of 128 MB. The worst read performance is obtained using POSIX interface into the PVFS2 file system. It gives a peak read performance of only ~368 MB/sec for a message size of 512 MB and with 6 PVFS clients.

However, as far as the write performances are concerned, MPI I/O over the POSIX layer offers significantly better performance compared to MPI I/O over the PVFS2 layer. The difference in the write performance between the different MPI I/O stacks becomes bigger and bigger as the number of PVFS2 clients increases. Also, for bigger message sizes, the gap between the write bandwidths gets wider. MPI I/O over PVFS offers a peak write bandwidth of ~370 MB/sec with 6 PVFS2 clients and a message size of 16 MB. In contrast, MPI I/O using POSIX offers a peak bandwidth of ~731 MB/sec, almost double the write bandwidth, for a message size of 256 MB. Native PVFS2 write performance almost exactly matches the write performance of MPI I/O over POSIX. It offers a peak write performance of ~706 MB/sec for a similar message size of 256 MB.

The performance difference between native PVFS2 and POSIX interfaces is because reads and writes in native PVFS2 happen through the user-space on the client side when the data is transferred to the pvfs2-server. In the POSIX model, reads and writes get copied from the user-space to the kernel space and back from kernel space to the user-space before the data gets transmitted to the pvfs2-server. Hence native PVFS2 calls perform better in terms of reads and writes.

The performance difference between MPI I/O over PVFS and MPI I/O over POSIX follows a similar theory. The I/O calls in each of the cases is different. As already discussed in section 2.5, the POSIX semantics use the kernel VFS layer, while the PVFS calls go via the libpvfs2 library. Since MPI I/O is an abstraction layer above POSIX or PVFS2, the I/O path underneath causes difference in their overall performance.

MPI I/O calls in general suffer from the disadvantages of protocol overhead. It however has support for passing hints, like the stripe size, to the underlying file system. We use the default PVFS2 stripe size throughout our tests. The ROMIO implementation of MPI I/O is especially optimized for good parallel performance on PVFS. MPI I/O also has support for pre-fetching data, a feature that POSIX I/O lacks leading to a poor parallel I/O performance for POSIX. POSIX I/O also suffers from drawbacks due to atomic writes, read-after-write consistency and attribute freshness [16]. MPI I/O provides better metadata management. In a POSIX file model, all client processes are forced to open the shared file, causing system call storm, while MPI I/O uses a handle-based model where a single file system handle lookup by a master client node is broadcasted to remaining client nodes [17]. Native PVFS has less protocol overhead, and also provides a much richer API for describing I/O accesses, since it is specifically made for parallel I/O. Also, unlike the POSIX model, it has much lesser metadata overhead compared to MPI I/O. It matches the non-contiguous regions in memory and file more effectively as compared to POSIX. Thus it offers superior read/write performance as our results have shown.

To understand the write performance between different interfaces better, we used the “vmstat” utility to get the virtual memory statistics on the server nodes. Since we used the same nodes as both clients and servers, it didn’t really matter where we ran the vmstat tool. In particular, we found that the number of context switches between different interfaces varied significantly. Following are the results captured for writing a 256 MB message by 6 clients (Table 3). We chose this message size since large messages result in the most performance difference compared to small messages. Table 3 shows the number of context switches happened for each interface when performing writes on the PVFS2 file system.

Table 5. Context Switch counts across interfaces for writes

Interface	Run1	Run2	Run3	Mean	Standard Deviation
MPI I/O over PVFS	152176	143114	157768	151000	6038.12
POSIX	131114	135840	133521	134000	1929.49
MPI I/O over POSIX	109792	103745	113928	109000	4181.52
Native PVFS	100029	99182	102219	100000	1279.62

As seen above, the number of context switches for MPI I/O over POSIX is less compared to MPI I/O over PVFS. This might account for the better write performance of MPI I/O over POSIX as compared to MPI I/O over PVFS. To understand why the context switches are more for MPI I/O over PVFS, we looked at the time spent by the CPU in the user-space and kernel-space while performing I/O, again using the vmstat utility. For 3 runs, following is the mean CPU time with an error margin of less than 2% -

Table 6. Mean CPU time

Interface	User-space	Kernel-space
MPI I/O over PVFS	30%	6%
POSIX	35%	10%
MPI I/O over POSIX	40%	12%
Native PVFS	35%	13%

As seen above, the CPU time spent in I/O for both user-space and kernel-space is less for MPI I/O over PVFS compared to MPI I/O over POSIX. Since the amount of I/O is same for both the cases, it seems that CPU is more actively involved in I/O for MPI I/O using POSIX as compared to MPI I/O over PVFS, thus achieving fewer context switches since CPU is not handling many non I/O tasks while doing I/O. In contrast, for MPI I/O over PVFS, since the CPU is less involved in I/O, it switches contexts to perform other tasks, leading to more context switches. This observation led us to hypothesize that the number of packets generated for MPI I/O over POSIX was perhaps more and with smaller packet sizes as compared to MPI I/O over PVFS. This somehow seems to lead to better parallelization and better batching at the server while performing disk I/O. The PVFS2 server interfaces to disk using POSIX asynchronous I/O routines, which are implemented via glibc using threads and blocking read/write. So if there is more I/O, it appears that the I/O gets batched together better. We believe that the higher IB traffic generated for MPI I/O over POSIX compared to MPI I/O over PVFS leads to a better write performance for larger message sizes. For larger message sizes, MPI I/O over POSIX seems to fragment the application data more leading to a higher number of packet traffic on the InfiniBand interconnect. More packets lead to higher parallelization of the packets to the I/O servers, leading to better performance. On the contrary, the higher number of IB packets generated for MPI I/O over POSIX for larger message sizes leads to a degraded read performance compared to MPI I/O over PVFS. This is because, reading more packets means more overhead for reading the actual file data from the file system.

To test our hypothesis that MPI I/O over POSIX results in more I/O packets compared to MPI I/O over PVFS, we used the “perfquery” tool. But we did not get any conclusive evidence.

The “perfquery” tool is part of the OpenIB diagnostics. The perfquery tool can query the host/switch InfiniBand ports to get a measure of the number of IB packets transmitted and received through these ports. We used the tool to query all the ports of both the IB switches used in our setup to compute the IB traffic generated for one run of the IOR benchmark tool for a 6 PVFS2 client setup. Basically, perfquery will compute the aggregated sum of the IB packets passing through the 24 ports of each of the two IB switches in our test environment. Since the difference in I/O performance was significant mostly for the larger message size, IOR was run with a transfer size of 256 MB. The counts of IB packets were –

MPI I/O and PVFS2 - 244,086,007

MPI I/O and POSIX - 244,241,310

Native PVFS2 - 244,126,754

POSIX - 244,785,626

But the difference between the packet counts was not significant enough to draw any conclusions. There is a possibility that the tool might also contain bugs that would give incorrent results.

In order to test the theory that parallelization helps the write performance for higher number of I/O packets, we reduced the number of I/O servers from 6 to 4 to decrease the degree of I/O parallelization. We then ran IOR with 4 PVFS2 clients on the MPI I/O over POSIX file system interface. IOR would write file of the same size as before (24 GB), but this time on a PVFS2 file system striped across only 4 I/O servers. We then compared the results so obtained with the results of MPI I/O over PVFS performance on 4 PVFS2 clients and 6 PVFS2 servers. We observed that their write bandwidths almost matched each other. Following is the graph of the same –

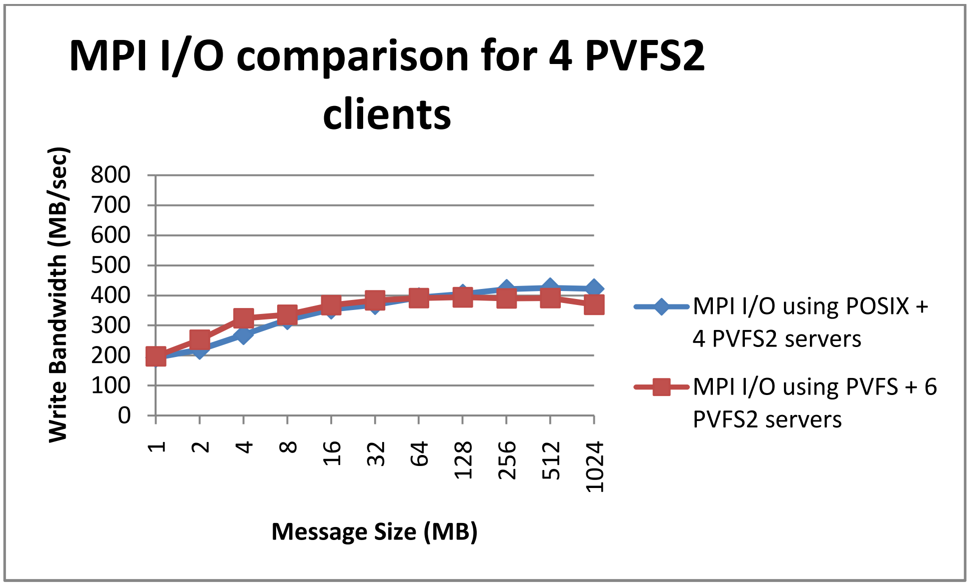

Figure 12. MPI I/O write bandwidth comparison

As seen above,

WriteBandwidth(Loss in the degree of parallelization for MPI I/O over POSIX, Higher number of IB packets) = WriteBandwidth(Higher degree of parallelization for MPI I/O over PVFS, Lesser number of IB packets).

Native PVFS2 outperforms MPI I/O over PVFS2 in terms of writes since it involves less redundant buffering in the I/O path on account of less protocol overhead. For larger messages, the write performance of MPI I/O over PVFS gets saturated perhaps on account of lack of support for transfer of large messages as a unit. In other words, the MPI I/O layer could be fragmenting the larger packets, leading to saturation in performance. As far as MPI I/O over POSIX is concerned, the performance benefit offered perhaps by its higher packet count matches with the reduced protocol overhead and support for larger message size of native PVFS2. Hence their performances are almost similar to each other. We have already discussed the limitations of POSIX for parallel I/O and our results reflect the limitations. In the current work, we have not been able to exactly determine the causes leading to the performance differences between different interfaces. We hope that in our future work, we can isolate the causes more effectively to explain the performance results and also help improve I/O performance.

CHAPTER 6. CONCLUSION

In this thesis we studied the I/O performance of PVFS2 over the InfiniBand technology. In particular, we studied the performance of concurrent reads and writes by multiple processes, characterizing the typical parallel I/O access pattern used in parallel applications. We examined in detail the effects of different application programming interfaces into the PVFS2 file system by making use of the IOR parallel I/O microbenchmark. In addition to the POSIX and MPI I/O interfaces supported by IOR, we extended its functionality to include a PVFS2 module allowing IOR to access the PVFS2 file system using native PVFS2 interfaces. To study the effects of different MPI I/O stacks on the I/O performance of a PVFS2 file system, we ran our benchmarks in the MPI I/O over POSIX as well as MPI I/O over PVFS environments. We also tested I/O performance with different I/O transfer unit sizes to find out the optimum I/O message size for each configuration.

Our results showed that MPI I/O over PVFS fetches better read performance for large I/O message sizes, while MPI I/O over POSIX fetches better write performance for large I/O message sizes. We also showed that a native PVFS2 interface performs really well in both reads and writes in a multi-client parallel I/O scenario. With these results, on the whole we characterize the performance of PVFS2 on the InfiniBand interconnect by achieving impressive I/O performance results. By extending IOR functionality to include PVFS2 support, we make available a single parallel I/O benchmarking tool suitable for comparing different file system interfaces in multiple client experiments in a PVFS2 environment. Using this work, parallel applications can configure their I/O environment according to the performance parameters we identified and get better I/O performance.

CHAPTER 7. FUTURE WORK

We showed our benchmarking results obtained from the IOR benchmarking tool. Now that we have identified the application programming interface and the I/O transaction unit size that will most benefit reads and writes in a PVFS2/InfiniBand environment, the next step is to run a real-world parallel application with these performance parameters as a case study. A measurement of the I/O efficiency of such an application will prove the effectiveness of our analysis in a real-world scenario. In this thesis, we measured PVFS2 performance on native InfiniBand, where PVFS2 clients communicate with PVFS2 servers using native InfiniBand calls. It may be worthwhile to test PVFS2 with clients and servers communicating using the IP over IB protocol stack, wherein the communication happens using the regular TCP/IP stack on InfiniBand as the physical medium. We are also planning to benchmark I/O performance on a Lustre file system that is currently being setup on the "ibmcluster" in Ames Lab. The goal is to do a comparative study of PVFS2 I/O performance and Lustre I/O performance on InfiniBand. Since this kind of comparative study does not exist as of today to our best knowledge, we hope that this will give us some interesting insights. Also, as we have already mentioned, we hope to analyze the performance of different interfaces more effectively to isolate the exact causes of performance bottlenecks. Lastly, in the pipeline is a asynchronous I/O implementation with NetPipe wherein application I/O overlaps with PVFS2 client/server I/O. Such an implementation might offer a significantly better parallel I/O performance.

BIBLIOGRAPHY

[1] InfiniBand Trade Association. http://www.infinibandta.com .

[2] Parallel Virtual File System. http://www.pvfs.org .

[3] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317-327, Atlanta, GA, 2000. USENIX Association.

[4] Y. Tsujita: Implementation of an MPI I/O Mechanism Using PVFS in Remote I/O to a PC Cluster. In *Proceedings of the High Performance Computing and Grid in Asia Pacific Region, 7th International Conference*, pages 136-139, 2004. IEEE Computer Society.

[5] ROMIO. http://www-unix.mcs.anl.gov/romio .

[6] R. Thakur, W. Gropp, and E. Lusk, An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces, in *Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180-187, 1996.

[7] J. Wu, P. Wyckoff, D. K. Panda: PVFS over InfiniBand: design and performance evaluation. In *Proceedings of the International Conference on Parallel Processing*, pages 125-132, 2003.

[8] L. Chai, X. Ouyang, R. Noronha, D. K. Panda: pNFS/PVFS2 over InfiniBand: early experiences. In *Proceedings of the 2nd international workshop on Petascale data storage*, pages 5-11, 2007.

[9] Iozone Filesystem Benchmark. http://www.iozone.org .

[10] Effective I/O Bandwidth Benchmark.

http://www.hlrs.de/organization/par/services/models/mpi/b_eff_io .

[11] H. Shan and J. Shalf: Using IOR to analyze the I/O performance of HPC platforms. In *Cray Users Group Meeting (CUG) 2007*, Seattle, Washington, May 7-10, 2007.

[12] Peter Corbett, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, Parkson Wong, and Dror Feitelson: *MPI-IO: A parallel file I/O interface for MPI*, version 0.4. http://lovelace.nas.nasa.gov/MPI-IO, December 1995.

[13] IOR, http://www.llnl.gov/asci/purple/benchmarks/limited/ior .

[14] OpenIB, http://www.openib.org .

[15] HPC Analytics, Arizona State University, http://plato.asu.edu/slides/stanzione.pdf .

[16] Clustered and Parallel Storage System Technologies FAST09,

http://www.usenix.org/events/fast09/tutorials/T1.pdf .

[17] R. Ross: PVFS2 and Parallel I/O on BG/L. Invited talk at *Third BG/L Systems Software and Applications Workshop 2006*, Tokyo, Apr 19-20, 2006; see

http://www.cbrc.jp/symposium/bg2006/PDF/Ross.pdf .

APPENDIX

IOR source changes for PVFS2 support.

Makefile changes

```
...
posix:      $(OBJS) aiori-POSIX.o aiori-noPVFS2.o aiori-noMPIIO.o \
            aiori-noHDF5.o aiori-noNCMPI.o \
            $(CC) -o IOR $(OBJS) \
                  aiori-POSIX.o aiori-noPVFS2.o aiori-noMPIIO.o \
                  aiori-noHDF5.o aiori-noNCMPI.o \
                  $(LDFLAGS)
pvfs2:      $(OBJS) aiori-PVFS2.o aiori-POSIX.o aiori-noMPIIO.o \
            aiori-noHDF5.o aiori-noNCMPI.o \
            $(CC) -o IOR $(OBJS) \
                  aiori-PVFS2.o aiori-POSIX.o aiori-noMPIIO.o \
                  aiori-noHDF5.o aiori-noNCMPI.o \
                  -I$(PVFS_INCLUDE) $(LDFLAGS) $(PVFS_LDFLAGS) -lpvfs2
mpiio:      $(OBJS) aiori-POSIX.o aiori-PVFS2.o aiori-MPIIO.o \
            aiori-noHDF5.o aiori-noNCMPI.o \
            $(CC) -g -o IOR $(OBJS) \
                  aiori-POSIX.o aiori-PVFS2.o aiori-MPIIO.o \
                  aiori-noHDF5.o aiori-noNCMPI.o \
                  $(LDFLAGS)
...
```

IOR.c changes

This is the main benchmarking logic that tests file system I/O.

sudhindra@da12:/IO-tools/IOR-2.10.2/src/C$ diff IOR.c.PVFS IOR.c.withoutPVFS

```
187,195d186
< } else if (strcmp(api, "PVFS2") == 0) {
< IOR_Create         = IOR_Create_PVFS2;
< IOR_Open           = IOR_Open_PVFS2;
< IOR_Xfer           = IOR_Xfer_PVFS2;
< IOR_Close          = IOR_Close_PVFS2;
< IOR_Delete         = IOR_Delete_PVFS2;
< IOR_SetVersion     = IOR_SetVersion_PVFS2;
< IOR_Fsync          = IOR_Fsync_PVFS2;
< IOR_GetFileSize    = IOR_GetFileSize_PVFS2;
```

aiori.h changes

This is the header file containing the definitions and prototypes needed for the abstract I/O interfaces invoked by IOR.

sudhindra@da12:/IO-tools/IOR-2.10.2/src/C$ diff aiori.h.pvfs aiori.h.withoutPVFS

```
202,212d201
< /* PVFS2-specific functions */
< void *IOR_Create_PVFS2(char *, IOR_param_t *);
< void *IOR_Open_PVFS2(char *, IOR_param_t *);
< IOR_offset_t IOR_Xfer_PVFS2(int, void *, IOR_size_t *,
<                              IOR_offset_t, IOR_param_t *);
< void IOR_Close_PVFS2(void *, IOR_param_t *);
< void IOR_Delete_PVFS2(char *, IOR_param_t *);

< void IOR_SetVersion_PVFS2(IOR_param_t *);

< void IOR_Fsync_PVFS2(void *, IOR_param_t*);

< IOR_offset_t IOR_GetFileSize_PVFS2(IOR_param_t *, MPI_Comm, char *);
```

aiori-PVFS2.c

This file contains the implementation of abstract I/O interfaces for PVFS2.

```
#include "aiori.h"            /* abstract IOR interface */

#ifdef __linux__

#  include <sys/ioctl.h>    /* necessary for: */

#  define __USE_GNU         /* O_DIRECT and */

#  include <fcntl.h>        /* IO operations */

#  undef __USE_GNU

#endif                      /* __linux__ */

#include <errno.h>          /* sys_errlist */

#include <fcntl.h>          /* IO operations */

#include <stdio.h>          /* only for fprintf() */

#include <stdlib.h>

#include <sys/stat.h>

#include <unistd.h>

#include <pvfs2.h>

#include <limits.h>

#include <string.h>

#include <sys/time.h>

#include <sys/types.h>

#include <sys/stat.h>
```

```
#include <time.h>

#include <libgen.h>

#define STRIP_SIZE -1

#define NUM_DATAFILES -1

/*******************P R O T O T Y P E S*********************/

/******************D E C L A R A T I O N S*******************/

extern int errno;

extern int rank;

extern int rankOffset;

extern int verbose;

extern MPI_Comm testComm;

int flag = 0;

PVFS_credentials credentials;

PVFS_object_ref ref;

PVFS_fs_id fs_id;

/********************F U N C T I O N S**********************/

/***********************************************************/

/*

 * Create and open a file through the PVFS2 interface.

*/

void make_attribs(PVFS_sys_attr *attr, PVFS_credentials *credentials,

                  int nr_datafiles, int mode)

{

      attr->owner = credentials->uid;

      attr->group = credentials->gid;

      attr->perms = PVFS_util_translate_mode(mode, 0);

      attr->atime = time(NULL);

      attr->mtime = attr->atime;
      attr->ctime = attr->atime;

      attr->mask = (PVFS_ATTR_SYS_ALL_SETABLE);

      attr->dfile_count = nr_datafiles;

      if (attr->dfile_count > 0)

      {

            attr->mask |= PVFS_ATTR_SYS_DFILE_COUNT;

      }

} /* make_attribs */

void *IOR_Create_PVFS2(char *testFileName, IOR_param_t *param)

{

      PVFS_sys_attr attr;

      PVFS_permissions perms;

      PVFS_sysresp_lookup resp_lookup;

      PVFS_sysresp_getattr resp_getattr;

      PVFS_sysresp_create resp_create;

      PVFS_object_ref parent_ref;

      PVFS_sys_dist *new_dist;

      int ret = 0;

      char pvfs2_path[PVFS_NAME_MAX];

      char   *entry_name;            /* name of the pvfs2 file */

      char  str_buf[PVFS_NAME_MAX]; /* basename of the pvfs2 file */

      /* so things like debug files go the right place */

      if (!flag)

      {

            ret = PVFS_util_init_defaults();

            if (ret < 0)

            {

                  ERR("PVFS_util_init_defaults");
            }
            flag = 1;
      }
      /* Translate path into pvfs2 relative path */
      ret = PVFS_util_resolve(testFileName, &fs_id, pvfs2_path,
                              PVFS_NAME_MAX);
      if (ret < 0)
      {
            ERR("Unable to map requested name to a pvfs2 file\n");
      }
      PVFS_util_gen_credentials(&credentials);
      entry_name = str_buf;
      if (PINT_remove_base_dir(pvfs2_path, str_buf, PVFS_NAME_MAX))
      {
            if(pvfs2_path[0] != '/')
            {
                  ERR("Error: poorly formatted path.\n");
            }
            ERR("Error: cannot retrieve entry name for creation");
      }
      ret = PINT_lookup_parent(pvfs2_path, fs_id, &credentials,
                               &parent_ref.handle);
      if (ret < 0)
      {
            ERR("PVFS_util_lookup_parent");
      }
      /*we are always dealing with a dest full path with file name */
      parent_ref.fs_id = fs_id;
```

```
memset(&resp_lookup, 0, sizeof(PVFS_sysresp_lookup));
if (!param->filePerProc && rank != 0)
{
      MPI_CHECK(MPI_Barrier(testComm), "barrier error");
}
ret = PVFS_sys_ref_lookup(parent_ref.fs_id, entry_name,
                          parent_ref, &credentials, &resp_lookup,
                          PVFS2_LOOKUP_LINK_FOLLOW);
if (ret == 0)
{
      /* file exists, open it */
      ref = resp_lookup.ref;
}
else
{
      int nr_datafiles = NUM_DATAFILES;
      PVFS_size stripe_size = STRIP_SIZE;
      PVFS_sys_dist *new_dist;
      PVFS_sysresp_create resp_create;
      PVFS_sys_layout layout;
      make_attribs(&attr, &credentials, NUM_DATAFILES,
                  (int)(S_IFREG | S_IRUSR | S_IWUSR));
      if (stripe_size > 0)
      {
            new_dist = PVFS_sys_dist_lookup("simple_stripe");
            ret = PVFS_sys_dist_setparam(new_dist, "strip_size",
                                         &stripe_size);
            if (ret < 0)
```

```
                  {
                        ERR("PVFS_sys_dist_setparam");
                  }
            }
            else
            {
                  new_dist = NULL;
            }
            layout.algorithm = PVFS_SYS_LAYOUT_NONE;
            ret = PVFS_sys_create(entry_name, parent_ref, attr,
                                 &credentials, new_dist, &layout,
                                 &resp_create);
            if (ret < 0)
            {
                  ERR("PVFS_sys_create");
            }
            ref = resp_create.ref;
      }
      if (rank == 0)
            MPI_CHECK(MPI_Barrier(testComm), "barrier error");
      return (void *)fs_id;
} /* IOR_Create_PVFS2() */

/****************************************************************/
/*
* Open a file through the PVFS2 interface.
 */
void *IOR_Open_PVFS2(char *testFileName, IOR_param_t * param)
```

```
{
      return((void *)fs_id);
} /* IOR_Open_PVFS2() */
/**************************************************************/
/*
* Write or read access to file using the PVFS2 interface.
*/
IOR_offset_t IOR_Xfer_PVFS2(int access,
                            void *file,
                            IOR_size_t   * buffer,
                            IOR_offset_t   length,
                            IOR_param_t  * param)
{
      char *ptr = (char *)buffer;
      int ret = 0;
      PVFS_Request mem_req, file_req;
      PVFS_sysresp_io resp_io;
      file_req = PVFS_BYTE;
      ret = PVFS_Request_contiguous(length, PVFS_BYTE, &mem_req);
      if (ret < 0)
      {
            ERR("PVFS_Request_contiguous");
      }
      if (access == WRITE)
      {
            ret = PVFS_sys_write(ref, file_req, param->offset,
                                 ptr, mem_req, &credentials, &resp_io);
            if (ret == 0)
            {
                  PVFS_Request_free(&mem_req);
            }
            else
            {
                  ERR("PVFS_sys_write");
            }
      }
      else
      {
            ret = PVFS_sys_read(ref, file_req, param->offset,
                                ptr, mem_req, &credentials, &resp_io);
            if (ret == 0)
            {
                  PVFS_Request_free(&mem_req);
            }
            else
            {
                  ERR("PVFS_sys_read");
            }
      }
      return(length);
} /* IOR_Xfer_PVFS2() */
/***********************************************************/
/*
* Perform fsync().
*/
void IOR_Fsync_PVFS2(void *fd, IOR_param_t *param)
```

```
{
} /* IOR_Fsync_PVFS2() */
/***********************************************************/
/*
* Close a file through the POSIX interface.
 */
void IOR_Close_PVFS2(void *fd, IOR_param_t *param)
{
} /* IOR_Close_PVFS2() */

/********************************************************/
/*
 * Delete a file through the PVFS2 interface.
*/
void IOR_Delete_PVFS2(char * testFileName, IOR_param_t * param)
{
      int rc = 0, num_segs = 0;
      char filename[PVFS_SEGMENT_MAX];
      char directory[PVFS_NAME_MAX];
      PVFS_fs_id cur_fs;
      PVFS_sysresp_lookup resp_lookup;
      PVFS_object_ref parent_ref;
      char pvfs2_path[PVFS_NAME_MAX];
      if (!flag)
      {
            rc = PVFS_util_init_defaults();
            if (rc < 0)
            {
```

```
            ERR("PVFS_util_init_defaults");
      }
      flag = 1;
}
PVFS_util_gen_credentials(&credentials);
/* Translate path into pvfs2 relative path */
rc = PVFS_util_resolve(testFileName, &cur_fs, pvfs2_path,
                       PVFS_NAME_MAX);
if (rc < 0)
{
      ERR("PVFS_util_resolve");
}
// break into file and directory
rc = PINT_get_base_dir(pvfs2_path, directory, PVFS_NAME_MAX);
if (rc < 0)
{
      ERR("PINT_get_base_dir");
}
num_segs = PINT_string_count_segments(testFileName);
rc = PINT_get_path_element(testFileName, num_segs - 1, filename,
                           PVFS_SEGMENT_MAX);
if (rc)
{
      ERR("Unknown file path format");
}
memset(&resp_lookup, 0, sizeof(PVFS_sysresp_lookup));
rc = PVFS_sys_lookup(cur_fs, directory, &credentials,
                     &resp_lookup, PVFS2_LOOKUP_LINK_NO_FOLLOW);
      if (rc)
      {
            ERR("PVFS_util_resolve");
      }
      parent_ref = resp_lookup.ref;
      rc = PVFS_sys_remove(filename, parent_ref, &credentials);
      if (rc)
      {
            ERR("Error: An error occurred while removing file");
      }
} /* IOR_Delete_PVFS2() */
/************************************************************/
/*
 * Determine api version.
*/
void IOR_SetVersion_PVFS2(IOR_param_t *test)
{
      strcpy(test->apiVersion, test->api);
} /* IOR_SetVersion_PVFS2() */

/********************************************************/
/*
* Use PVFS2_sys_attr to return aggregate file size.
*/
IOR_offset_t IOR_GetFileSize_PVFS2(IOR_param_t *test,
                                   MPI_Comm testComm,
                                   char *testFileName)
{
```

```
IOR_offset_t aggFileSizeFromStat, tmpMin, tmpMax, tmpSum;

int ret = 0;

PVFS_sys_attr *attr;

PVFS_sysresp_getattr getattr_response;

memset(&getattr_response,0, sizeof(PVFS_sysresp_getattr));

PVFS_util_gen_credentials(&credentials);

ret = PVFS_sys_getattr(ref, PVFS_ATTR_SYS_ALL_NOHINT,

                       &credentials, &getattr_response);

if (ret < 0)

{

      ERR("PVFS_sys_getattr");

}

attr = &getattr_response.attr;

aggFileSizeFromStat = attr->size;

if (test->filePerProc == TRUE)

{

      MPI_CHECK(MPI_Allreduce(&aggFileSizeFromStat, &tmpSum, 1,

                MPI_LONG_LONG_INT, MPI_SUM, testComm),

                "cannot total data moved");

      aggFileSizeFromStat = tmpSum;

}

else

{

      MPI_CHECK(MPI_Allreduce(&aggFileSizeFromStat, &tmpMin, 1,

                               MPI_LONG_LONG_INT, MPI_MIN,

                               testComm),

                               "cannot total data moved");

      MPI_CHECK(MPI_Allreduce(&aggFileSizeFromStat, &tmpMax, 1,
                                          MPI_LONG_LONG_INT, MPI_MAX,
                                          testComm),
                                          "cannot total data moved");
                  if (tmpMin != tmpMax)
                  {
                        if (rank == 0)
                        {
                           WARN("inconsistent file size by different tasks");
                        }
                        /* incorrect, but now consistent across tasks */
                        AggFileSizeFromStat = tmpMin;
                  }
             }
            return(aggFileSizeFromStat);
      } /* IOR_GetFileSize_PVFS2() */
```

benchmark script

This is the benchmarking script we used to capture the I/O performance of our experimental setup.

```
#!/bin/sh

echo "MPI I/O over PVFS"

i=1
size=1024
```

```
while [ $i -lt 2 ]; do


        sleep 30

        new_size="$size"M


        echo "Testing with $new_size"


        j=0


        while [ $j -lt 5 ]; do

                /usr/src/mvapich2-1.2rc2IB/bin/mpiexec -n 6 \

                /IO-tools/IOR-2.10.2/src/C/IOR -a MPIIO -t $new_size -b 4G \

                -i 5 -o pvfs2:/mnt/pvfs2/mpiio_pvfs2_krsna

                sleep 10

                j=`expr $j + 1`

                rm -rf /mnt/pvfs2/*

        done



        echo "******Run $i ended*******"


        i=`expr $i + 1`

        size=`expr $size \* 2`

done


echo "MPI I/O over POSIX"


i=1

size=1024
```

```
while [ $i -lt 2 ]; do
        sleep 30
        new_size="$size"M

        echo "Testing with $new_size"

        j=0

        while [ $j -lt 5 ]; do
                /usr/src/mvapich2-1.2rc2IB/bin/mpiexec -n 6 \
                /IO-tools/IOR-2.10.2/src/C/IOR -a MPIIO -t $new_size -b 4G \
                -i 5 -o /mnt/pvfs2/mpiio_posix_krsna
                sleep 10
                j=`expr $j + 1`
                rm -rf /mnt/pvfs2/*
        done

        echo "******Run $i ended*******"

        i=`expr $i + 1`
        size=`expr $size \* 2`
done

echo "POSIX"

i=1
size=1024
```

```
while [ $i -lt 2 ]; do
        sleep 30
        new_size="$size"M

        echo "Testing with $new_size"

        j=0

        while [ $j -lt 5 ]; do
                /usr/src/mvapich2-1.2rc2IB/bin/mpiexec -n 6 \
                /IO-tools/IOR-2.10.2/src/C/IOR -a POSIX -t $new_size -b 4G \
                -i 5 -o /mnt/pvfs2/posix_krsna
                sleep 10
                j=`expr $j + 1`
                rm -rf /mnt/pvfs2/*
        done

        echo "******Run $i ended*******"

        i=`expr $i + 1`
        size=`expr $size \* 2`
done

echo "Native PVFS2"

i=1
size=1024

while [ $i -lt 2 ]; do
```

```
        sleep 30
        new_size="$size"M

        echo "Testing with $new_size"

        j=0

        while [ $j -lt 5 ]; do
                /usr/src/mvapich2-1.2rc2IB/bin/mpiexec -n 6 \
                /IO-tools/IOR-2.10.2/src/C/IOR -a PVFS2 -t $new_size -b 4G \
                -i 5 -o /mnt/pvfs2/pvfs2_krsna
                sleep 10
                j=`expr $j + 1`
                rm -rf /mnt/pvfs2/*
        done

        echo "******Run $i ended*******"

        i=`expr $i + 1`
        size=`expr $size \* 2`
done
```

perfquery/vmstat script

This is the script we wrote to capture the count of IB traffic for each interface, and to get virtual memory statistics.

```
#!/bin/sh
# reset all counters
```

```
reset_counters() {

        echo "### & reset all counters"

        perfquery -r -e -a 4

        perfquery -r -e -a 5

        perfquery -r -e -a 6

}

# function to query port counters

dump_counters() {

        echo "### da12 (lid 8) port on switch 4"

        perfquery -e 4 4

        echo "### 12XDDR link from switch 4 to switch 5"

        perfquery -e 4 22

        echo "### 12XDDR link from switch 5 to switch 4"

        perfquery -e 5 22

        echo "### 12XDDR link from switch 5 to switch 6"

        perfquery -e 5 10

        echo "### 12XDDR link from switch 6 to switch 5"

        perfquery -e 6 13

}


echo "MPI I/O over PVFS"

vmstat 1 &

i=0

while [ $i -lt 5 ]; do

        sleep 30

        reset_counters

        dump_counters

        /usr/src/mvapich2-1.2rc2IB/bin/mpiexec -n 6 \

        /IO-tools/IOR-2.10.2/src/C/IOR -a MPIIO -t 256M -b 4G -i 1 -w -o \
        pvfs2:/mnt/pvfs2/mpiio_pvfs2
        dump_counters
        echo "Run $i ended"
        i=`expr $i + 1`
done
echo "MPI I/O over POSIX"
i=0
while [ $i -lt 5 ]; do
        sleep 30
        reset_counters
        dump_counters
        /usr/src/mvapich2-1.2rc2IB/bin/mpiexec -n 6 \
        /IO-tools/IOR-2.10.2/src/C/IOR -a MPIIO -t 256M -b 4G -i 1 -w –o \
         /mnt/pvfs2/mpiio_posix
        dump_counters
        echo "Run $i ended"
        i=`expr $i + 1`
done
kill %1
```