

# A Windows-Based Parallel File System

Lungpin Yeh, Juei-Ting Sun, Sheng-Kai Hung, and Yarsun Hsu

Department of Electrical Engineering,  
National Tsing Hua University HsinChu, 30013, Taiwan  
{lungpin, posh, phinex}@hpcc.ee.nthu.edu.tw, yhsu@ee.nthu.edu.tw

**Abstract.** Parallel file systems are widely used in clusters to provide high performance I/O. However, most of the existing parallel file systems are based on UNIX-like operating systems. We use the Microsoft .NET framework to implement a parallel file system for Windows. We also implement a file system driver to support existing applications written with Win32 APIs. In addition, a preliminary MPI-IO library is also developed. Applications using MPI-IO could achieve the best performance using our parallel file system, while the existing binaries could benefit from the system driver without any modifications. In this paper, the design and implementation of our system are described. File system performance using our preliminary MPI-IO library and system driver is also evaluated. The results show that the performance is scalable and limited by the network bandwidth.

## 1 Introduction

As the speed of CPU becomes faster, we might expect that the performance of a computer system should benefit from the advancement. However, the improvements of other components in a computer system (i.e. memory system, data storage system) cannot catch up with that of CPU. Although the capacity of a disk has grown with time, its mechanical nature limits its read/write performance. In this data-intensive world, it is significant to provide a large storage subsystem with high performance I/O[1]. Using a single disk with a local file system to sustain this requirement is impossible nowadays. Disks combined either tightly or loosely to form a parallel system provide a possible solution to this problem. The success of a parallel file system comes from the fact that accessing files through network can have higher throughput than fetching files through local disks. This could be attributed to the emergence of high-speed networks such as Myrinet [2], InfiniBand [3], Gigabit Ethernet, and more recently 10 Gigabit Ethernet.

A parallel file system can not only provide a large storage space by combining several storage resources on different nodes but also increase the performance. It could provide high-speed data access by using several disks at the same time. With suitable striping size, the workload in the system can be distributed among these disks instead of being centralized in a single disk. For example, whenever a write happens, a parallel file system would split these data into a lot of small

chunks, which are then stored on different disks across the network in a round-robin fashion.

Most of parallel file systems are based on Unix or Linux. As far as we know, WinPFS[4] is the only parallel file system based on Microsoft Windows. However, it does not allow users to specify the striping size of a file across nodes. Furthermore, it does not provide a user level library for high performance parallel file access. In this paper, we implement a parallel file system for Microsoft Windows Server 2003 allowing users the flexibility to specify different striping size. Users can specify the striping size to satisfy the required distribution or using the default striping size provided by the system. We have implemented a file system driver to trap Win32 APIs such that existing binaries can access files stored on our parallel file system without recompilation. Besides, some MPI-IO functions (such as noncontiguous accesses) are also provided for MPI jobs to achieve the best performance. We have successfully used our parallel file system as a storage system for VOD (Video On Demand) services, which can deliver the maximum bandwidth and demonstrate the successful implementation of our parallel file system.

This paper is organized as follows: Section 2 presents some related works. Design and implementation will be discussed in section 3, with the detailed description of our system driver. Section 4 depicts the results of performance evaluation of our windows-based parallel file system, along with the prototype VOD system. Finally, we would make some conclusions and provide some directions in section 5.

## 2 Related Works

PVFS[5,6] is a parallel file system publicly available in the Linux environment. It provides both user level library for performance and a kernel module package that makes existing binaries working without recompiling.

WinPFS [4] is a parallel file system for Windows and integrated within the Windows kernel components. It uses the existing client and server pairs in the Windows platform (i.e. NFS [7], CIFS [8], ...) and thus no special servers are needed. It also provides a transparent interface to users, just like what does when accessing normal files. The disadvantage is that the user can not specify the striping size of a file across nodes. Besides, its performance is bounded by the slowest client/server pairs if the load balancing among servers is not optimal. For example, if it uses NFS as one of the servers, the overall performance may be gated by NFS. This heterogeneous client/server environment helps but it might also hurt when encountering unbalanced load.

Microsoft adds the support of dynamic disks starting from Windows 2000. Dynamic disks are the disk formats in Windows necessary for creating multi-partition volumes, such as spanned volumes, mirrored volumes, striped volumes, and RAID-5 volume. The striped volumes contain a series of partitions with one partition per disk. However, only up to 32 disks can be supported, which is not very scalable[9].

### 3 Design and Implementation

The main task of the parallel file system is to stripe data or split files into several small pieces. Files are equally distributed among different I/O nodes and can be accessed directly from applications. Applications can access the same file or different files in parallel rather than sequentially. The more I/O nodes in a system, the more bandwidth it could provide (only limited by the network capacity).

#### 3.1 System Architecture

Generally speaking, our parallel file system consists of four main components: Metadata server, I/O daemons (Iod), a library and a file system driver. Metadata server and I/O daemons set up the basic parallel file system architecture. The library provides high performance APIs for users to develop their own applications on top of the parallel file system. It communicates with the metadata server and Iods, and does the tedious work for users. The complexity behind the parallel file system is hidden by the library and users do not need to concern about how the metadata server and Iods co-operate. With the help of file system driver, we can trap I/O related Win32 API calls and provide transparent file accesses. Most of the user mode APIs have the kernel mode equivalent implementation. The overall architecture is shown in Fig. 1.

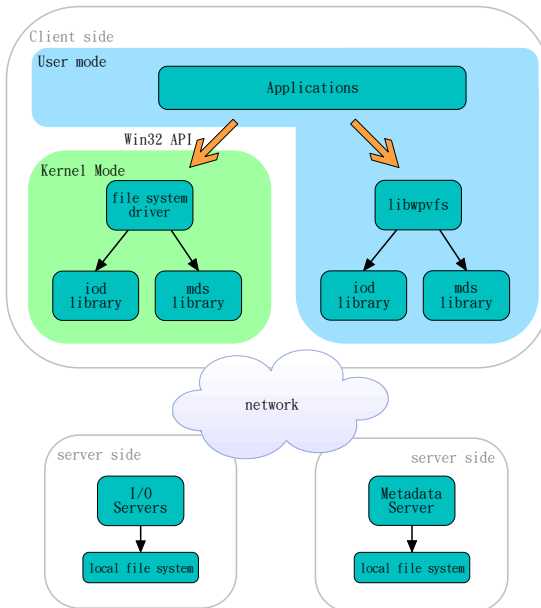


Fig. 1. The overall system architecture

**Metadata Server.** Metadata means the information about a file except for the contents that it stores. In our parallel file system, metadata contains five parts:

- File size: It describes the size of a file.
- File index: It is a 64-bit number, which uniquely identifies the file stored on the metadata server. Its uniqueness is maintained by the underlying file system, such as the inode number of the UNIX operating systems. It is used as the filename of the striped data stored on I/O nodes.
- Striping size: The size that a file is partitioned.
- Node count: The number of I/O nodes that the file is spread across.
- Starting I/O node: The I/O node that the file is first stored on.

The metadata server runs on a single node, managing the metadata of a file and maintaining the directory hierarchy of our parallel file system. It does not communicate with I/O daemons or users directly, but only converses with the library, `libwpvfs`. Whenever a file is requested, users may call the library to connect with the metadata server and get the metadata of that file. Before a file can be accessed, its metadata must be fetched in advance.

**I/O Daemons.** The I/O daemon is a process running on each of the I/O nodes responsible for accessing the real data of a file. It can run on a single node or several nodes, and you can run several I/O daemons on an I/O node if you want. After users get the metadata of a file, the library could connect to the required I/O nodes, and the Iods would access the requested file and send stripes back to the client.

Each of the I/O nodes maintains a flat directory hierarchy. The file index is used as the filename of the striped data regardless of the file's real filename. No matter what the real path of a file is, the striped data is always stored in a directory whose name is hashed from the file index. In our implementation, we use modulation as the hash function.

### 3.2 Library

As mentioned before, a library can hide the complexity of a parallel from users. In this subsection, we would discuss how the different libraries are implemented.

**User Level Library.** We provide a class library that contains six most important file system methods, including `open`, `create`, `read`, `write`, `seek`, and `close`. These methods are mostly similar to those of the `File` class in C# but with more capabilities support. Users can specify the striping size, starting Iod, and Iod counts when accessing a file. The library separates the users from the Iods and the metadata server. All the tedious jobs will be handled by the library. With the help of the library, users only need to concern how to efficiently partition and distribute the file.

**Kernel Level File System Driver.** In the Windows operating system, NT I/O Manager, which is a kernel component, is responsible for the I/O subsystem. To allow I/O Manager and drivers to communicate with other components in the operating system, a data structure called I/O Request Packet (IRP) is frequently used. An IRP contains lots of information to describe requests and parameters. Most important of all is the major function code and the minor function code. These two integers contained in an IRP precisely indicate the operation that should be performed. I/O related Win32 APIs will eventually be sent to the I/O Manager, which then allocates an IRP sent to the responsible driver.

With the help of a virtual disk driver, a file system driver, and a mount program, our parallel file system can be mounted as a local file system for Windows. Fig. 2 illustrates the mounting process of our parallel file system. The virtual disk driver presents itself as a normal hard disk to Windows when it is loaded into the system. The mount program invokes the `DefineDosDevice` function call to create a new volume on the virtual disk. After the new volume is created, the mount program tries to create a file on the volume. This request will be routed to the NT I/O Manager. Upon receiving this request, the I/O Manager finds that this volume is not handled by any file system driver yet. Thus, it sends an IRP containing a mount request to each of registered file system drivers in the system. File system drivers check the on disk information when they receive such a request to determine if it recognizes this volume.

We implement a crafted read function in the virtual disk driver. The driver returns a magic string “-pfs-” without quotes when a file system driver tries

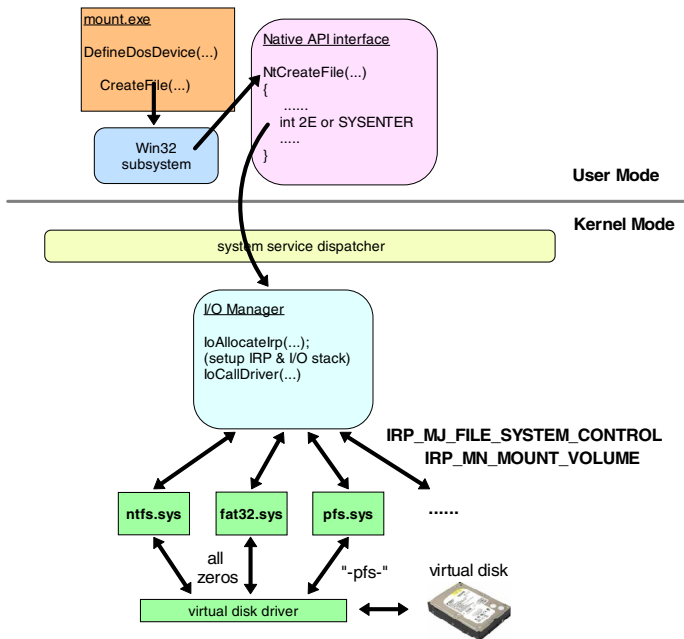


Fig. 2. The process of mounting our parallel file system

to read 6 bytes from the disk. Otherwise, it returns zeros. In this way, while any other file system drivers try to check the on disk information, they do not recognize the volume. “-pfs-” is the magic string that only our parallel file system driver recognizes. When the mount request is sent to our file system driver, it reads 6 bytes from the disk, recognizes the magic string, and tells the I/O Manager that this volume is under our control. The mount operation completes and all I/O operations targetting at this volume will be routed to our file system driver hereafter.

On loading the file system driver into the system, persistent connections are established to all I/O daemons. The connection procedures are performed once at the loading time, and all operations are made through these sockets. This eliminates the connection overhead of all I/O operations from user mode applications. The file system driver effectively does the same thing as the user mode library when it receives a read or write operation.

**MPI-IO Library.** MPI-IO[10] is the parallel I/O part of MPI and its objective is to provide high performance parallel I/O interface for parallel MPI programs. A great advantage of MPI-IO is the ability to access noncontiguous data with a single function call, which is known as collective I/O. Our parallel file system is built on .NET framework using C#.

## 4 Performance Evaluation

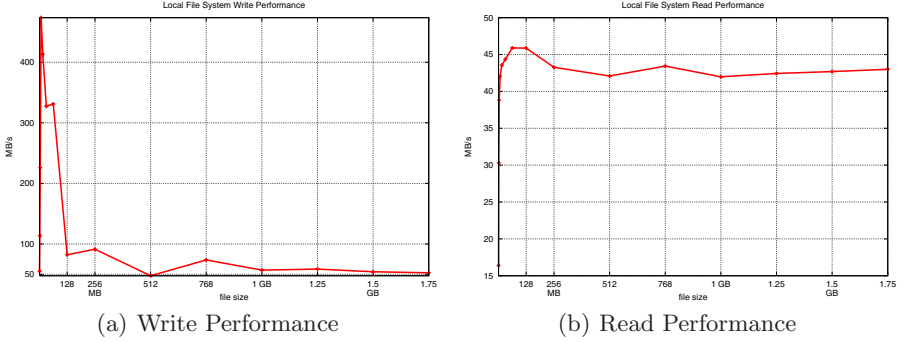
In this section, the local file system performance is measured along with read and write performance of our parallel file system. The hardware used is IBM eServer xSeries 335 with five nodes connected through Gigabit Ethernet, each housing:

- One Intel Xeon processor at 2.8 GHz
- 512 MB DDR memory
- 36.4 G Ultra 320 SCSI disk
- Microsoft Windows Server 2003 SP1

### 4.1 Local File System Performance

Our parallel file system doesn’t maintain the on disk information itself, but relies on the underlying file system. The root directory for Iods or the metadata server is set in an NTFS partition. To test I/O performance of the local file system and the .NET framework, we write a simple benchmark using C#. The tests are performed on a single node, running the tests ten times and averaged the results.

A 64 KB buffer is filled with random data and written to the local file system continuously until the number of bytes written to the local file system reaches the file size. Note that the write operations are carried out by the Microsoft .NET Framework and the NTFS file system driver which has some caching mechanism



**Fig. 3.** Performance evaluation of local file system

internally. In Fig. 3(a), we observe that write performance of local file system converges to about 55 MB/s when the file size is larger than 768 MB, but the performance varies when the file size is smaller than 512 MB. We think this is the effect of the caching mechanism.

To make sure that the files written are not cached in memory, the system is rebooted before measuring the read performance. The same file is read from the disk into a fixed-size buffer and the buffer is used over and over again. The data read is ignored and overwritten by later reads. As you can see from Fig. 3(b), read performance converges to about 43 MB/s.

## 4.2 Performance Evaluation Using User Level Library

The performance of our parallel file system are evaluated on five nodes. One of them is served both as a metadata server and a client which runs our benchmark program written with our library. The other four nodes are running I/O daemons, one for each.

Again, a fixed-size memory buffer is filled with random data. After that, a create operation is invoked, and the buffer content is written to the parallel file system continuously until the number of bytes written reaches the file size. The test program then waits for the acknowledgements sent by the I/O daemons to make sure all the data sent by the client are received by all I/O daemons. Note that though the Iods have written received data to their local file systems, this does not guarantee that the data is really written to their local disks. They may be cached in the memory by the operating system and written back to the physical disks later. We ran the tests ten times and averaged the results.

In Fig. 4(a), we measure write performance with various file sizes and various number of I/O nodes. The striping size is 64 KB. The size of the memory buffer used equals to the number of I/O nodes multiplied by the striping size. Write performance converges to about 53 MB/s when only one I/O node is used. We consider that the write performance is bounded by the local file system in this case, since this is almost equal to the local file system write performance as

shown in the previous test. The performance of writing to two I/O nodes is about twice of writing to only one node when the file size is large enough.

However, write performance reaches a peak of 110 MB/s when writing to three or four I/O nodes. For these two cases, they almost have the same performance since the bottleneck is the network bandwidth rather than the physical disks. Since all the cluster nodes are connected by a Gigabit Ethernet which has the theoretical peak bandwidth of 125 MB/s, it is conceivable that a client can not write out faster than 125 MB/s due to protocol overhead. The same behavior has been observed in PVFS[11] and IBM vesta parallel file system[12]. However, we expect the write performance to be scalable if a higher bandwidth network is available in the future.

The size of the memory buffer for read is also the number of I/O nodes multiplied by the striping size. The data read into the memory buffer is ignored and overwritten by later data. As you can see in Fig. 4(b), read performance is not as good as write performance. But when we increase the number of I/O nodes, the performance increases too. For four I/O nodes, read performance reaches a peak of 75 MB/s. With the use of more than two I/O nodes, read performance of our parallel file system is better than that of a local disk.

We have made some tests to figure out why the read performance can not fully utilize the theoretical network bandwidth. The Iod program is modified such that when it receives a read request, it does not read the data from the local file system, but just sends the contents of a memory buffer to the client directly. The contents in the memory buffer are non-deterministic. In this process of measuring read performance, the behavior is exactly the same as previous tests except that no local file system operations are involved. We run the test several times. The results show that when only one I/O node is used, the curves are almost identical and the performance reaches a peak of 90 MB/s. In the case of two I/O nodes, it has the same behavior but the performance reaches a peak of 93 MB/s. For three and four I/O nodes, the curves are desultory and the average performance reaches a peak of around 78 MB/s which is lower than the performance of using only one or two I/O nodes. We think this is due to network congestion and packet collision. Whenever multiple I/O nodes try to send large

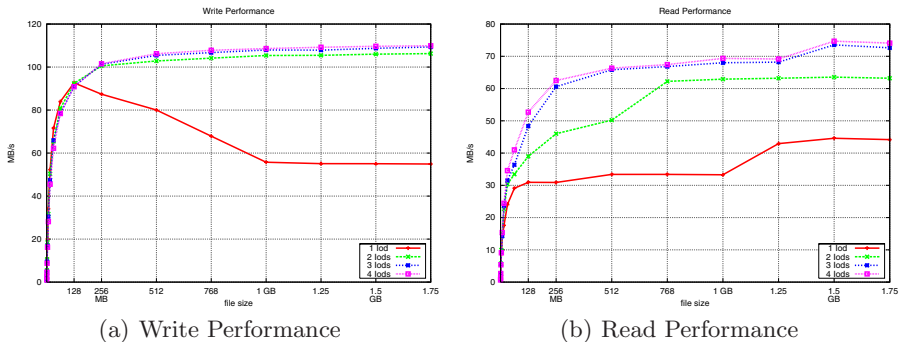


Fig. 4. Performance evaluation using variable I/O nodes

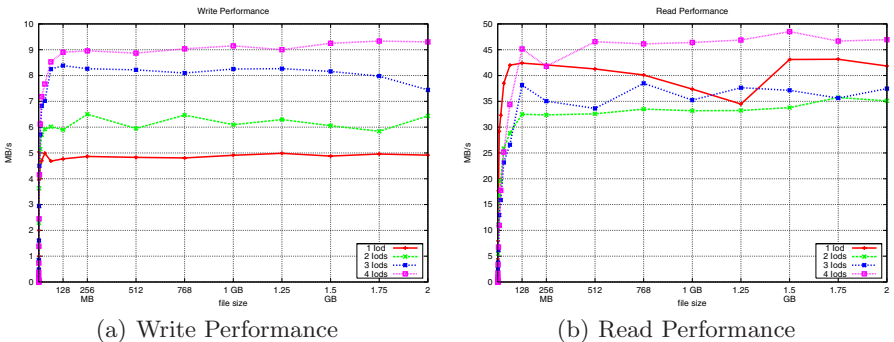
amount of data to a client simultaneously, the receiving speed of the client can not catch up with the overall sending speed of I/O nodes. Therefore some packets may collide with others and get dropped. The I/O nodes have to back off and resend packets as required by the protocol design of Ethernet architecture. This explains why the read performance is not as good as the write performance and saturated around 75 MB/s when three or four I/O nodes are used. In Fig. 4(b), the read performance of one I/O node and two I/O nodes are bounded by the local file system. In the case of three and four I/O nodes, it is bounded by the network due to network congestion. Again, as in write performance, we expect the read performance to be improved significantly when a higher performance network is available in the future.

### 4.3 Performance Evaluation Using Kernel Driver

To measure the performance of our parallel file system when using the file system driver, we write a simple benchmark program which has the same functionality as the one written in C#. But this benchmark uses Win32 APIs directly to create, read and write files. We also repeat the tests ten times and average the results.

Fig. 5(a) shows the write performance with various number of I/O nodes. The striping size is 64 KB and the user supplied memory buffer is 1 MB. When we increase the number of I/O nodes, the performance increases too, but it is worse than that of the local file system even when four I/O nodes are used. As observed from Fig. 5(b), the read performance is much better compared with the write performance. The performance increases with the number of I/O nodes when more than two I/O nodes are used, but the performance with only one I/O node is between that of four I/O nodes and three I/O nodes.

Windows is a commercial product and the source codes are not available. Therefore, the detail operations are opaque. Furthermore the file system driver resides in kernel mode and needs a socket library to communicates with daemons in the parallel file system. Microsoft doesn't provide a socket library for kernel



**Fig. 5.** Performance evaluation using the system driver with 64KB striping size and 1MB user buffer

mode programmers. The lack of a sophisticated kernel mode socket library also makes it difficult to write high performance programs.

The main purpose of implementing a kernel driver is to enable existing binaries to run over this parallel file system. However the parallel file system is developed mainly for high performance applications. We expect users to write high performance applications using our new APIs created especially to take advantage of this parallel file system.

#### 4.4 Performance Evaluation Using MPI-IO

We use the MPI-IO functions that we have implemented to write a benchmark program. In this case, we set the size of `etype[10]` to 64 KB which is equal to the striping size of the previous three cases. This is an obvious selection, since an `etype` (elementary datatype) is the basic unit of data access and all file accesses are performed in units of `etype`. The visible portion of the `filetype` is set to an `etype` and the `stride[10]` (i.e. the total length of the `filetype`) is set to the number of I/O nodes multiplied by the `etype`. Finally, the displacement is set to the rank of the I/O node multiplied by the `etype`. All the others are set based on the previous settings.

We measure the performance by varying the number of I/O nodes from one to four. The buffer size is set to be the number of I/O nodes multiplied by the `etype`. Each test is performed ten times and averaged to get the final result. The write and read performance are shown in Fig. 6(a) and Fig. 6(b) respectively.

The trends of the write and read performance resemble those which we have discussed above. Compared with the library of our parallel file system, `libwpvfs`, the MPI-IO functions have some added function calls and operations, but they do not influence the performance deeply. Consequently, the MPI-IO functions are provided without suffering serious overhead.

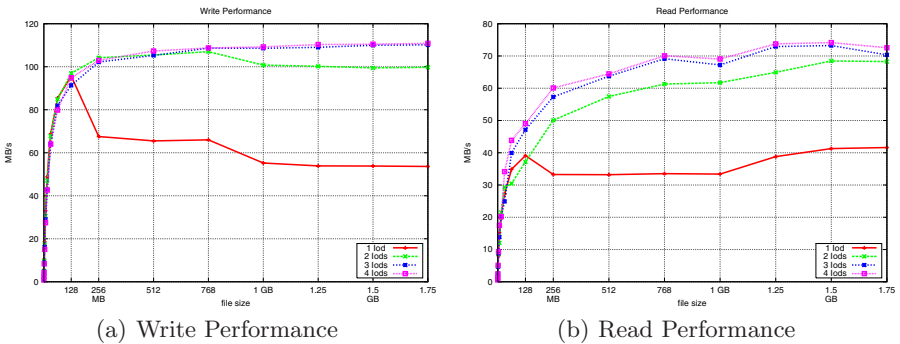


Fig. 6. Performance evaluation using our MPI-IO library

## 4.5 VOD Prototype System

Besides, we have set up a distributed multimedia server on top of our parallel file system. Microsoft DirectShow is used to build a simple media player. Using DirectShow with `libwpvfs`, we build a media player, which could play multimedia files distributed across different I/O nodes. Since DirectShow can only play media files stored on disks or from a URL, we establish a web server as an agent to gather striped files from I/O nodes. This web server is inserted between the media player and our library, `libwpvfs`, and it is the web server that uses the library to communicate with the metadata server and I/O nodes.

The data received by the web server from I/O nodes is passed to the media player. The media player plays a media file coming from the http server through a URL rather than from the local disk. Both the media player and the web server run on the local host. The web server is bound with our media player and transparent to the end user. A user is not aware of the existence of the web server and could use our media player as a normal one.

Any existing media player programs which support playing media files from an URL, such as Microsoft Media Player, can take advantage of our parallel file system by accessing the video file on our web server. In this way, we may provide a high performance VOD service above our parallel file system.

## 5 Conclusions and Future Work

PC-based clusters are getting more and more popular these days. Unfortunately almost all of the parallel file systems are developed in UNIX-based clusters. It is hard to implement a Windows-based parallel file system because Windows is a commercial product and the source codes are not available. In this paper, we have implemented a parallel file system which provides parallel I/O operations for PC clusters running Windows operating system. A user mode library using .NET framework is also developed to enable users writing efficient parallel I/O programs. We have also successfully implemented a simple VOD system to demonstrate the feasibility and usefulness of our parallel file system. In addition we have implemented key MPI-IO functions on top of our parallel file system and found that the overhead of implementing MPI-IO is very minimal. The performance of MPI-IO is very close to the performance provided by the parallel file system. Furthermore, we have also implemented a file system driver which provides a transparent interface for accessing files stored on our parallel file system so that existing programs written with Win32 APIs can still run on our system.

We have found that both write and read performance are scalable and only limited by the performance of the Ethernet network we use. We plan to further evaluate the performance of this parallel file system when we can obtain a higher performance network such as Infiniband under Windows and believe that our parallel file system can automatically achieve much better performance.

The prototyping VOD system proves the usability of our parallel file system in the Windows environment. Varying the striping size in the VOD system under different load conditions may have distinct behavior. The impact of the striping

size may hurt or help the VOD system under different load conditions. We would perform some detailed experiments and analysis in the near future. This would help us develop a more realistic and high performance VOD system that can benefit from our parallel file system.

## References

1. Adiga, N.R., Blumrich, M., Liebsch, T.: An overview of the BlueGene/L supercomputer. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, Baltimore, Maryland, pp. 1–22 (2002)
2. Myricom: Myrinet, <http://www.myri.com/>
3. InfiniBand Trade Association: Infiniband, <http://www.infinibandta.org/>
4. Pérez, J.M., Carretero, J., García, J.D.: A Parallel File System for Networks of Windows Workstations. In: ACM International Conference on Supercomputing (2004)
5. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: A parallel file system for linux clusters. In: 4th Annual Linux Showcase and Conference, Atlanta, GA, pp. 317–327 (2000)
6. Ligon III, W. B., Ross, R.B.: An Overview of the Parallel Virtual File System. In: 1999 Extreme Linux Workshop (1999)
7. Kleiman, S., Walsh, D., Sandberg, R., Goldberg, D., Lyon, B.: Design and implementation of the sun network filesystem. In: Proc. Summer USENIX Technical Conf., pp. 119–130 (1985)
8. Hertel, C.R.: Implementing CIFS: The Common Internet File System. Prentice-Hall, Englewood Cliffs (2003)
9. Russinovich, M.E., Solomon, D.A.: Microsoft Windows Internals, Microsoft Windows Server 2003, Windows XP, and Windows 2000, 4th edn. Microsoft Press, Redmond (2004)
10. Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Netzberg, W., Prost, J., Snir, M., Traverset, W., Wong, P.: 32. In: Overview of the MPI-IO Parallel IO Interface. IEEE and Wiley Interscience, Los Alamitos (2002)
11. Ligon III, W.B., Ross, R.B.: Implementation and performance of a parallel file system for high performance distributed applications. In: Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, pp. 471–480. IEEE Computer Society Press, Los Alamitos (1996)
12. Feitelson, D.G., Corbett, P.F., Prost, J.-P.: Performance of the vesta parallel file system. In: 9th International Parallel Processing Symposium, pp. 150–158 (1995)