A Parallel File System for Windows

Lungpin Yeh, Juei-Ting Sun, Sheng-Kai Hung and Yarsun Hsu
Department of Electrical Engineering, National Tsing-Hua University
HsinChu, Taiwan, 30055, ROC
{lungpin, posh, phinex, yarsun}@hpcc.ee.nthu.edu.tw

Abstract— **Parallel file system is widely used in clusters to provide high performance I/O. But most of the existing parallel file systems are based on UNIX-like operating systems. We implement a parallel file system for Windows using the Microsoft .NET framework. In this paper, the design and implementation of our system are described and the performance is also evaluated. The results show that the write performance reaches a peak of 109MB/s and the read performance reaches a peak of 85MB/s.**

I. Introduction

As the speed of CPU becomes faster, we might expect that the performance of a computer system should benefit from the advancement of CPU. However, the improvements of other components in a computer system (ie. memory system, data storage system) cannot catch up with that of CPU. This phenomenon lowers the overall performance. By Amdahl's law[1], the performance of a computer system is dominated by the component with the slowest speed. In a computer system, that is the storage system. Although the capacity of a disk has grown with time, its read/write performance is still an issue. In this data-intensive world, it is significant to provide a large storage capacity with high performance[2]. Using a single disk 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.

A parallel file system provides high-speed data access by using several disks at the same time. When we want to write a file to disks, a parallel file system will split these data into a lot of small chunks. Each of these chunks is stored on different disks in a round-robin fashion. Similarly, when reading a file from disks, a parallel file system will take out these chunks from each disk and then combine them to get the original file. With suitable striping size, the workload in the system can be distributed among these disks instead of centralized in a single disk. 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.

We implement a parallel file system for Microsoft Windows Server 2003 with striping support. User can specify the striping size using our user interface to obtain the required distribution. Default striping size and the number of I/O nodes will be used if not specified. We have successfully used our parallel file system as a storage system for VOD (Video On Daemon) services. This VOD system can deliver its maximum bandwidth to users with suitable network interface support. This paper is organized as follows: section 2 presents some related works, design and implementation will be discussed in section 3. Our VOD prototype system will be shown in section 4. Finally, we will discuss the performance of our parallel file system and make conclusion in section 5 and 6.

This work was supported by National Science Council, R.O.C., under Grant NSC 94-2213-E-007-004

II. Related Works

A special mention should go to PVFS[3][4] which is a popular parallel file system publicly available and working in the true world. PVFS provides both user level library for easy use and a kernel module package that makes existing binaries working without recompiling. However, it is only available for Linux clusters.

WinPFS[5] is a parallel file system for Windows. It is integrated within the Windows kernel components. It uses the existing client and server pairs in the Windows platform (ie. NFS[6], CIFS[7], ...) and thus no special servers needed to be installed. It also provides a transparent interface to users, just like what does when accessing normal files. The disadvantage is that the user can not control the striping size of a file across nodes. Besides, its performance is bounded by the slowest client/server pairs used. For example, if it uses NFS as one of the servers, the overall performance may be dominated by NFS. This heterogeneous client/server environment helps but it also hurts unless all client and server pairs have the same performance.

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, up to 32 disks are supported, which is not scalable.[8]

III. 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 in serial. Evidently, the more I/O nodes we have the more widely the files are distributed and the faster we can access the files.

A. Architecture

Generally speaking, our parallel file system consists of three main components:

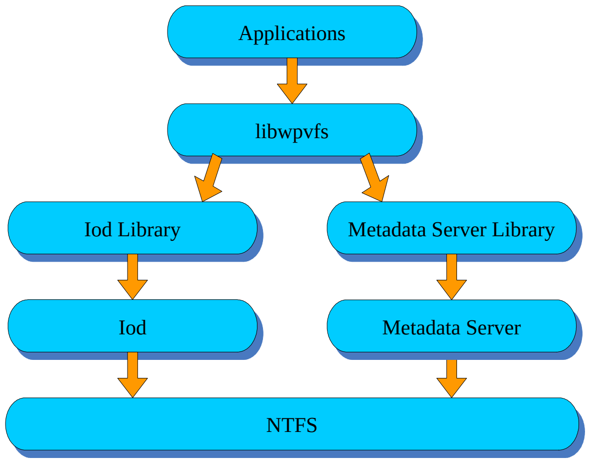

Fig. 1. The overall architecture.

File name	keroro.mp3
File size	512 KB
File index	2293117120
Striping size	64 KB
Starting I/O node	0
Node count	4

Fig. 2. The metadata stored on the metadata server.

- Metadata server
- I/O daemons (Iod)
- Library

Metadata server and I/O daemons set up the basic parallel file system architecture. The library provides convenient APIs for users to develop their own applications on top of the parallel file system. They do not need to concern about how the metadata server and Iods co-operate. The library communicates with the metadata server and Iods, and does the tedious work for users. The overall architecture is shown in Figure 1.

B. Metadata Server

Metadata means the information about a file except for the real data that it contains. In our parallel file system, metadata contains six parts shown in Figure 2:

- File name: It is unique and is used to distinguish a file from others. It is impossible to have files with the same name.
- File size: It describes the total length of a file.
- File index: It is a 64-bit number and also unique. This is the file index of the unique file name stored on the metadata server itself. Its uniqueness is maintained by the underlying file system. It is just like the inode number of the UNIX operating systems. This unique value is used as the filename of the striped data stored on the 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.

Metadata server runs on a single node. It manages the metadata of a file and maintains the directory hierarchy of our parallel file system. It does not contact with I/O daemons or users directly, but only communicates with the library, *libwpvfs*. When users use the library to access a file, the library will connect with the metadata server and acquire the metadata of that file in the first stage. The process is shown in Figure 3. The library can not access a file from I/O nodes until it

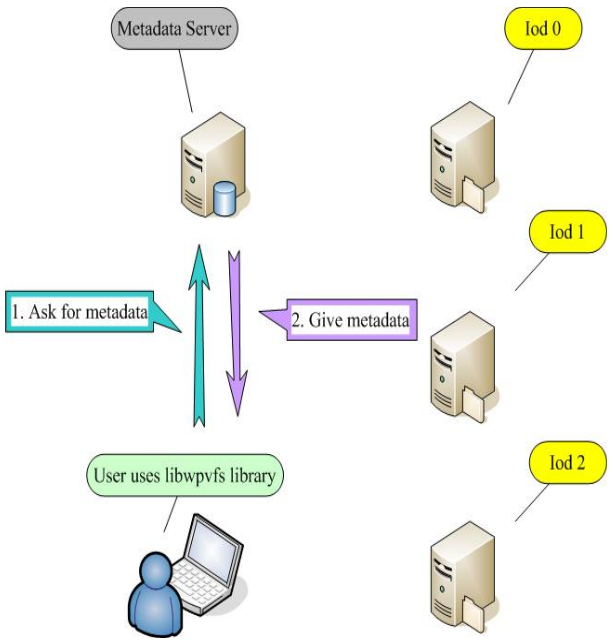

Fig. 3. Acquiring metadata from the metadata server.

obtains the metadata from the metadata server, or it does not know where the file locates. This centralized metadata server causes a single point of failure if some errors happen within it. In this case, our parallel file system can not service again unless the metadata server comes back online. However, this could be solved by mirroring. If faults occur, we just need to replace the broken one with the mirrored one.

C. I/O Daemons

The I/O daemon is a process running on 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. Like the metadata server, it does not contact with the metadata server and users directly, but only communicates with the library, *libwpvfs*. When users want to

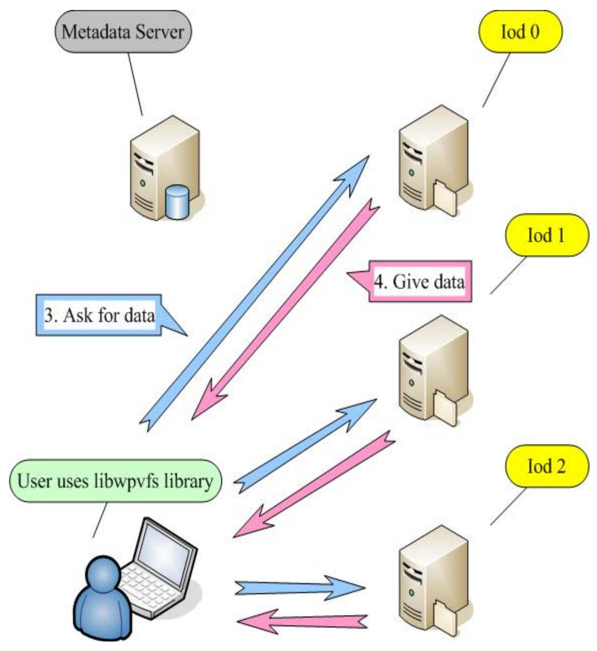

Fig. 4. Reading data from Iods.

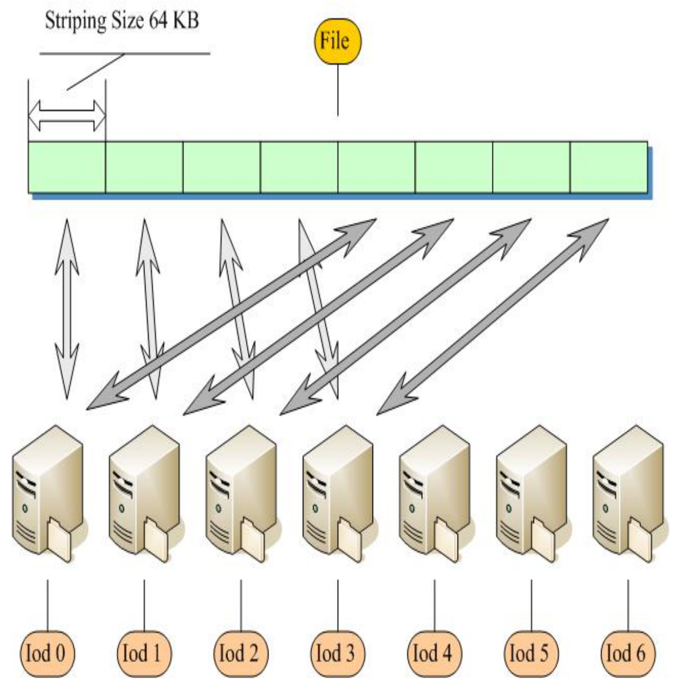

Fig. 5. A file is striped over 4 Iods.

access a file, the library obtains the metadata of that file from the metadata server first. Then, the library will connect to proper I/O daemons according to the metadata, and the Iod will access the correct file and send stripes back to the client. This process is shown in Figure 4. Unlike the metadata server, if one of the Iods fails, the parallel file system can still work. In this situation, the library can make use of the other healthy Iods, but the data within the failed Iod is not available anymore.

The way how data is stored on Iods is determined by some parameters of the metadata, those are, *file index, striping size, node count,* and *starting I/O node*. While the library receives the metadata, it will decide which I/O nodes should be used, and connect to them based on the starting I/O node and the node count parameters. After choosing the Iods, the library splits the file into small blocks with the dimension of striping size, and then writes each block to the corresponding I/O node. On the Iod side, the file index is modulated by 101 to get the directory where this file should be stored in and it is also used as the filename of the file. Certainly, these 101 directories must be created when the first time Iod runs. Eventually, the file stored on the Iod is named with the file index number rather than its original name.

An example is given in Figure 5, a file of size 512 KB is stored on the parallel file system. The metadata of this file has been shown in Figure 2. As the metadata exhibits, this file is striped over four Iods with 64 KB striping size and the file is stored on Iods starting from number 0.

D. Library

We provide a class library that contains six basic file system methods, including *open, create, read, write, seek,* and *close*, with *open* and *create* overloaded. The methods of our class library are mostly similar to those of the *File* class in C#. The difference between these two classes is that our class is used to access files stored in our parallel file system and the *File* class in C# can only be used to access files in local disks. When accessing a file, you can specify the striping size, starting Iod, and Iod counts as stated in Figure 2. If users use our library to write programs, they can build a variety of application programs having high performance I/O executed above the parallel file system.

The library separates the users from the Iods and the metadata server. For this reason, users do not have to worry about how to communicate with the metadata server, which Iods should be connected to, how to read or write data to Iods, and so on. All these tedious jobs will be handled by the library. Users only have to understand how to use the methods provided by the library and how to efficiently distribute their own data. Anyone can easily develop programs on our parallel file system by using the library.

IV. Prototype

In this section, we build a distributed multimedia server on top of our parallel file system. Microsoft DirectShow is used to build a simple media player. DirectShow is an architecture that contains many helpful classes and methods for media streaming on the Microsoft Windows platform, including Windows 9X, Windows Me, Windows 2000 and Windows XP. It supports lots of media formats such as AVI, MIDI, MP3, MPG, WMV, and so forth. Using DirectShow with *libwpvfs*, we build a media player, which could play multimedia files distributed across different I/O nodes.

Howerver, DirectShow can only play media files stored on disks or from a URL. It makes no sense if we copy the media file to the local disk and then play it. To paly multimedia files just in time, we have to read the media file from I/O nodes and

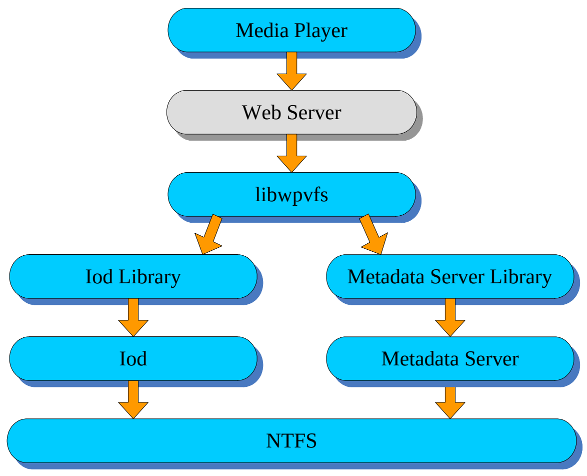

Fig. 6. A media player built on top of our file system.

play it at the same time. To solve this problem, we establish a web server as a 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 talk to the metadata server and I/O nodes (see Figure 6).

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. The process is shown in Figure 7. In this figure, the web server receives striped data from four I/O nodes and sends the combined stream to the media player immediately.

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.

V. Performance Evaluation

In order to measure the performance of our parallel file system for Windows, we have made some preliminary tests. In this section, the disk 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 CPU 2.8 G
- 512 MB memory
- 33 G SCSI disk
- Windows 2003 Server

To test I/O performance of the disk and the .NET framework, we write a simple benchmark using C# to measure the performance. The tests are performed on a single node. We ran the tests ten times and averaged the results.

The testing method is very straightforward. A fixed-size buffer is filled with random data and written to the disk

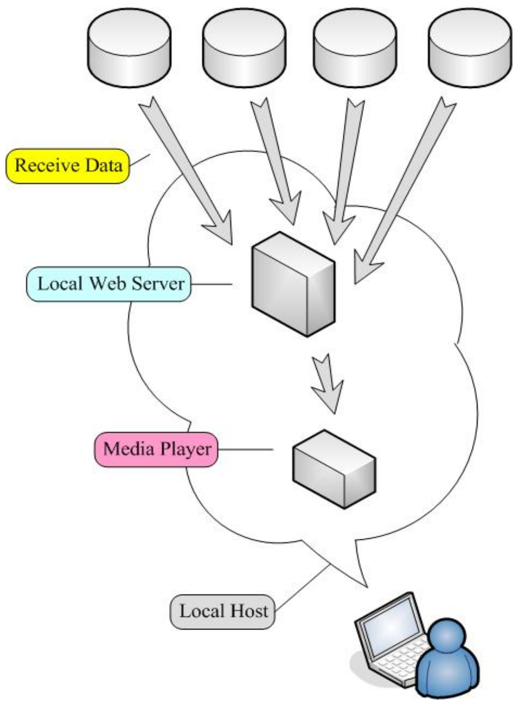

Fig. 7. The process of playing a media file.

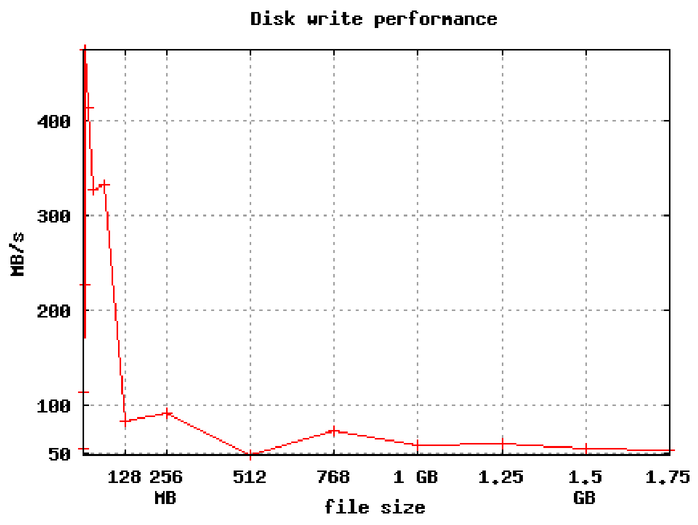

Fig. 8. The write performance of the local disk.

continuously until the number of bytes written to the disk reach the file size. In Figure 8, we observe that write performance of local SCSI disks converges to about 50 MB/s when the file size is larger than 768 MB, but the performance varies when the file size is smaller than 512 MB.

To make sure that the files written are not cached in memory, the system is rebooted before measuring the read performance. The 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 Figure 9, read performance converges to about 43 MB/s.

We measure the performance of our parallel file system on

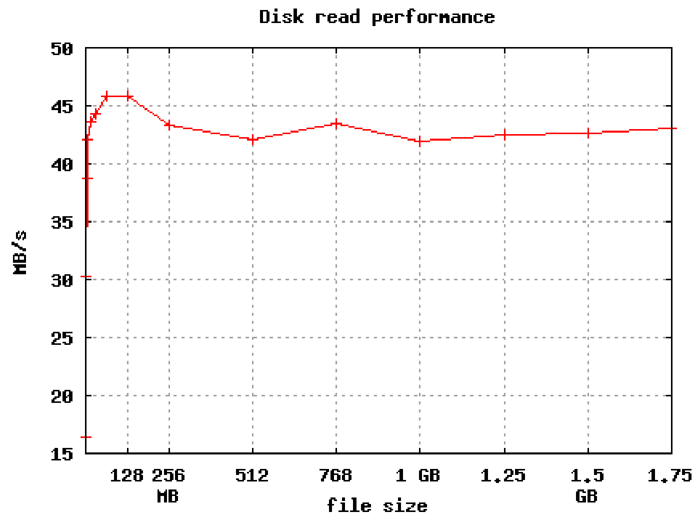

Fig. 9. The read performance of the local disk.

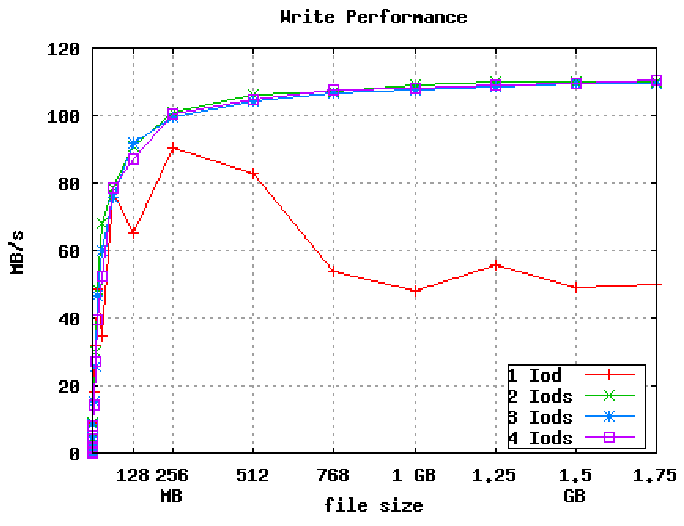

Fig. 10. The Write performance with increasing number of I/O nodes.

five nodes. One of them is running our test benchmark written with our library. The rest four nodes are running I/O daemons, one for each. One of the four nodes is running the metadata server too.

A fixed-size memory buffer is filled with random data and written to the parallel file system continuously until the number of bytes written reach the file size. The test program then waits for the acknowledgement sent by the I/O daemons to make sure all the data sent by the client are received by all I/O daemons.

In Figure 10, we measure write performance with different file sizes with 64 KB striping size. The memory buffer used is the number of Iod multiplied by the striping size. Write performance reaches a peak of 109 MB/s which almost utilizes the available network bandwidth. Writing to two, three or four I/O nodes almost has the same performance since the bottleneck is the network bandwidth rather than the disk system. Write performance converges to about 50 MB/s when only one I/O node participates. This is almost the local disk write performance as shown in the previous test.

The size of the memory buffer for read is also the number of Iod multiplied by the striping size. The data read into the memory buffer is ignored and overwritten by later data. As you can see in Figure 11, read performance is not so good compared with 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 85 MB/s. With more than two I/O nodes participate, read performance is better than that of a local disk.

VI. Conclusions and Future Work

Our parallel file system for Windows helps the integration of existing storages and provides parallel I/O operations for PC clusters running Windows operating system. A user mode library using the .NET framework is provided to users. It is still in the early stages of development, but the performance is better than only using a local disk.

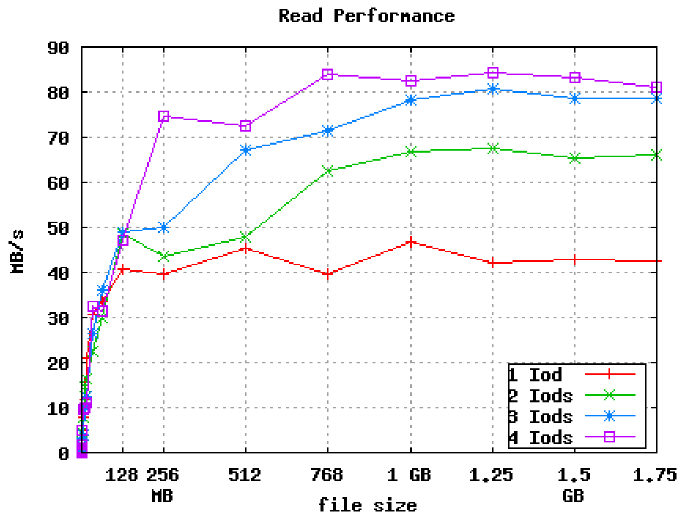

Fig. 11. The read performance with increasing number of I/O nodes.

Currently, the existing binaries need to be recompiled and linked with our library to benefit from our system. We are planning to implement the file system in the kernel mode using Microsoft IFS(Installable File System)[9] or using the CIFS(Common Internet File System). This provides a virtual disk for any existing binaries to take the advantage of our parallel file system. In addition, we will also provide fault tolerance facility, thus the system still works even if some of the I/O nodes failed.

References

[1] Gene M. Amdahl, *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*, pp. 79–81, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[2] NR Adiga, M Blumrich, and et al T Liebsch, “An overview of the BlueGene/L supercomputer”, in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Baltimore, Maryland, 2002, pp. 1 – 22.

[3] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, “PVFS: A parallel file system for linux clusters”, in *4th Annual Linux Showcase and Conference*, Atlanta, GA, October 2000, pp. 317–327.

[4] W. B. Ligon III and R. B. Ross, “An Overview of the Parallel Virtual File System”, in *1999 Extreme Linux Workshop*, June 1999.

[5] José María Pérez, Jesús Carretero, and José Daniel García, “A Parallel File System for Networks of Windows Worstations”, in *ACM International Conference on Supercomputing*, 2004.

[6] S. Kleiman D., Walsh R. Sandberg, D. Goldberg, and B. Lyon, “Design and implementation of the sun network filesystem”, in *Proc. Summer USENIX Technical Conf.*, 1985, pp. 119–130.

[7] Christopher R. Hertel, *Implementing CIFS: The Common Internet File System*, Prentice-Hall, 2003.

[8] Mark E. Russinovich and David A. Solomon, *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, Microsoft Press, 2004.

[9] Rajeev Nagar, *Windows NT File System Internals : A Developer’s Guide*, O’Relly, 1st edition, September 1997.