中華民國八十六年全國計算機會議

以 MPI 為基礎之平行檔案系統的設計與實作
The Design and Implementation of A MPI-Based Parallel File System

蔡永裕 Yung-Yu Tsai　謝德慶 Te-Ching Hsieh　李國華 Guo-Hwa Lee　張明峰 Ming-Feng Chang

國立交通大學資訊工程學系
Department of Computer Science and Information Engineering
National Chiao-Tung University
{yytsai,tchsieh,ghlee,mfchang}@csie.nctu.edu.tw

摘要

這篇論文說明以 MPI 為基礎之平行檔案系統 (MPFS) 的設計。自 MPI 標準延伸而來的 MPI-IO，能支援具有彈性的邏輯檔案分割及實體檔案組織，並提供豐富的檔案存取函式。MPFS 讓使用者能夠指定處理程序之間的邏輯檔案分割，及在資料伺服器上的實體檔案資料分佈。目前的實作完整支援下列功能：檔案資料分布，非同步 I/O，共享檔案指標，群集式 I/O 操作；以及對資料分佈指示的有限支援。然而，MPFS 並不支援在 MPI-IO 規格當中的 FORTRAN 介面，錯誤處理以及分析。MPFS 已經實作在由高速乙太網路所連接的工作站群上。 MPFS 的效能測量也將在這篇論文中說明。

Abstract

This paper presents the design of a MPI-based parallel file system, MPFS. MPI-IO is an extension of MPI to support flexible logical file partition and physical file organization, as well as a rich set of file access functions. MPFS enables users to specify both logical file partition among user processes and physical file data layout across data servers. Present implementation includes full support of file data distribution, asynchronous I/O, shared file pointers, collective I/O operations, and limited support of data layout hints. However, MPFS does not support FORTRAN interface, error handling and profiling in MPI-IO specification. MPFS has been implemented on a workstation cluster connected by Fast Ethernet. The performance measurements of MPFS are also presented in this paper.

keywords: parallel file system, MPI, MPI-IO

1.Introduction

The processor speed and memory capacity of computer systems have been increasing rapidly in the last decade. However, the I/O subsystem has not improved at the same rate due to the limitation of disk mechanical operations. This mismatch between computing speed and I/O bandwidth has impeded the performance of computer systems. In order to improve the performance of I/O subsystem, most recent researchers focus on parallel disk schemes that use several independent disks in parallel to aggregate disk bandwidth [1-3]. Moreover, in order to effectively exploit I/O performance of the parallel disk scheme, a parallel file system is needed to provide programmers a parallel file interface. An efficient parallel file interface should enable users to specify data distribution, caching, and prefetching policies that allow a parallel file system to reduce the number of physical input/output operations and to overlap physical input/output with computation. In addition, for parallel computing systems, the parallel file interface should enable a group of user processes to access disjoint parts of a file in a parallel fashion.

MPFS, a MPI-based Parallel File System, is designed to provide high I/O bandwidth, an efficient and yet easy-to-use parallel file interface, as well as high system portability for parallel computer systems. MPI is a message passing interface standard prompted by MPI Forum [4] and it is gaining popularity as a parallel programming environment. However, due to the lack of I/O interface support in MPI, IBM and NASA proposed MPI-IO, a parallel file I/O interface extension for MPI. MPI-IO supports a high-level interface to specify the logical partitioning of file data among processes and the physical file layout on data servers. In addition, it provides a rich set of file access functions. In general, MPI data type specifies dynamic user-defined partition of file data among a group of processes. This logical partition specification is independent of the physical data layout of file on data servers. The design of MPFS provides a user-level library which is an implementation of MPI-IO interface.

2.Related Work

A simple approach to improve the performance of a file system is to stripe file data and to store them across multiple disks or across multiple I/O nodes. RAID (Redundant Array of Inexpensive Disks) is an example of striping file data across multiple disks to aggregate disk bandwidth [2-3]. RAID enable the parallel inputs/outputs at the disk level; we can apply the same concept at a higher level - the file system level. File data can be striped and stored across multiple file servers so that the data transfer rate is not limited by the performance of a single file server. Example file systems that utilize I/O parallelism at both the disk level and the host level include Intel's Concurrent File System (CFS) [5] , nCUBE [6], Bridge [7], IBM's Vesta [8], Pasda [9] and PIOFS [10]. In contrast, Zebra [11] uses a different approach to stripe file data across the data servers. Instead of striping a single user file, Zebra stripes the logical file stream generated by a single client process. Small files are combined into a stripe fragment, and then written to the data servers. In this way, the read/write overhead of small files is lowered because the number of physical I/O operations is reduced..

Most parallel file systems are designed for particular parallel systems. Therefore, it is difficult to port these parallel file systems to other machines. For example, Intel's Concurrent File System(CFS) is built for iPSC/860 and iPSC/2, the Vesta parallel file system for IBM Vulcan, and the scalable file system(sfs) for the CM-5 [12]. On the other hand, some parallel file systems are built for virtual parallel environments to maximize their portability. Instead of modifying system kernel, user-level libraries are developed to provide the parallel file interface and disk accesses are performed by the underlying Unix file system. For example, PPFS [13] is a portable parallel file system that is based on MPI and NXLIB [14]. PIOUS [15] is another example designed for PVM [16] environment which is a portable software platform that aggregates networked computing resources. However, both PPFS and PIOUS do not allow users to specify a flexible parallel view in file access.

MPI-IO is an extension of MPI to support a flexible user-defined logical file partition and a rich set of file access functions. To date, there are some implementations of parallel file system that is also based on MPI-IO. For example, PMPIO [17], a portable implementation of MPI-IO, is developed at the Numerical Aerodynamic Simulation (NAS) facility located at NASA Ames Research Center. PMPIO is a portable MPI-IO library that runs on Intel Paragon, IBM SP2 and SGI workstation clusters. They employ a collective buffering design to collect and merge I/O operations, and thus effectively reduce physical I/O overhead. However, their present implementation does not support asynchronous I/O, shared file pointer, profiling and hints. PIOFS [10] is another implementation of MPI-IO, developed by the Scalable Parallel Systems Department at the IBM T. J. Watson Research Center. PIOFS runs on the IBM SP2, but it does not support asynchronous data access functions, file pointers (nether individual nor shared file pointers) operations, error handling and profiling in their present implementation. Our MPFS implementation includes full supports of MPI-IO filetype and buftype, asynchronous I/O, shared file pointers, collective I/O operations, and limited support of hints. However, present implementation does not support FORTRAN interface, error handling and profiling.

3.Design & Implementation of MPFS

The overall architecture of MPFS is shown in Figure 1. MPFS consists of three major components: a user library, data servers and file managers. The user library of MPFS is an implementation of MPI-IO, a parallel file I/O interface for user processes to access files. The file managers maintain the metadata of MPFS files. When a MPFS file is opened, the user library obtains the metadata of the file from a file manager to determine the physical data distribution pattern over the data servers. User processes can access a MPFS file in parallel through the user library which in turn requests data from the data servers according to the file data distribution pattern. File data are striped and stored over the data servers so that the file access can be done in parallel. To maximize system portability, the data servers use the underlying Unix file system. Moreover, our design is built on the top of MPICH which is a portable implementation of MPI. The communications between the user library, the data servers and the file mangers are carried out by the message passing schemes of MPICH. We will explain the design of the three components in following sections.

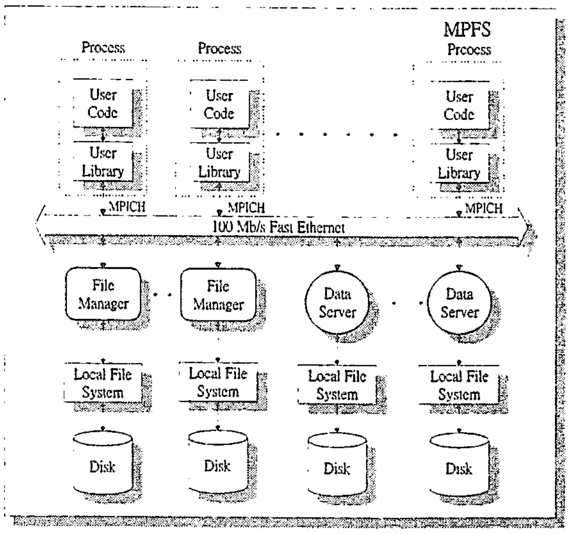

Figure 1 : The architecture of MPFS

3.1 MPI-IO and MPFS user library

MPI-IO provides flexible logical file partition schemes and a rich set of file access functions. MPI-IO extends the MPI datatype to create etype, filetype and buftype in order to specify the logical data partitioning among user processes. etype defines the basic access unit of a file; it is used as the elementary datatype to ensure the consistency between buftype and filetype. filetype describes the data layout in the file, and buftype describes the data layout in process' buffer; both the filetype and buftype are specified in terms of etype. The filetype specifies the a data pattern that a process intends to access in the file; as shown in Figure 2, the 'holes' in the filetype represent the data that the process cannot access. Several disjoint filetypes can form a logical file partition. For example, in Figure 2, three processes partition a MPFS file logically by using disjoint filetypes. The three processes can access the same file in a interleaving fashion. On the other hand, buftype specifies a data pattern in which file data are placed in the user buffer. This allows non-contiguous memory buffers to be read from or written to the file system. An Example file access of process 1 using the filetype and buftype is also shown in Figure 2.

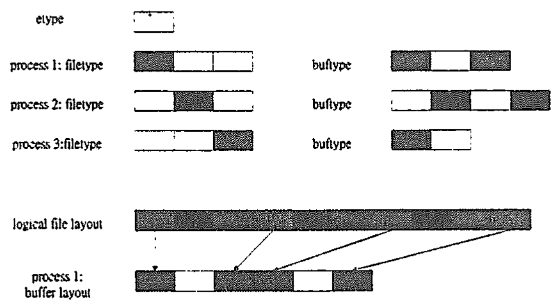

Figure 2: An example of filetype and buftype

MPFS user library is an implementation of MPI-IO interface; it consists of two types of functions: the file manipulation functions and the data access functions. In addition, the user library supports a local data cache to reduce the number of physical I/O operations. The file manipulation functions including file open, file close, file delete, file resize, and file seek. When a file is opened, the user library communicates with the file managers to determine which data servers will serve the input/output requested for the given file on behalf of the user program. Users can also set system parameters by using file manipulation functions including the number of data servers, stripe factor, striping size, cache size, enabling/disabling cache, cache policy, and enabling/disabling prefetching.

The file data access functions are classified according to three orthogonal aspects: positioning, synchronism and coordination (see Table 1). In the positioning aspect, user processes describe the position of file pointer in three ways: explicit offsets, individual file pointers and shared file pointers. The operations using an explicit offset need to specify the file pointer position in every read or write operation; they act like seek-and-read or seek-and-write operations. On the other hand, the operations using individual and shared file pointers use implicit file pointers maintained by MPFS. Individual file pointers are maintained for each process individually, and this is implemented in the MPFS user library. In contrast, the shared file pointer is maintained by the file managers for the group of processes that opened the file.

In the synchronism aspects, all operations can be either synchronous or asynchronous. A synchronous I/O operation will be blocked until the I/O request is completed. On the other hand, an asynchronous I/O operation only initiates the corresponding I/O operation, but does not wait for its completion. A separate MPI-IO function call, MPIO_Test or MPIO_Wait, is needed to check whether the asynchronous I/O request is completed. Using asynchronous data access functions, user program can overlap I/O operations with computation to improve performance.

The coordination of I/O functions can be either collective or independent. The independent accesses do not need any coordination between processes, while the collective accesses imply that the functions cannot be performed until all processes in the process group associated with the target file invokes the I/O operation. The collective I/O calls have the potential to fully utilize the disk bandwidth, because the disk accesses in the data servers are more likely to be segmented accesses. MPFS implements all combinations of these data access functions defined in MPI-IO (see Table 1).

When a data access function is invoked, the user library first calculates the absolute offset of the target block for the associated file stored in the data server. In this way, it maintains individual file pointers for the files opened. The user library issues only simple read/write commands to the data servers. MPI_Send and MPI_Recv functions supported by MPICH are used to transfer data between the data servers and the user library. Moreover, the user library of MPFS guarantees that the data stream returned to user processes would be in order, like Unix file system does.

Table 1: Data access functions defined by MPI-IO

positioning	synchronism	coordination	
		independent	collective
explicit offset	blocking (synchronous)	MPIO_Read MPIO_Write	MPIO_Read_all MPIO_Write_all
	nonblocking (asynchronous)	MPIO_Iread MPIO_Iwrite	MPIO_Iread_all MPIO_Iwrite_all
individual file pointer	blocking (synchronous)	MPIO_Read_next MPIO_Write_next	MPIO_Read_next_all MPIO_Write_next_all
	nonblocking (asynchronous)	MPIO_Iread_next MPIO_Iwrite_next	MPIO_Iread_next_all MPIO_Iwrite_next_all
shared file pointer	blocking (synchronous)	MPIO_Read_shared MPIO_Write_shared	MPIO_Read_shared_all MPIO_Write_shared_all
	nonblocking (asynchronous)	MPIO_Iread_shared MPIO_Iwrite_shared	MPIO_Iread_shared_all MPIO_Iwrite_shared_all

The user library also supports a local data cache. User can enable or disable the caching. When enabled, the cache replacement policy can be either LRU or FIFO. To reduce the complexity in maintaining data consistence, write-through policy is used, i.e., all write operations write data first to the data servers, and then to the local cache. When a read operation is a hit in the cache, the user library still has to verify the data by requesting the data serves to check the time-stamp of the cache data. In this way, the communications with the data servers are still required, but the data transfer may be omitted.

3.2 File managers

The file managers maintain file metadata, authenticate user access requests and coordinate file manipulation operations. Metadata describes the parallel file organization (i.e., the file name, the data servers, the striping size, the stripe factor, file length, and file access privilege). In addition, the file manager maintains a file handle table that contains the status of the files opened; the status data include the group of processes that opened the file, their process ID, the file handle number, the access mode, and the shared file pointer.

The file managers service file manipulation operations including file open, file close, file delete, and file control. When user processes open a file, the file mangers notify all data servers that contain file data that the file is opened, and then send the corresponding metadata and a file handle number back to the user library. When more than one group of processes open the same file, a different file handle number is returned to each process group. Moreover, the file managers maintain the shared file pointers of the opened files. When a file access function using a shared file pointer is invoked, the user library needs to retrieve the current file pointer from the file manager before it send access requests to the data server. In addition, the file managers also coordinate file close operations, detecting that all user processes of a file have closed and then collecting any updated metadata from the data servers.

3.3 Data servers

The data servers respond to the file access requests of user processes; they read data from and write data to the local disks. The number of data servers is configurable at the time PMFS is initialized. The data server maintains a table to store the metadata of the files opened. When a data server receive a file access request, it first checks whether the file handle and the process ID is valid. Before the data server issues a physical I/O operation, it first tries to lock the target data block. Present data server design uses the approach of single-writer/multiple-reader lock so that more than one process can read a single data block at the same time, but only one process can write a particular data block. If the locking attempts fails, the access request will be placed in a waiting queue. Moreover, the data server splits each access request into a series of data block access commands, and then send each data block back to the client separately. In this way, the disk access and the network data transfer of the data blocks can be overlapped thus reduce the overall response time. The size of a data block can be specified by users.

Each data server contains a cache of file data to reduce the number of disk accesses, and thus the access latency. This file cache is also used as the data buffer of the message passing operations between the data servers and user processes. In this way, the overhead of memory-to-memory copy is reduced. Moreover, a simple prefetching algorithm is designed to read additional data blocks from the underlying file system when collective operation requests are received. In order to maximize system portability, the data servers use local Unix file system to access disk data.

4.Performance Measurements

In order to measure the performance of MPFS, sequential file access experiments are carried out on a workstation cluster connected by 100Mb/s Fast Ethernet. The workstation cluster consists of four workstations; each workstation contains a Pentium-150 CPU, 32MB memory and FreeBSD 2.2.1-RELEASE operating system. The disk drives are connected using SCSI bus; the maximum data transfer rate of a hard disk is measured at about 6.5 MB/s. In the experiments described below, one user process accesses 64 MB of file data sequentially in MPFS. To examine the scalability of present implementation, the number of data servers varies form 1 to 4. In addition, to estimate the message passing overhead, the physical block size of data striping varies from 1KB to 128KB. The data stripe factor is set to one, i.e., file data are stored across the data servers in a round-robin fashion..

The measurements on synchronous read operations (MPIO_Read) are shown in Figure 5. Note that the performance of MPIO_Read improves significantly as the block size increases. This is due to the less number of read commands issued to the data server, and thus the smaller communication overhead. When the block size is greater than 16 KB, the data throughput using one data server is 5.5 MB/sec which is about 20% less than the maximum data throughput of a local disk access. This indicates the message passing overhead of MPFS is about 20%. In addition, when the block size is large, the performance of MPFS improve by only a small margin as the number of data servers increases. This is because the total data transfer rate is limited by the network bandwidth which is about 7 MB/sec. On the other hand, when small block size is used, the performance indeed improves significantly as the number of data servers increases, but still cannot reach the maximum network bandwidth because of the message passing overhead. Figure 6 shows the experiment results of synchronous write (MPIO_Write) operations. The performance measurements are almost identical to those of synchronous read operations. However, note that MPIO_Write is faster than MPIO_Read by a very small margin in each case. This is because the data server issues asynchronous write operations to the local Unix file system.

Figure 7 shows the experiment results of asynchronous read functions (MPIO_Iread). An

MPIO_Iread function returns as soon as it sends the read requests to the data servers; it does not wait for the data server to send the data back. The results in Figure 7 show only the time to issue the requests to the data servers, not including the time to wait for the completion of the request. Compared with synchronous read functions in Figure 5, asynchronous read functions save a great amount of time without waiting the data requested. This provides the asynchronous operation the potential of overlapping I/O operations and computation. Furthermore, we consider the overhead that may be incurred if a user process use MPIO_Wait to wait for the return of the data requested by asynchronous read functions. The results in Figure 8 indicates that there is no overhead for asynchronous read functions using MPIO_Wait when compared with synchronous read functions.

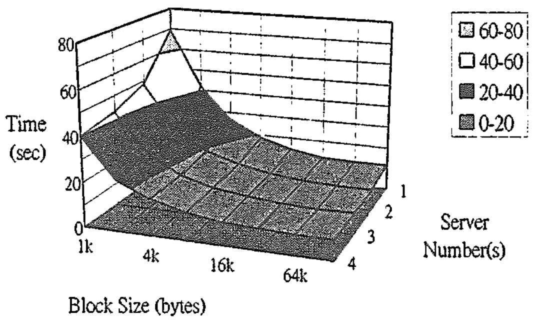

Figure 5: The performance of synchronous read.

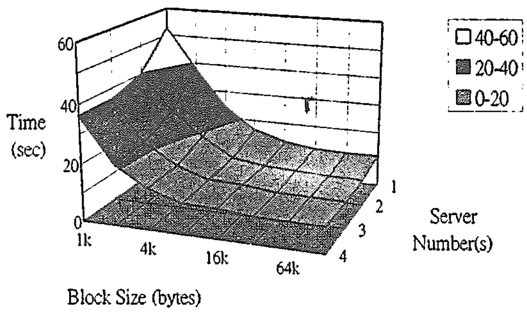

Figure 6: The performance of synchronous write.

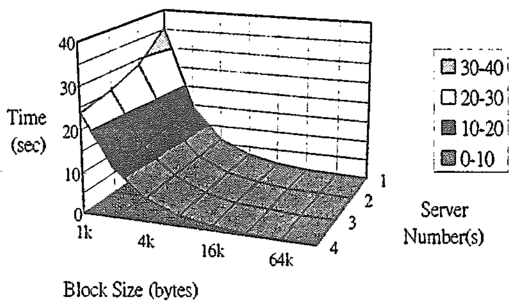

Figure 7: The performance of asynchronous read

Figure 9 shows the overhead of synchronous read functions using a shared file pointer. Since user process using a shared file pointers needs to fetch current file pointer position before each read/write operation, the overhead is proportional to the number of read/write operations. In Figure 9, we can see that the overhead in using a shared file pointer is larger as the block size gets smaller. Note that the overhead using a shared file pointer for small block size almost doubles the access time.

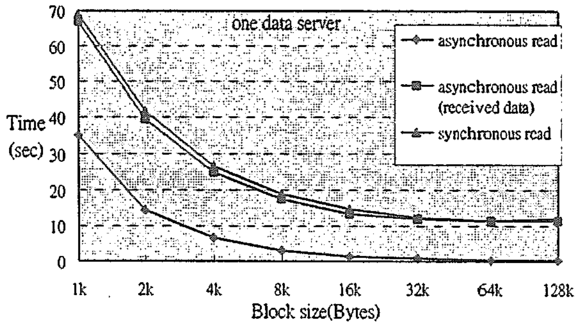

Figure 8: The comparison between synchronous read and asynchronous read

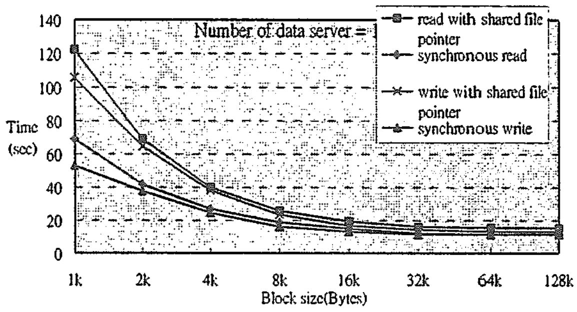

Figure 9: the overhead of a shared file pointer

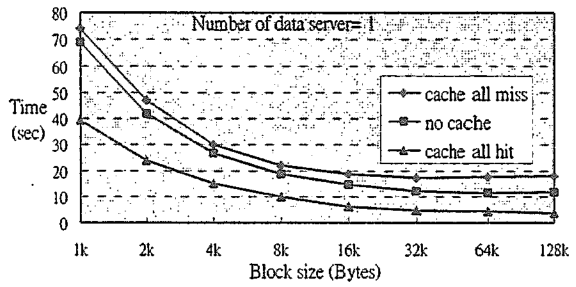

Figure 10: The cache performance for synchronous read

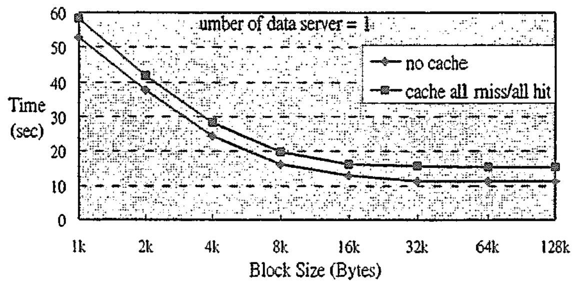

Figure 11: The cache overhead for synchronous write

We examine the performance and overhead of data cache in two extreme cases: one in which all cache references are missed, and the other in which all cache references are hit. The caching is implemented at both the user library and the data servers. The results in Figure 10 shows that compared with synchronous read without caching, cache-hit provides 50% performance improvement in average, while cache-miss calls for 25% performance loss in average. By simple calculation, one can derive that if the cache hit-rate is above 33%, read operations benefit from the caching in average. On the other hand, the overhead for synchronous write using caching is 25% in average, as shown in Figure 11. The overhead includes writing the data to the data servers and to the local cache.

5.Conclusions

MPFS, a MPI-based parallel file system is implemented on a workstation cluster. The implementation includes full support of MPI-IO filetype and buftype, asynchronous I/O, shared file pointers, collective I/O operations, and limited support of data layout hints. MPFS enables users to specify both logical file partition among user processes and physical file data layout across data servers. It also provides parallel file access interface to allow a group of user processes to access a MPFS file in parallel. Since our MPFS implementation is a user library, it can be easily ported to any platforms that support Unix file systems and MPICH which is used as our underlying message passing environment. The present implementation is built on a workstation cluster connected by 100Mbit Fast Ethernet. To date, MPFS does not support FORTRAN interface, error handling and profiling in MPI-IO specifications.

The experimental measurements show that the performance of MPIO_Read improves significantly as the block size increases. When compared with the local file access, the message passing overhead of MPFS is about 20%. Since the bandwidth of Fast Ethernet becomes the performance bottleneck if more than one data server is used, we believe the system performance can improve significantly once the network is upgraded. In addition, the results also indicates that there is no overhead for asynchronous read functions using MPIO_Wait when comparing with synchronous read functions. For caching, if the cache hit-rate is above 33%, read operations benefit from the data caching.

References

[1] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *SIGMOD Intl. Conf. Management of Data,* pp.109-116, Jun 1988.

[2] P. M. Chen, et al., "Performance and design evaluation of the RAID-II storage server," *IPPS'93 Workshop on Input/Output in Parallel Computer Systems,* pp. 110-120, 1993.

[3] Kent Teriber and Jai Menon. "Simulation Study of Cacaed RAID5 Designs," *Proceedings of the First Conference on High-Performance Computer Architecture,* pp. 186-197, 1995.

[4] M. P. I. Forum, "MPI: A Message Passing Interface Standard," May 1994.

[5] P. Pierce, "A concurrent file system for a highly parallel mass storage subsystem," *4th Conf. Hypercubes, Concurrent* Comput. & Appl., vol.I, pp.155-160, May 1989.

[6] Erik DeBenedictis and Juan Miguel del Rosario, "nCUBE Parallel I/O Software," *Proceedings of the Eleventh Annual IEEE International Phoenix Conference on Computers and Communications,* pp.117-124, 1992.

[7] Peter Dibble, Michael Scott, and Carla Ellis, "Bridge: A High Performance File System for Parallel Processors," *Proceedings of the Eighth International Conference on Distributed Computer Systems,* pp.154-161, 1988.

[8] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson, "Overview of the Vesta parallel file system," *IPPS'93 Workshop on Input/Output in Parallel Computer Systems,* pp.1-16, 1993.

[9] M. C. Jih, L. C. Feng and R. C. Chang, "The design and implementation of the Pasda file system," *Proc. Intl. Computer Symposium,* 1994.

[10] IBM, "MPI-IO/PIOUS," *http://www.research.ibm.com/people/prost/sections/mpiio.html*

[11] J. H. Hartman and J. K. Ousterhout, "Zebra: A Striped Network File System," *Proceedings of the USENIX File Systems Workshop,* pp.71-78, May, 1992.

[12] J. Susan, et al., "sfs: A parallel file system for the CM-5," *Proceedings of the Summer 1993 USENIX Conf.,* pp. 291-307.

[13] Jay Huber, et al., "PPFS: a high performance portable parallel File System," *Proceedings of the 9th ACM International Conference on Superence on Supercomputing,* pp. 385-394, July 1995.

[14] G. Stellner, A. Bode, S. Lamberts, and T. Ludwig, "Developing applications for multicomputer systems on workstation clusters," *The Intl. Conf. And Exhib. On High-Performance Computing and Networking,* 1994.

[15] Steven A. Moyer and V. S. Sunderam, "PIOUS: A scalable parallel I/O system for distributed computing environments," *Proceedings of the Scalable High-Performance Computing Conf.,* pp.71-78, 1994.

[16] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "Integrated PVM framework supports heterogeneous network computing," *Computers in Physics,* 7(2):166-75, April 1993.

[17] S. Finberg, B. Nitzberg, and P. Wong, "PMPIO - A portable MPI-IO library," *http://lovelace.nas.nasa.gov/MP-IO/pmpio/pmpio.html*