THE ARCHITECTURE OF BPFS: A BASIC PARALLEL FILE SYSTEM

ROBERT D. RUSSELL *
E-mail: rdr@unh.edu

Department of Computer Science
University of New Hampshire
Durham, New Hampshire 03824, USA

ABSTRACT

BPFS is a distributed, modular parallel file system designed for use on clusters of workstations. It is specified as a set of active components, functions utilized by the components, and protocols for communication between the components. Components can be implemented in many different ways, depending on the hardware and software available, the performance desired, etc. A key idea is to be able to experiment with different implementations and configurations under various operating conditions to achieve a general, flexible system that delivers good performance.

To date, three application program interfaces have been defined for BPFS. API0 is the first of an anticipated series of low-level, experimental client interfaces. It provides asynchronous block-oriented operations, time-regulated "data streaming", and user-level control of both server-side caching and per-file striping onto disks. CLI is a "C Library Interface" implemented over API0 that mimics the Standard C I/O library interface, but accesses parallel files stored by BPFS rather than sequential files. The third interface is the ROMIO version of the standard MPI-IO interface which has been implemented on top of API0.

Keywords: parallel file systems, distributed file systems, file system architecture.

1 INTRODUCTION

Parallel file systems represent an important area of research that has recently gained attention as a significant contributor to performance of parallel programs. This paper describes the architecture of BPFS — a Basic Parallel File System. It is designed to be one component of a more general system for multi-media parallel computation on clusters of workstations interconnected by high-speed communication networks. The goal of BPFS is to be a general-purpose distributed block storage server. Therefore, it does not provide or utilize any mechanisms, such as synchronization, locks, barriers, etc., that could reasonably be provided by other components of a parallel computation system and that may be necessary at the application level to properly coordinate parallel access to parallel files.

Because of its specification in terms of open protocols between network nodes, the design approach taken with BPFS is unlike that taken for other parallel file systems reported in the literature. This approach is very low level and is intended for clusters of workstations — it makes no sense whatsoever for "traditional" parallel machines such as the Intel Paragon, for example. The intention is to encourage experimentation with and evolution of these protocols, and to stress the "glue" between components rather than the components themselves, so that as newer technology, such as network attached disks, becomes available it can be utilized without redesigning the existing framework.

BPFS provides a tool by which files can be stored and accessed "in parallel" on workstation clusters. The data in a single file is stored on several disks on different nodes using a technique known as "striping" or "declustering". All disks are accessed in parallel in order to obtain the complete file. Many processes will have simultaneous access to the file system as a whole, as well as simultaneous access to individual files. Thus the active components of BPFS operate in parallel on behalf of many user applications. This requires attention to the semantics of file access to guarantee consistency and integrity between file components while allowing files to be shared. The goal of all this parallel activity is to provide user applications, both parallel and non-parallel, with the ability to overlap computation with I/O in order to obtain higher I/O bandwidth and lower I/O latency.

The organization of the rest of this paper is as follows: Section 2 gives an overview of the BPFS architecture, introducing the components and showing how they are related. Section 3 gives a brief review of

*Work performed while on leave at the Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France.

the literature on parallel file systems, stressing those systems that most influenced the design of BPFS. Section 4 introduces three application program interfaces to BPFS. Section 5 discusses some performance results of the first implementation. Section 6 concludes with a brief discussion of work in progress.

2 GENERAL OVERVIEW

BPFS assumes a model of a parallel computing system that consists of a set of heterogeneous “logical nodes” capable of communicating with each other via a “communication network”. There are three types of logical nodes, although in any real system several different types of logical nodes may (and usually will) coexist on a single physical node.

- A “logical client” node consists of a user application process (i.e., the end user of the parallel file system) linked with a client component.
- A “logical manager” node consists of a manager component, some amount of permanent storage for metadata components that are controlled by the manager, and some subset of the global file name space controlled by the manager.
- A “logical server” node consists of a server component, a number of agent components, and some large amount of storage for data components that are controlled by the server.

A node may contain two types of “components”:

- “active components” — the processes and/or threads that perform the operations of the parallel file system. There are four types of active components: clients, managers, servers and agents. These active components communicate with each other by using specific “protocols”.
- “passive components” — the disk storage areas containing the information in the parallel files. There are two types of passive components: data and metadata.

Figure 1 shows the organization of logical nodes, the components on each, and their network connections.

A single instance of the parallel file system is “constantly available” to all applications simultaneously (i.e., it is not restricted to being run as part of a single application). It is expected that logical nodes will dynamically join and depart from a running system.

Data is organized, stored and communicated in fixed-size blocks. The block size is determined by the implementation, but should be at least the size of a physical disk block (usually 4096 bytes). It is the same for all applications using the same BPFS system.

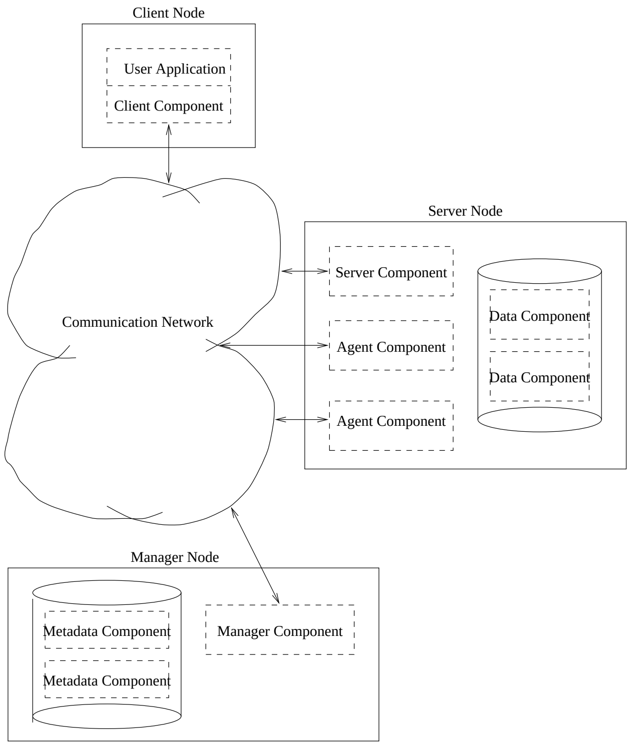

FIGURE 1: OVERVIEW OF THE LOGICAL NODES, COMPONENTS AND COMMUNICATION NETWORK IN BPFS. SOLID RECTANGLES INDICATE LOGICAL NODES, DASHED RECTANGLES ARE COMPONENTS, THE IRREGULAR “CLOUD” IS THE COMMUNICATION NETWORK, AND ARROWS INDICATE NETWORK CONNECTIONS.

2.1 NETWORK

The network can be any general purpose facility, such as TCP over a Local Area Network, over a special purpose facility such as BIP/Myrinet [1], or over the global Internet. Each active component in BPFS must have a unique “location” in this network. There are two aspects to a “location” — its internal representation and its external representation — and the contents of each depends on the implementation of the network layer. The internal representation is an opaque value in network byte order, since locations are themselves sent over the network between potentially heterogeneous nodes. The external representation is an ASCII equivalent that can be stored as part of the human-readable metadata of a file.

2.2 FILES

In the BPFS model, a “file” is a contiguous sequence of logical blocks that are stored in “permanent” storage under a single name. Each block contains the same fixed number of 8-bit bytes except possibly the last block in a file, which can be smaller. A “parallel file” consists of one “metadata” component stored on a manager node, and one or more "data" components each stored on a separate server node. Different data components of the same file may contain different numbers of blocks. Depending on the implementation, the metadata component and each data component may be stored as normal files in the underlying host file system using a suitable naming convention.

A client maps a file onto a parallel file by a "mapping function", often called a "declustering function" or "striping function" in the literature. For each logical block in the file, this function determines which data component contains the block and the corresponding number of the relative block within that data component. Each file can have a different mapping function.

Metadata consists of the mapping function, the "thickness" of each stripe, and the list of server nodes on which data components are stored.

2.3 ACTIVE COMPONENTS

BPFS consists of the following active components:

1. Clients residing on client nodes, any number of clients per logical node. This software is implemented as a library which is called by user processes to communicate with managers and agents in order to accomplish all access to parallel files. The application calling the library can be a parallel or non-parallel process.

2. Managers residing on manager nodes, one manager per logical node. Each manager is an independent daemon process that creates, deletes and maintains metadata about a set of parallel files. Each file has exactly one manager, regardless of the number of clients accessing it.

3. Servers residing on server nodes, one server per logical node. Each server is an independent daemon process that creates, deletes and maintains the data components of a set of parallel files.

4. Agents residing on server nodes, any number of agents per logical node. Each agent controls one open file, performing all I/O and caching for it. An agent is a process or thread spawned and controlled by the server on that node. Once a file has been opened by the server, data flows directly between a client and the unique agent for that file, with no interference from the server or manager.

2.4 FUNCTIONS

BPFS contains the following functions by which the actions of the components can be tailored to particular application and system requirements:

1. Functions to supply a manager with a set of default metadata to use when creating new files.
2. Functions to convert between the internal and external representations of a network "location".
3. A function to map file names onto managers.
4. A function that gathers access control information necessary for clients to access files.
5. For each mapping, a set of functions to map logical block numbers onto data components and relative block numbers within a data component.
6. For each agent-side cache replacement policy, functions to add, remove, and reorder blocks on the cache list.

An implementation will define default values for each of these functions, and will provide a mechanism to allow users to specify alternative functions.

2.5 PROTOCOLS

BPFS specifies four protocols for communications between a subset of the active components: client and manager, client and agent, manager and server, and server and agent. More details can be found in [2].

These protocols are designed to be efficient on both 32- and 64-bit architectures. Numeric fields are never bigger than 32 bits, and major data structures within messages are aligned on 8-byte boundaries and are multiples of 8 bytes in size. However, the limitation to 32-bit values does not mean that file sizes are limited to 2^{32} bytes, because BPFS deals only with block numbers, not byte numbers. Therefore, a file is limited to BLOCKSIZE $\times$ 2^{32} bytes. If BLOCKSIZE is 4096 (2^{12}) bytes, the maximum file size would be 2^{44} bytes, more than 17 terabytes. Larger block sizes would allow correspondingly larger files, assuming the underlying file system is capable of storing and addressing such large files.

2.6 FIRST IMPLEMENTATION

All software for the first implementation of BPFS was written in Standard C for a "POSIX 1003.1 standard" UNIX extended with the BSD 4.3 socket interface to the TCP/IP protocol stack. Testing was performed on the following platforms:

- Linux 2.0.29, Intel Pentium Pro, 1 Gbps Myrinet and 10 Mbps ethernet.
- SunOS 5.5.1, SUN SPARC and i86pc, 10 Mbps ethernet.
- OSF1 4.0, DEC ALPHA, 10 Mbps ethernet.

In principle this software should be trivially portable onto any collection of UNIX-based workstations interconnected by a TCP/IP network.

Standard UNIX does not provide real-time or asynchronous I/O facilities. These have been simulated in this implementation using a software layer called the "q-interface" [3]. This layer provides time resolution of one microsecond for data streaming operations. It also isolates the use of TCP/IP networking from the rest of BPFS, so that to utilize a different protocol stack would require changing only this layer.

3 RELATED WORK

This work was inspired by the large number of parallel and distributed file systems that have been developed in the past few years.

RAMA [4] (Rapid Access to Massive Archive) was designed by Miller and Katz to utilize a system-wide hashing function for pseudo-randomly distributing blocks in a parallel file across many disks. This function is computable directly by a client application on any node, thereby giving each application direct access to any block without consulting a central node. BPFS generalizes this idea by allowing applications to provide arbitrary block mapping functions for each file.

PPFS [5] [6] (Portable Parallel File System) was developed by Reed and Chien as a portable user-level input/output library designed to permit rapid and flexible experimentation with policies for buffering, caching, data distribution and prefetching parallel files. It provides predefined policies for all necessary functions, but also allows users to specify file layouts, access patterns, and prefetching patterns. BPFS generalizes this notion to allow many file system functions to be provided by the application programmer and/or the system implementor. A difference in design philosophy is that BPFS is intended to be a utility that is always up and running, simultaneously serving all applications for all users on the physical hardware. With PPFS, each user's parallel virtual machine gets its own copy of PPFS that is independent of other copies in other virtual machines on the same physical hardware.

The work by Gibson on SPSS [7] (Scotch Parallel Storage Systems) incorporates into a running system earlier work by Patterson and Gibson [8] stressing the importance of prefetching and caching on parallel file system performance. BPFS provides two mechanisms for prefetching: the ability to request groups of blocks at once, and the ability to request streams of blocks in which the agent "pumps" data to the client at a specified rate. Control of these mechanisms lies in the layers above BPFS itself, as does any client-side caching mechanism. Agent-side caching on a per-file basis is provided directly in BPFS, although it can be controlled by higher levels. BPFS also follows the SPSS philosophy of providing just a basic parallel storage component within a broader parallel programming system. Neither BPFS nor SPSS provide any of the common synchronization or locking mechanisms that applications may need to properly coordinate parallel access to parallel files. Such mechanisms have to be provided by a "message passing" subsystem, such as MPI [9], that would be used in a layer on top of BPFS.

DPSS [10] (Distributed Parallel Storage System), developed by Tierney, is a collection of disk servers operating in parallel over a wide area network to provide logical block-level access to large data sets. It is implemented with user-level software for UNIX workstations interconnected by an ATM Internet. It was built with commercial, off-the-shelf (COTS) components, which is also a major goal for BPFS.

xFS [11], developed by Anderson, is based on the use of a striped log-structured file system in which new blocks are always written at the end of the available file storage space, and old blocks are reclaimed later via a background collection procedure. The complexity of this system made it difficult to implement and debug, and it required some changes to the operating system kernel, a serious impediment to portability. The design of BPFS attempts to avoid these complexities. It also avoids the cooperative caching designed into xFS, taking instead the same view as SPSS that such synchronization and process cooperation properly belong in higher-level layers above the basic file system.

The work of Kotz on the Galley file system [12] [13] maintains that a single strategy for various aspects of parallel file systems, such as caching, cannot suit all applications. He stresses the idea of building a simple, general purpose core storage system, and then layering a set of high-level libraries on top of it to provide many different application programmer interfaces (APIs). This is exactly the philosophy adopted by BPFS. The ViC* preprocessor [14] illustrates a high-level tool that is layered on top of a "commodity" parallel file system to provide a convenient API for manipulating large, "out of core" arrays. Even the MPI-IO interface [15], which is considered "low level" compared to ViC*, is "high level" for BPFS, and has been ported to run on top of BPFS using the ADIO (Abstract Device Interface for Implementing Portable Parallel-I/O Interfaces) [16].

4 INTERFACES

To date, three application program interfaces to BPFS have been implemented: API0, CLI, and MPI-IO. API0 is a direct interface to BPFS, while the others are implemented over API0, as shown in Figure 2. More details about these interfaces are given in [17].

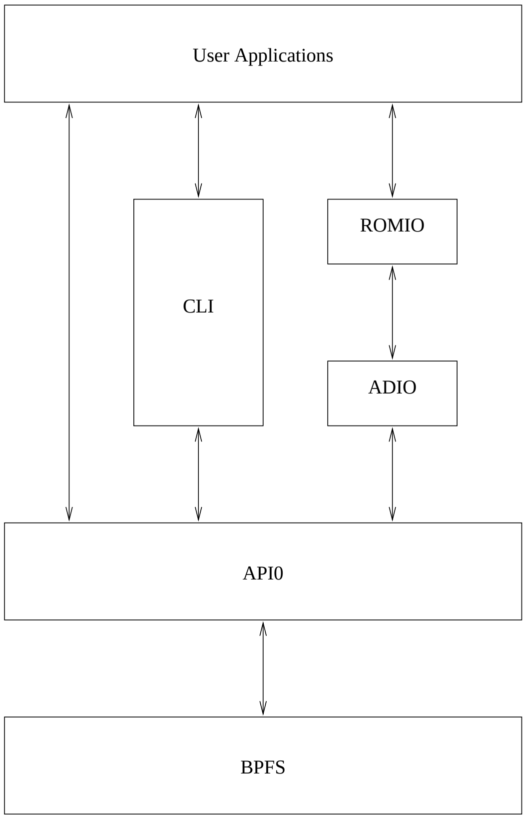

FIGURE 2: LAYERING OF THE API0, CLI, AND MPI-IO INTERFACES TO BPFS.

API0 is the first of an anticipated set of experimental, low-level client interfaces to BPFS. It provides direct access to the facilities of BPFS, and is "unconventional" in many respects:

- it is not particularly "UNIX-like",
- it is block-oriented rather than byte-oriented,
- it reads and writes system buffers as well as user-defined data areas,
- it has no notion of "current position" within a file, requiring a user to indicate a specific block number for every I/O operation,
- it is asynchronous in that functions generally come in pairs — the first in the pair to "start" an operation, the second in the pair to "wait-for" the completion of the operation,
- its functions all return NULL pointers on correct operation, pointers to a system buffer containing an error message on incorrect operation,
- it provides no communication or synchronization between clients,

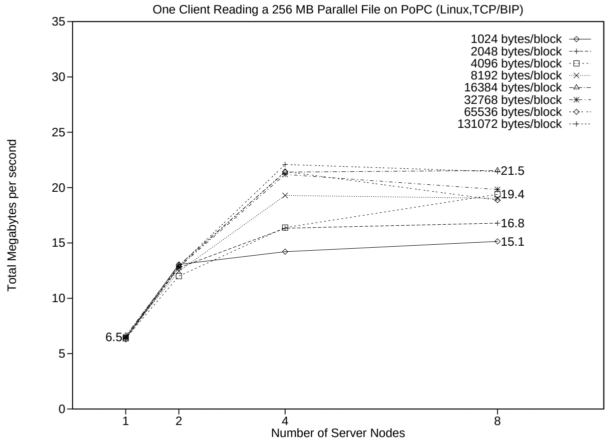

FIGURE 3: TOTAL MEGABYTES PER SECOND FOR ONE CLIENT PROCESS READING ONE PARALLEL FILE.

- it provides time-regulated "data streaming" as well as conventional request-reply operations,
- it provides user control over server-side caching,
- it provides user control over striping onto disks.

CLI is a "C Library Interface" implemented over API0 that mimics the Standard C I/O library interface but accesses BPFS rather than the host file system. This interface is designed for high-level applications that wish to access parallel files. These applications can be parallel programs or non-parallel programs that wish to access the parallel file system because of its greater bandwidth and lower latency than the normal host file system.

MPI-IO is the ROMIO version of the standard MPI-IO interface which has been implemented on top of API0 to support access to BPFS files from parallel programs that use MPI [15].

5 PERFORMANCE

A number of experiments have been performed to determine the performance of BPFS on a cluster of 200 MHz Pentium Pros with 64 Megabytes of memory and 4 Gigabyte SCSI disks, interconnected by a 1 Gigabit per second Myrinet, using the Linux 2.0 operating system and TCP/IP over BIP [1].

Figure 3 shows the number of megabytes per second available to a single client process as the block size and number of file servers varies. As expected, increasing the block size and the number of servers gives better performance. However, communication bottlenecks at the client's network interface cause the increase to flatten out after four servers, regardless of the block size.

Figure 4 shows the total number of megabytes per second provided by BPFS to multiple clients simulta-

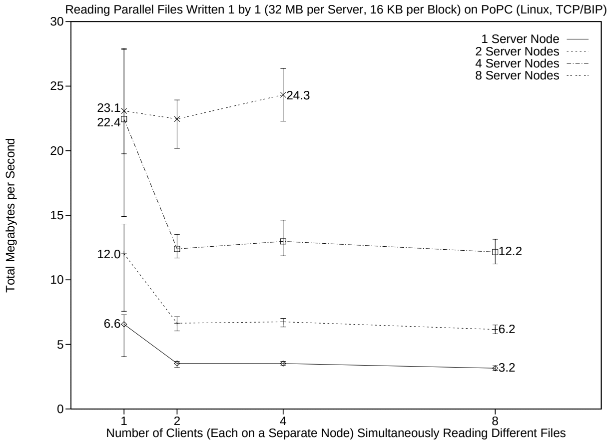

FIGURE 4: TOTAL MEGABYTES PER SECOND WHEN MULTIPLE CLIENTS READ FROM MULTIPLE SERVERS.

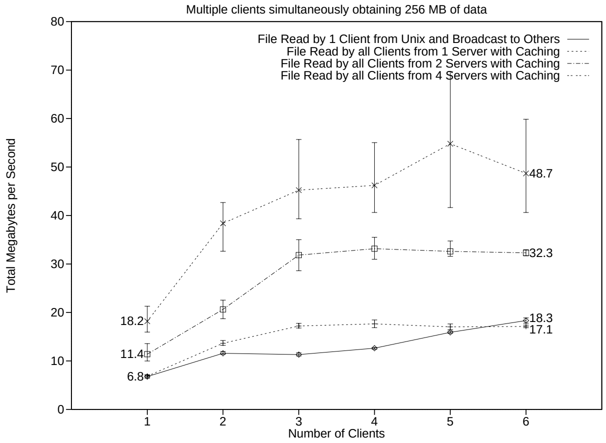

FIGURE 5: COMPARISON OF MPI BROADCAST VS BPFS READ FOR OBTAINING INITIALIZATION DATA.

neously reading different files. In this graph, the block size is fixed at 16k bytes per block, while the number of clients and number of servers varies.

Figure 4 illustrates two interesting points. The first is a sharp drop in total bandwidth when a second client becomes active, thereby forcing each server node to simultaneously accesses two files. This is due to an unexpected phenomenon of SCSI disks, reported in [18], in which simultaneous access to two or more different files on the same disk decreases performance by 50%.

The second observation is that scalability is good: as the number of servers increases up to four, speedup is nearly linear; as the number of clients increases beyond two, total available bandwidth remains flat.

Many parallel programs begin with an initialization phase, in which initial values are made available to all nodes participating in the parallel computation. There are at least two ways to do this. The first is to have one node read the initialization data from a file using a "normal" file system and then broadcast it to all other nodes. The second is to have all nodes simultaneously read the initialization file using a parallel file system. Figure 5 is a comparison of these two techniques. The solid line at the bottom of the figure is the total megabytes per second provided by the broadcast option, using MPI_Bcast and the host UNIX filesystem. The other lines give the bandwidth provided by BPFS with different numbers of servers. The block size is fixed at 16k bytes per block, while the number of clients varies. It is obvious that BPFS performance is superior, especially when there are multiple servers. Even when the file resides entirely on a single server, BPFS is at least as good as broadcasting. This is due to the effect of server-side caching in BPFS and the parallelism inherent in having direct data streams between each client-server pair.

6 CONCLUSION

BPFS has been implemented and tested on a number of platforms. Preliminary results are encouraging and demonstrate the potential for a simple, portable parallel file system to improve the performance of parallel computations on clusters of work-stations.

A group at ENS-Lyon is working on a more general parallel computation system of which BPFS is just one part. This system is intended to support real-time video streaming and editing applications. We are also currently in the process of porting the out-of-core array manipulation package ScaLAPACK to use BPFS.

We plan to try other networking technologies which promise much higher network bandwidth and lower latency than is possible with TCP/IP. Several high performance SAN protocols are now available for this purpose [1]. We would also like to try some of the newer disk technologies, such as network attached disks, to increase the performance of BPFS.

REFERENCES

[1] Loic Prylli and Bernard Tourancheau. BIP: A new protocol designed for high performance networking on myrinet. In *Workshop PC-NOW, IPPS/SPDP*, 1998.

[2] Robert D. Russell. The architecture of BPFS: a basic parallel file system, version 1.0. Technical Report RR98-21, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Lyon, France, March 1998.

[3] Robert D. Russell. The q-interface to UNIX. Technical Report TR98-03, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Lyon, France, July 1998.

[4] Ethan L. Miller and Randy H. Katz. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23(4):419–446, June 1997.

[5] Chris Elford, Chris Kuszmaul, Jay Huber, and Tara Madhyastha. Portable parallel file system detailed design. Technical report, University of Illinois at Urbana-Champaign, November 1993.

[6] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995. ACM Press.

[7] Garth A. Gibson, Daniel Stodolsky, Pay W. Chang, William V. Courtright II, Chris G. Demetriou, Eka Ginting, Mark Holland, Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachad Youssef, and Jim Zelenka. The Scotch parallel storage systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410, San Francisco, Spring 1995.

[8] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[9] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville, Tennessee, July 1997.

[10] William E. Johnston, Hanan Herzog, Gary Hoo, Guojun Jin, Jason Lee, Ling Tony Chen, and Doron Rotem. Distributed parallel data storage system: A scalable approach to high speed image servers. In *Proceedings of ACM Multimedia - San Francisco*, October 1994.

[11] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serveless network file systems. In *5th Symposium on Operating Systems Principles, ACM Transactions on Computer Systems*, 1995.

[12] David Kotz and Nils Nieuwejaar. Flexibility and performance of parallel file systems. *ACM Operating Systems Review*, 30(2):63–73, April 1996.

[13] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[14] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[15] MPI-2: Extensions to the message-passing interface. The MPI Forum, July 1997.

[16] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[17] Robert D. Russell. Application interfaces to BPFS: a basic parallel file system. Technical Report RR98-28, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Lyon, France, June 1998.

[18] Robert D. Russell. A study of end-user SCSI disk performance with parallel and non-parallel files. Technical Report RR98-38, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Lyon, France, August 1998.