Flexibility and Performance of Parallel File Systems

David Kotz   Nils Nieuwejaar

Department of Computer Science
Dartmouth College
Hanover, NH 03755
{dfk,nils}@cs.dartmouth.edu

February 14, 1996

Abstract

Many scientific applications for high-performance multiprocessors have tremendous I/O requirements. As a result, the I/O system is often the limiting factor of application performance. Several new parallel file systems have been developed in recent years, each promising better performance for some class of parallel applications. As we gain experience with parallel computing, and parallel file systems in particular, it becomes increasingly clear that a single solution does not suit all applications. For example, it appears to be impossible to find a single appropriate interface, caching policy, file structure, or disk management strategy. Furthermore, the proliferation of file-system interfaces and abstractions make application portability a significant problem.

We propose that the traditional functionality of parallel file systems be separated into two components: a fixed core that is standard on all platforms, encapsulating only primitive abstractions and interfaces, and a set of high-level libraries to provide a variety of abstractions and application-programmer interfaces (APIs). We think of this approach as the "RISC" of parallel file-system design.

We present our current and next-generation file systems as examples of this structure. Their features, such as a three-dimensional file structure, strided read and write interfaces, and I/O-node programs, are specifically designed with the flexibility and performance necessary to support a wide range of applications.

1 Introduction

Scientific applications are increasingly dependent on multiprocessor computers to satisfy their computational needs. Many scientific applications, however, also use tremendous amounts of data [dC94]: input data collected from satellites or seismic experiments, checkpointing output, and visualization output. Worse, some applications manipulate data sets too large to fit in main memory, requiring either explicit or implicit virtual memory support. The I/O system becomes the

This research was funded by NSF under grant number CCR-9404919 and by NASA Ames under agreement numbers NCC 2-849 and NAG 2-936.

bottleneck in all of these applications, a bottleneck that is worsening as processor speeds continue to improve more rapidly than disk speeds.

Fortunately, it is now possible to configure most parallel systems with sufficient I/O hardware [Kot96]. Most of today's parallel computers interconnect tens or hundreds of processor *nodes*, each of which has a processor and memory, with a high-speed network. Nodes with attached disks are usually reserved as *I/O nodes*, while applications run on some cluster of the remaining *compute nodes*.

In the past few years, many parallel file systems have been described in the literature, including Bridge/PFS [Dib90], CFS [Pie89], nCUBE [DdR92], OSF/PFS [Roy93], sfs [LIN$^+$93], Vesta/PIOFS [CFP$^+$95], HFS [KS96], PIOUS [MS94], RAMA [MK95], PPFS [HER$^+$95], Scotch [GSC$^+$95], and Galley [NK96a, NK96b]. Many more techniques for improving the performance of parallel file systems have been described, including caching and prefetching [KE93b, KE93a, PGG$^+$95], two-phase I/O [dBC93], disk-directed I/O [Kot94], compute-node caching [PEK96], chunking [SW94], compression [SW95], filtering [Kot95, BP88], and so forth.

The diversity of current systems and techniques indicates that there is clearly no consensus about the structure of, interface to, or even functionality of parallel file systems. Indeed, it seems that no one interface or structure will be appropriate for all parallel applications; for maximum performance, flexibility of the underlying system is critical [KS96]. It is important that applications be able to choose the interface and policies that work best for them, and for application programmers to have control over I/O [WGRW93, CK93].

This diversity of current systems, particularly of the application-programmer's interface (API), also makes it difficult to write portable applications. Nearly every file system mentioned above has its own API. A standard interface is being developed, MPI-IO [CFH$^+$95], but even that interface is appropriate only for a certain class of applications.

2 Solution

We believe that flexibility is needed for performance. An application programmer should be able to choose the interfaces and abstractions that work best for that application. To be practical, however, these interfaces and abstractions should be available on all platforms, so the application is portable, and each platform should support multiple interfaces and abstractions, so the platform is usable by many applications.

Consider Figure 1. Most traditional parallel file-system solutions attempt to provide a common file system that hopes to fit all applications. This common "core" file system is fixed, in that it must be used by all applications accessing parallel files.[1] To increase flexibility, we propose to move much of the functionality out of the core and into application libraries. Our new Galley Parallel File System takes this "RISC"-like approach.

The new core file system provides only a minimal set of services, leaving higher-level interfaces, semantics, and functionality to application-selectable libraries. While the implementation of the core is platform dependent, and provided by the platform vendor, its interface is standard across all platforms. This approach has proven successful with the MPI message-passing standard [MPI94].

Application programmers may then choose from a variety of different languages and libraries,

[1]We avoid the term "kernel," as the core may be comprised of user-level libraries, server daemons, and kernel code.

to select one that best fits the application's needs. Some languages or libraries would provide a traditional read-write abstraction; others (probably with compiler support) would provide transparent out-of-core data structures; still others may provide persistent objects. Some libraries may be designed for particular application classes like computational chemistry [FN96] or to support a particular language [CC94, CBH+94]. Finally, some compilers and programmers may choose to generate application-specific code using the core interface directly.

The concept of I/O libraries is not new; the C `stdio` library and the C++ `iostreams` library are common examples, both layered above the "core" kernel interface. Yet few parallel file systems have been designed specifically to support a variety of high-level libraries. The difficulty is in deciding how to divide features between the core and the application libraries, and then in designing an appropriate core interface. In our research to explore this issue, we are building two generations of file systems. In the first, Galley, we investigate the underlying file abstraction, a low-level read/write interface, and resource-scheduling alternatives. In the second, with the tentative name Galley2, we go a step further and allow user code to run on the I/O nodes. The next two sections discuss each file system in more detail.

3 The Galley Parallel File System

Our current parallel file system, Galley [NK96a, NK96b], looks like Figure 1b. A more detailed picture is shown in Figure 2. The core file system includes servers that run on the I/O nodes and a tiny interface library that runs on the compute nodes. The I/O-node servers manage file-system metadata, I/O-node caching, and disk scheduling. The interface library translates library calls into messages to servers on the I/O nodes and arranges the movement of data between compute and I/O nodes. The higher-level application library, if any, is responsible for providing a convenient API, data declustering, file-access semantics, and any compute-node caching.

Galley's servers provide a unified global file-name space. Each file is actually a collection of *subfiles*, each of which resides entirely on one I/O node. Each subfile is itself a collection of one or more named *forks*. Each fork is a sequence of bytes, the traditional file abstraction. Galley's core file system provides no automatic data declustering; a library may choose to stripe data across subfiles, for example.

Galley's forks are specifically designed to support libraries. In particular, some libraries may wish to store metadata in one or more forks of the subfile, with data in other forks. The traditional approach is to place the metadata in an auxiliary file or in a "header" at the beginning of the data. The former approach makes file management awkward, as there is more than one file name involved in a single data set. The latter approach makes it difficult to access the file through multiple libraries, each of which expects its own header, and can complicate declustering calculations. In Galley each library can add its own fork to the subfiles, containing its own metadata.

The structure of parallel files, beyond the fact that they are collections of local files, is completely determined by library code. Multiple applications wishing to use the same parallel files must maintain a mutually agreed structure, by convention.

In an extensive characterization of parallel scientific applications [NKP+95], we found that many applications access files in small pieces, typically in a regular "strided" pattern. To allow application libraries to support these patterns efficiently, the Galley interface supports both structured (e.g., strided and nested strided) and unstructured read and write requests. This interface leads to dramatically better performance [NK96b].

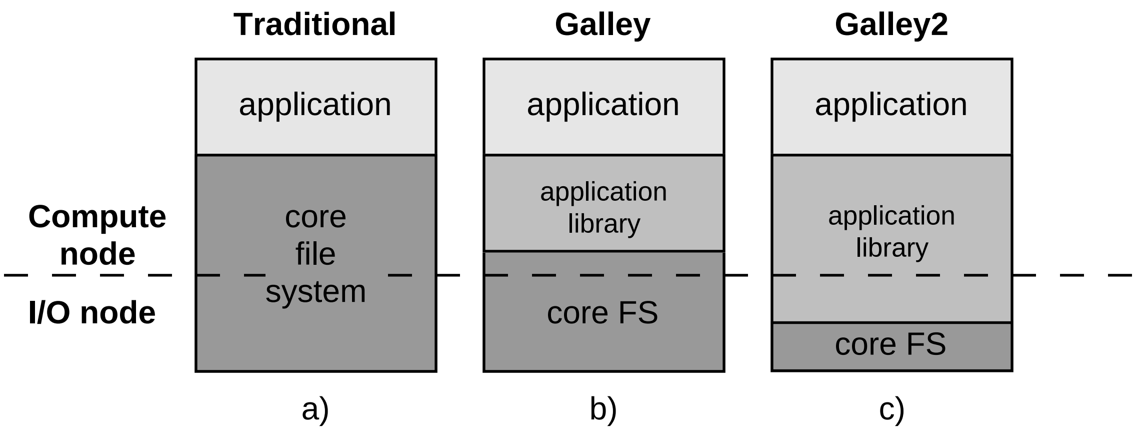

Figure 1: Our proposed evolution of parallel file-system structure. Traditional systems depend on a fixed "core" file system that attempts to serve all applications. In our Galley File System, we shrink the core to leave the API and many of the parallel features to an application-selectable library. In our next-generation Galley2 File System, we shrink the core further to allow user-selected code to run on the I/O nodes.

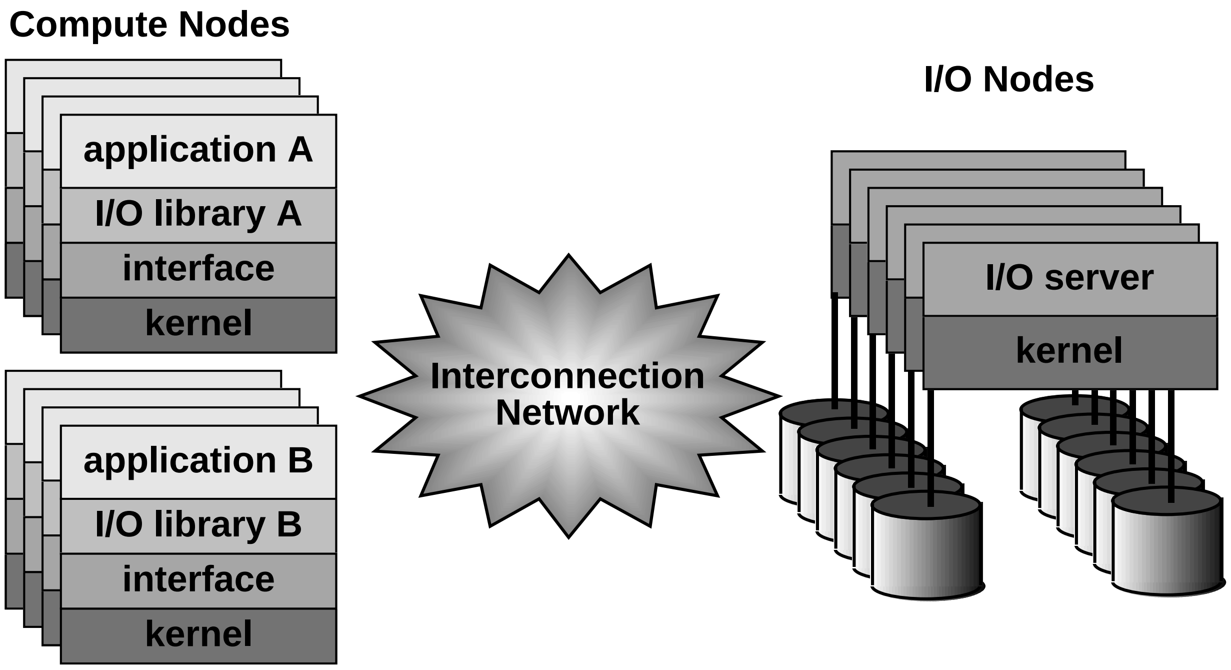

Figure 2: The structure of the Galley parallel file system includes a tiny interface library on the compute node, which coordinates communication between application I/O libraries on the compute nodes and servers on the I/O nodes.

Galley's features, including the global name space, three-dimensional file structure, and structured read and write requests, make it a suitable and efficient base for constructing parallel file systems, much more so than building directly on distributed Unix systems.

More information about Galley is available on the WWW[2] and in forthcoming papers [NK96a, NK96b].

4 The Galley2 Parallel File System

Our next-generation file system, which we so far call "Galley2" for lack of a better name, goes beyond Galley to allow application control over I/O-node activities. We keep the same three-dimensional file structure of subfiles and forks, and we keep the global name space, but we otherwise reduce the core file system to a minimal local file system on each I/O node, and allow application-supplied code to run on the I/O nodes (see Figure 1c). Indeed, we expect that an I/O node would have an active process (or thread) for each application with files on that I/O node. Figure 3 gives a more detailed picture of this structure.

This structure breaks away from the traditional client-server structure to allow for "programmable" servers. A fixed, common server always forces designers to choose between specific high-level services that may not fit the needs of all applications, and primitive low-level operations that permit flexibility in the clients but at the cost of extensive client-server communications. Galley makes a reasonable choice here, but (for example) uses a fixed caching policy.

In Galley2 the core file system is extremely simple: there is no caching, prefetching, or remote access. It provides a (local) interface to open, close, read and write forks through a block-level interface, and it arbitrates among I/O-node programs competing for processor time, memory, disk access, and network access. In short, it focuses on the shared aspects of the file system.

Thus, Galley2 applications can choose nearly all features of the parallel file system, including the API, caching, prefetching, declustering, inter-node communication protocols, synchronization and consistency, and so forth. Again, we expect most applications to choose from pre-defined libraries, but we also encourage use of application-specific code written by application programmers, generated automatically by compilers, or generated at run time [PAB$^+$95]. We refer to all of these choices as "application-selected code."

There are many reasons to allow application-selected code on the I/O node. Application-specific optimizations can be applied to I/O-node caching and prefetching. Mechanisms like disk-directed I/O [Kot94] can be implemented, using application-specific data-distribution information. File data can be distributed among memories according to a data-dependent mapping function, for example, in applications with a data-dependent decomposition of unstructured data [Kot95]. Incoming data can be filtered in a data-dependent way, passing only the necessary data on to the compute node, saving network bandwidth and compute-node memory [Kot95, BP88]. Blocks can be moved directly between I/O nodes, for example, to rearrange blocks between disks during a copy or permutation operation, without passing through compute nodes. Format conversion, compression, and decompression are also possible. In short, there are many ways that we can optimize memory and disk activity at the I/O node, and reduce disk and network traffic, by moving what is essentially application code to run at the I/O node in addition to the compute nodes.

Although it would be feasible to use a Unix file system as the local file system, the semantics

[2] `http://www.cs.dartmouth.edu/~nils/galley.html`

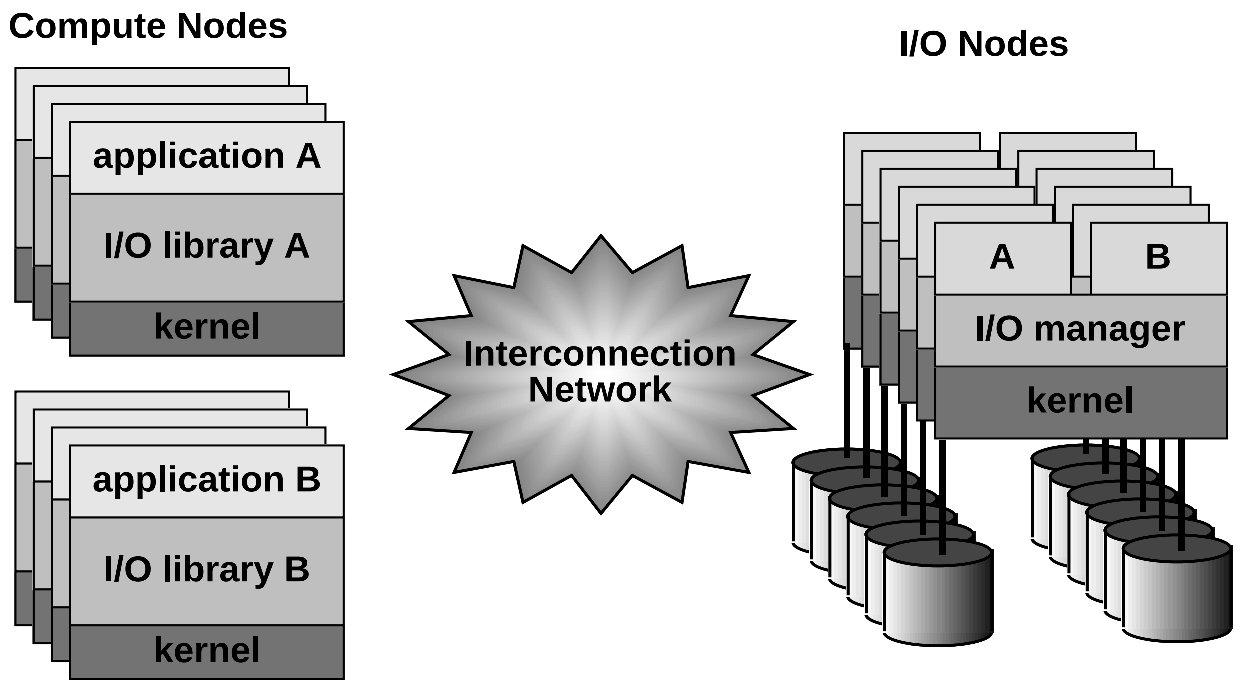

Figure 3: The structure of the Galley2 parallel file system depends on application I/O libraries that have components on both the compute and I/O nodes. The I/O-node servers shrink down to simple I/O managers that arbitrate resources among the local user-selected library modules.

and interface are not appropriate for the highest performance. In particular, the Unix file-system interface does not give the applications enough control, would have no global name space, and has an inefficient copy-based interface.

5 Research directions

The success of our design clearly depends on the ability of the I/O-node operating system to efficiently manage its resources while providing the necessary functionality. We are exploring the following issues:

- resource management: how should the I/O node manage its shared resources in the presence of competing applications? The result must be a tradeoff between overall system throughput and individual application performance. Traditional uniprocessor policies do not directly apply to this distributed situation; local resource decisions can have a disproportionate global impact on performance.
- physical memory allocation: how should we best allocate physical memory among I/O-node programs?
- processor scheduling: how shall we schedule the CPU among I/O-node programs? What about applications that choose to move some non-I/O-related computation to the I/O node?
- disk transfers: what is an appropriate interface for requesting I/O to and from buffers?
- message-passing: what is the best interface for I/O-node programs to communicate with the compute nodes, and with each other?
- What is the appropriate mechanism to support I/O-node programs? We are considering three alternatives: processes, threads within a safe language like Java [GM94] or Python[3], and threads running sandboxed code [WLAG93]. There are three primary issues in this consideration:
    1. how is the I/O-node manager protected from I/O-node programs? With normal hardware protection, in the case of processes; with type-safe languages like Java; or with sandboxing.
    2. how is the code loaded onto the I/O node? Presumably they can be loaded from disk in the same way as the compute-node code. The tricky part might be dynamic linking of sandboxed code.
    3. what is the overhead?

6 Related work

The Hurricane File System (HFS) [KS96], a parallel file system for the Hector multiprocessor, is also designed with the philosophy that flexibility is critical for performance. Indeed, their results clearly demonstrate the tremendous performance impact of choosing the right file structure and

[3] `http://www.python.org/`

management policies for the application's access pattern. HFS is actually a collection of building-block objects that can be plugged together differently according to application needs. For example, some building blocks distribute data across multiple disks, others provide prefetching policies, and others define an API. HFS allows the programmer to replace or extend application-level building blocks, but these do not include the objects that control declustering, replication, parity, or other server-side attributes. Galley permits, but does not enforce, a building-block approach to library design; other approaches are possible. Finally, the Hurricane operating system does not dedicate nodes to I/O, so it is not unusual for application code to run on "I/O" nodes.

The Portable Parallel File System (PPFS) [HER+95] is a testbed for experimenting with parallel file-system issues. It includes many alternative policies for declustering, caching, prefetching, and consistency control, and allows application programmers to select appropriate policies for their needs. It also supports user-defined declustering patterns through an upcall function. Unlike Galley, however, there is no clearly defined lower-level interface to which programmers may write new high-level libraries. Unlike Galley2, it does not allow application-selected code (beyond that already included in PPFS) to execute on the I/O nodes.

In the Transparent Informed Prefetching (TIP) system [PGG+95] an application provides a set of *hints* about its future accesses to the file system. The file system uses these hints to make intelligent caching and prefetching decisions. While this technique can lead to better performance through better prefetching, it only affects prefetching and caching behavior. It is possible to provide "hints that disclose," in their words, for other aspects of the system, but it is unclear that these hints can provide the same amount of flexibility offered by Galley and Galley2.

All three of these systems provide the application programmer some control over the parallel file system, primarily by selecting existing policies from the built-in alternatives.

Galley2 promotes the use of application-selected code on the I/O nodes. Several operating systems can download user code into the kernel [Gai72, LCC94, BSP+95]. Other researchers have noted that it is useful to move the function to the data rather than to move the data to the function [CBZ95, SG90, Gra95]. Some distributed database systems execute part of the SQL query in the server rather than the client, to reduce client-server traffic [BP88]. Hatcher and Quinn hint that allowing user code to run on nCUBE I/O nodes would be a good idea [HQ91].

7 Status

Galley runs on the IBM SP-2 and on workstation clusters [NK96a], and has so far been extremely successful [NK96b]. We are currently porting several application libraries on top of Galley, including a traditional striped-file library, Panda [SCJ+95], Vesta [CFP+95], and SOLAR [TG96]. We are also using Galley to investigate policies for managing multi-application workloads.

We are building a simulator for Galley2, to evaluate some of the key ideas, and a full implementation, to experiment with real applications. There is no question that it will be a much more flexible system than Galley and its predecessors. We will declare success if that flexibility provides better performance on a wider range of applications. That will occur if the benefits of application-specific I/O-node programs outweigh the cost of the extension mechanism (sandboxing, context switching, or interpretation). We are optimistic!

More information about our research can be found at

```
http://www.cs.dartmouth.edu/research/pario.html
```

Interested readers should also plan to visit the upcoming Workshop on I/O in Parallel and Distributed Systems (IOPADS), at FCRC on May 27, 1996. See

`http://www.cs.dartmouth.edu/iopads/`

References

[BP88] Andrea J. Borr and Franco Putzolu. High performance SQL through low-level system integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 342–349, 1988.

[BSP+95] Brian Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[CBH+94] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[CBZ95] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[CC94] 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.

[CFH+95] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: a parallel file I/O interface for MPI. Technical Report NAS-95-002, NASA Ames Research Center, January 1995. Version 0.3.

[CFP+95] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, pages 222–248, 1995.

[CK93] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Revised as Dartmouth PCS-TR93-188 on 9/20/94.

[dBC93] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[dC94] Juan Miguel del Rosario and Alok Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[DdR92] Erik DeBenedictis and Juan Miguel del Rosario. nCUBE parallel I/O software. In *Proceedings of the Eleventh Annual IEEE International Phoenix Conference on Computers and Communications*, pages 0117–0124, April 1992.

[Dib90] Peter C. Dibble. *A Parallel Interleaved File System*. PhD thesis, University of Rochester, March 1990.

[FN96] Ian Foster and Jarek Nieplocha. ChemIO: High-performance I/O for computational chemistry applications. WWW http://www.mcs.anl.gov/chemio/, February 1996.

[Gai72] R. Stockton Gaines. An operating system based on the concept of a supervisory computer. *Communications of the ACM*, 15(3):150–156, March 1972.

[GM94] James Gosling and Henry McGilton. The Java language: A white paper. Sun Microsystems, 1994.

[Gra95] Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995.

[GSC+95] Garth A. Gibson, Daniel Stodolsky, Pay W. Chang, William V. Courtwright 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.

[HER+95] 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.

[HQ91] Philip J. Hatcher and Michael J. Quinn. C*-Linda: A programming environment with multiple data-parallel modules and parallel I/O. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, pages 382–389, 1991.

[KE93a] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17(1–2):140–145, January and February 1993.

[KE93b] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[Kot95] David Kotz. Expanding the potential for disk-directed I/O. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, pages 490–495, October 1995.

[Kot96] David Kotz. Introduction to multiprocessor I/O architecture. In Ravi Jain, John Werth, and J. C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*. Kluwer Academic Publishers, January 1996. To appear.

[KS96] Orran Krieger and Michael Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 95–108, Philadelphia, May 1996.

[LCC94] Chao Hsien Lee, Meng Chang Chen, and Ruei Chuan Chang. HiPEC: High performance external virtual memory caching. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 153–164, 1994.

[LIN+93] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[MK95] Ethan L. Miller and Randy H. Katz. RAMA: Easy access to a high-bandwidth massively parallel file system. In *Proceedings of the 1995 Winter USENIX Conference*, pages 59–70, January 1995.

[MPI94] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1.0 edition, May 5 1994. http://www.mcs.anl.gov/Projects/mpi/standard.html.

[MS94] Steven A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[NK96a] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996. To appear.

[NK96b] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, May 1996.

[NKP+95] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. Technical Report PCS-TR95-263, Dept. of Computer Science, Dartmouth College, August 1995. Submitted to IEEE TPDS.

[PAB+95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 314–324, December 1995.

[PEK96] Apratim Purakayastha, Carla Schlatter Ellis, and David Kotz. ENWRICH: a compute-processor write caching scheme for parallel file systems. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 55–68, May 1996.

[PGG+95] 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, December 1995.

[Pie89] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160. Golden Gate Enterprises, Los Altos, CA, March 1989.

[Roy93] Paul J. Roy. Unix file access and caching in a multicomputer environment. In *Proceedings of the Usenix Mach III Symposium*, pages 21–37, 1993.

[SCJ+95] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995.

[SG90] James W. Stamos and David K. Gifford. Remote execution. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[SW94] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, November 1994.

[SW95] K. E. Seamons and M. Winslett. A data management approach for handling large compressed arrays in high performance computing. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 119–128, February 1995.

[TG96] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 28–40, Philadelphia, May 1996.

[WGRW93] David Womble, David Greenberg, Rolf Riesen, and Stephen Wheat. Out of core, out of mind: Practical parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–16, Mississippi State University, October 1993.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.