Flexibility and Performance of Parallel File Systems

David Kotz and Nils Nieuwejaar

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

Abstract. As we gain experience with parallel file systems, 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 applications difficult to port.

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 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 [11]: 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 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 [22]. Most of today's parallel computers interconnect tens

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.

This paper appeared previously in *ACM Operating Systems Review* 30(2), April 1996, pp. 63–73. The only changes are the format, a shorter abstract, and updates to Section 7 and the references.

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 [12], CFS [35], nCUBE [9], OSF/PFS [38], sfs [27], Vesta/PIOFS [6], HFS [25], PIOUS [30], RAMA [29], PPFS [19], Scotch [15], and Galley [31, 32]. Many more techniques for improving the performance of parallel file systems have been described, including caching and prefetching [24, 23, 34], two-phase I/O [10], disk-directed I/O [20], compute-node caching [37], chunking [40], compression [41], filtering [21, 2], 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 [25]. 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 [46, 8].

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 [5], 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 [28].

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

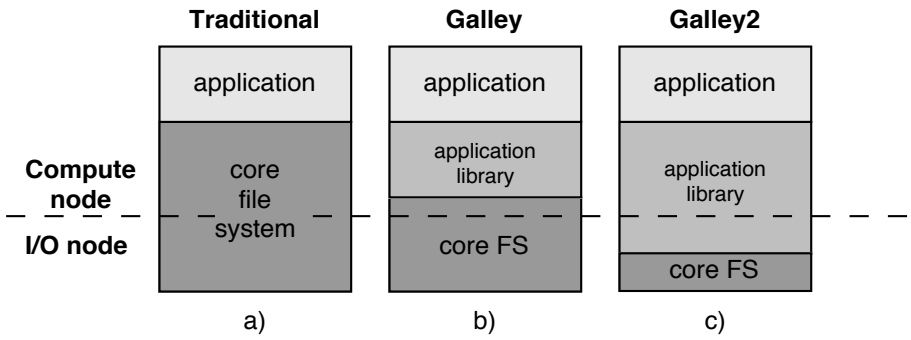

Fig. 1. Our proposed evolution of parallel file-system structure. Traditional systems depend on a fixed "core" file system that attempts to serve all applications. In our Galley File System, we shrink the core to leave the API and many of the parallel features to an application-selectable library. In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

Application programmers may then choose from a variety of different languages and libraries, 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 [13] or to support a particular language [7, 4]. 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 [31, 32], 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.

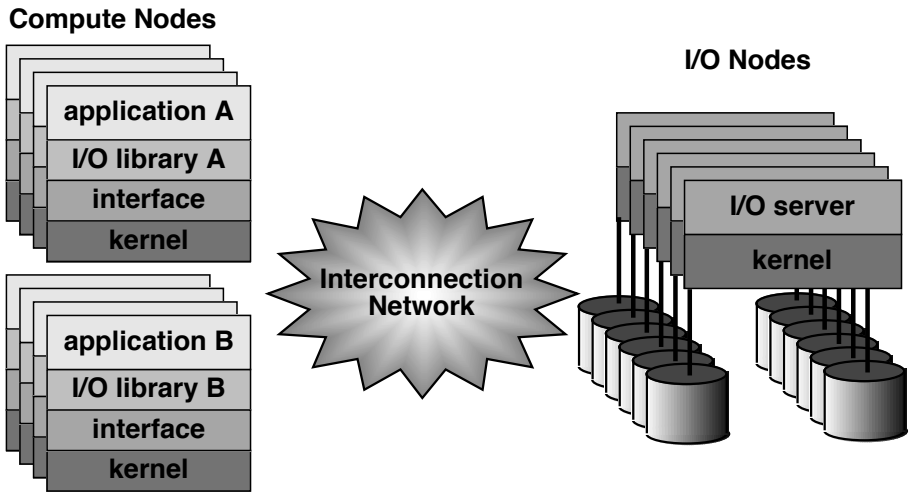

Fig. 2. The structure of the Galley parallel file system includes a tiny interface library on the compute node, which coordinates communication between application I/O libraries on the compute nodes and servers on the I/O nodes.

Galley's 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 [33], 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 [32].

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 [31, 32].

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 [36]. We refer to all of these choices as "application-selected code."

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

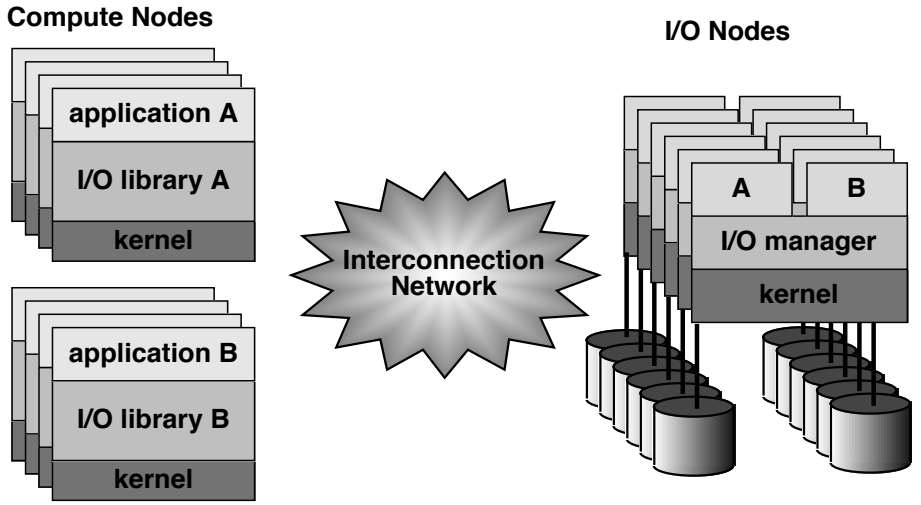

Fig. 3. The structure of the Galley2 parallel file system depends on application I/O libraries that have components on both the compute and I/O nodes. The I/O-node servers shrink down to simple I/O managers that arbitrate resources among the local user-selected library modules.

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 [20] 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 [21]. 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 [21, 2]. 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 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 [16] or Python[3], and threads running sandboxed code [45]. 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) [25], 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 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

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

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) [19] 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 [34] 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 [14, 26, 1]. Other researchers have noted that it is useful to move the function to the data rather than to move the data to the function [3, 42, 17]. Some distributed database systems execute part of the SQL query in the server rather than the client, to reduce client-server traffic [2]. Hatcher and Quinn hint that allowing user code to run on nCUBE I/O nodes would be a good idea [18].

7 Status

Galley runs on the IBM SP-2 and on workstation clusters [31], and has so far been extremely successful [32]. We have ported several application libraries on top of Galley, including a traditional striped-file library, Panda [39, 43], Vesta [6], and SOLAR [44]. 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`

References

1. B. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of the 15th ACM SOSP*, pages 267–284, Dec. 1995.
2. A. J. Borr and F. Putzolu. High performance SQL through low-level system integration. In *Proc. of the ACM SIGMOD Conf.*, pages 342–349, 1988.
3. J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM TOCS*, 13(3):205–243, Aug. 1995.
4. A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, Sept. 1994.
5. P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: a parallel file I/O interface for MPI. Technical Report NAS-95-002, NASA Ames Research Center, Jan. 1995. Version 0.3.
6. P. F. Corbett, D. G. Feitelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T. R. Morgan, and A. Zlotek. Parallel file systems for the IBM SP computers. *IBM Sys. Journal*, 34(2):222–248, Jan. 1995.
7. T. H. Cormen and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, Nov. 1994.
8. T. H. Cormen and D. Kotz. Integrating theory and practice in parallel file systems. In *Proc. of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Inst. for Adv. Graduate Studies. Revised as Dartmouth PCS-TR93-188 on 9/20/94.
9. E. DeBenedictis and J. M. del Rosario. nCUBE parallel I/O software. In *Proc. of the 11th IPCCC*, pages 0117–0124, Apr. 1992.
10. J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on I/O in Par. Comp. Sys.*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.
11. J. M. del Rosario and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, Mar. 1994.
12. P. C. Dibble. *A Parallel Interleaved File System.* PhD thesis, University of Rochester, Mar. 1990.
13. I. Foster and J. Nieplocha. ChemIO: High-performance I/O for computational chemistry applications. WWW http://www.mcs.anl.gov/chemio/, Feb. 1996.
14. R. S. Gaines. An operating system based on the concept of a supervisory computer. *Comm. of the ACM*, 15(3):150–156, Mar. 1972.

15. G. A. Gibson, D. Stodolsky, P. W. Chang, W. V. Courtright II, C. G. Demetriou, E. Ginting, M. Holland, Q. Ma, L. Neal, R. H. Patterson, J. Su, R. Youssef, and J. Zelenka. The Scotch parallel storage systems. In *Proc. of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410, San Francisco, Spring 1995.
16. J. Gosling and H. McGilton. The Java language: A white paper. Sun Microsystems, 1994.
17. R. 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, Dec. 1995.
18. P. J. Hatcher and M. J. Quinn. C*-Linda: A programming environment with multiple data-parallel modules and parallel I/O. In *Proc. of the 24th HICSS*, pages 382–389, 1991.
19. J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proc. of the 9th ACM Int'l Conf. on Supercomp.*, pages 385–394, Barcelona, July 1995.
20. D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proc. of the 1994 Symp. on OS Design and Impl.*, pages 61–74, Nov. 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.
21. D. Kotz. Expanding the potential for disk-directed I/O. In *Proc. of the 1995 IEEE SPDP*, pages 490–495, Oct. 1995.
22. D. Kotz. Introduction to multiprocessor I/O architecture. In R. Jain, J. Werth, and J. C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 4, pages 97–123. Kluwer Academic Publishers, 1996.
23. D. Kotz and C. S. Ellis. Caching and writeback policies in parallel file systems. *J. of Par. and Dist. Comp.*, 17(1–2):140–145, January and February 1993.
24. D. Kotz and C. S. Ellis. Practical prefetching techniques for multiprocessor file systems. *J. of Dist. and Par. Databases*, 1(1):33–51, Jan. 1993.
25. O. Krieger and M. Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. In *4th Workshop on I/O in Par. and Dist. Sys.*, pages 95–108, Philadelphia, May 1996.
26. C. H. Lee, M. C. Chen, and R. C. Chang. HiPEC: High performance external virtual memory caching. In *Proc. of the 1994 Symp. on OS Design and Impl.*, pages 153–164, 1994.
27. S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler. *sfs*: A parallel file system for the CM-5. In *Proc. of the 1993 Summer USENIX Conf.*, pages 291–305, 1993.
28. 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.
29. E. L. Miller and R. H. Katz. RAMA: Easy access to a high-bandwidth massively parallel file system. In *Proc. of the 1995 Winter USENIX Conf.*, pages 59–70, Jan. 1995.
30. S. A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proc. of the Scalable High-Perf. Comp. Conf.*, pages 71–78, 1994.
31. N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proc. of the 10th ACM Int'l Conf. on Supercomp.*, pages 374–381, May 1996.
32. N. Nieuwejaar and D. Kotz. Performance of the Galley parallel file system. In *4th Workshop on I/O in Par. and Dist. Sys.*, pages 83–94, May 1996.

33. N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best. File-access characteristics of parallel scientific workloads. Technical Report PCS-TR95-263, Dept. of Computer Science, Dartmouth College, Aug. 1995. To appear in IEEE TPDS.
34. R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. of the 15th ACM SOSP*, pages 79–95, Dec. 1995.
35. P. Pierce. A concurrent file system for a highly parallel mass storage system. In *Proc. of the Fourth Conf. on Hypercube Concurrent Comp. and Appl.*, pages 155–160. Golden Gate Enterprises, Los Altos, CA, Mar. 1989.
36. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. of the 15th ACM SOSP*, pages 314–324, Dec. 1995.
37. A. Purakayastha, C. S. Ellis, and D. Kotz. ENWRICH: a compute-processor write caching scheme for parallel file systems. In *4th Workshop on I/O in Par. and Dist. Sys.*, pages 55–68, May 1996.
38. P. J. Roy. Unix file access and caching in a multicomputer environment. In *Proc. of the Usenix Mach III Symposium*, pages 21–37, 1993.
39. K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proc. of Supercomp. '95*, Dec. 1995.
40. K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proc. of Supercomp. '94*, pages 650–659, Nov. 1994.
41. K. E. Seamons and M. Winslett. A data management approach for handling large compressed arrays in high performance computing. In *Proc. of the 5th Symp. on the Frontiers of Massively Par. Comp.*, pages 119–128, Feb. 1995.
42. J. W. Stamos and D. K. Gifford. Remote execution. *ACM TOPLAS*, 12(4):537–565, Oct. 1990.
43. J. T. Thomas. The Panda array I/O library on the Galley parallel file system. Technical Report PCS-TR96-288, Dept. of Computer Science, Dartmouth College, June 1996. Senior Honors Thesis.
44. S. Toledo and F. G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *4th Workshop on I/O in Par. and Dist. Sys.*, pages 28–40, Philadelphia, May 1996.
45. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of 14th ACM SOSP*, pages 203–216, 1993.
46. D. Womble, D. Greenberg, R. Riesen, and S. Wheat. Out of core, out of mind: Practical parallel I/O. In *Proc. of the Scalable Par. Libraries Conf.*, pages 10–16, Mississippi State University, Oct. 1993.

This article was processed using the LaTeX macro package with LLNCS style