PPFS: A High Performance Portable Parallel File System

James V. Huber, Jr.* Christopher L. Elford* Daniel A. Reed*
Andrew A. Chien* David S. Blumenthal*

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

Rapid increases in processor performance over the past decade have outstripped performance improvements in input/output devices, increasing the importance of input/output performance to overall system performance. Further, experience has shown that the performance of parallel input/output systems is particularly sensitive to data placement and data management policies, making good choices critical. To explore this vast design space, we have developed a user-level library, the Portable Parallel File System (*PPFS*), which supports rapid experimentation and exploration. The *PPFS* includes a rich application interface, allowing the application to advertise access patterns, control caching and prefetching, and even control data placement. *PPFS* is both extensible and portable, making possible a wide range of experiments on a broad variety of platforms and configurations. Our initial experiments, based on simple benchmarks and two application programs, show that tailoring policies to input/output access patterns yields significant performance benefits, often improving performance by nearly an order of magnitude.

1 Introduction

The widespread acceptance of massively parallel systems as the vehicle of choice for high-performance computing has produced a wide variety of machines and an even wider variety of potential input/output configurations, most with inadequate input/output capacity and performance. This disparity between computation and input/output rates reflects similar limitations on sequential systems. Processor performance has increased by several orders of magnitude in the last decade, but improvements in secondary storage access times have not kept pace (e.g., disk seek times have decreased by less than a factor of two even as data density has increased dramatically).

In an effort to balance impressive peak processing rates with sufficient input/output bandwidth, most parallel systems support multiple, redundant arrays of inexpensive disks (RAIDs) [21, 12]. Although multiple RAIDs provide *peak* input/output bandwidth equal to the product of the number of RAIDs and their individual bandwidths, little is known about the *effective* input/output bandwidth of an array of RAIDs in parallel scientific computing environments.

Developing models for how such massively parallel systems should be configured to be balanced, and how to manage the input/output resources to achieve high *sustained* performance requires exploration of a broad space of choices (e.g., disk data striping factors, file and disk block prefetch policies, and caching policies). Simply put, we must determine the most effective combinations of buffering, caching, data distribution 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.

To explore these issues, we have developed a Portable Parallel File System (*PPFS*) to study the interaction of application access patterns, file caching and prefetching algorithms, and application file data distributions. *PPFS* consists of a user-level input/output library that is portable across both parallel systems and workstation clusters, permitting a wide range of experimentation with modest human resources. *PPFS* allows application control of a variety of file caching, prefetching, data layout, and coherence policies. This enables rapid exploration of a wide range of possibilities, facilitating rapid convergence on good policies for each file in the application. The *PPFS* application interface supports a number of predefined policies, but can also extend its repertoire by accepting user-defined layouts, access pattern declarations, and prefetching patterns. Interposing the libraries between the application and the system software has allowed us to more quickly experiment with a variety of data distribution and data management algorithms than would be possible via system software modifications.

In a preliminary study, we have measured the performance of *PPFS* on simple input/output benchmarks and on two input/output intensive programs, a gene sequencing matching application and a low temperature plasma electron scattering code. All of these experiments were conducted on a variety of Intel Paragon XP/S hardware configurations, using the Intel parallel file system (PFS)

*Supported in part by the National Science Foundation under grant NSF ASC 92-12369, by the National Aeronautics and Space Administration under NASA Contracts NGT-51023, NAG-1-613, and USRA 5555-22 and by the Advanced Research Projects Agency under ARPA contracts DAVT63-91-C-0029 and DABT63-93-C-0040. Andrew Chien is supported in part by NSF Young Investigator Award CDA-94-01124.

ICS '95 Barcelona, Spain 1995 ACM 0-89791-726-6/95/0007..$3.50

and *PPFS* atop multiple Intel Paragon Unix file systems (UFS).

The simple benchmarks show that *PPFS* performance is less dependent than PFS on changes in request size or the number of concurrently active input/output operations. The two larger application codes have dramatically different input/output behavior (the genome code is read-intensive and the electron scattering code is write-intensive), but *PPFS* demonstrates significant performance benefits for each, improving performance by nearly an order of magnitude in some cases.

With this context, the remainder of the paper is organized as follows. In §2 we elaborate on the motivations for a portable parallel file system. This is followed in §3–§4 by a detailed description of the *PPFS* design philosophy, design overview, examples of its use, and current implementation status. In §5 we analyze the performance of our current implementation, based on simple input/output benchmarks. In §6, we describe two large application codes, their input/output behavior, and the performance improvements possible with *PPFS*. Finally, §7 describes related work, summarizes our results and discusses current research directions.

2 Motivations

Input/output performance depends on the distribution of file data across storage devices, the file caching and prefetching algorithms, *and* parallel, spatio-temporal file access patterns. Simply put, the potential design space of input/output software systems is enormous, and our current knowledge is extremely limited. Hence, it is critical that we develop experimental methodologies that allow us to quickly explore large portions of the software design space.

In turn, rapid exploration requires broad control over file management policies. Few extant file systems provide such malleable interfaces, and the effort to build a low-level parallel file system is substantial. Moreover, to be credible, building such a system requires access to commercial operating system source code; this is rarely possible or practical.

We believe that many of the important adaptivity issues are more easily explored via a portable parallel file system (*PPFS*). A *PPFS* implementation interposes a portable input/output library between the application and a vendor's basic system software. In this model, a parallel file consists of a collection of standard vendor-supported files and a set of file metadata that describes the mapping from these files to a single *PPFS* parallel file.

This approach allows one to explore a variety of caching and prefetching policies and their performance tradeoffs on disparate architectures, without becoming mired in costly and time-consuming modifications to system software. Although this compromise does sacrifice some low-level control (e.g., disk block placement and disk arm scheduling), we have found that it is repaid in increased experimental flexibility and coverage.

Based on the premise of a portable file system infrastructure, the goals of the *PPFS* project are threefold.

- First, we are developing a flexible application programming interface (API) for specifying access pattern hints and for controlling file system behavior.
- Second, we are using *PPFS* to explore a variety of data distribution, distributed caching and prefetching strategies via both direct execution of input/output-intensive applications and via trace-driven simulation.
- Third, we are exploring distributed techniques for dynamically classifying file access patterns and automatically adapting data management algorithms to minimize access latencies for those patterns.

3 *PPFS* Design Principles

The portable parallel file system (*PPFS*) is a tool for exploring the design space of parallel input/output systems. To support experimentation and optimization of input/output performance, *PPFS* has an open application interface, and it is portable across a number of parallel platforms. An open application interface supports the advertising of access pattern information, control of data management policies, and extension of the parallel file system. Portability enables experimentation on a range of platforms.

To maximize experimental flexibility, *PPFS* is implemented as a user-level input/output library designed to be portable across several parallel systems. Building portable user-level libraries involves a number of challenges; when building atop an extant system, one can dependably assume only that the system software provides basic Unix file service. The basic challenge is building a coordination framework amongst multiple input/output servers that allows them to synchronize and exchange data. *PPFS* implements *parallel* file operations, providing global naming and location for the components of a parallel file and a rich application interface. *PPFS*'s API includes library calls for control of file striping, distribution, and caching, enabling the application to export policy and access pattern information to the input/output system.

3.1 Malleable Access

In any parallel file system, caching, prefetching, data placement, and data sharing policies constrain the range of file access patterns where high performance is possible. In traditional file systems, these policies are frozen during implementation. Given our current, limited understanding of the range of possible parallel access patterns, we believe the file system must provide application interfaces for specifying these and other policies. A portable, extensible file system must provide interfaces for extending, changing, and controlling these policies.

As described in §4 and documented in [11], *PPFS* incorporates a rich set of input/output primitives that allow a parallel application to specify the data distribution across disks, prefetching and caching algorithms, and the anticipated file access characteristics (e.g., sequential, random, or block-sequential). This infrastructure is far more flexible than current vendor parallel systems; both Thinking Machines' SFS [17] and Intel's CFS/PFS [9, 12] support only a small number of access modes and an even smaller number of automatic file data distributions among the disks. This means that *PPFS* can be used to explore a far broader range of input/output system designs than is possible with incremental modification of existing parallel file systems. For example, with *PPFS* one can distribute a file across disks by groups of N blocks, specify that every k^{th} block will be accessed, and request that blocks be prefetched into distributed client caches based on access pattern.

The design of an experimental tool for exploring parallel input/output systems must include support for controlling the following key policies.

- **Caching:** global, client side, server side, with particular control over aggregation (granularity) and write-back policies.
- **Prefetching:** client level, server level, and aggregate prefetching to hide access latency and share work. Prefetching must be coordinated with caching.
- **Data Distribution:** controlling data layout across input/output devices.
- **File Sharing:** coordinated control over metadata and file data consistency.

3.2 Portability

Software portability is always predicated on some design assumptions, and no software is painlessly portable to all systems. However, to increase the range of experiments possible with *PPFS* we must maximize its portability, enabling experiments on as many systems as possible. Hence, the only system requirements to support a port of *PPFS* are:

- an underlying Unix file system for each input/output node,
- a typed message passing library (currently NXLIB [26] and MPI [19]), and
- a C++ language implementation.

We view these porting requirements as minimal. Unix, the sequential file system base, is a *de facto* standard. The NXLIB message passing library provides Intel Paragon XP/S message passing semantics on a workstation network and compatibility with the native XP/S software environment; however, it can be and has been easily replaced with other message passing substrates (e.g., MPI). Finally, C++ has allowed us to isolate implementation details in objects and to inherit implementation features when porting the software to new environments. By minimizing system dependencies and isolating system dependencies behind object firewalls, *PPFS* is portable to any system that meets our three software assumptions, including both parallel systems and workstation clusters [1].

4 *PPFS* Software Design

Our *PPFS* design is based on a client/server model. Clients are application processes that include both the user program and a *PPFS* client. Clients issue requests to one or more *PPFS* input/output servers that manage logical input/output devices. Together the *PPFS* client and *PPFS* servers provide parallel file service to the application program. Below we describe the components of *PPFS*, their interactions with other components, and illustrate their use with an example.

4.1 *PPFS* Components

[1] The current implementation of PPFS is available from http://www-pablo.cs.uiuc.edu/Projects/PPFS/.

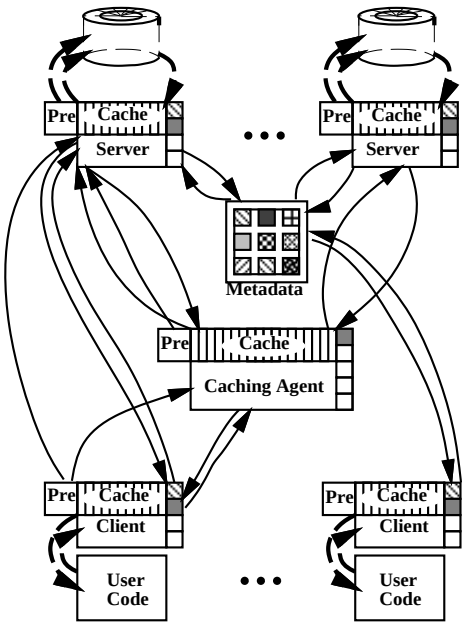


Figure 1: Portable Parallel File System Design

As Figure 1 shows, the key software elements in *PPFS* are the clients, servers, the metadata server, and caching agents. These components are briefly described below, coupled with a discussion of their interactions. Dashed arrows in Figure 1 represent local library or system calls, and solid arrows (e.g., those between clients and servers) correspond to message passing interactions. A more detailed description of the file system can be found in [7, 6, 8].

Clients In an SPMD computation, a client consists of an instance of the user application code and the local caching, prefetching, and bookkeeping software that permits an application to use *PPFS*. The client code communicates with the metadata server to determine which data servers and caching agents will satisfy input/output requests on behalf of the application code for a given file. Application-specific file system configuration and policies originate from the application code, and can affect all elements in the *PPFS*. For example, the *PPFS* client may maintain a local cache to hold recently used and prefetched data. Such a client cache can be configured, controlled (with respect to replacement policy for example), or disabled (to ensure data coherence) by the application program.

Servers An input/output server is an abstraction of an input/output device in *PPFS*. Input/output servers are the ultimate resolvers of requests; all physical input/output flows through them. Each server consists of a cache of file data, a prefetch engine that fetches data from the underlying sequential file system based on one of a variety of prefetch algorithms, storage for file metadata associated with open files, server message passing support, and an underlying Unix file system for storing data.

The number of input/output servers is configurable at the time *PPFS* is initialized and may be larger or smaller than the number of physical input/output devices. Server caching and prefetch policies may also be affected by application calls and can be changed during application exe-

PERSISTENT

File Name:		"foo"
Clustering Table:		
Server	Segment	Size
2	2:0	40
5	5:1	40
2	2:3	40
1	1:2	40
0	0:3	40
File Size:		200
Record Size:		1024
Distribution:		
Index Scheme:		

TRANSIENT

File Handle:	18
File Size:	225
Access Pattern:	
Agent Info:	

Figure 2: Example *PPFS* Metadata

cution.

The data within a parallel file is divided into fixed or variable size records, which are stored within *segments*, as specified by the file's distribution; each segment is managed by a single *PPFS* server.

Metadata Metadata describes the parallel file organization (i.e., the mapping of logical records to physical records) and information about anticipated file access patterns. Intuitively, the metadata, shown in Figure 2, is analogous to Unix *inodes* (persistent data) and open file pointers (transient data). The metadata server is the metadata repository for *PPFS*.

The metadata servers in *PPFS* service file open and close requests from clients. For opens, the metadata server generates messages that notify all servers that contain file data that the file is now open. Metadata servers also coordinate file close operations, detecting that all clients of a file have closed and then collecting any updated metadata from the input/output servers. Metadata can also be forwarded directly between clients, avoiding a bottleneck at the metadata server.

Caching Agents Caching agents are shared caches via which multiple clients share files. Such shared caches support request aggregation across clients, "constructive interference," allowing one client to effectively prefetch data for another. Each caching agent caches data for a single file, and provides a coherent view to multiple clients (both data and metadata). All requests to the shared file pass through the caching agent rather than directly to the input/output servers, supporting a wide range of file data consistency policies.

4.2 *PPFS* File Control Interfaces

In addition to supporting sequential access patterns, *PPFS* provides a more general set of interfaces to guide both local (i.e., client) and global (i.e., servers and caching agents) data management and the placement of file data on storage devices. These features enable an application to control or customize policies to match application requirements, supporting *malleable access*. Applications can

- declare per client or aggregate access patterns for the file (random, sequential, strided, etc.),
- control data layout of files over servers (block, cyclic, etc.),
- control caching policies at servers, clients, and
- control prefetching policies at servers, clients.

Together, the file control interfaces of *PPFS* provide a wide range of information advertisement and policy control capabilities to the application. They provide a range of policies beyond what is available from commercial file systems, and *PPFS*'s portability enables experimentation with such policies on a number of platforms. These file control interfaces can be used to advertise application information (helping the file system to make good policy choices), to control policies (explicitly optimizing input/output performance), and to develop new policies (rapid evaluation).

4.3 Performance Instrumentation

We have exploited our existing Pablo performance instrumentation and analysis suite [24, 23] to instrument *PPFS*. In addition to capturing client input/output requests and message passing among *PPFS* clients and servers, client and server cache hit ratios, cache utilizations, request counts, and access latencies, we also record the times, durations, and sizes of all input/output requests that *PPFS* servers generate for the underlying file system. Because all performance data is written in the Pablo self-defining data format (SDDF) [2], extensive, off-line analysis and behavioral comparisons are possible via the Pablo performance analysis environment.

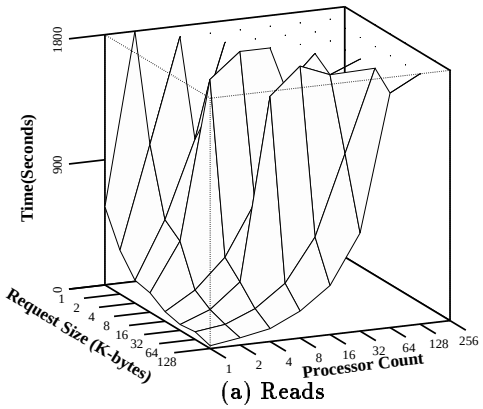
5 PFS and *PPFS* Benchmarks

To assess the performance of any parallel file system, one must have a baseline for performance comparison. We used the 512 node Paragon XP/S at Caltech to compare the performance of Intel's native parallel file system (PFS) to that of our *PPFS*, which runs atop the individual Unix file systems on each of the input/output nodes. Each input/output node in the Intel Paragon includes a RAID. Measurements in this section reflect the time to read and write a 64 MB file using a variety of request sizes and numbers of compute nodes. For the read benchmark, each processing node reads the entire file. For the write benchmark, each processing node writes an equal portion of the file, with record level interleaving (i.e., writer k wrote records $k, k+p, k+2p, ...$).

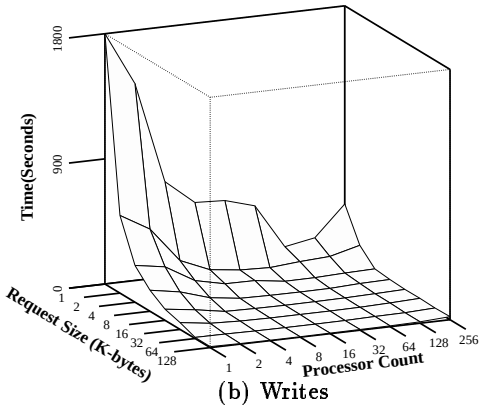
5.1 PFS

Figure 3 shows PFS performance as a function of both number of active readers or writers and the size of the requests.[2] For the read benchmark, PFS synchronous reads were used in conjunction with the `M_UNIX` PFS file mode. For the file write benchmark, we use the `M_RECORD` PFS file mode which supports writing of interleaved records. Because PFS does not cache or prefetch data, large numbers of small reads are inefficient — each read sees the full input/output latency and the input/output nodes each see large numbers of small requests. Similarly,

[2] Points at the top of each bounding volume are clipped; actual values are higher.

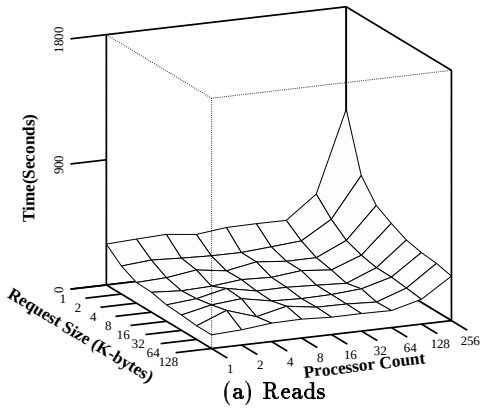


(a) Reads

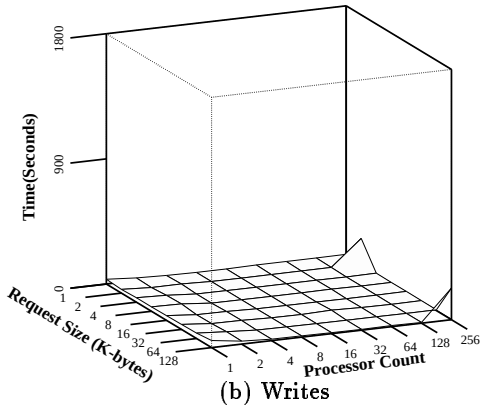


(b) Writes

Figure 3: Intel Paragon XP/S PFS Benchmark Performance (16 I/O nodes)



(a) Reads



(b) Writes

Figure 4: Intel Paragon XP/S *PPFS* Benchmark Performance (16 I/O nodes)

the lack of a cache means that concurrent readers do not benefit from the prefetch implicit in earlier requests from other readers for the same data.

Small writes suffer from many of the same problems as small reads — file system call overhead dominates total execution time. As either the write size or the number of writers increases, the number of writes per writer decreases. Larger writes better exploit the 64 KB striping factor of the underlying RAIDs, and greater writer parallelism increases the aggregate request rate and allows both aggregation of requests at the input/output nodes and more efficient disk arm scheduling. Simply put, PFS is optimized for modest numbers of large requests.

5.2 PPFS

To assess *PPFS* performance, we constructed benchmarks with behavior identical to the PFS benchmarks, but annotated the benchmark code with file distribution, caching, and prefetching directives.

The Paragon which we used for our *PPFS* measurements has sixteen dedicated input/output nodes. We used sixteen *PPFS* input/output servers. The Paragon was in multi-user mode during the tests.[3] Each input/output server had a 1 MB cache, and did no prefetching. Each client used a 512 KB cache. For writes, the client cache policy was configured as "to client," meaning that the client cache writes back to the server only when it fills; server caches behave similarly, writing to input/output nodes only when filled.

Figure 4 shows that in contrast to the sensitivity of PFS to the number of input/output requesters and request size, *PPFS* is resilient to variations in both. For file reads, the *PPFS* server combines multiple pending requests for the same record, sending only one request to the input/output node. Subsequent requests for the same record can be serviced from the cache. Although data returned from the server is stored in the client cache, for the read pattern, there is no reuse, so client caches give no benefit.

For writes, data is cached at each client and is written back to the server caches only when a client cache fills. Likewise, server caches are written to input/output nodes only when they fill. These extra levels of indirection add a fixed overhead, visible in the front left corner of Figure 4b, but it is more than compensated by the benefits of aggregating requests, particularly for small requests. At the client caches, this aggregation reduces the number of transfers between the clients and servers (messages), and at the server caches, the aggregation creates large, contiguous blocks of data that can be written efficiently by the input/output node RAIDs.

In general, the performance of *PPFS* is less brittle than that of PFS over a range of request sizes and compute par-

[3] In [11], we explore the performance of Paragon XP/S configurations with varying numbers of input/output nodes.

I/O Nodes	Number of Clients								
	1	2	4	8	16	32	64	128	256
PFS									
1	2364	1422	877	681	656				
2	2144	1334	791	611	554	541			
16	2071	1244	753	649	431	453	486		
PPFS									
1	1630	824	425	226	141				
2	1585	793	401	202	107	64			
16	1607	896	481	250	230	152	59	24	30

Table 1: Gene Sequence Execution Times (seconds)

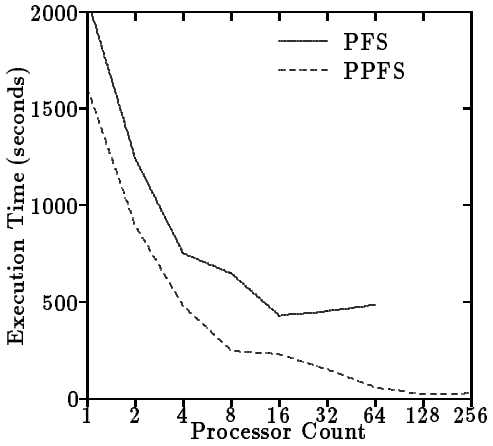


Figure 5: Relative Genome Timings

tition sizes. Intuitively, most of the benefit comes from the aggregation of requests in multiple levels of caching, reducing the file system call overhead, and also organizing the data for more efficient transfer to the secondary storage.

6 Application Codes

To assess the utility and performance of *PPFS*, we selected two input/output intensive parallel applications whose behavior and performance on the Intel Paragon XP/S had been previously analyzed. The input/output behavior of the first of these, a parallel gene sequence matching code [1, 25], is strongly data-dependent. The second, a low temperature plasma electron scattering code [27], has more regular spatial and temporal input/output patterns. Using these codes, we measured execution times, and captured detailed input/output performance information on both Intel's PFS and *PPFS*.

6.1 Gene Sequence Matching

6.1.1 Application Structure

Because the synthesis methods currently used to determine genetic sequences produce non-trivial numbers of errors, exact string matching algorithms are inappropriate for biological sequences.

This code uses an approximate sequence matching approach is based on a generalization of the Needleman, Wunsch, and Sellers (NWS) [20] dynamic programming algorithm, with a K-tuple heuristic to improve performance.[4] The input sequence is processed against all entries in the genome data base, and the data base entry that results in the best score is declared the best match for the input sequence.

In the native parallel implementation of the NWS algorithm, each Paragon XP/S node independently compares the test sequence against disjoint portions of the sequence data base. The results of each comparison are reported to a coordinating node that maintains a record of the best score.

6.1.2 Experiments

To port the gene sequence code from PFS to *PPFS*, we replaced the PFS input/output calls with *PPFS* routines that opened and read the sequence data base. Unlike the simple benchmarks of §5, where fixed size records are read, the gene sequence comparison code reads variable size records, each corresponding to a sequence. We used the *PPFS* variable size record access mode to retrieve sequences from a *PPFS* file.[5] We configured 1 MB server caches to always prefetch groups of 32 records (i.e., 32 gene sequences). Input/output servers also attempt to stay at least 16 sequences ahead of the client request stream.

Because the computing processors collectively compare a test sequence against the entire sequence data base, which sequences are processed by each processor has no effect on the result. On PFS, the application processors use a shared file pointer to access a list of sequence database offsets. The processors then seek to the specified location and read the needed sequences. Using the *PPFS read_any* access mode, which returns any record that has not yet been requested, exploits the order insensitivity of the algorithm, eliminating the shared file overhead.

Table 1 and Figure 5 show the execution times for the PFS and *PPFS* variations of the gene sequencing code. Empty fields in Table 1 correspond to machine size or compute time limitations. *PPFS* execution time improves dramatically with larger numbers of processors, even when there is only a single input/output node. For 256 processors, the speedup is over fifty with sixteen input/output nodes.

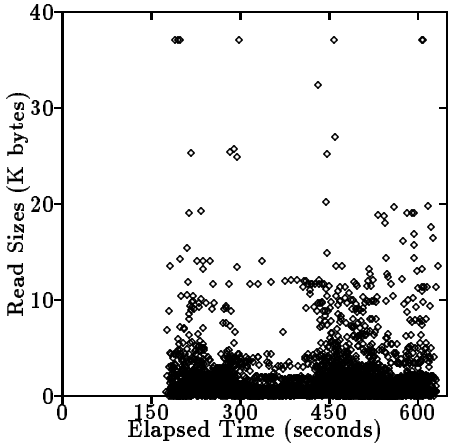
The reasons for the large disparity between PFS and *PPFS* are evident in Figures 6–7. These figures show the duration of the read and seek calls experienced by both file systems on a Paragon XP/S with sixteen input/output nodes (note scale differences). In both cases, the durations are for the native file system calls (i.e., Figure 7 does not show *PPFS* input/output operation times).

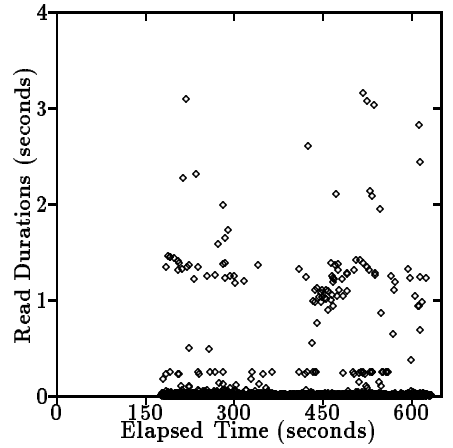
The data in Figures 6–7 was obtained by modifying both versions of the sequence code to trace input/output operations via the Pablo instrumentation library. Comparing the total execution time in the figures to the run times in Table 1 shows that input/output instrumentation adds roughly thirty percent to the total execution time of both code versions. However, the relative magnitudes of the input/output durations are preserved.

Figure 6 illustrates that contending, non-sequential file access is extraordinarily expensive. In contrast, Figure 7 shows that the *PPFS* client and server caches eliminate

[4] The original version of this code was developed for the Cray X-MP by Dan Davidson, then at Los Alamos National Laboratory.

[5] See [11] for details on the vread_any routine.





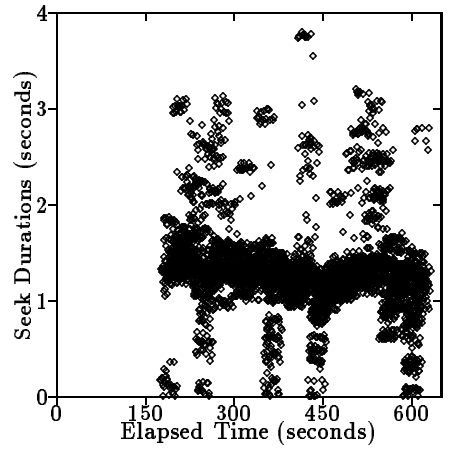
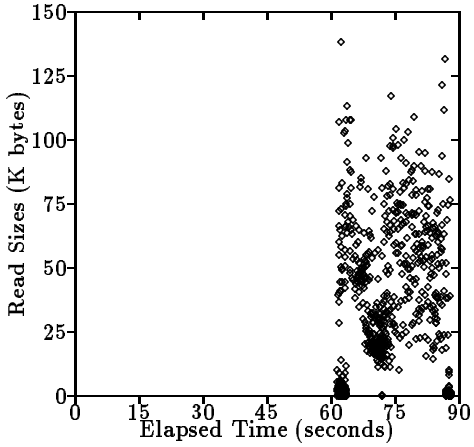
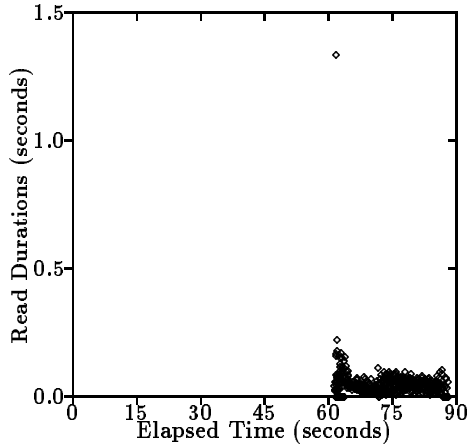


Figure 6: Gene Sequence Comparison (Intel PFS with 64 nodes)





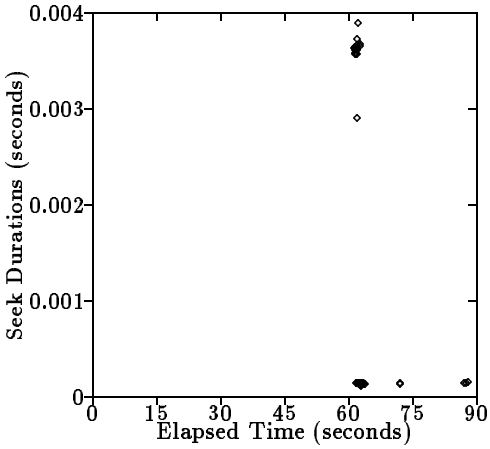


Figure 7: Gene Sequence Comparison (*PPFS* with 64 clients)

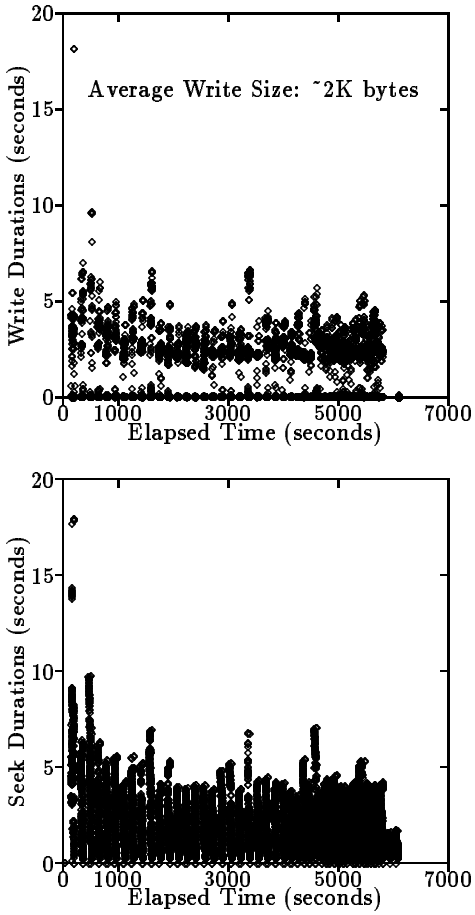


Figure 8: Electron Scattering Comparison (PFS with 128 nodes)

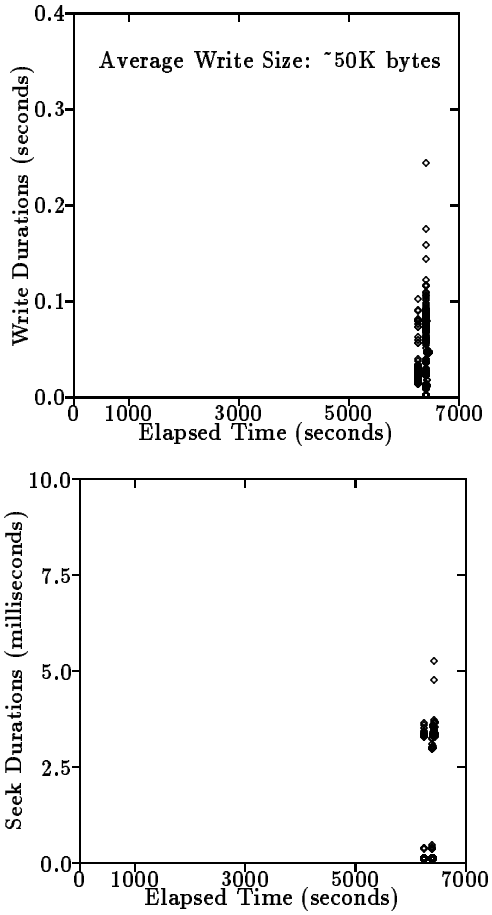


Figure 9: Electron Scattering Comparison (*PPFS* with 128 client nodes)

most contention at the native file system – 99 percent of the client reads are satisfied from the caches, and the server prefetch algorithm fetches large blocks of data from the input/output nodes.[6] PPFS aggregates read requests, reducing the number of reads by more than a factor of ten and increasing the average read size dramatically. As an indication that these larger requests are better exploiting the RAID devices, despite the dramatic increase in request size, the average read time increases by less than 20 percent. Thus, the net effect is that PPFS transforms the input/output request stream to better exploit the performance characteristics of the underlying file system; best performance for a few large requests.

The large performance improvement with *PPFS* buttresses our belief that a richer file system interface allows one to tune the file system's behavior to match application needs. For the gene sequencing code, using a shared PFS file to parcel sequence pointers, results in high file consistency overhead, with consequent performance loss. The *PPFS* approach with *read_any* mode embodies the real semantics of the read — which sequence is fetched is unimportant, only that it be previously unread.

[6] In Figure 7, the sixty second delay before *PPFS* input/output operations begin is due to *PPFS* server initialization.

6.2 Electron Scattering

6.2.1 Application Structure

Plasma processing plays an important role in semiconductor manufacturing; plasmas are used to perform a variety of tasks, including cleaning, mask stripping, passivation, doping, and etching. The electron scattering code computes plasma behavior from first principles [27], predicting the rates for surface reactions and the probabilities, or cross sections, for collision processes in the plasma.

The approach involves numerical solution of a linear system for each possible collision energy. However, the terms in this linear system require evaluation of a Green's function via numerical quadrature. The quadrature data is the major source of input/output; it is too voluminous to retain in memory, so it is written to disk, then re-read. The total volume of quadrature data grows as N^3, where N is the number of possible electron scattering outcomes. Current input/output limitations on the Paragon XP/S limit N to roughly 10, though the desired range is approximately 50 (i.e., over two orders of magnitude beyond current practice).

The version of the electron scattering code we studied begins with one processor reading initialization data from a small number of shared files and broadcasting this data

to all other nodes. This implementation was chosen to reduce contention in the Paragon's parallel file system (PFS). After initialization and during normal execution, all processors repeatedly write quadrature data to an output file for subsequent reuse.

6.2.2 Experiments

As with the gene sequencing code, we measured the execution time of both PFS and *PPFS* variants of the electron scattering code and captured input/output traces from both. Although the scattering code both reads and writes multiple data files, writes are the dominant operation; reads occur during initialization and near the end of the computation.

Figure 8 illustrates the duration of PFS file writes and seeks during execution on 128 nodes. All writes are a single size, roughly 2K bytes, and occur in bursts at the end of each code iteration. All nodes repeatedly compute, synchronize, and then write 2K bytes of quadrature data. To simplify reloading of the data in the next phase, each node seeks to a calculated offset dependent on the node number, iteration, and PFS stripe size before writing the data. Intel's M_UNIX file mode is used for these writes.

Figure 9 shows similar data from an execution of the scattering code under *PPFS*. Comparing Figures 8–9 shows the pattern of writes and seeks differs dramatically. *PPFS* generates far fewer requests to the underlying file system than does PFS. In Figure 9, the file writes that occur after each iteration are fully cached; this reduces the time for each iteration by between ten and fifteen seconds. Only near the end of execution is the file cache flushed to secondary storage, efficiently writing large blocks of data.

Comparing total program execution time under PFS and *PPFS* shows that the elapsed times are similar. In large part, this is because only about ten percent of the execution of the scattering code is attributable to input/output. Under *PPFS* the total input/output time is lower. However, because *PPFS* is a user-level library there are some compensating overheads.

In particular, *PPFS* incurs a small overhead for library initialization that is not present with PFS.[7] In addition, the scattering code relies heavily on the NX global synchronization routine (`gsync()`). Because *PPFS* executes in the same partition as the application, *PPFS* must emulate the NX functionality. Global operation emulation is unnecessary when using MPI because the client nodes can execute "global" operations and synchronizations using separate client communicator group.

7 Conclusions

We have argued that a portable, parallel file system (*PPFS*) provides the requisite infrastructure for exploring the critical issues in achieving high input/output performance, data distribution, caching, and prefetching policies and their performance tradeoffs on disparate architectures, without becoming mired in costly and time-consuming modifications to system software. Such an infrastructure requires a rich application interface with library calls for control of file striping, distribution, and caching, enabling the application to export policy and access pattern information to the input/output system.

[7] By analogy, PFS incurs this overhead when the operating system is booted.

Our experiments with large research codes have shown that investment in a malleable infrastructure is repaid with increased input/output performance. Tuning the file system policies to application needs, rather than forcing the application to use inappropriate and inefficient file access modes, is the key to performance. Simple access pattern hints and cache policy controls yield large performance increases with modest coding effort.

7.1 Related Research

PPFS is but one of several new parallel file systems based on user-level libraries. The major distinguishing feature of *PPFS* is its rich interface that supports advertising a variety of access information as well as control and augmentation of input/output system policies. PIOUS [18] is a portable input/output system designed for use with PVM. PIOUS enforces sequential consistency on file accesses; in contrast, *PPFS* allows data consistency to be controlled by the application, enabling higher performance as with the genome matching application. PASSION [4] supports out-of-core algorithms in a user-level library, but focuses on a high-level array oriented interface. IBM's Vesta parallel file system [5] allows applications to define logical partitions, data distributions, and some access information. However applications have little control over caching.

Related work also includes a number of commercial parallel file systems — the CM-5 Scalable Parallel File System [17, 16], the Intel Concurrent File System [9] for the iPSC/2 and iPSC/860, and the Intel Paragon's Parallel File System [12]. These provide data striping and a small number of parallel file access modes. In many cases, these access modes provide insufficient control for the application to extract good performance from the input/output system. Distributed file systems, such as Zebra [10] and Swift [3], stripe data over distributed input/output servers, but do not provide distribution or policy control to the application layer. Further, because the performance requirements in this environment are quite different (users are not as willing to tune for input/output performance), these systems provide little control to the application program.

Several groups have proposed schemes for exploiting access pattern information both in sequential and parallel systems [22, 13]. In [13], Kotz uses pattern predictors to anticipate an application's future access patterns. A variety of data management strategies for parallel input/output systems are explored in [15, 14].

Acknowledgments

Our thanks to Tara Madhyastha and Chris Kuszmaul for their early contributions to the design of *PPFS*. Some of the experimental data presented in this paper is derived from runs on the Intel Paragon at the CalTech Concurrent Supercomputing Facility.

References

[1] Arendt, J. W. Parallel Genome Sequence Comparison Using an iPSC/2 with a Concurrent File System. Master's thesis, University of Illinois at Urbana–Champaign, Department of Computer Science, Jan. 1991.

[2] AYDT, R. A. SDDF: The Pablo Self-Describing Data Format. Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Sept. 1993.

[3] CABRERA, L.-F., AND LONG, D. D. E. Exploiting Multiple I/O Streams to Provide High Data-Rates. In *Proceedings of the 1991 Summer Usenix Conference* (1991), pp. 31–48.

[4] CHOUDHARY, A., BORDAWEKAR, R., HARRY, M., KRISHNAIYER, R., PONNUSAMY, R., SINGH, T., AND THAKUR, R. PASSION: Parallel And Scalable Software for Input-Output. Tech. rep., Department of Electrical and Computer Engineering, Syracuse University, Nov. 1994.

[5] CORBETT, P., AND FEITELSON, D. Design and Implementation of the Vesta Parallel File System. In *Scalable High-Performance Computing Conference* (May 1994), pp. 63–70.

[6] ELFORD, C. L., KUSZMAUL, C., HUBER, J., AND MADHYASTHA, T. M. Design of a Portable Parallel File System. Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Nov. 1993.

[7] ELFORD, C. L., KUSZMAUL, C., HUBER, J., AND MADHYASTHA, T. M. Portable Parallel File System Detailed Design. Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Nov. 1993.

[8] ELFORD, C. L., KUSZMAUL, C., HUBER, J., AND MADHYASTHA, T. M. Scenarios for the Portable Parallel File System. Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Nov. 1993.

[9] FRENCH, J. C., PRATT, T. W., AND DAS, M. Performance Measurement of the Concurrent File System of the Intel iPSC/2 Hypercube. *Journal of Parallel and Distributed Computing 17*, 1–2 (January and February 1993), 115–121.

[10] HARTMAN, J. H., AND OUSTERHOUT, J. K. Zebra: A Striped Network File System. In *Proceedings of the Usenix File Systems Workshop* (May 1992), pp. 71–78.

[11] HUBER, JR., J. V., ELFORD, C. L., REED, D. A., CHIEN, A. A., AND BLUMENTHAL, D. S. PPFS: A High Performance Portable Parallel File System. Tech. Rep. UIUCDCS-R-95-1903, University of Illinois at Urbana Champaign, January 1995.

[12] INTEL SUPERCOMPUTER SYSTEMS DIVISION. *Paragon XP/S Product Overview.* Beaverton, Oregon, Nov. 1991.

[13] KOTZ, D. Disk-directed I/O for MIMD Multiprocessors. Tech. rep., Department of Computer Science, Dartmouth College, July 1994.

[14] KOTZ, D., AND ELLIS, C. S. Practical Prefetching Techniques for Parallel File Systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems* (December 1991), pp. 182–189.

[15] KOTZ, D., AND ELLIS, C. S. Caching and Writeback Policies in Parallel File Systems. *Journal of Parallel and Distributed Computing 17*, 1–2 (January and February 1993), 140–145.

[16] KWAN, T. T., AND REED, D. A. Performance of the CM-5 Scalable File System. In *Proceedings of the 1994 ACM International Conference on Supercomputing* (July 1994).

[17] LOVERSO, S. J., ISMAN, M., NANOPOULOS, A., NESHEIM, W., MILNE, E. D., AND WHEELER, R. *sfs*: A Parallel File System for the CM-5. In *Proceedings of the 1993 Summer Usenix Conference* (1993), pp. 291–305.

[18] MOYER, S. A., AND SUNDARAM, V. S. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *1994 Scalable High Performance Computing Conference* (May 1994), pp. 71–78.

[19] MPI. MPI: A Message Passing Interface Standard. Tech. rep., Message Passing Interface Forum, May 1994.

[20] NEEDLEMAN, S. B., AND WUNSCH, C. D. An Efficient Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins. *Journal of Molecular Biology 48* (1970), 444–453.

[21] PATTERSON, D., GIBSON, G., AND KATZ, R. A Case For Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD* (December 1988), pp. 109–116.

[22] PATTERSON, D., GIBSON, G., AND SATYANARAYANAN, M. A Status Report on Research in Transparent Informed Prefetching. Tech. Rep. CMU-CS-93-113, Department of Computer Science, Carnegie-Mellon University, Feb. 1993.

[23] REED, D. A. Performance Instrumentation Techniques for Parallel Systems. In *Models and Techniques for Performance Evaluation of Computer and Communications Systems*, L. Donatiello and R. Nelson, Eds. Springer-Verlag Lecture Notes in Computer Science, 1993.

[24] REED, D. A. Experimental Performance Analysis of Parallel Systems: Techniques and Open Problems. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (May 1994).

[25] SHIELDS, K., TAVERA, L., SCULLIN, W. H., ELFORD, C. L., AND REED, D. A. Virtual Reality for Parallel Computer Systems Analysis. In *ACM SIGGRAPH '94 Visual Proceedings* (July 1994), p. 261.

[26] STELLNER, G., BODE, A., LAMBERTS, S., AND LUDWIG, T. Developing Applications for Multicomputer Systems on Workstation Clusters. In *HPCN Europe, The International Conference and Exhibition on High-Performance Computing and Networking* (1994).

[27] WINSTEAD, C., AND MCKOY, V. Studies of Electron-Molecule Collisions on Massively Parallel Computers. In *Modern Electronic Structure Theory*, D. R. Yarkony, Ed., vol. 2. World Scientific, 1994.