Computer Architecture News, Sept. 1989

Beyond Striping: The Bridge Multiprocessor File System

Peter C. Dibble and Michael L. Scott
Computer Science Department
University of Rochester

August 14, 1989

Abstract

High-performance parallel computers require high-performance file systems. Exotic I/O hardware will be of little use if file system software runs on a single processor of a many-processor machine. We believe that cost-effective I/O for large multiprocessors can best be obtained by spreading both data and file system computation over a large number of processors and disks. To assess the effectiveness of this approach, we have implemented a prototype system called Bridge, and have studied its performance on several data intensive applications, among them external sorting. A detailed analysis of our sorting algorithm indicates that Bridge can profitably be used on configurations in excess of one hundred processors with disks. Empirical results on a 32-processor implementation agree with the analysis, providing us with a high degree of confidence in this prediction. Based on our experience, we argue that file systems such as Bridge will satisfy the I/O needs of a wide range of parallel architectures and applications.

1 Introduction

As processing power increases, high-performance computing systems require increasing I/O bandwidth. The problem is especially severe for machines with many processors. Internally-parallel I/O devices can provide a conventional file system with effectively unlimited data rates, but a bottleneck remains on a multiprocessor if the file system software itself is sequential or if interaction with the file system is confined to only one process of a parallel application. Ideally, one would write parallel programs so that each process accessed files on a private disk, local to its processor. Such files could be maintained under separate file systems, which could execute in parallel. Unfortunately, such an approach would force the programmer to assume complete responsibility for the physical partitioning of data among file systems, destroying the coherence of data logically belonging to a single file, and introducing the possibility of creating programs with incompatible ideas about how to organize the same data.

We believe that it is possible to combine convenience and performance by designing a file system that operates in parallel and that maintains the logical structure of files while physically distributing the data. Our approach is based on the notion of an *interleaved file*, in which consecutive logical records are assigned to different physical nodes. We have realized this approach in a prototype system called Bridge [4]. For applications in which I/O performance is critical, Bridge allows user-provided code to be incorporated into the file system at run time, so that arbitrary application-specific operations can be performed on the processors local to the data.

In order to demonstrate the feasibility of Bridge, we must show not only that it is possible to parallelize most I/O-intensive applications, but also that the parallelization can be accomplished in the context of a file system that manages data distribution automatically. We have therefore focussed our research on applications requiring non-trivial amounts of interprocessor communication and data movement. Examples include file compression, image transposition, and sorting. Sorting is a particularly significant example; it is

important to a large number of real-world users, and it re-organizes files thoroughly enough that very few operations can be confined to a single processor in Bridge.

We have used Bridge to implement a straightforward parallel external merge sort. We have analyzed the behavior of this sort at a level that allows us to predict its performance on a wide range of architectures. Under reasonable assumptions regarding the speed of processors, disks, and IPC, the analysis indicates that parallel I/O will dominate sequential portions of the code for configurations in excess of one hundred processors with disks. Experimental results on a 32-processor implementation of Bridge agree closely with this analysis, providing us with a high degree of confidence in its accuracy. These results suggest that a parallel interleaved file system can exceed the performance of a uniprocessor file system by some two orders of magnitude.

2 Parallel Interleaved Files

The typical disk drive is only about twice as fast now as it was ten years ago. Processor performance has improved by a factor of ten during the same interval. I/O system designers have therefore resorted to parallelism to increase I/O bandwidth. Several vendors sell parallel transfer disks that read or write more than one disk head at a time. Others sell storage arrays that manage a collection of disk drives under a single controller, spreading data across all drives so that it can be accessed in parallel. The RAID project at Berkeley is working to extend the concept of storage arrays to as many as one thousand disks, with internal redundancy to increase reliability [8]. Each of these systems seeks to eliminate a bottleneck at or near the disk drive.

A bottleneck at the disk controller can also be attacked with parallelism. Salem and Garcia-Molina have proposed *disk striping* [9] to interleave data across independent disks under the control of a single file system. The net effect of storage arrays and striping is that commercially-available I/O systems are able to provide data fast enough to satisfy almost any uniprocessor. The challenge for multiprocessors is to eliminate the principal remaining bottleneck on the disk-to-application path: the processor running the file system.

The *Data Vault* of the second-generation Connection Machine [10] addresses this challenge in a SIMD environment by interleaving data bits across multiple processors and disks. Each data word is read or written in parallel by the processors attached to the Vault. This technique is appropriate for a fine-grained SIMD computer, but does not extend well to machines with larger, individually-programmed processors. Several other vendors (e.g. of hypercubes) have begun to market systems in which disks are attached to multiple processors, but to the best of our knowledge their file systems are still limited by the decision to provide a more or less sequential interface to applications.

Our work focuses on the design and use of general-purpose parallel file system software, with particular emphasis on the use of what we call *tools* to dynamically add high-level operations to the file system. Like the designers of the Connection Machine, we scatter data across processors, but we do it at a level of granularity comparable to that of a storage array or striped file system. We also interleave *logical* records of potentially varying size, and we make the interleaving visible to applications that wish to write tools to exploit it.

An interleaved file can be regarded as a two dimensional array. Each of p disk drives (or multi-drive subsystems) is attached to a distinct processor, and is managed by a separate local file system (LFS). Files span all the disks, interleaved with a granularity of logical records. The main file system directory lists the names of the constituent LFS files for each interleaved file. Given that logical records are distributed in round-robin order, this information suffices to map an interleaved file name and record number to the corresponding local file name and record number. Formally, with p instances of the LFS, numbered $0 \ldots p-1$, record R of an interleaved file will be record (R div p) in the constituent file on LFS (R mod p). The part of the file located on node x consists of records $\{y | y \bmod p = x\}$. Round-robin interleaving guarantees that programs can access p consecutive records in parallel. For uniformly random access, it scatters records at least as well as any other distribution strategy.

We do not regard parallel interleaved files as an alternative to storage arrays or striping. They address a different bottleneck. In fact, we believe that the best performance for a multiprocessor file system is

likely to be obtained by a parallel interleaved file system in which each LFS uses striping or storage arrays internally. RAID-style storage arrays on each processor are a particularly attractive option because of their high reliability. Any file system that spreads data across multiple devices becomes increasingly vulnerable to hardware failures unless it incorporates redundancy. Such redundancy can be provided in software, but only by performing several disk operations for every write requested by the user. RAID devices implement redundancy in hardware, and can be made reliable enough that even a large collection of them has a many-year mean time to data loss [6].

2.1 The Bridge File System

Bridge is an implementation of a parallel interleaved file system on the BBN Butterfly Parallel Processor [1]. Bridge has two main functional layers. The lower layer consists of a Local File System (LFS) on each of the processors with disks. The upper layer is called the Bridge Server; it maintains the integrity of the file system as a whole and provides the initial interface to user applications. Except for a few functions that act on the state of the server itself, the Bridge Server interprets I/O requests and dispatches them to the appropriate LFSs.

Our LFS implementation is based on the Elementary File System developed for the Cronus distributed operating system [7]. Since our goal is simply to demonstrate the feasibility of Bridge, we have not purchased real disk drives. Instead of invoking a device driver, the lowest level of the LFS retrieves data from a (very large) "file" in RAM and returns it after an appropriate delay. We have programmed the delay to approximate the performance of a CDC Wren IV disk drive.

In order to meet the needs of different types of users, the Bridge Server implements three different system views. Two of the views hide significant amounts of the underlying parallel structure. These are designed for applications in which a familiar interface is more important that I/O performance. The third view of Bridge reveals the interleaved structure of files to I/O-critical applications. It is based on the concept of *tools*, discussed in the following section.

2.2 Bridge Tools

Parallel interleaved files effectively address any I/O bottlenecks at the LFS level of the file system or below. Between the file system and the application, interprocessor communication remains a potential bottleneck. It can be addressed by reducing communication to the minimum required for each file operation—by exporting as much functionality as possible out of the application, across the communication medium, and into the processors that run the LFS.

Bridge tools are applications that become part of the file system. A standard set of tools (copy, sort, grep, etc.) can be viewed as part of the top layer of the file system, but an application need not be a standard utility program to become a tool. Any process that can benefit from knowledge of the LFS structure may be written as a tool. Tools communicate with the Bridge Server to obtain structural information from the Bridge directory. Thereafter they have direct access to the LFS level of the file system. In essence, Bridge tools communicate with the Server as application programs, but they communicate with the local file systems as if they were the Server. Since tools are application-specific and may be written by users, they are likely to be structured in a variety of ways. We expect, however, that most tools will use their knowledge of the low-level structure of files to create processes on the nodes where the LFS instances are located, thereby minimizing interprocessor communication.

Our simplest tool copies files. The bulk of the algorithm runs in parallel on all the local file systems. A controlling process (1) opens the file to be copied; (2) creates an interleaved file to contain the copied data; (3) creates worker processes on each processor in the file system (this is a single operation from the controlling process's point of view); and (4) waits for the workers to complete. Each of the worker processes reads its local records and writes them back out to the local component of the output file. The startup time is $O(\log p)$ for p processors, and the run time is $O(N/p)$ for N records. Any one-to-one filter will display the same behavior; simple modifications to the Copy Tool would allow us to perform a large number of

```
In parallel perform local external sorts on each LFS
Consider the resulting files to be "interleaved" across only one processor
x := 2
while (x <= p)
    Merge pairs of files in parallel
    Consider the new files to be interleaved across x processors
    Discard the old files in parallel
    x := 2 * x
```

Figure 1: Merge Sort Pseudo-Code

other tasks, including character translation, block encryption, or lexical analysis within logical records. By returning a small amount of information at completion time, we could also perform sequential searches or produce summary information.

Other tools can be expected to require non-trivial communication or data movement between parallel components. In some, the communication pattern will be regular. We have implemented a bitwise image transpose tool, for example, that performs the same series of disk operations regardless of the data in the file. In other applications, communication will be much less predictable. We focus in the following section on the problem of sorting, first because it is an important operation in practice (files are frequently sorted) and second because it is in some sense an inherently hard problem for interleaved files—most records can be expected to belong to a different LFS after sorting, and substantial data-dependent communication will be needed in order to determine that location. Support for a fast sort is certainly not sufficient proof that our file system is practical, but it is impressive evidence.

3 An Example Tool, for Sorting

An astonishing number of parallel sorting algorithms have been proposed [3]. Most are ill-suited for external sorting because they access data randomly, read the data too often, or require a very large number of processors. Among the few algorithms specifically designed for external sorting, several require special-purpose hardware, assume a very small number of processors, or have significant phases during which only a fraction of the processors or disks are active. In a recent paper Beck, Bitton, and Wilkinson [2] detail their construction of a functional parallel external sort. They chose an algorithm similar to ours: local quicksort followed by parallel mergesort. Since they used comparatively few processors (five), they were able to pipeline the entire multi-phase merge with one disk read and one write. This gives them excellent performance, but poor speedup past three processors and no speedup of the merge stage in the worst case.

For our algorithm we have chosen a more or less conventional merge sort because it is easy to understand and has a straightforward parallelization. Our primary objective is to demonstrate that Bridge can provide significant speedups for common file operations. Rather than invest a great deal of effort in constructing the fastest possible sort, we deliberately chose to limit ourselves to an implementation with reasonable performance that could be derived from its sequential counterpart with only modest effort. We conjecture that most I/O-intensive applications can be implemented on Bridge with straightforward parallel versions of conventional algorithms; our merge sort constitutes a case in point.

3.1 The Algorithm

A parallel interleaved file can be viewed as a whole or, at the other extreme, as p sub-files, each local to a node. The first phase of our algorithm sorts the records on each LFS independently. The second phase merges the sorted records in neighboring pairs of LFSs. Assuming for the sake of simplicity that p is a power of two, the final phase merges the records from two collections of LFSs, each consisting of $p/2$ processors. Pseudo-code for the sort tool appears in figure 1.

```
Define token = { WriteAll, Key, Source, Number }

Setup for the merge
Read a record
If this process initiates the merge
     Build a token {false, key, MyName, 0}
          where key is the first key in the local file
     Send the token to the first reading process for the other sub-file

Loop
     Receive token
     If (token.Key ≥ record.key and not EOF) or token.WriteAll
          Increment token.Number
          Send token to next reading process for this source sub-file
          Send an output record to the writing process
               on LFS (token.Number −1) mod p
          Read a new record
     Else
          Build a token {EOF, file key, MyName, token.Number}
          Send the new token to token.Source
While not EOF

If (not token.WriteAll)
     Build a token {true, MAXKEY, MyName, token.Number}
     Send the token to old token.source
```

Figure 2: Reader Process Pseudo-Code

In each of the merge phases, the algorithm employs two sets of reading processes (one set for each of the source sub-files, one process per node) and one set of writing processes (again, one process per node). The algorithm passes a token among the reading processes of the two source sub-files. The token contains the least unwritten key from the other source sub-file and the location of the process ready to write the next record of the output sub-file. When a process receives the token it compares the key in the token to the least unwritten key among its source records. If the key in the token is greater than or equal to its local key, the process sends an output record to the appropriate writing process, and forwards the token to the next reading process in its sub-file. If the key in the token is less than the local key, the process builds a new token with its own key and address, and sends that token back to the originator of the token it received.

Special cases are required to deal with termination, but the algorithm generally follows the outline above. Figure 2 contains pseudo-code for an individual reader process. The token is never passed twice in a row without writing, and all records are written in nondecreasing order. The program therefore writes all of its input as sorted output and halts.

The parallelism of a merge phase is limited by sequential forwarding of the token. On at least every other hop, however, the process with the token initiates a disk read and write at the same time it forwards the token. For disk sorting on a machine like the Butterfly multiprocessor, the token can undergo approximately two hundred fifty hops in the time required for the parallel read and write. This implies that the sequential component will be entirely hidden by I/O latency on configurations of well over 100 processors. Performance scales almost linearly with p within that range.

3.2 Analysis and Empirical Results

We have constructed a mathematical model of our sorting algorithm on Bridge [5]. We find that the time T_{sort} to sort an N record file on p processors can be expressed as

$$T_{sort} = T_{local\_sort} + T_{parallel\_merge} \approx \frac{N}{p}\left[C_{local}\left(1 + \log\frac{N}{pB}\right) + \log p\,(T_{access} + T_{overhead})\right], \qquad (1)$$

where C_{local} is a constant reflecting the speed of the local (sequential) external merge-sort, B is the size of the in-core sort buffer in records, T_{access} is the time required to write an output record to a (probably

Table 1: Sort Tool Performance (10 Mbyte file)

	Merge Phases		Local Sort		Merge Sort Total		
Processors	Measured (Minutes)	Predicted (Minutes)	Measured (Minutes)	Predicted (Minutes)	Measured (Minutes)	Predicted (Minutes)	Rate (kbytes/sec)
2	7.8	7.68	19.6	19.58	27.4	27.26	6.26
4	7.6	7.68	7.6	7.83	15.2	15.51	11.00
8	5.7	5.76	2.7	2.94	8.4	8.70	19.62
16	3.8	3.84	1.0	0.98	4.8	4.82	35.41
32	2.4	2.40	0.3	0.24	2.7	2.65	64.52
64		1.44		0.12		1.56	109.21
128		0.84		0.06		0.90	189.28
256		0.51		0.03		0.54	318.06
512		0.33		0.02		0.35	491.29

non-local) disk and read a new input record from a local disk, and $T_{overhead}$ is the time required to create and destroy one record of a temporary file (including file system directory maintenance). This formula holds so long as T_{access} is greater than the expected time to pass a token all the away around a set of reading processes; i.e., so long as

$$p \leq p_{max} = \frac{T_{access}}{T_{act} + \frac{1}{2}T_{pass}}, \tag{2}$$

where T_{act} is the time required to pass the token on to the next reading process of the current input sub-file (when the current local record should be written to the output) and T_{pass} is the time to pass the token to a reading process of the other input sub-file (when the local record should not be written).

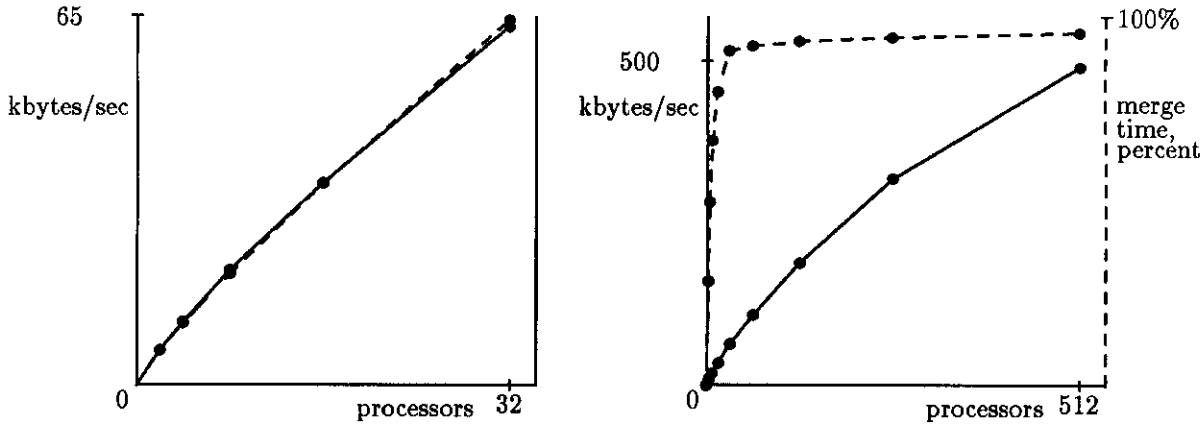
Given appropriate values for constants, our analysis of the sort tool can be used to predict execution time (T_{sort}) and level of available parallelism (p_{max}) for a wide range of processors, disks, and architectures. In our Butterfly implementation of Bridge, B is 500, C_{local} is 45.9 ms, and $T_{disk} + T_{overhead}$ is 90 ms. The limiting value p_{max} is 160. Above this number of processors the final phases of the parallel merge begin to be limited by token sequential passing but early phases continue to display speedup, so performance as a whole continues to improve, if more slowly.

In an attempt to evaluate the accuracy of our analysis, we have measured the sort tool's performance on configurations of up to 32 processor/disk nodes. Though our large Butterfly machine has 96 processors, memory space requirements for simulated disk drives made larger experiments impossible.

Actual and predicted performance figures are shown in table 1. The better-than-linear performance "improvements" with increasing p in the local sort are not remarkable; they reflect the fact that each individual processor has fewer records to sort. Our predicted performance figures differ only slightly from measured performance between 2 and 32 processors. A detailed examination of timing data suggests that the remaining disagreement stems from measurement error and from minor contention for the local file systems that is not accounted for in the analysis.

Figure 3 displays the data pictorially. The graph on the left compares predicted and actual performance figures up to 32 processors. The graph on the right plots predicted performance for larger numbers of nodes. The dotted line in the second graph indicates the percentage of total execution time devoted to parallel merge phases (as opposed to local sorts); it demonstrates that the local sorts consume an insignificant fraction of the execution time beyond 8 or 16 processors. Speedup (the solid line) begins to taper off noticeably beyond the center of the graph, where I/O ceases to dominate sequential token passing. Performance continues to improve, but at a slower rate. The merge that uses 256 processors to merge two files will run at its IPC speed. The earlier stages of the merge and the local sorts will, however, run with p-way parallelism. This should cause the algorithm to show some improvement with thousands of processors even though the last stages of the merge will reach a speedup of only about 160.

Figure 3: (a) Predicted (dashed) Versus Actual Performance;
(b) Predicted Performance for Larger Systems



4 Discussion

An application for a parallel interleaved file system is in some sense "trivially parallel" if the processing for each record of the file is independent of activities on other processors. From our point of view, the most interesting problems for Bridge are those in which data movement and inter-node cooperation are essential. The critical observation is that algorithms will continue to scale so long as all the disks are busy all the time (assuming they are doing useful work). In theory there is a limit to the parallelism that can be exploited in any algorithm with a sequential component, but the time scale difference between disk accesses and CPU operations is large enough that one can hope to run out of money to buy disks before reaching the point of diminishing returns. In the merge sort tool, the token is generally able to pass all the way around a ring of more than one hundred processes before a given process can finish writing out its previous record and reading in the next. It is clear that the tool can offer "only" a constant factor of speedup, but this observation misses the point entirely. Constant factors are all one *ever* looks for when replicating hardware.

This argument suggests that asymptotic performance may be the wrong way to think about I/O-intensive parallel applications. It also suggests that the exporting of user code onto the processors local to the disks, as supported in Bridge, is essential to obtaining good performance. It is precisely this exported code, embodied in a Bridge tool, that enables each disk to perform useful work as steadily as possible. Our experience with a variety of applications also suggests that the tool-based interface to Bridge is simple and convenient enough to be used for any application in which very high performance is important. There is no doubt that tools are harder to write than programs that use the more naive Bridge interfaces. At the same time, they are substantially easier than the alternative: explicit management of multiple files under multiple local file systems. They also allow separate applications to process the same data without worrying about compatible use of subfiles.

Round-robin interleaving for data distribution has proven to be both simple and effective. In the case of the merge sort tool, a process that produces a record to be written may need to write it to any disk in the system, depending on the distribution of runs. Though the records read from a particular source file or written to a particular destination file are always used in order, the records accessed by the application as a whole are more randomly distributed. Even so, a small amount of buffering allows each disk to remain busy almost all the time. We have observed this same phenomenon in other tools, such as those devised for file compression and image transposition. We have been unable to find a practical application for which our interleaving scheme does not suffice.

We see the token-passing technique as a promising way of coping with round-robin interleaving in a large number of file-manipulating programs. A tool that performs lexical analysis, data compression, or some

other size-altering translation could use a token to keep track of the location of the next output record in a fashion very similar to that of the sort tool; as in the sort tool we would expect the token to make an entire circuit of the worker processes faster than a record could be read, processed, and combined with other data to be written.

The successful analysis and implementation of a sorting tool on Bridge supports our thesis that a parallel interleaved file system can effectively address the I/O bottleneck on parallel machines. Intuitively, it seems fitting that parallel file system software should be used to solve a problem introduced by the parallel execution of application code. Practically, our experience with the merge sort tool (together with similar experience with other tools) has shown that parallel interleaved file systems will scale well to very large numbers of nodes.

References

[1] Butterfly$_{tm}$ parallel processor overview. Technical Report 6149, Version 2, BBN Laboratories, June 1986.

[2] Micah Beck, Dina Bitton, and W. Kevin Wilkinson. Sorting large files on a backend multiprocessor. *IEEE Transocations on Computers*, 37(7):769–778, July 1988.

[3] Dina Bitton, David J. DeWitt, David K. Hsaio, and Jaishankar Menon. A taxonomy of parallel sorting. *ACM Computing Surveys*, 16:287–318, September 1984.

[4] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 154–161, June 1988.

[5] Peter C. Dibble and Michael L. Scott. External sorting on a parallel interleaved file system. In *1989-90 Computer Science and Computer Engineering Research Review*. Department of Computer Science, University of Rochester, 1989.

[6] Garth A. Gibson, Lisa Hellerstein, Richard M. Karp, Randy H. Katz, and David A. Patterson. Failure correction techniques for large disk arrays. In *Third Internation Conference on Architectural Support for Pragramming Languages and Operating Systems*, pages 123–132, April 1989.

[7] R. F. Gurwitz, M. A. Dean, and R. E. Schantz. Programming support in the Cronus distributed operating system. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 486–493, May 1986.

[8] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference*, pages 109–116, June 1988.

[9] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986. Expanded version available as TR 332, EECS Department, Princeton University.

[10] Connection machine model CM-2 technical summary. Technical Report HA87-4, Thinking Machines Inc., April 1987.