

Bridge: A High-Performance File System for Parallel Processors *

Peter C. Dibble Michael L. Scott
 Department of Computer Science
 University of Rochester
 Rochester, NY 14627

Carla Schlatter Ellis
 Department of Computer Science
 Duke University
 Durham, NC 27706

Abstract

Faster storage devices cannot solve the I/O bottleneck problem for large multiprocessor systems if data passes through a file system on a single processor. Implementing the file system as a parallel program can significantly improve performance. Selectively revealing this parallel structure to utility programs can produce additional improvements, particularly on machines in which interprocessor communication is slow compared to aggregate I/O bandwidth.

We have designed and prototyped a parallel file system that distributes each file across multiple storage devices and processors. Naive programs are able to access files just as they would with a conventional file system, while more sophisticated programs may export pieces of their code to the processors managing the data, for optimum performance. Theoretical and early empirical data indicate nearly linear speedup on critical operations for up to several dozen devices.

1 Introduction

Parallel operation is a widely-applicable technique for maximizing computer performance. Within limits imposed by algorithms and interprocessor communication, the computing speed of a multiprocessor system is directly proportional to the number of processing nodes. As processing speed increases, however, programs that perform I/O become increasingly I/O bound. For all but the most compute-intensive applications, overall system throughput cannot increase without corresponding improvements in the speed of the I/O subsystem. From the point of view of parallel processing, any performance limit on the path between secondary storage and application program must be considered an I/O bottleneck.

The current state of the art in parallel storage device hardware can deliver effectively unlimited data rates to the file system. A bottleneck remains, however, if the file system itself uses sequential software 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 individual process accessed only local files. Such files could be maintained under separate file systems, on separate processors, with separate storage devices. Such an approach, unfortunately, would force the programmer to assume complete responsibility for the physical partitioning of data among file systems, destroying the coherence of data logi-

cally belonging to a single file. In addition, frequent restructuring might be required if the same data was to be used in several applications, each of which had its own idea about how to organize explicit partitions.

The thesis of this paper is that it is possible to have one's cake and eat it too, by designing a file system as a parallel program 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 blocks are assigned to different physical nodes. Following a bit of background information, we introduce the notion of interleaving in section 3 and (in section 4) describe its realization in an experimental file system known as Bridge. Our prototype runs on a BBN Butterfly parallel processor [1], but the ideas on which it is based are equally applicable to a large number of other parallel architectures and to locally-distributed collections of conventional machines.

For the most critical file operations, early analytical and experimental results indicate that Bridge will deliver good parallel speedup for configurations in excess of 32 nodes with disks. High performance is achieved by *exporting* the I/O-related portions of an application into the processors closest to the data. Section 5 presents a pair of applications that illustrate this technique. Section 6 discusses general issues in the design of algorithms for Bridge. The final section describes the status of our work and discusses future plans.

2 Background

I/O bottlenecks have been a source of concern for computer system designers since at least the early 1950's [2]. Since that time, parallelism has been built into most of the individual components of the I/O data path. Busses, for example, have transferred bits in parallel for many years. Similar "trivial parallelism" in hardware can increase the performance of controllers and connecting cables almost indefinitely.

Parallelism in storage devices can be achieved in several ways. Traditional memory interleaving can be employed in solid-state "RAM disks" to produce very high transfer rates. Mechanical disks that read several heads at once are available from such vendors as CDC and Fujitsu [3, 4]. Other manufacturers have introduced, or are at least investigating, so-called *storage arrays* that assemble multiple drives into a single logical device with enormous throughput [5, 6]. Unlike multiple-head drives, storage arrays can be scaled to arbitrary levels of parallelism, though they have the unfortunate tendency to maximize rotational latency: each operation must wait for the most poorly positioned disk.

As an alternative to storage arrays, Salem and Garcia-Molina have studied the notion of *disk striping* [7], in which conventional

This research was supported by NSF CER grant DCR8320136, DARPA/ETL contract number DACA76-85-C-0001, and an IBM Faculty Development Award.

devices are joined logically at the level of the file system software. Consecutive blocks are located on different disk drives, so the file system can initiate I/O operations on several blocks in parallel. Striped files are not limited by disk or channel speed, but like storage arrays and parallel devices they are limited by the throughput of the file system software.

Our work may be seen as an attempt to eliminate the remaining serialization points on the path between storage devices and a parallel application. To this end, we address the possibility that the communication channels between the components of the application program and the components of the file system may also constitute a bottleneck, even if no individual process is overloaded. This is certainly the case for locally distributed processors on a broadcast network like the Ethernet. Ideally, each application process would be located on the same node as the file system processes that perform its I/O. This would allow the application to run without transferring data between nodes. Though there is no way to ensure this locality in general, we can encourage local access by allowing the file system to execute complex operations provided by the user.

The Bridge File System distributes the blocks of each file across p disks using an interleaving scheme, and gives application programs access to the blocks both sequentially and in parallel. In addition, Bridge allows any program to become part of the file system by requesting a package of information from the main directory server. There is sufficient information in the package to allow the new program to find the processors attached to the disks, and to arrange for subprocesses to read and write blocks locally.

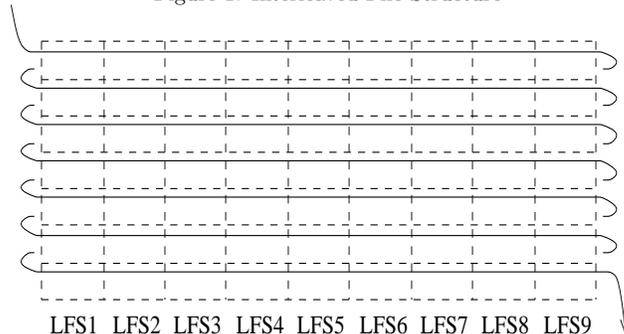
In its emphasis on distribution of data, parallel file system code, and the local execution of application-specific operations, Bridge shares a number of ideas with the Gamma project at the University of Wisconsin [8]. The fundamental difference is that Bridge is a general-purpose file system while Gamma is a relational database system. Our interest in traditional file operations requires that we present a more general-purpose interface. In addition, as described in the following section, our anticipated access patterns suggest a different file distribution strategy, with different algorithms for common operations.

3 Interleaved Files

An interleaved file can be regarded as a two dimensional array of blocks in row major order (see figure 1). Each column of the array is placed on a separate storage device attached to a separate processor, and managed by a separate local file system (LFS). The main file system directory lists the names of the constituent LFS files for each interleaved file. Given that blocks are distributed in round-robin order, this information suffices to map an interleaved file name and block number to the corresponding local file name and block number. Formally, with p instances of the LFS, the n th block of an interleaved file will be block $(n \text{ div } p)$ in the constituent file on LFS $(n \text{ mod } p)$ (assuming that processor and block numbering starts at zero). If the round-robin distribution can start on any node, then the n th block will be found on processor $((n + k) \text{ mod } p)$, where block zero belongs to LFS k .

Of course, round-robin distribution is not the only possible strategy for allocating blocks to nodes. Gamma [8], for example, allows a file to be divided into exactly p equal-size *chunks* of contiguous blocks. Each chunk is allocated to a processor in its entirety. Gamma also allows the blocks of a file to be scattered

Figure 1: Interleaved File Structure



randomly among nodes according to a hash function. The argument of the function can be any appropriate database key; in our world it could just as well be the number of the block. Both of these approaches appear to be more attractive in the context of a relational database than they are for general files.

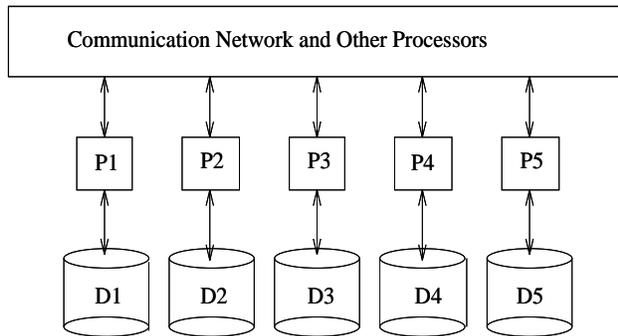
The principal disadvantage of chunking is that it requires *a priori* information on the ultimate size of a file being written. Significant changes in size (when appending, for example) require a global reorganization involving every LFS. The principal *advantage* of hashing is that it tends to randomize the locations of blocks required by any particular query. This advantage is much less compelling for ordinary files (where sequential access is the usual case) than it is for database files. The usual hashing algorithms cannot guarantee that p consecutive blocks can be accessed in parallel. If placement of blocks on processors were controlled by a hash function, the probability that two consecutive blocks would be on different processors would be high, but with p processors in the file system the probability that p consecutive blocks would be on p different processors would be extremely low. Round-robin interleaving guarantees that consecutive blocks will all be on different nodes. For parallel execution of sequential file operations this guarantee is optimal.

At the other extreme from chunking, one can imagine distributing data at a level of granularity smaller than the block. Thinking Machines Corporation is pursuing bit-level distribution for its second-generation Connection Machine [9]. The number of processors with disks is slightly larger than half the number of bits in a block. Each block¹ has two bits on each drive. The extra drives contain error-correction bits, so that loss of any one drive will leave the data intact. Such a strategy may well make sense for a massively-parallel SIMD machine with one-bit ALUs. For a more conventional multiprocessor it certainly does not. Our system of block distribution provides a relatively fine granularity of distribution without always forcing the programmer to reassemble logical records from multiple sources.

The most appropriate distribution strategy for parallel files will ultimately depend on the role that files assume in parallel applications. Unfortunately, the information that is currently available about file usage patterns [10, 11, 12] in uniprocessor systems does not necessarily apply to the multiprocessor environment. Preliminary experience allows us to make some educated guesses about what to expect. The principal role of sequential file systems,

¹Blocks on the Connection Machine are called "chunks" and contain 64 bits of data and 14 bits of ECC. Each chunk is subdivided into two 32-bit words, with each word spanning 39 disk drives.

Figure 2: Hardware Layout



namely long-term storage of large amounts of data, is likely to be even more predominant. Given the generally large size of parallel applications and the tendency to confine program development activity to a front-end machine, files should generally be larger than on an interactive system. Large amounts of main memory should also reduce the need for temporary files on disk, either by means of packages such as the Butterfly RAMFile system [13] or as a result of extensive caching.

These considerations lead us to believe that sequential access to relatively large files will overwhelm all other usage patterns. It therefore appears that round-robin interleaving will support common operations well. Given the sorts of performance results reported in section 5, we are hopeful that this distribution strategy will be all we ever need. If the potential gain in efficiency of some alternative ordering is large enough, files can always be sorted by the user.

Our only serious concern with strict interleaving is that it may result in unreasonable amounts of inter-processor data movement for certain applications. Deleting a block from the middle of a file, for example, would require all subsequent blocks to be moved. Traditional file systems do not usually support deletion in the middle, but that is not necessarily an excuse for precluding it in interleaved files as well. We are considering the relaxation of interleaving rules for a limited class of files, possibly with off-line reorganization. Our prototype implementation supports an explicit linked-list representation of files that permits arbitrary scattering of blocks at the expense of very slow random access. We will assume in the following sections that every file is strictly interleaved. A full exploration of “disordered files” is beyond the scope of this paper.

4 Bridge System Structure

The Bridge file system has three main functional layers. The top layer consists of the Bridge Server and a group of special purpose programs we call *tools*. The middle layer consists of local file systems on individual nodes. The lowest layer manages physical storage devices.

Our implementation runs under the Chrysalis operating system [1] on the BBN Butterfly Parallel Processor [14]. The components of the file system are implemented as user processes; no changes to the operating system were required. All components communicate via message passing. Messages are implemented on the Butterfly with atomic queues and buffers in shared memory, but could be realized equally well on any local area network.

Table 1: Bridge Server Commands

Command	Arguments	Returns
Create File		File id
Delete File	File id	
Open	File id	LFS file ids
Sequential Read	File id	Data
Random Read	File id, Block number	Data
Sequential Write	File id, Data	
Random Write	File id, Block number, Data	
Parallel Open	File id, Worker list	
Get Info		LFS handles

4.1 Bridge Server

The Bridge Server is the interface between the Bridge file system and user programs. Its function is to glue the local file systems together into a single logical structure. In our implementation the Bridge Server is a single centralized process, though this need not be the case. If requests to the server are frequent enough to cause a bottleneck, the same functionality could be provided by a distributed collection of processes. Our work so far has focused mainly upon the tool-based use of Bridge, in which case access to the central server occurs only when files are opened.

In order to meet the needs of different types of users, the Bridge Server implements three different system views. The simplest view supports operations reminiscent of an ordinary sequential file system: open,² read, and write (see table 1). Users who want to access data without bothering with the interleaved structure of files can use this simple interface. The Bridge Server transparently forwards requests to the appropriate LFS.

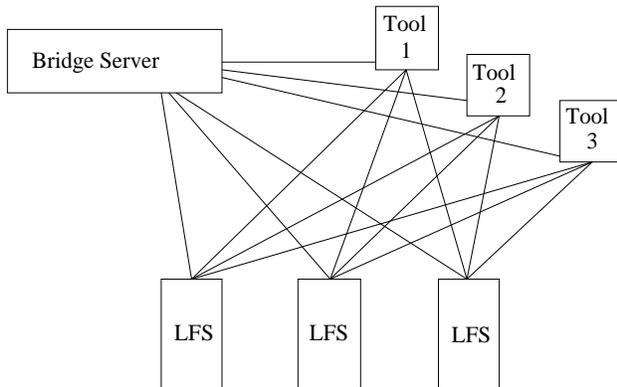
The second view of Bridge supports parallel reading and writing of multiple blocks of a file. A parallel open operation groups several processes into a “job.” The process that issues the parallel open becomes the job controller. It provides the Bridge Server with the names of all its workers. Suppose there are t such workers. When the job controller performs a read operation, t blocks will be transferred (one to each worker) with as much parallelism as possible. When the job controller performs a write operation, t blocks will be received from the workers in parallel.

Although the performance of parallel operations is limited by the number of nodes in the file system (p), the Bridge Server will simulate any degree of parallelism. If the width of a parallel open is greater than p , the server will perform groups of p disk accesses in parallel until the high-level request is satisfied. Application programs may thus be ignorant of the actual amount of interleaving in a file while still taking advantage of its parallelism.

The hidden serialization that can be introduced by specifying too many workers in a parallel open cannot cause incorrect results, but it may lead to unexpected performance. More importantly, it forces the workers to proceed in lock step, and does not increase the speed of random access. In order to take full advantage of the interleaved structure of files, sophisticated users can construct so-called *tools* that use the Get Info operation to view the third of the Bridge Server interfaces.

²The open operation is interpreted by Bridge as a hint. The Bridge Server responds by setting up an optimized path to the file. There is no close operation. This semi-stateless interface is not particularly significant, or intrinsic to interleaved files; it was adopted for the sake of consistency with our local file systems, which are also stateless (see below).

Figure 3: Software Connectivity



4.2 Bridge Tools

Bridge tools are applications that become part of the file system. A standard set of tools (copy, sort, grep, ...) 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 with knowledge of the middle-layer structure is 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 (see figure 3). We insist that all accesses to the Bridge directory (Create, Delete, and Open) be performed only by the Bridge Server in order to ensure consistency by providing what amounts to a monitor around all file management operations. 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 wide 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. Typical interaction between tools and the other components of the system involves (1) a brief phase of communication with the Bridge Server to create and open files, and to learn the names of the LFS processes, (2) the creation of subprocesses on all the LFS nodes, and (3) a lengthy series of interactions between the subprocesses and the instances of LFS. The Open operation provides the tool with LFS local names for all the pieces of a file, allowing it to translate between global and local block names.

Many operations on files (copying is the obvious example) require communication between nodes only for startup and completion. Where a sequential file system requires time $O(n)$ to copy an n -block file, the Bridge tool described in section 5.1 can accomplish the same thing in time $O(n/p)$, plus $O(\log(p))$ for startup and completion. Any one-to-one filter will display the same behavior; simple modifications to the copy tool allow us to perform character translation, encryption, or lexical analysis on fixed-length lines. By returning a small amount of information at completion time, we can also perform sequential searches or produce summary information. The implementation and performance of a sorting tool are discussed in section 5.2.

4.3 Local File System

The local file servers for Bridge are an adaptation of the Elementary File System (EFS) constructed for the Cronus distributed system project at BBN [15, 16]. EFS is a simple, stateless file system with a flat name space and no access control. File names are numbers that are used to hash into a directory. Files are represented as doubly linked circular lists of blocks. A pointer to the first block of a file can be found in the file's EFS directory entry. In addition to its neighbor pointers, each block also contains its file number and block number.

Every request to EFS can provide a disk address hint. In order to find a given block, EFS searches through the linked list from the closest of three locations: the beginning, the end, and the hint (assuming, of course, that the hint points into the correct file). A cache of recently-accessed blocks makes sequential access more efficient by keeping neighboring blocks (and their pointers) in memory.

The decision to use EFS was made for both pragmatic and technical reasons. On the pragmatic side, the simplicity, availability, and portability of the Cronus code made it easy to get our prototype up and running quickly. On the technical side, the level of functionality provided was almost exactly what we needed. Access control, for example, could be provided by the Bridge Server, and is unneeded in the LFS. Statelessness is also an advantage. Individual LFS instances must be informed when a Bridge file is created, but not when it is opened. The Bridge Server softens the potential performance penalty of statelessness by including an appropriate hint in each call to EFS.

EFS has been modified to fit into Bridge, but most of the modifications have involved adding instrumentation, improving performance, or adapting to the Butterfly and Chrysalis. An additional 40 bytes for Bridge-related header information have been taken from the data storage area of each block (leaving 960 bytes for data). The pointers in the original 24 byte EFS header lead to blocks that are interpreted as adjacent within the local context. In other words, the block pointed to by the *next* pointer is p blocks away in the Bridge file.

The revised version of EFS can still be used by a program that knows nothing about Bridge. In particular, the portion of a Bridge file controlled by one EFS server can be viewed locally as a complete file. The EFS server can ignore the fact that it holds every p th block of a more global abstraction. The instances of EFS are self-sufficient, and operate in ignorance of one another.

4.4 Physical Storage Management

The lowest layer of Bridge consists of device drivers that manage physical data storage. In principle, we see no reason why this layer should not be entirely conventional. For practical reasons, we have chosen in our implementation to simulate the disks in memory. Since our goal is simply to demonstrate feasibility there was no need to purchase real drives. Our large Butterfly has 120 Megabytes of RAM; we use part of that memory to simulate 64 Megabytes of "disk" on which to test our programs.

Our device driver code includes a variable-length sleep interval to simulate seek and rotational delay for different types of disk drives. For the performance figures in this paper, the delay has been set to 15 ms, to approximate the performance of a CDC Wren-class hard disk. Back-of-the-envelope calculations suggest that such disks are near the knee of the price/performance curve for currently-available hardware.

Table 2: Bridge Operations

Delete	$20 \cdot \text{filesize}/p$ ms
Create	$145 + 17.5p$ ms
Open	80 ms
Read	$9.0 + 500p/\text{filesize}$ ms
Write	31 ms

4.5 Basic Benchmarks

The figures in table 2 are taken from a simple program that uses the naive interface to the Bridge server in order to read and write files sequentially. The performance of Open and Write operations is essentially independent of p (the number of file system nodes). Read operations pay an amortized price for startup tasks that would be borne by the Open operation in a more traditional LFS, including initial reads of file header and directory information. Average read time for typical files is substantially less than disk latency because of full-track buffering in our version of EFS.³

The Delete operation runs in parallel on all instances of the LFS, but it takes time $O(n/p)$ where n is the size of the file being deleted. The original Cronus version of EFS included a substantial amount of code to increase resiliency to failures. One remnant of this code is a file deletion algorithm that traverses the file sequentially, explicitly freeing each block.

The Create operation must create an LFS file on each disk. Bridge gets some parallelism for this operation by starting all the LFS operations before waiting for them, but the initiation and termination are sequential, leading to an almost linear increase in overhead for additional processors. Performance could be improved somewhat by sending startup and completion messages through an embedded binary tree.

5 Example Tools

5.1 Copy Tool

Copying is in practice the most common file operation. It is also one of the simplest. An ordinary file system can copy a file of length n in time $O(n)$. If the copy program is written as a Bridge tool, files can be copied in time $O(n/p + \log(p))$ with p -way interleaving. The code looks something like this:

```

Send Get Info () to Bridge; Receive (LFS Names)
Send Open (source) to the Bridge Server; Receive (source_information)
Send Create () to the Bridge Server; Receive (destination)
Send Open (destination) to the Bridge Server;
  Receive (destination_information)
Start an instance of ecopy on each LFS node
Wait for all ecopies to complete

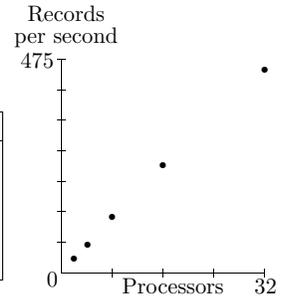
ecopy (LFS, local_src, local_dest)
  Send Read (local_src) to LFS; Receive (data)
  while not end of file
    Send Write (local_dest, data) to LFS
    Send Read (local_src) to LFS; Receive (data)
  endwhile
exit ecopy

```

³Write operations actually pay an amortized startup price as well, but its effect on average time is almost negligible, partly because writes take so much longer than reads, and partly because of EFS peculiarities that make caching of directory information less effective for writes than it is for reads.

Table 3: Copy Tool Performance (10 Mbyte file)

Processors	Copy Time
2	311.6 sec
4	156.0 sec
8	79.3 sec
16	41.0 sec
32	21.6 sec



The copy tool ignores the Bridge headers in the file it is copying. Since all the header pointers are block-number/LFS-instance pairs, the pointers are still valid in the new file. The while loop in ecopy could contain any transformation on the blocks of data that preserves their number and order. Any of the filter programs produced by inserting such transformations should run within a constant factor of the copy tool's time.

The copy tool displays nearly linear speedup as processors are added.

5.2 Sort Tool

The standard algorithm for external sorting is the merge sort. It makes no unusual demands on the file system and runs in $O(n \log(n))$ time without pathological special cases. Given a parallel merging algorithm, a log-depth parallel merge sort is easy to write. With p processors and n records, a parallel merge sort builds p sorted runs of length n/p in parallel. It merges the sorted runs in a $\log(p)$ -depth merge tree.

The basic structure of the algorithm is as follows:

```

In parallel perform local external sorts on each LFS.
Consider the resulting files to be "interleaved" across
  only one processor
x := p
while x > 1
  Merge pairs of files in parallel
  x := x/2
  Consider the new files to be interleaved across p/x processors
  Discard the old files in parallel
endwhile

```

Each pass of the merge sort has different parallel behavior, but they all use p processors. Pass k of the sort runs $p/2^k$ merges, each of which uses 2^k processors to merge $2kn/p$ records. On the first pass the sort will provide $p/2$ -way parallelism, and the merge will provide 2-way parallelism. On the last pass the sort will provide no parallelism, but the merge will provide p -way parallelism.

The best possible performance for a parallel sorting algorithm is $O(n \log(n)/p)$. For a conventional merge sort, the time is consumed by $\log(n)$ passes through merge at $O(n)$ total time per pass. We have developed a parallel merge algorithm that runs in time $O(n/p)$ for limited values of p . An optimal sorting algorithm (for limited p) can be built from this merge. For the sake of simplicity we assume that the records to be sorted are the same size as a disk block. Odd-sized records make the algorithm significantly messier, but do not affect its asymptotic complexity.

Figure 4: Merge Pseudo Code

```

type token = {StartFlag, EndFlag, Key, Originator, SeqNum}

Read a record
loop
  Receive token
  if token.StartFlag then
    Build a token {false, false, K, MyName, 0}
    where K is the first key in the local file
    and MyName is the name of this process
    Send token to first process of other input file
  elif token.EndFlag then
    if End of file then
      DONE
    else
      Increment token.SeqNum
      Send token to next process of current input file
      Send record to destination process for token.SeqNum - 1
      Read a new record
    end if
  else (usual case)
    if End of file then
      Build a token {false, true, 0, MyName, token.SeqNum}
      Send new token to token.Originator
    else if record.key ≤ token.key then
      Increment token.SeqNum
      Send token to next process of current input file
      Send record to destination process for token.SeqNum - 1
      Read a new record
    else
      Build a token {false, false, record.key,
                    MyName, token.SeqNum}
      Send new token to token.Originator
    endif
  end if
end loop

```

The algorithm to merge two $t/2$ -way interleaved files into one t -way interleaved file involves three sets of processes. The first set contains $t/2$ processes and is devoted to reading one of the input files. The second set also contains $t/2$ processes and is devoted to reading the other input file. The third set contains t processes to write the destination file.

The algorithm proceeds by passing a token among the processors controlling the two files to be merged. When received by a given process, the token contains the least unwritten key from the other input file, the name of the process that holds the record with that key, and the sequence number of the next destination record to be written, from which we can derive the name of the process ready to write that destination record. When a process receives the token it compares the key inside to the least unwritten key in its local file. If the key in the token is greater than or equal to its local key, the process forwards the token to the next process for its input file and sends an output record to the appropriate process for the destination file. If the key in the token is less than the local key, the process builds a new token with its own key and name, and sends this new token to the originator of the token it received.

Special cases are required to deal with termination, but the algorithm always follows the above outline. Except for the fact that records are scattered among a collection of communicating processes, the logical flow of control is directly analogous to the standard sequential merge. Correctness can be proven by observing that the token is never passed twice in a row without writing, and all records are written in nondecreasing order. The code for

a process handling the input-file part of a merge can be found in figure 4.

Sorting proceeds in two distinct phases. The first phase sorts the records of each LFS locally. The second phase performs a series of successively global merges. For reasonable values of p , the merge phase should take time $O(n \log(p)/p)$, for $\log(p)$ passes at n/p per pass. Speedup in practice is nearly linear, and improves as p increases. With sufficiently large p , the token will eventually be unable to complete a circuit of the nodes in the time it takes to read and write a record. At that point performance should begin to taper off, but only in the merge phase of the tool, and only for the final pass of the merge (since earlier passes involve shorter circuits). 32 nodes is clearly well below the point at which the merge phase of the sort tool would be unable to take advantage of additional parallelism.

The local sorting phase consumes the bulk of the sort tool's time for small values of p . It improves much more than linearly as processors are added. The expected time for local sorting is the sum of the times for in-core sorting and local merges, or $O((n/p)(1 + \log c) + (n/p) \log(n/cp))$, where c is the size of the in-core buffer (in our case 512 records). Doubling the number of processors not only doubles the number of disks that can be reading or writing at once (which would explain nearly linear speedup), but also moves one pass of merging out of the local sorting phase and into the global merging phase. In our implementation the constant for a local merge is higher than the constant for a global merge, with the net result that the sort tool as a whole displays super-linear speedup. With a faster (e.g. multi-way) local merge, this anomaly should disappear.

6 Algorithm Design Considerations

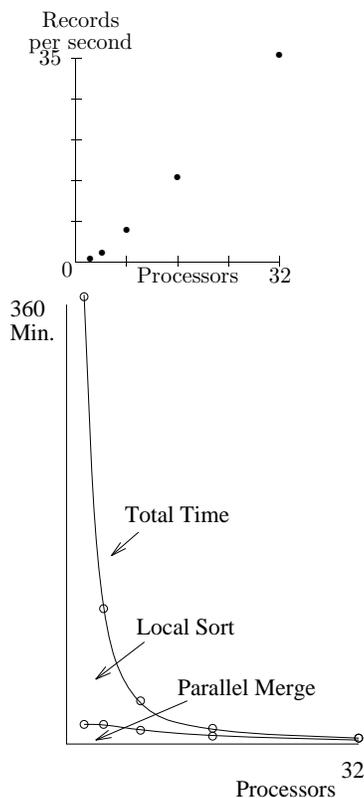
The primary design goal for the Bridge file system is high performance for common operations in a multiprocessor environment. A second goal is that the system perform well enough even on uncommon operations that it could be used as the *only* file system on a parallel machine. We are particularly anxious to ensure that Bridge never perform significantly worse than a conventional file system on any operation. This concern has played a major role in several of our design decisions, including the adoption of a stateless LFS and the rejection of data distribution via chunking. It has also led us to provide the three different user views described in section 4.1.

Programs can take advantage of the parallelism in Bridge to differing degrees, depending on the view they use. For naive programs that use the conventional interface, Bridge offers the same advantage as storage arrays or disk striping: its aggregate transfer rate from disk can be arbitrarily high. Assuming that the local file systems perform read-ahead and write-behind, virtually any program that uses the naive interface will be compute- or communication-bound.

For applications that read or write files sequentially (in predictable chunks), the parallel open can eliminate computational bottlenecks by transferring data to and from multiple processes. The parallel-open access method offers true parallelism up to the interleaving breadth of the Bridge file or the bandwidth of inter-processor communication, whichever is least. It also offers virtual parallelism to any reasonable degree, thus freeing the user from the need to know about implementation details. Unfortunately, the hiding of low-level details also means that users are unable to

Table 4: Merge Sort Tool Performance (10 Mbyte file)

Processors	Local Sort	Merge	Total
2	350 min	17 min	367 min
4	98 min	16 min	111 min
8	24 min	11 min	35 min
16	6 min	7 min	13 min
32	0.67 min	4.45 min	5.12 min



place their workers on appropriate LFS nodes. Communication is likely to remain a bottleneck in many situations.⁴

The Bridge tool-level interface addresses the limitation of communication bandwidth by allowing the user to move functionality across the bottleneck toward the data. Even if processing cannot be completed on the LFS node, the exportation of user-level code allows data to be filtered (and presumably compressed) before it must be moved. Since tools communicate directly with the LFS instances, they also gain a modest performance benefit by avoiding indirection through the Bridge Server. Finally, unrestricted access to the LFS constituents of a file frees the programmer from the constraints of lock-step multi-block access, allowing workers to proceed at their own rate of speed and permitting random access. The penalty, of course, is that access to the file system becomes vastly less abstract.

An application for a parallel interleaved file system is in some sense “trivially parallel” if the processing for each block of the file is independent of activities on other processors. From a research

⁴In order to prevent arbitrary programs from competing with the file system for cycles, we suggest that user processes never run on an LFS node unless they are part of a tool.

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 several dozen 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.

Our current impression is that the limiting factor for speedup in interleaved files (given infinite financial resources) is not Amdahl’s law but rather Murphy’s law: interleaved files (like striped files and storage arrays) are inherently intolerant of faults. A failure anywhere in the system is fatal; it ruins every file. Replication helps, but only at very high cost. Storage capacity must be doubled in order to tolerate single-drive failures. One might hope to reduce the amount of space required by using an error-correcting scheme like that of the Connection Machine, but we see no obvious way to do so in a MIMD environment with block-level interleaving.

7 Conclusion

In this paper we have discussed the design and initial implementation of a file system based on interleaved files and exported user code. Interleaving appears to be a natural approach to providing I/O parallelism, but previous experience with systems built upon this idea has been minimal. The prototype version of Bridge provides a concrete implementation that can be measured and used experimentally. It serves as a testbed for investigating design tradeoffs and for studying how interleaved files can be put to good use in parallel applications. The success of interleaving for unsophisticated users is likely to depend on the popularity of the parallel open feature. It is already clear that for critical tool-based applications there are significant benefits to making the interleaved structure explicit. Although the current implementation has not been tuned for high performance, the empirical measurements so far are encouraging.

Our early decision to build upon the Cronus elementary file system has worked out fairly well. The linked structure of files, inherited from EFS and expanded upon with global pointers, provides a degree of flexibility that is important in a testbed. The resulting system remains relatively simple, yet rich enough to allow us to explore alternative data-distribution strategies.

We expect over the course of the upcoming year to develop several additional tools. We have developed an unconventional mathematical analysis of the merge sort algorithm that expresses the maximum available degree of parallelism in terms of the relative performance of processors, communication channels, and physical devices [17]. The results we obtain for the constants on the Butterfly agree quite nicely with empirical data. We hope to produce a similar analysis for future tools as well.

References

- [1] “ButterflyTM parallel processor overview,” Tech. Rep. 6149, Version 2, BBN Laboratories, June 1986.
- [2] C. J. Bashe, W. Buchholz, B. V. Hawkins, J. J. Ingram, and N. Rochester, “The architecture of IBM’s early computers,” *IBM Journal of Research and Development*, vol. 25, pp. 363–375, September 1981.
- [3] M. Gamerl, “Maturing parallel transfer disk technology finds more applications,” *Hardcopy*, vol. 7, pp. 41–48, February 1987.
- [4] H. Boral and D. J. DeWitt, “Database machines: An idea whose time has passed: A critique of the future of database machines,” Tech. Rep. 288, Technion, August 1983.
- [5] J. Voelcker, “Winchester disks reach for a gigabyte,” *IEEE Spectrum*, vol. 24, pp. 64–67, February 1987.
- [6] T. Manuel and C. Barney, “The big drag on computer throughput,” *Electronics*, vol. 59, pp. 51–53, November 1986.
- [7] K. Salem and H. Garcia-Molina, “Disk striping,” Tech. Rep. 332, EECS Department, Princeton University, December 1984.
- [8] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna, “Gamma: A high performance dataflow database machine,” Tech. Rep. 635, Department of Computer Sciences, University of Wisconsin – Madison, March 1986.
- [9] “Connection machine model CM-2 technical summary,” Tech. Rep. HA87-4, Thinking Machines Inc., April 1987.
- [10] R. Floyd, “Short-term file reference patterns in a UNIX environment,” Tech. Rep. 177, Department of Computer Science, University of Rochester, March 1986.
- [11] J. Porcar, “File migration in distributed computer systems,” Tech. Rep. LBL-14763, Lawrence Berkeley Laboratory, July 1982.
- [12] J. Ousterhout, H. DaCosta, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, “A trace driven analysis of the UNIX 4.2 BSD file system,” *Proceedings of 10th Symposium on Operating Systems Principles, Operating Systems Review*, vol. 19, pp. 15–24, December 1985.
- [13] “The Butterfly RAMFile system,” Tech. Rep. 6351, BBN Advanced Computers Incorporated, September 1986.
- [14] BBN Advanced Computers Inc., *Chrysalis Programmers Manual*, April 1987.
- [15] R. F. Gurwitz, M. A. Dean, and R. E. Schantz, “Programming support in the Cronus distributed operating system,” in *Sixth International Conference on Distributed Computing Systems*, pp. 486–493, May 1986.
- [16] R. Schantz, “Elementary file system,” Tech. Rep. DOS-79, BBN, April 1984.
- [17] P. C. Dibble and M. L. Scott, “Analysis of a parallel disk-based merge sort,” tech. rep., Department of Computer Science, University of Rochester. In preparation.