External Sorting on a Parallel Interleaved File System

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

Abstract

Parallel computers with non-parallel file systems find themselves limited by the performance of the processor running the file system. We have designed and implemented a parallel file system called Bridge that eliminates this problem by spreading both data and file system computation over a large number of processors and disks. To assess the effectiveness of Bridge we have used it as the basis of a parallel external merge sort, an application requiring significant amounts of interprocessor communication and data movement. A detailed analysis of this application 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 closely 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

Parallelism is a widely-applicable technique for maximizing computer performance. Within limits imposed by algorithms and interprocessor communication, the computing speed of a multiple-processor system is directly proportional to the number of processing nodes, but for all but the most compute-intensive applications, overall system throughput cannot increase without corresponding improvements in the speed of the I/O subsystem.

Internally-parallel I/O devices can provide a conventional file system with effectively unlimited data rates [Manuel and Barney 1986], but a bottleneck remains 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 individual process accessed only local files. Such files could be maintained under separate file systems, on separate processors, with separate storage devices. 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. In addition, frequent restructuring might be required if the same data were to be used in several applications, each of which had its own idea about how to organize explicit partitions.

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 [Dibble et al. 1988].

To validate Bridge, we must demonstrate that it can provide most I/O-intensive applications with significant speedups on a significant number of processors. We have therefore implemented several data-intensive applications, including utilities to copy and sort sequential files and to transpose image bitmaps. Sorting is a particularly significant example; it is important to a large number of real-world users, and it reorganizes files thoroughly enough to require a large amount of interprocessor communication.

We have analyzed and implemented a parallel external merge sort on Bridge. Our analysis suggests that the parallel portions of the algorithm, especially disk I/O, will overlap the sequential portions under reasonable assumptions regarding the number and speed of processors and disks and the speed of interprocessor communication. Not until disks are attached to over a hundred different processors will the system become CPU or communication-bound. In an attempt to confirm this analysis, we have measured the merge sort on a 32-processor prototype of Bridge. The results are within three percent of the analytical predictions.

2. Parallel Interleaved Files

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. For example, the lines in a text file would be distributed such that consecutive lines would be located on logically adjacent disks. The main file system directory lists the names of the constituent LFS files for each interleaved file. 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 $\{y \mid y \bmod p = x\}$. Round-robin interleaving guarantees that programs can access p consecutive records in parallel. For random access, it scatters records at least as well as any other distribution strategy.

† This work was supported in part by DARPA/ETL Contract DACA76-85-C-0001, NSF/CER Grant DCR-8320136, Microware Systems Corporation, and an IBM Faculty Development Award.

Our approach to data distribution resembles that of several other researchers. At the file system level, *disk striping* can be used to interleave data across the disks comprising a sequential file system [Salem and Garcia-Molina 1986]. In the second-generation Connection Machine [Thinking Machines Inc. 1987], data is interleaved across processors and disks at the level of individual bits. At the physical device level, *storage arrays* encapsulate multiple disks inside a single logical device. Work is underway at Berkeley to construct such arrays on a very large scale [Patterson et al. 1988]. Our work is distinguished by its emphasis on the design and use of parallel file system software, particularly its use of *tools* to dynamically add high-level operations to the file system.

2.1 The Bridge File System

Bridge is an implementation of a parallel interleaved file system on the BBN Butterfly Parallel Processor [BBN Laboratories 1986]. 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.

An LFS sees only one column sliced out of each interleaved file, but this column can be viewed locally as a complete file. The LFS instances are self-sufficient, fully-competent file systems. They operate in ignorance of one another. LFSs can even maintain local files outside the Bridge file system without introducing any problems. Our LFS implementation is based on the Elementary File System developed for the Cronus distributed operating system [Gurwitz et al. 1986]. 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 maintains an image of the disk in RAM and executes an appropriate delay with each I/O request.

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 where a familiar interface is more important than 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 requires 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. All accesses to the Bridge directory (Create, Delete, and Open) are performed by the Bridge Server in order to ensure consistency. 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.

Our simplest tool copies files. It requires 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 Copy Tool 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 would allow us to perform a large number of other tasks, including character translation, block encryption, or lexical analysis. 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 between parallel components. We focus in this paper 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.

3. Sorting Parallel Interleaved Files

Several researchers have addressed the problem of parallel external sorting [Bitton et al. 1984; Kwan 1986]. In a recent paper, Beck, Bitton, and Wilkinson [1988] detail their construction of a functional parallel external sort. They chose an algorithm consisting of 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. We chose a straightforward parallelization of the most conventional merge sort algorithm.

A sequential external merge sort makes no unusual demands on the file system (no random access, indexing, etc.) and runs in $O(n \log n)$ time. Given a parallel merge algorithm, a log-depth parallel merge sort is easy to write. With p processors and N records a parallel merge sort concurrently builds p sorted runs of length N/p. It then merges the sorted runs in a $\log p$ depth merge tree. Pseudo-code for this algorithm appears in Figure 1. The first phase of the 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.

3.1 Merging Parallel Interleaved Files

An interleaved file can be viewed as a whole or, at the other extreme, as p sub-files, each local to a node. It may also be regarded as some intermediate number of sub-files, each of which spans a non-trivial subset of the file system nodes. The merge algorithm takes two sub-files, each spread across k nodes, and produces a single sub-file spread across $2k$ nodes. To do so it 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 writes an output record and forwards the token to the next processor 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 in

```
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

Figure 1. Figure 2 contains pseudo-code for an individual reader process.

The parallelism of the merge algorithm 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, we will show that the token can undergo approximately one hundred sixty 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 should scale almost linearly with p within that range.

3.2 Analysis

Merge

The parallel merge algorithm is a close analog of the standard sequential merge. 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.

For the purposes of this analysis, let p be the number of nodes across which the output sub-file is to be inter-

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

Setup for the merge                                          k1
Read a record                                                -
If this process initiates the merge                          -
    Read a record                                            -
    Build a token {false, key, MyName, 0}                    -
        where key is the first key in the local file         -
    Send the token to the first process for the other file   -

Loop
    Receive token                                            k2
    If (token.Key ≥ file key and not EOF) or
            token.WriteAll                                   -
        Increment token.Number                               k3
        Pass token to next process for this source file      -
        Send a write-request message to
            Write-Process[token.Number-1 mod p]              k4
        Send a read-request message to the local LFS         k5
    Else
        Build a token {EOF, file key,
            MyName, token.Number}                            k6
        Send the new token to token.Source                   -
While not EOF

If (not token.WriteAll)                                      k7
    Build a token {true, MAXKEY,
        MyName, token.Number}                                k6
    Send the token to old token.Source                       -
```

Figure 2 Merge Pseudo-Code

leaved and let N be the number of records in this file. Let us also refer to source and destination sub-files simply as files. Each source file will of course be interleaved over half as many nodes as the destination file, and will consist of half as many records. Moreover, the merge steps that make up an overall merge sort will often manipulate significantly fewer records than comprise the entire file, and will use significantly fewer nodes.

We will call the sequential part of the algorithm its *limiting section*. The rest of the code can execute in parallel with disk I/O. The critical code comprises the loop in Figure 2.

There are two cases in the loop, one taking time $T_{act} = k_2+k_3$ and the other taking time $T_{pass} = k_2+k_6$. Since the first case will be executed N times before all the records are written and the algorithm terminates, the total time used by that code will be $T_{act}N$. The second case is executed whenever the token passes from one file to the other.

A run is a string of records merged from the same file. A crossover stands between runs. If C is the number of crossovers in the merge, the total time used by case 2 is $T_{pass}C$.

To analyze the behavior of the algorithm as a whole, we must consider the extent to which the limiting section can execute in parallel with disk I/O. The second case of the loop has no parallel part, but case 1 includes both a read and a write: $T_{read} = k_5$ and $T_{write} = k_4$. The limiting section has the potential to become significant when a process of a source file finishes reading its next record before the token returns, or when a process of the destination file finishes writing a record before being given another one. The time required for the token to return to the same reading process can be as small as $T_{act}p/2$, or as large as $T_{act}(p/2+N/2) + T_{pass}\min(C, p/2)$, since it is possible for the entire other source file to be traversed before returning. Similarly, the time that elapses between writes to the same output process can be as small as $T_{act}p$, or as large as $T_{act}p + T_{pass}\min(C, p)$. On average, the time to complete either a read or a write "circuit" should be $p(T_{act} + T_{pass}C/N)$. The extent to which individual circuits deviate from the average will depend on the uniformity of the distribution of crossovers.

We want to discover the number of processors that can be used effectively to sort. We must therefore determine the point at which I/O begins to wait for the sequential token passing. Average case behavior can only be used with care because an unusually brief circuit saves no time (I/O is still the limiting factor), whereas an unusually long circuit loses time by allowing the sequential component to dominate. Fortunately, we can make the fluctuations negligible in practice by allowing source file processes to read ahead and write behind.

Since the output file is interleaved across the same disks as the input files, we will obtain linear speedup so long as every disk is kept continually busy reading or writing useful records. A source file process whose disk has nothing else to do should simply read ahead. Finite buffer space will not constitute a problem until the limiting section begins to dominate overall. In our implementation, the timing anomalies caused by uneven crossover distribution (on random input data) are rendered negligible with only one record of read ahead.

Execution time for the merge algorithm as a whole can be approximated as

$$T_{merge} = T_{fixed} + \frac{N}{p}T_{delete} + \max\left(NT_{act} + C\,T_{pass}, \frac{N}{p}T_{disk}\right) \quad (1)$$

where $T_{disk} = T_{read} + T_{write}$, and T_{fixed} is overhead independent of p. Each pass of the merge sort algorithm should delete its temporary files; since the delete operation for our LFS takes time k_{10} per record, $T_{delete} = k_{10}$. T_{fixed} includes the time required within each process for initialization and finalization. It also includes the time T_{eof} required in one of the token circuits to recognize the end of the first source file and build a WriteAll token. Per-phase initialization time is k_1. $T_{eof} = k_2 + k_7 + k_6$. If we let k_{11} be per-phase termination time, we have $T_{fixed} = k_1 + (k_2 + k_7 + k_6) + k_{11}$.

The merge algorithm will display linear speedup with p so long as p is small enough to keep the minimum token-passing time below the I/O times; in other words, so long as

$$p \le p_{max} = \min\left(\frac{2T_{read}}{T_{act}}, \frac{T_{write}}{T_{act}}\right) \quad (2)$$

If crossovers are close to uniformly distributed, the algorithm should actually display linear speedup so long as p is small enough to keep the *average* token-passing time below the I/O time; in other words, so long as

$$p \le \overline{p_{max}} = \frac{T_{disk}}{T_{act} + \frac{C}{N}T_{pass}} \quad (3)$$

The expected value of C can be shown to be $N/2$, so

$$\overline{p_{max}} = \frac{T_{disk}}{T_{act} + \frac{1}{2}T_{pass}}$$

and

$$\overline{T_{merge}} = T_{fixed} + \frac{N}{p}T_{delete} + \max\left(NT_{act} + \frac{N}{2}T_{pass}, \frac{N}{p}T_{disk}\right) \quad (4)$$

which we will henceforth use as our value for T_{merge}.

Merge Sort

The local external sorts in the first phase of the merge sort are ordinary external sorts. Any external sorting utility will serve for this phase. Standard external sorts will run (in parallel with each other) in time $O((N/p) \log (N/p))$. We can approximate this as

$$T_{local} = C_{local} \frac{N}{p} \left(1 + \log \frac{N}{pB}\right)$$

where B is the size in records of the in-core sort buffer. The 1 inside the parentheses accounts for one initial read and write of each record, used to produce sorted runs the size of the buffer. Internal sort time is negligible compared to the cost of I/O for merging.

Referring back to Figure 1, there are $\log p$ phases of merge, for $x = 2, 4, 8, ..., p$ (again assuming that p is a power of two). Phase x runs p/x merges, each of which uses x processors to merge Nx/p records. The expected time for phase x is therefore

$$T_x = T_{fixed} + \frac{N}{p} T_{delete} + \max \left(\frac{Nx}{p} T_{act} + \frac{Nx}{2p} T_{pass}, \frac{N}{p} T_{disk} \right)$$

If T_{create} is the time required to create $2p$ processes and to verify their termination, then the expected time for the merge sort as a whole is

$$\begin{aligned} T_{sort} &= T_{create} + T_{local} + \sum_{x=2,4,8,...,p} T_x \\ &= T_{create} + T_{local} + \log p \left(T_{fixed} + \frac{N}{p} T_{delete} \right) \\ &+ \sum_{\substack{x=2^i \\ 2 \leq x \leq p_{max}}} \frac{N}{p} T_{disk} + \sum_{\substack{x=2^i \\ p_{max} < x \leq p}} \frac{Nx}{p} T_{act} + \frac{Nx}{2p} T_{pass} \end{aligned} \quad (5)$$

If p is small enough that I/O always dominates, this is

$$T_{sort} = T_{create} + T_{local} + \log p \left(T_{fixed} + \frac{N}{p} (T_{disk} + T_{delete}) \right)$$

Otherwise our equation for T_{sort} is

$$\begin{aligned} T_{sort} &= T_{create} + T_{local} \\ &+ \lfloor \log p_{max} \rfloor \left(T_{fixed} + \frac{N}{p} (T_{disk} + T_{delete}) \right) \\ &+ \frac{N}{p} \left(T_{act} + \frac{1}{2} T_{pass} \right) \left(2p - 2^{\lfloor \log p_{max} \rfloor + 1} \right) \end{aligned} \quad (6)$$

4. Empirical Results

We have measured the sort tool's performance on our implementation of Bridge. 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 reduced local file size, n, with greater p, and the $n \log n$ local sort. Figure 4 displays our predicted results pictorially. The dashed line indicates the percentage of time spent in the merge phase. The graph clearly illustrates that merging becomes proportionally much more important with increasing p, as work moves out of the local sort and into additional merges.

Our local sort algorithm is relatively naive: a simple two-way external merge with 500-record internal sort buffers. It runs at about a quarter the speed of the Unix sort utility.

Our predicted performance figures differ only slightly from measured performance between 2 and 32 processors (see Figure 3). A detailed examination of timing data suggests that the remaining inaccuracy stems from minor contention for the local file systems that is not accounted for in the analysis.

Figure 3 plots predicted and actual performance figures up to 32 processors for the local sorts, the merge phases, and the overall sort tool. Figure 4 extends these graphs with predicted performance on larger numbers of nodes. The dashed line plots the percentage of total execution time devoted to parallel merge phases (as opposed to

Table 1 Merge Tool Performance (10 Mbyte file, times in minutes)

	Merge Phases		Local Sort		Merge Sort Total		
Processors	Measured	Predicted	Measured	Predicted	Measured	Predicted	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

local sorts). Speedup begins to taper off noticeably beyond the point where I/O ceases to dominate sequential token passing. Performance continues to improve, but at a slower rate. The merge phase that uses 256 processors to merge two files will run at its IPC speed. The earlier phases of the merge and the local sorts will, however, run with p-way parallelism. This causes the algorithm to show some improvement with thousands of processors even though (as shown in the following section) the last stages of the merge will reach a speedup of only about 160.

Values for Constants

The figures in Table 2 were obtained by inserting timing code in the merge program. The measurement code inevitably introduced overhead but not enough to make a difference in performance. The predicted times in Table 1 were calculated with equation 6 and the constants from Table 2.

Perhaps the most important figure in Table 2 is the value for p_{max}. With the I/O, communication, and computation times found in our implementation (which we believe would be closely matched by a production-quality version of Bridge), and with fewer than 160 processors in use, there is no way for a process to complete an I/O operation before being asked to perform another one.

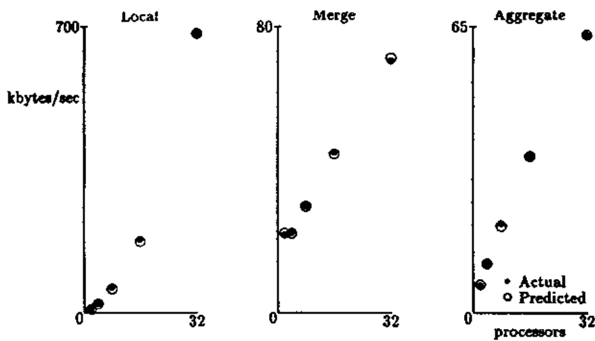

Figure 3 Predicted versus Actual Performance

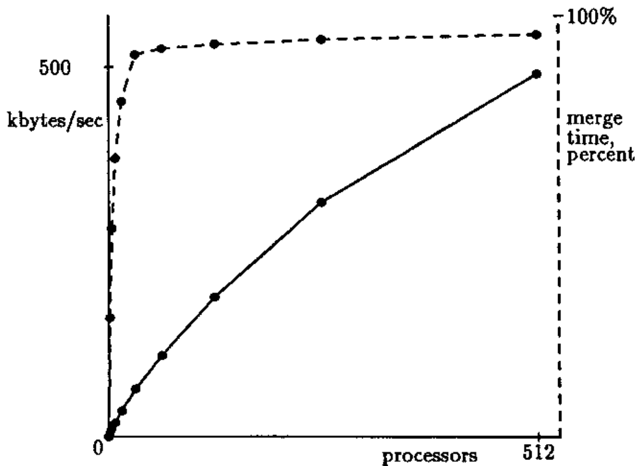

Figure 4 Predicted Aggregate Performance

5. Experiences

5.1 Analysis

Our research is focussed more on systems issues than theory, but we find nonetheless that the mathematical analysis of algorithms can play an important role in this work. In the case of the merge sort tool, our analysis has proven extremely successful. Informal analysis guided our initial choice of algorithm. Detailed analysis uncovered an im-

Table 2 Constant Values

Constant	Value	Source
k_1 — merge setup time	20000 μsec	measured (avg.)
Maximum	30951 μsec	measured
Minimum	116 μsec	measured
k_{11} — terminate a merge phase	0 μsec	hidden in start of next phase
k_2 — receive token and following tests	253 μsec	measured
k_3 — update token number and forward token	28 μsec	measured
k_4 — write a record (T_{write})	45000 μsec	measured
k_5 — read a record (T_{read})	25000 μsec	measured
k_6 — build & send a token	31 μsec	measured
k_7 — simple *if*	45 μsec	measured
k_8 — start a process	48000 μsec	measured
k_9 — end a process	300 μsec	measured
k_{10} — delete a record	20000 μsec	measured
C_{local}	45900 μsec	measured
T_{disk}	70000 μsec	$k_4 + k_5$
T_{act}	281 μsec	$k_2 + k_3$
T_{pass}	284 μsec	$k_2 + k_5$
T_{fixed}	20300 μsec	$k_1 + k_2 + k_6 + k_7 + k_{11}$
p_{max}	160	see eqn. 2
$\overline{p_{max}}$	165	see eqn. 3

portant flaw in our implementation, and yielded a trustworthy estimate of the number of nodes that could be utilized effectively.

An informal analysis of our sorting algorithm suggested that it would parallelize well despite its sequential part. Further analysis confirmed that conclusion, but only a full analysis, including all constant factors, could show the range over which we could expect the algorithm to scale. The analysis played the role of a lower bound while we tuned our implementation of the algorithm.

Our sorting algorithm is *not* parallel under asymptotic analysis. It is, however, simple and it is parallel over the range of parallelism that we have chosen to address. There are numerous truly parallel sorting algorithms, but they don't have the simplicity of our merge sort. Some of these parallel algorithms would use more than $(n \log n)/p$ reads; others are too casual about access to non-local data.

Our analytical predictions accurately match the experimental results we report in this paper, but our first experiments fell well below the predicted performance. There were wide variations in read times and there was less parallelism than expected. This cast doubt on our analysis. When we included a simple model of contention for disk drives in our analysis, we obtained a far better match with the experimental data. Unfortunately, the equations became much more complex than those in Section 3. We then collected more timing measurements, which confirmed that contention was a serious problem. Alerted to the problem, we were able to implement a simple read-ahead scheme that eliminated almost all of the contention, thereby improving performance and matching the predictions of the simpler version of the analysis.

This approach to program optimization is too painful for us to recommend as a general practice, but it was a useful tool in this specific case and is likely to be so in others. The analysis plays the role of a lower bound on run time, providing a self-sufficient benchmark for comparison with experimental results. When experimental data fails to match the analysis, there is either a problem with one's understanding of the algorithm (as reflected in the analysis) or with one's realization of the algorithm in code. In our case, the problem was the latter.

5.2 Bridge

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 many 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.

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 round-robin interleaving does not suffice.

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.

Acknowledgement

Carla Ellis contributed heavily to the early design of Bridge.

References

BBN Laboratories, "Butterfly® parallel processor overview," TR 6149, Version 2, June 1986.

Beck, M., D. Bitton, and W.K. Wilkinson, "Sorting large files on a backend multiprocessor," *IEEE Trans. Computers 37*, 7, 769-778, July 1988.

Bitton, D., D.J. DeWitt, D.K. Hsaio, and J. Menon, "A taxonomy of parallel sorting," *ACM Computing Surveys 16*, 287-318, Sept. 1984.

Dibble, P.C., M.L. Scott, and C.S. Ellis, "Bridge: A high-performance file system for parallel processors," *Proc., 8th Int'l. Conf. on Distributed Computing Systems*, 154-161, June 1988.

Gurwitz, R.F., M.A. Dean, and R.E. Schantz, "Programming support in the Cronus distributed operating system," *Proc., 6th Int'l. Conf. on Distributed Computing Systems*, 486-493, May 1986.

Kwan, S.C., "External sorting: I/O analysis and parallel processing techniques," Ph.D. Thesis, U. Washington, Jan. 1986.

Manuel, T. and C. Barney, "The big drag on computer throughput," *Electronics 59*, 51-53, Nov. 1986.

Patterson, D.A., G. Gibson, and R.H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *Proc., ACM SIGMOD Conf.*, 109-116, June 1988.

Salem, K. and H. Garcia-Molina, "Disk striping," *Proc., IEEE Conf. on Data Engineering*, 336-342, 1986; a slightly more detailed version can be found in TR 332, EECS Dept., Princeton University.

Thinking Machines, Inc., "Connection machine model CM-2 technical summary," TR HA87-4, Apr. 1987.