RAMA: Easy Access to a High-Bandwidth Massively Parallel File System

Ethan L. Miller
University of Maryland Baltimore County
Randy H. Katz
University of California at Berkeley

Abstract

Massively parallel file systems must provide high bandwidth file access to programs running on their machines. Most accomplish this goal by striping files across arrays of disks attached to a few specialized I/O nodes in the massively parallel processor (MPP). This arrangement requires programmers to give the file system many hints on how their data is to be laid out on disk if they want to achieve good performance. Additionally, the custom interface makes massively parallel file systems hard for programmers to use and difficult to seamlessly integrate into an environment with workstations and tertiary storage.

The RAMA file system addresses these problems by providing a massively parallel file system that does not need user hints to provide good performance. RAMA takes advantage of the recent decrease in physical disk size by assuming that each processor in an MPP has one or more disks attached to it. Hashing is then used to pseudo-randomly distribute data to all of these disks, insuring high bandwidth regardless of access pattern. Since MPP programs often have many nodes accessing a single file in parallel, the file system must allow access to different parts of the file without relying on a particular node. In RAMA, a file request involves only two nodes — the node making the request and the node on whose disk the data is stored. Thus, RAMA scales well to hundreds of processors. Since RAMA needs no layout hints from applications, it fits well into systems where users cannot (or will not) provide such hints. Fortunately, this flexibility does not cause a large loss of performance. RAMA's simulated performance is within 10-15% of the optimum performance of a similarly-sized striped file system, and is a factor of 4 or more better than a striped file system with poorly laid out data.

1. Introduction

Massively parallel computers are becoming a common sight in scientific computing centers because they provide scientists with very high speed at reasonable cost. However, programming these computers is often a daunting task, as each one comes with its own special programming interface to allow a programmer to squeeze every bit of performance out of the machine. This applies to file access as well; massively parallel file systems require hints from the application to provide high-bandwidth file service. Each machine's file system is different though, making programs difficult to port from one machine to another. In addition, many scientists use workstations to aid in their data analysis. They would like to easily access files on a massively parallel processor (MPP) without explicitly copying them to and from the machine. Often, these scientists must use tertiary storage to permanently save the large data sets they work with [7,15], and current parallel file systems do not interface well with mass storage systems.

Traditional MPPs use disk arrays attached to dedicated I/O nodes to provide file service. This approach is moderately scalable, though the single processor controlling many disks is a bottleneck. Recent advances in disk technology have resulted in smaller disks which may be spread around an MPP rather than concentrated on a few nodes.

RAMA addresses both ease of use and bottlenecks in massively parallel file systems using an MPP with one or more disks attached to every node. The file system is easily scalable: a file read or write request involves only the requesting node and the node with the desired data. By distributing data pseudo-randomly, RAMA insures that applications receive high bandwidth regardless of the pattern with which the data is accessed.

2. Background

Massively parallel processors (MPPs) have long had file systems, as most applications running on them require a stable store for input data and results. Disk is also used to run out-of-core algorithms too large to fit in the MPP's memory. It is the last use that generally places the highest demand on file systems used for scientific computation [14].

2.1. Parallel File Systems

Early MPP file systems made a concerted effort to permit the programmer to place data on disk near the processor that would use it. This was primarily done because the interconnection network between nodes was too slow to support full disk bandwidth for all of the disks. In the Intel iPSC/2 [16], for example, low network bandwidth restricted disk bandwidth. The Bridge file system [4], on the other hand, solved the problem by moving the computation to the data rather than shipping the data across a slow network.

Newer file systems, such as Vesta [2] run on machines that have sufficient interconnection network bandwidth to support longer paths between disked nodes and nodes making requests. These file systems must still struggle with placement information, however. Vesta uses a complex file access model in which the user establishes various views of a file. The file system uses this information to compute the best layout for data on the available disks. While this system performs well, it requires the user to tell the system how the data will be accessed. Vesta provides a default data layout, but using this arrangement results in performance penalties if the file is read or written with certain access patterns. Supplying hints may be acceptable for MPP users accustomed to complicated interfaces, but it is difficult for traditional workstation users who want their programs to be portable to different MPPs.

The CM-5 *sfs* [11] is another example of a modern MPP file system. The CM-5 uses dedicated disk nodes, each with a RAID-3 [8] attached, to store the data used in the CM-5. The data on these disks is available both to the CM-5 and, via NFS, to the outside world. The system achieves high bandwidth on a single file request by simultaneously using all of the disks to transfer data. However, this arrangement does not allow high concurrency access to files. Since a single file block is spread over multiple disks, the file system cannot read or write many different blocks at the same time. This restricts its ability to satisfy many simultaneous small file requests such as those required by compilations.

A common method of coping with the difficulties in efficiently using parallel file systems is to provide additional primitives to control data placement and manage file reads and writes efficiently. Systems such as PASSION [1] and disk-directed I/O [9] use software libraries to ease the interface between applications and a massively parallel file system. These systems rely on the compiler to orchestrate the movement of data between disk and processors in a parallel application. While this solution may work well for specialized MPP applications, though, it does not allow the integration of parallel file systems with the networks of workstations used by scientific researchers. The compute-intensive applications that run on the parallel processor may get good performance from the file system, but files must be explicitly copied to a standard file system before they can be examined by workstation-based tools.

Another shortcoming of parallel file systems is their inability to interface easily with tertiary storage systems. Traditional scientific supercomputer centers require terabytes of mass storage to hold all of the data that researchers generate and consume [6]. Manually transferring these files between a mass storage system and the parallel file system has two drawbacks. First, it requires users to assign two names to each file — one in the parallel file system, and a different one in the mass storage system. Second, it makes automatic file migration difficult, thus increasing the bandwidth the mass storage system must provide [15].

2.2. Parallel Applications

Parallel file systems are primarily used by compute-intensive applications that require the gigaflops available only on parallel processors. Many of these applications do not place a continuous high demand on the parallel file system because their data sets fit in memory. Even for these programs, however, the file system can be a bottleneck in loading the initial data, writing out the final result, or storing intermediate results.

Applications such as computational fluid dynamics (CFD) and climate modeling often fit this model of computation. Current climate models, for example, require only hundreds of megabytes of memory to store the entire model. The model computes the change in climate over each half day, storing the results for later examination. While there is no demand for I/O during the simulation of the climate for each half day, the entire model must be quickly stored to disk after each time period. The resulting large I/Os, are large and sequential.

Some applications, however, have data sets that are larger than the memory available on the parallel processor. These algorithms are described as running *out-of-core*, since they must use the disk to store their data, staging it in and out of memory as necessary. The decomposition of a 150,000 × 150,000 matrix requires 180 GB of storage just for the matrix itself; few parallel processors have sufficient memory to hold the entire matrix. Out-of-core applications are written to do as little I/O as possible to solve the problem; nonetheless, decomposing such a large matrix may require sustained bandwidth of hundreds of megabytes per second to the file system. Traditionally, the authors of these programs must map their data to specific MPP file system disks to guarantee good performance. However, doing so limits the application's portability.

3. RAMA Design

We propose a new parallel file system design that takes advantage of recent advances in disk and network technology by placing a small number of disks at every processor node in a parallel computer, and pseudo-randomly distributing data among those disks. This is a change from current parallel file systems that attach many disks to each of a few specialized I/O nodes. Instead of statically allocating nodes to either the file system or computation, RAMA (Rapid Access to Massive Archive) allows an MPP to use all of the nodes for both computation and file service.

The location of each block in RAMA is determined by a hash function, allowing any CPU in the file system to locate any block of any file without consulting a central node. This approach yields two major advantages: good performance across a wide variety of workloads without data placement hints, and scalability from fewer than ten to hundreds of node-disk pairs. This paper provides a brief overview of RAMA; a more complete description may be found in [13].

RAMA, like most file systems, is I/O-bound, as disk speeds are increasing less rapidly than network and CPU speeds. While physically small disks are not necessary for RAMA, they reduce hardware cost and complexity by allowing disks to be mounted directly on processor boards rather than connected using relatively long cables. High network bandwidth allows RAMA to overcome the slight latency disadvantage of not placing data "near" the node that will use it; thus, RAMA requires interconnection network link bandwidth to be an order of magnitude higher than disk bandwidth; this is currently the case, and the gap in speeds will continue to widen. Network latency is less important for RAMA, however, since each disk request already incurs a latency on the order of 10 ms.

3.1. Data Layout in RAMA

Files in RAMA are composed of file blocks, just as in most file systems.However, RAMA uses reverse indices, rather than the direct indices used in most file systems, to locate individual blocks. It does this by grouping file blocks into *disk lines* and maintaining a per-line list of the file blocks stored there. Since a single disk line is 1 - 8 MB long, each disk in RAMA may hold many disk lines. A disk line, shown in Figure 1, consists of hundreds of sequential disk blocks and the table of contents (called a *line descriptor*) that describes each data block in the disk line. The exact size of a disk line depends on two file system configuration parameters: the size of an individual disk block (8 KB for the studies in this paper) and the number of file blocks in each disk line. The number of blocks per disk line was not relevant for the simulations discussed in this paper, since they did not run long enough to fill a disk line.

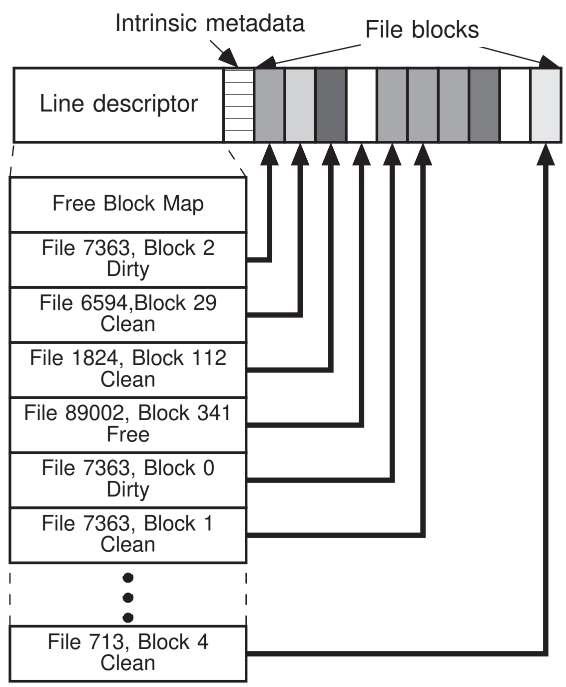

Figure 1: Layout of a disk line in RAMA.

The line descriptor contains a bitmap showing the free blocks in the disk line and a *block descriptor* for each block in the disk line. The block descriptor contains the identifier of the file that owns the block, its offset within the file, an access timestamp for the block, and a few bits holding the block's state (free, dirty, or clean).

Every block of every file in RAMA may be stored in exactly one disk line; thus, the file system acts as a set-associative cache of tertiary storage. File blocks are mapped to disk lines using the function *diskline* = hash(*fileId*, *blockOffset*). This mapping may be performed by any node in the MPP without any file-specific information beyond the file identifier and offset for the block, allowing RAMA to be scaled to hundreds of processor-disk pairs.

The hash algorithm used to distribute data in RAMA must do two things. First, it must insure that data from a single file is spread evenly to each disk to insure good disk utilization. Second, it must attempt to map adjacent file blocks to the same line, allowing larger sequential disk transfers without intermediate seeks. This is done in RAMA by dividing the block offset by an additional hash function parameter s. This scheme yields the same hash value, and thus the same mapping from file block to disk line, for s sequential blocks in a single file. The optimal value for s depends on both disk characteristics and the workload [13]. S is set to 4 for the simulations in this paper.

While any node in the MPP can compute the disk line in which a file block is stored, direct operations on a disk line are only performed by the processor to which the line's disk is attached. This CPU scans the line descriptor to find a particular block within a disk line, and manages free space for the line. The remainder of the nodes in the MPP never know the exact disk block where a file block is stored; they can only compute the disk line that will hold the block. Since the exact placement of a file block is hidden from most of the file system, each node may manage (and even reorder) the data in the disk lines under its control without notifying other nodes in the file system.

RAMA's indexing method eliminates the need to store *positional metadata* such as block pointers in a central structure similar to a normal Unix inode. This decentralization of block pointer storage and block allocation allows multiple nodes to allocate blocks for different offsets in a single file without central coordination. Since there is no per-file bottleneck, the bandwidth RAMA can supply is proportional to the number of disks. If all nodes has the same number of disks, performance scales as the number of nodes increases.

The remainder of the information in a Unix inode — file permissions, file size, timestamps, etc. — is termed *intrinsic metadata* since it is associated with the file regardless of the media on which the file is stored. The intrinsic metadata for a file is stored in the same disk line as the first block of a file in a manner similar to inodes allocated for cylinder groups in the Fast File System [12].

Since each block in RAMA may be accessed without consulting a central per-file index, the line descriptor must keep state information for every file block in the disk line. Blocks in RAMA may be in one of three states — free, dirty, or clean — as shown in Figure 1. Free blocks are those not part of any file, just as in a standard file system. Blocks belonging to a file are dirty unless an exact copy of the block exists on tertiary storage, in which case the block is clean. A clean block may be reallocated to a different file if additional free space is needed, since the data in the block may be recovered from tertiary storage if necessary.

RAMA, like any other file system, will fill with dirty blocks unless blocks are somehow freed. Conventional file systems have only one way of creating additional free blocks — deleting files. However, mass storage systems such as RASH [6] allow the migration of data from disk to tertiary storage, thus freeing the blocks used by the migrated files. RAMA uses this strategy to generate free space, improving on it by not deleting migrated files until their space is actually needed. Instead, the blocks in these files are marked clean, and are available when necessary for reallocation. RAMA also supports partial file migration, keeping only part of a file on disk after a copy of the file is on tape. This facility is useful, for example, for quickly scanning the first block of large files for desired characteristics without transferring an entire gigabyte-long file from tape.

3.2. RAMA Operation

A read or write request in RAMA involves only two nodes: the node making the request (the client) and the node on whose disk the requested data block is stored (the server). If several clients read from the same file, they do not need to synchronize their activities unless they are actually reading the same block. In this way, many nodes can share the large files used by parallel applications without a per-file bottleneck.

Figure 2 shows the flow of control and data for a file block read in RAMA; a similar sequence is used to write a file block. First, the client hashes the file identifier and offset, computing the disk line in which the desired block is stored. The client then looks up the owner of the disk line, and sends a message to that server. The server reads the line descriptor (if it is not already cached) and then reads or writes the desired block. If the operation is a write, the data to write goes with the first message. Otherwise, the data is returned after it is read off disk.

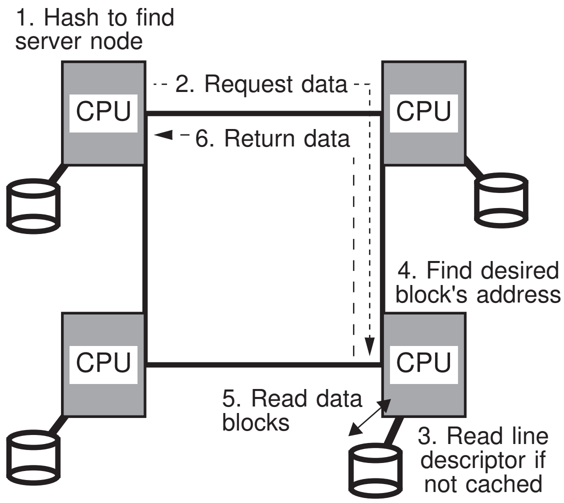

Figure 2: Steps required to read a file block in RAMA.

The common case for the file system is that the data is on disk. If a block not listed in the appropriate line descriptor is written, a free block is allocated to hold the data. If a read request cannot be satisfied, the block must exist on tertiary storage; a request is sent to the user-level tertiary storage manager process and the requested data is fetched from tape. While running the tertiary storage manager at user level adds context switch delays to the I/O time, the penalty of a few milliseconds pales in comparison to the tens of seconds necessary to fetch data from tape. Additionally, keeping tertiary storage management at user level allows much greater flexibility, as RAMA need not be recompiled (or even restarted) if a new tertiary storage device or migration policy is added.

4. Simulation Methodology

We used a simulator to compare RAMA's performance to that of a striped file system. The simulator modeled the pseudo-random placement on which RAMA is based, but did not deal with unusual conditions such as full disk lines. This limitation does not affect the findings reported later, since none of the workloads used enough data to fill the file system's disks. The simulator also modeled a simple striped file system using the same disk models, allowing a fair comparison between striping and pseudo-random distribution.

The interconnection network and disks in the MPP were both modeled in the simulator. While it would have been possible to model the applications' use of the network, this was not done for two reasons. First, modeling every network message sent by the application would have slowed down simulation by a factor of 10 or more. Second, the network was not the bottleneck for either file system, as Section 5.3 will show. The simulator did model network communications initiated by the file system, including control requests from one node to another and file blocks transferred between processors. This allowed us to gauge the effect of network latencies on overall file system performance.

The disks modeled in the simulator are based on 3.5" low-profile Seagate ST31200N drives. Each disk has a 1 GB capacity and a sustained transfer rate of 3.5 MB/s, with an average seek of 10 ms. The seek time curve used in the simulation was based on an equation from [10] using the manufacturer's seek time specifications as inputs.

The workload supplied to the simulated file systems consisted of both synthetic benchmarks and real applications. The synthetic access patterns all transfer a whole file to or from disk using different, but regular, orderings and delays between requests. For example, one simple pattern might require each of n nodes to sequentially read $1/n$th of the entire file in 1 MB chunks, delaying 1 second between each chunk. This workload generated access patterns analogous to row-order and column-order transfers of a full matrix.

Real access patterns, on the other hand, were generated by simulating the file system calls from a parallel application. All of the computation for the program was converted into simulator delays, leaving just the main loops and the file system calls. This allowed the simulator to model applications that would take hours to run on a large MPP and days to run on a workstation, and require gigabytes of memory to complete.

The modeled program used for many of the simulations reported in this paper was out-of-core matrix decomposition [5,19], which solves a set of n linear equations in n variables by converting a single $n \times n$ matrix into a product of two matrices: one upper- and one lower- triangular. Since large matrices do not fit into memory, the file system must be used to store intermediate results. For example, a $128K \times 128K$ matrix of double-precision complex numbers requires 256 GB of memory — more than most MPPs provide. The algorithm used to solve this problem stores the matrix on disk in segments — vertical slices of the matrix each composed of several columns. The program processes only one segment at a time, reducing the amount of memory needed to solve the problem. Before "solving" a segment, the algorithm must update a it with all of the elements to its left in the upper-triangular result. This requires the transfer of $c^2/2$ segments to decompose a matrix broken into c segments. The application prefetches segments to hide much of the file system latency; thus, the file system need only provide sufficiently fast I/O to prevent the program from becoming I/O bound. The point at which this occurs depends on the file system and several other factors — the number and speed of the processors in the MPP, and the amount of memory the decomposition is allowed to use.

5. Performance Comparison of RAMA and Striping

The RAMA file system is the product of a new of thinking about how a parallel file system should work. It is easier to use than other parallel file systems, since it does not require users to provide data layout hints. If this convenience came at a high performance cost, however, it would not be useful; this is not the case. I/O-intensive applications using RAMA may run 10% slower than they would if data were laid out optimally in a striped file system. However, improper data layout in a striped file system can lead to a 400% increase in execution time, a hazard eliminated by RAMA.

Parallel file system performance can be gauged by metrics other than application performance. Most parallel file systems use the MPP interconnection network to move data between processors. This network is also used by parallel applications; thus, a file system must that places a high load on the network links may delay the application's messages, reducing overall performance. Uniform disk utilization is another important criterion for parallel file systems with hundreds of disks. Using asynchronous I/O enables applications to hide some of the performance penalties from poorly distributed disk requests. However, uneven request distribution will result in lower performance gains from faster CPUs as some disks remain idle while others run at full bandwidth. RAMA meets or exceeds striped file systems in both network utilization and uniformity of disk usage.

5.1. Application Execution Time

The bottom line in any comparison of file systems is application performance. We simulated the performance of several I/O intensive applications, both synthetic and real, under both RAMA and standard striped MPP file system with varying stripe sizes. We found that RAMA's pseudo-random distribution imposed a small penalty relative to the best data layout in a parallel file system, while providing a large advantage over poor data layouts.

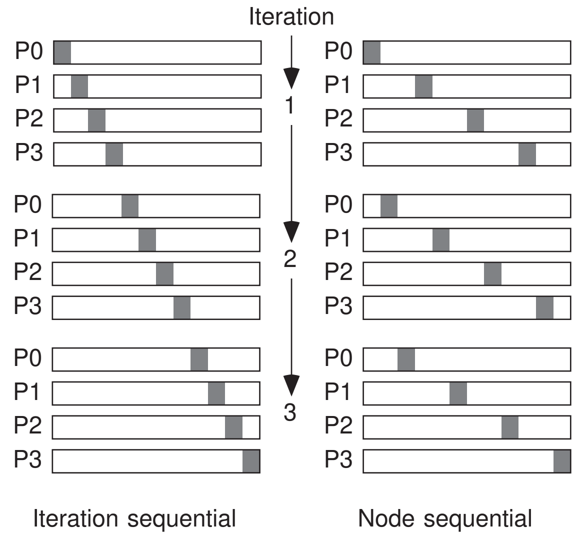

Figure 3: Transfer ordering for iteration sequential and node sequential access patterns.

The first benchmark we simulated read an entire 32 GB file on an MPP with 64 nodes and 64 disks. Each processor in the MPP repeatedly read 1 MB, waiting for all of the other processors to read their chunk of data before proceeding to the next one. These reads could be performed in two different orders resembling those shown in Figure 3: node sequential and iteration sequential. For node sequential access, the file was divided into 512 MB chunks, each of which was sequentially read by a single node. Iteration sequential accesses, on the other hand, read a contiguous chunk of 64 MB each iteration, using all 64 nodes to issue the requests. While the entire MPP appeared to read the file sequentially, each node did not itself issue sequential file requests.

We simulated this access pattern on several different configurations for the striped file system as well as the RAMA file system. Each curve in Figure 4 shows the time required to read the file for the striping configuration with N disked nodes and D disks per node, denoted by Nn, Dd on the graph. We varied the size of the stripe on each disked node to model different layouts of file data on disk. The horizontal axis of Figure 4 gives the amount of file data stored across all of the disks attached to a single disked node before proceeding to the next disked node in the file system. The dashed lines show the execution time for the iteration sequential access pattern, while the solid lines graph node sequential access. RAMA's performance for a given access pattern is constant since it does not

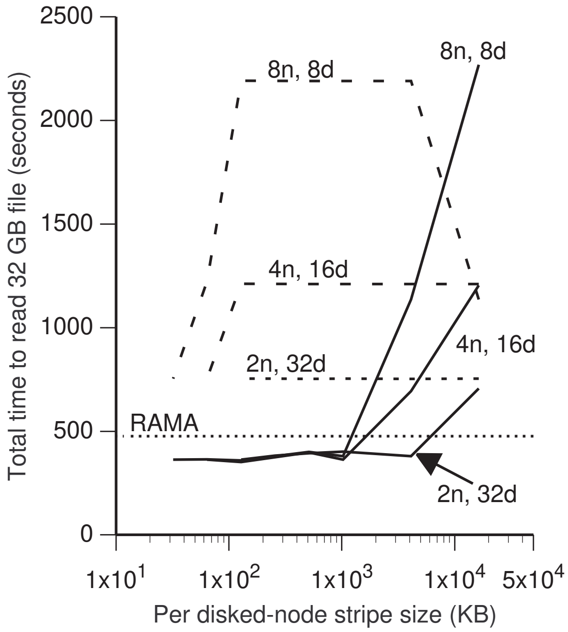

Figure 4: Time required to read a 32 GB file on an MPP with 64 processors and 64 disks.

use layout information; thus, there is only one execution time for each access pattern. Since the execution times for the different access patterns under RAMA were within 0.1%, RAMA's performance is shown as a single line.

As Figure 4 shows, RAMA is within 10% of the performance of a striped file system with optimal data layout. Non-optimal striping, on the other hand, can increase the time needed to read the file by a factor of four. Worse, there is no striped data layout for which both access patterns perform better than RAMA. There is thus no "best guess" the file system can make that will provide good performance for both access patterns. With RAMA, however, pseudo-random data layout probabilistically guarantees near-optimal execution time.

Real applications such as the out-of-core matrix decomposition described in Section 4, exhibit similar performance variations for different data layouts in a striped file system. As with the earlier benchmarks, however, RAMA provides consistent run times despite variations in the algorithm.

Matrix decomposition stresses striped file systems by only transferring a portion of the file each iteration. If each of these partial transfers is distributed evenly to all of the disks, the performance shown in Figure 5 results. Most of the data layouts for the striped file system allow the application to run without I/O delay, while only the largest file system stripe sizes are sub-optimal. Performance under RAMA matches that of the best striped arrangements, and is better than execution time for the worst data layouts.

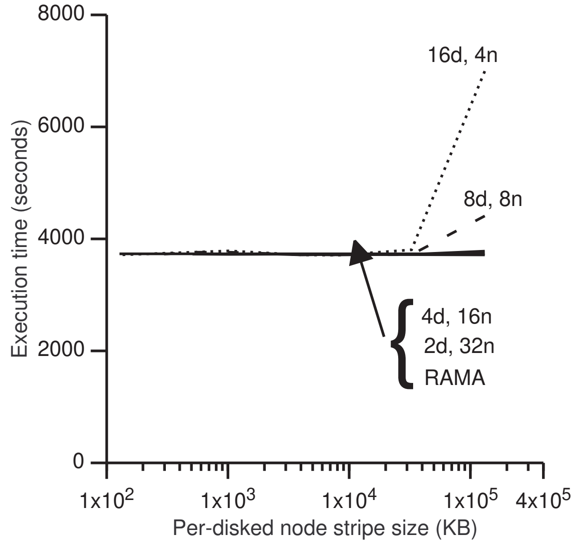

Figure 5: Execution time for LU decomposition under RAMA and striped file systems on a 64 node MPP with 64 disks.

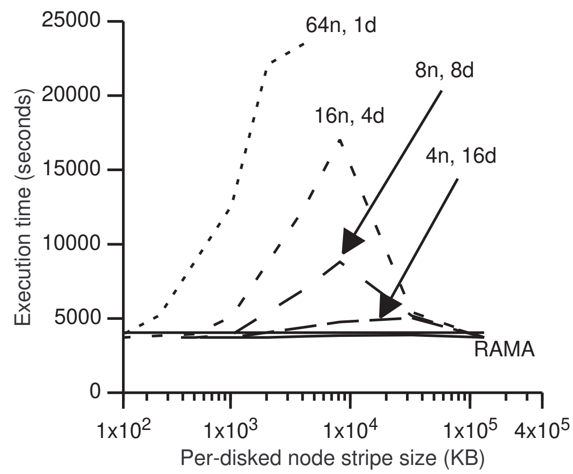

Figure 6: Execution time for LU decomposition with an alternate data layout.

Just a small change in the algorithm's source code governing the placement of data, however, can cause a large difference in performance for matrix decomposition under striping. The data in Figure 6 were gathered from a simulation of a matrix decomposition code nearly identical to that whose performance is shown in Figure 5. The sole difference between the two is a single line of code determining the start of each segment in the matrix. An minor arbitrary choice such as this should not result in a radical difference in performance; it is just this sort of dependency that makes programming parallel machines difficult. Performance under file striping, however, is very different for the two data layouts. The small stripe sizes that did well in the first case now perform poorly with the alternate data layout. On the other hand, large stripe sizes serve the second case well, in contrast to their poor performance with the first data layout. In contrast, the execution times for the two variants using RAMA are within 0.1%.

Simulation results from other synthetic reference patterns and application skeletons showed similar results. A global climate model, for example, attained its highest performance for medium-sized stripes. Execution time using either large or small stripes was two to four times longer. Using RAMA's pseudo-random distribution, the climate model was able to run at the same speed as the optimal striped data layout.

5.2. Disk Utilization

There are two reasons for the wide variation in program performance under file striping: poor spatial distribution and poor temporal distribution of requests to disks. The first problem occurs when some disks in the file system must handle more requests than others because the application needs the data on them more frequently. Even if all of the disks satisfy the same number of requests during the course of the application's execution, however, the second problem may remain. At any given time, the program may only need data on a subset of the disks; the remaining disks are idle, reducing the maximum available file system bandwidth. RAMA solves both of these problems by scattering data to disks pseudo-randomly, eliminating the dissonance caused by conflicting regularity in the file system and the application's data layouts.

Figure 7 shows the average bandwidth provided by each of 64 disks in a striped file system during the decomposition of a $32K \times 32K$ matrix. The bandwidth is generally highest for the lowest-number disks because they store the upper triangular portions of each segment which are read to update the current segment. The disks on the right, however, store the lower triangular parts of the segments which are not used during segment updates. Since the file system is limited by the performance of the most heavily loaded disks, it may lose as much as half of its performance because of the disparity between the disks servicing the most and fewest requests.

To prevent this poor assignment of data to disks, RAMA's hash function randomly chooses a disk for each 32 KB chunk of the matrix file. The result is the distribution of requests to disks shown in Figure 8.

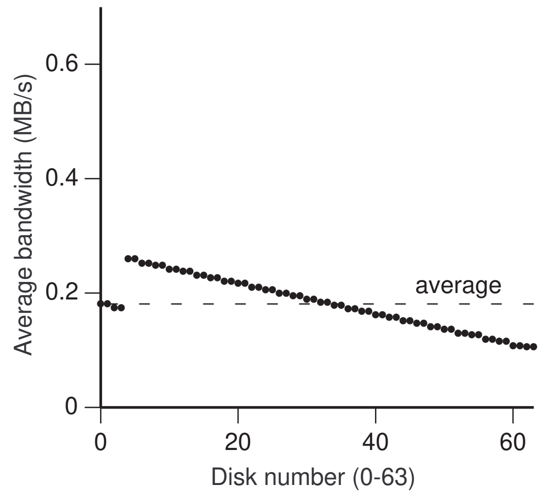

Figure 7: Disk utilization in a striped file system for a $32K \times 32K$ matrix decomposition.

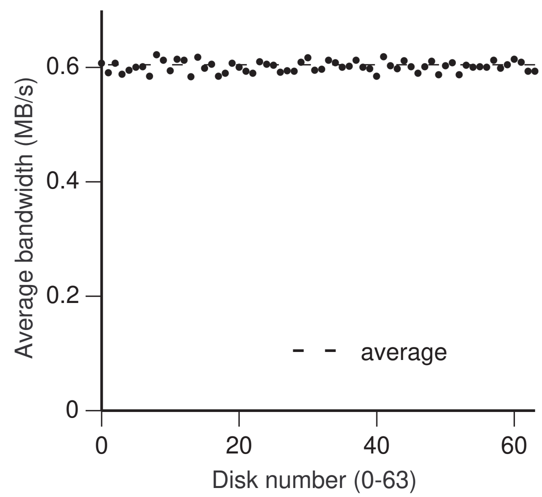

Figure 8: Disk utilization in RAMA for a $32K \times 32K$ matrix decomposition.

Each disk delivers between 0.584 and 0.622 MB/s for the $32K \times 32K$ matrix decomposition, a spread of less than 6.5%. This difference is much smaller than the factor of two difference in the striped file system. Thus, RAMA can provide full-bandwidth file service to the application, while the striped file system cannot.

Striped file systems can also have difficulties with temporal distribution, as shown by the disk activity during a 1 GB file read graphed in Figure 9. The top graph shows the ideal situation in which every disk is active all of the time. Often, however, poor disk layout results in the bottom situation. Though every disk satisfies the same number of requests during the program's execution, only half of the disks are busy at any instant, cutting overall bandwidth for the file system in half.

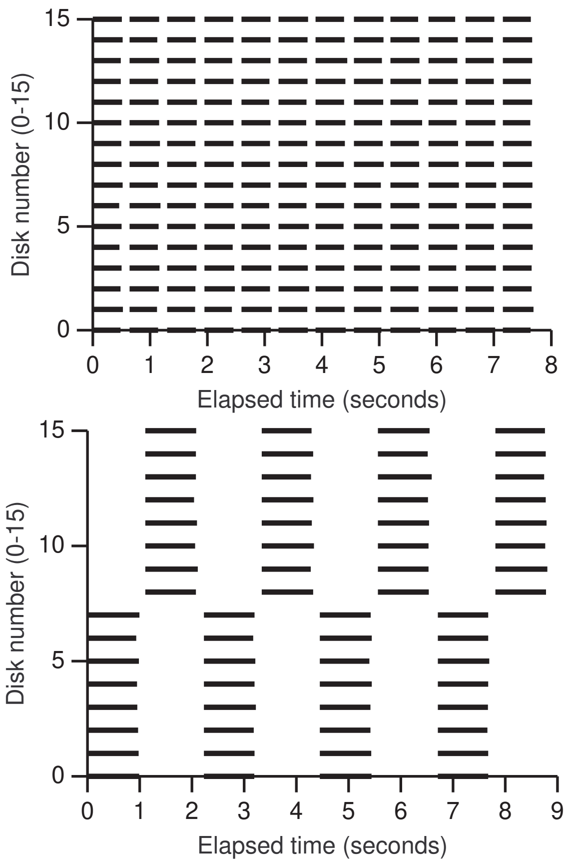

Figure 9: Disk activity over time for a striped file system during differently-ordered reads of a 1 GB file.

Here, too, RAMA's pseudo-random distribution avoids the problem. As Figure 10 shows, each disk is active most of the time. By randomly distributing the regular file offsets requested by the program, RAMA probabilistically assures that all disks will be approximately equally utilized at all times during a program's execution.

5.3. Interconnection Network Utilization

Many older parallel file systems [16,17] required data placement hints from programs to reduce network traffic as well as balance disk traffic. On some older machines, each interprocessor link was slower than 10 MB/s — hardly faster than a disk. On such a system, pseudo-random placement as done in RAMA would be a poor choice because it would place too high a load on the interprocessor links. However, interconnection networks have become faster; each processor in the Cray T3D [3] is connected to its

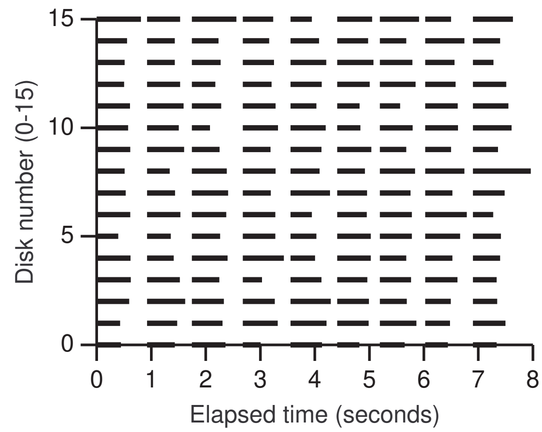

Figure 10: Disk activity over time for RAMA during a read of a 1 GB file. The access pattern is the same as the lower graph of Figure 9.

neighbors by six links each capable of transferring over 150 MB/s. The gap between network and disk speeds will only widen, since network technology is electronic while disk speeds are limited by mechanics.

As Figure 11 shows, the message traffic created by RAMA does not place high loads on a torus interconnection network with 100 MB/s links, even while all of the disks are transferring data at full speed. Average link utilization during the matrix decomposition ranged from 1.6% to 2.8%, leaving the remainder of the bandwidth for application-generated messages. The network load was evenly distributed throughout the matrix with no hot spots because the disks and requests to them were evenly spread. This uniform load decreases the travel time variation for "normal" messages, simplifying (a little bit) the creation of parallel programs.

The striped file system also caused relatively little congestion of the interconnection network — no link averaged more than 5.9% utilization over the course of the matrix decomposition. However, variation in file system link utilization was much higher than in RAMA. The links connecting to the disked nodes were, as expected, more heavily used by file system messages than those away from the disked nodes, as Figure 12 shows. The overall amount of file system message traffic was similar for RAMA and striping. However, RAMA's messages were more evenly spread through the interconnection torus. Thus, a side benefit for RAMA is better interconnection network load leveling from more uniform distribution of file system message traffic.

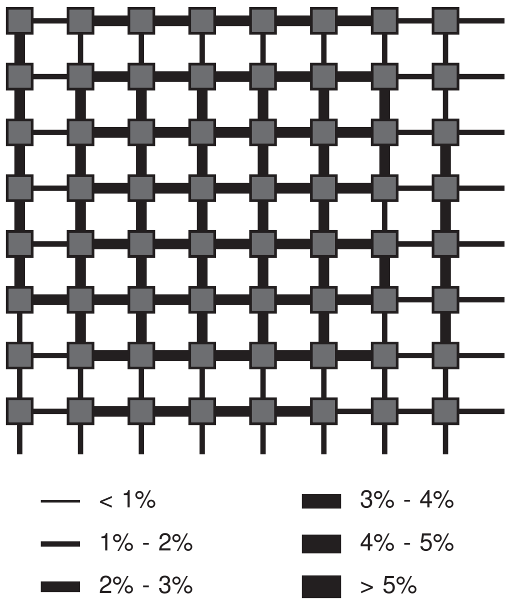

Figure 11: Interconnection network load under RAMA. The shaded nodes have disks attached, all of which are being used to read a file at full speed.

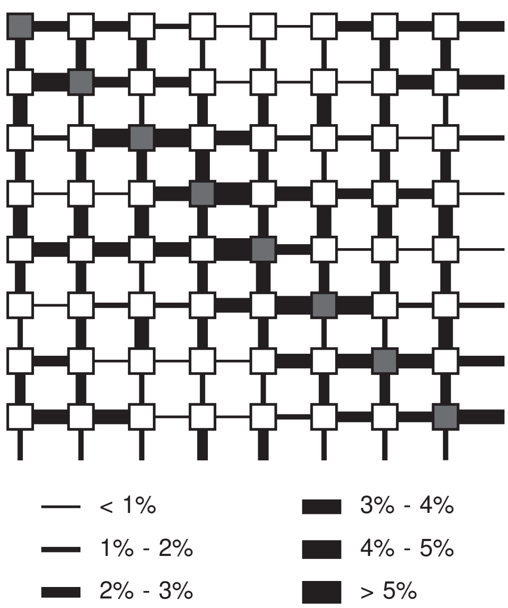

Figure 12: Interconnection network load under a striped file system. The program run is the same as in Figure 11.

6. Small File Performance

A major attraction of the RAMA file system is that it performs well on high-volume small file workloads as well as on supercomputer workloads. The workloads in Figure 13 use constant file sizes requested at different rates to generate each curve. Rather than use a closed system in which a fixed number of processes make requests as rapidly as possible, the RAMA simulator schedules requests according to a Poisson process whose average size and request rate are parameters to the workload. If there are too many outstanding requests, the simulator throttles the workload by delaying until an unfinished request completes, avoiding infinite queue growth.

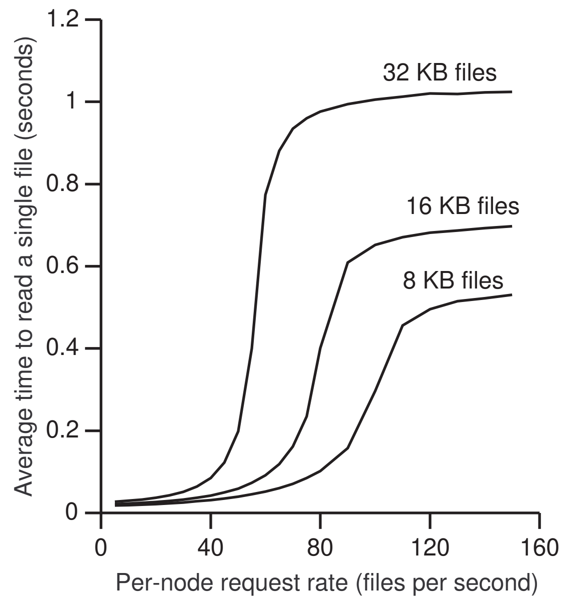

Figure 13: RAMA read performance for small files.

RAMA has low latency for small file transfers, enabling workstations connected to an MPP to access files directly instead of copying them to and from the MPP file system. Figure 13 shows RAMA's simulated performance on transfers of small files, 75% of which are reads. Even for 32 KB files, RAMA's performance does not begin to decline until the average request rate exceeds 40 requests per MPP node (and disk) per second. For the 16×8 processor mesh in Figure 13, this is an average rate of over 5000 requests per second.

RAMA is able to maintain this high level of performance for small files because file data is already distributed pseudo-randomly. The request stream from a workstation network results in a disk request stream similar to that for a single large file — both request lots of data in a somewhat random fashion. As with large files, no layout information need be supplied for small files, allowing workstations to use standard Unix file access semantics.

7. Future Work

The simulation results in this paper show that the RAMA file system design has great promise as a file system for future massively parallel machines. Many questions still remain to be answered, however. Issues to be explored further include RAMA's integration with tertiary storage and file migration, the testing of additional parallel applications, and the actual implementation of the RAMA file system using the experience gained from simulation.

One of the main attractions of RAMA for a scientific environment is its tight integration with tertiary storage. RAMA provides a facility unique among file systems for scientific storage — support for partial file migration. We will explore file migration algorithms, considering partial file migration and other developments in the fifteen years since [18].

We are also planning to build a prototype version of RAMA on a parallel processor. This can be done in two ways: as a software library layered over a generic file system, or as a replacement for an MPP file system. The first approach would be simpler, but the latter will prove a better test of RAMA's ideas. A true RAMA system will provide a good testbed for I/O-intensive parallel applications. Running real programs on this testbed will show that programmers need not spend their energy trying to lay data out on disk; the file system can do the job just as well using pseudo-random placement.

An implementation of RAMA will also be a good place to explore RAMA's design space. How much consecutive file data should be stored on a disk before randomly selecting another? How big should a disk line be? How will performance be affected as disk lines fill and allocation becomes more difficult? A RAMA prototype will allow us to address these issues by experimenting on a real system.

8. Conclusions

Traditional multiprocessor file systems use striping to provide good performance to massively parallel applications. However, they depend on the application to provide the file system with placement hints. In the absence of such hints, performance may degrade by a factor of four or more, depending on the interaction between the program's data layout and the file system's striping.

RAMA avoids the performance degradation of poorly configured striped file systems by using pseudo-random distribution. Under this scheme, an application is unlikely to create hot spots on disk or in the network because the data is not stored in an orderly fashion. Laying files on disk pseudo-randomly costs, at most, 10-20% of overall performance when compared to applications that stripe data optimally. However, optimal data striping can be difficult to achieve. Applications using striped file systems may increase their execution time by a factor of four if they choose a poor data layout. This choice need not be the fault of the programmer, as simply using a machine with its disks configured differently can cause an application's I/O to run much less efficiently. RAMA's performance, on the other hand, varies little for different data layouts in full-speed file transfers, matrix decomposition, and other parallel codes.

The flexibility that RAMA provides does not exact a high price in multiprocessor hardware, however. RAMA allows MPP designers to use inexpensive commodity disks and the high-speed interconnection network that most MPPs already have. It is designed to run on an MPP built from replicated units of processor-memory-disk, rather than the traditional processor-memory units. This method of building MPPs removes the need for a very high bandwidth link between an MPP and its disks; instead, the file system uses the high-speed network that already exists in a multiprocessor. Since the file system is disk-limited, though, the network is never heavily loaded.

Disks, too, are utilized well in RAMA. Pseudo-random distribution insures an even distribution of data to disks. Disk requests are evenly distributed to disks in time as well as in space. Thus, no disk serves as a bottleneck by servicing too many requests at any time. In addition, all disks are used nearly equally at every step of an I/O-intensive application without the need for data placement hints.

The simulations of both synthetic traces and cores of real applications show that the pseudo-random data distribution used in RAMA provides good performance while eliminating dependence on user configuration. While RAMA's performance may be 10-15% lower than an optimally configured striped file system, it provides a factor of four or more performance improvement over a striped file system with a poor layout. It is this portability and scalability that make RAMA an excellent file system choice for the multiprocessors of the future.

References

[1] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. "PASSION: Parallel And Scalable Software for Input-Output." Technical Report

NPAC Technical Report SCCS-636, NPAC and CASE Center, Syracuse University, Sept. 1994.

[2] P. F. Corbett, D. G. Feitelson, J.-P. Prost, and S. J. Baylor. "Parallel access to files in the Vesta file system." In *Proceedings of Supercomputing '93*, pages 472–483, Portland, Oregon, Nov. 1993.

[3] Cray Research, Inc. "Cray T3D system architecture overview manual," Sept. 1993. Publication number HR-04033.

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

[5] G. A. Geist and C. H. Romine. "LU factorization algorithms on distributed-memory multiprocessor architectures." *SIAM Journal of Scientific and Statistical Computing*, 9(4):639–649, July 1988.

[6] R. L. Henderson and A. Poston. "MSS-II and RASH: A mainframe UNIX based mass storage system with a rapid access storage hierarchy file management system." In *USENIX — Winter 1989*, pages 65–84, 1989.

[7] D. W. Jensen and D. A. Reed. "File archive activity in a supercomputer environment." Technical Report UIUCDCS-R-91-1672, University of Illinois at Urbana-Champaign, Apr. 1991.

[8] R. H. Katz, G. A. Gibson, and D. A. Patterson. "Disk system architectures for high performance computing." *Proceedings of the IEEE*, 77(12):1842–1858, Dec. 1989.

[9] D. Kotz. "Disk-directed I/O for MIMD multiprocessors." Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994.

[10] E. K. Lee and R. H. Katz. "An analytic performance model of disk arrays." In *Proceedings of SIGMETRICS*, pages 98–109, May 1993.

[11] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler. "*sfs*: A parallel file system for the CM-5." In *Proceedings of the 1993 Summer Usenix Conference*, pages 291–305, 1993.

[12] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. "A fast file system for UNIX." *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

[13] E. L. Miller. *Storage Hierarchy Management for Scientific Computing*. PhD thesis, University of California at Berkeley, to be published in early 1995.

[14] E. L. Miller and R. H. Katz. "Input/output behavior of supercomputing applications." In *Proceedings of Supercomputing '91*, pages 567–576, Nov. 1991.

[15] E. L. Miller and R. H. Katz. "An analysis of file migration in a Unix supercomputing environment." In *USENIX—Winter 1993*, pages 421–434, Jan. 1993.

[16] P. Pierce. "A concurrent file system for a highly parallel mass storage system." In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[17] T. W. Pratt, J. C. French, P. M. Dickens, and S. A. Janet, Jr. "A comparison of the architecture and performance of two parallel file systems." In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 161–166, 1989.

[18] A. J. Smith. "Long term file migration: Development and evaluation of algorithms." *Communications of the ACM*, 24(8):521–532, August 1981.

[19] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. "Beyond core: Making parallel computer I/O practical." In *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

Author Information

Ethan Miller is an assistant professor at the University of Maryland Baltimore County. He received a ScB from Brown in 1987 and an MS from Berkeley in 1990. He will complete his PhD at Berkeley in early 1995. His research interests are file systems and data storage for high performance computing, including both disk and tertiary storage. Surface mail sent to the Computer Science Department, UMBC, 5401 Wilkens Avenue, Baltimore, MD 21228 will reach him, as will electronic mail sent to elm@cs.umbc.edu.

Randy Katz has been on the Berkeley faculty since 1983. He received his MS and PhD at Berkeley in 1978 and 1980 respectively. He received his AB degree from Cornell University in 1976. He may be contacted by mail at the Computer Science Division, University of California, Berkeley, CA 94720, and by electronic mail at randy@cs.berkeley.edu.