ELSEVIER

Parallel Computing 23 (1997) 419–446

PARALLEL COMPUTING

# RAMA: An easy-to-use, high-performance parallel file system

Ethan L. Miller [a,*], Randy H. Katz [b,1]

[a] *Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250, USA*

[b] *Computer Science Division, University of California at Berkeley, Berkeley, CA 94720, USA*

Received 17 May 1996; revised 30 August 1996

**Abstract**

Modem massively parallel file systems provide high bandwidth file access by striping files across arrays of disks attached to a few specialized I/O nodes. However, these file systems are hard to use and difficult to integrate with workstations and tertiary storage. RAMA addresses these problems by providing a high-performance massively parallel file system with a simple interface. RAMA uses hashing to pseudo-randomly distribute data to all of its disks, insuring high bandwidth regardless of access pattern and eliminating bottlenecks in file block accesses. 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.

*Keywords:* Parallel file system; Parallel disk; Pseudo-random file block layout; High-performance I/O; Scientific computing

## 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

* Corresponding author. E-mail: elm@acm.org

[1] E-mail: randy@cs.berkeley.edu.


*PII* S0167-8191(97)00008-2

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 [9,19], 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. The 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 [18].

### *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 [21], for example, low network bandwidth restricted disk bandwidth. The Bridge file system [6], 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 [4] 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 so Vesta can determine the optimal layout for the data. This option works well for many programs that use regular matrices; however, other programs such as irregular grid methods and Fourier transforms either lack such regularity or have I/O access patterns that change over the course of the program, making an optimal layout difficult to obtain. Vesta provides a default data layout for

users that do not supply hints, 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* [15] is another example of a modern MPP file system. The CM-5 uses dedicated disk nodes, each with a RAID-3 [10] 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 [2] and disk-directed I/O [12] 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. Parallel I/O libraries can gather small requests into the large requests that the file system prefers, provide a data layout that leads to higher I/O performance, and give programmers a higher-level interface to the low-level file system. However, libraries cannot address basic shortcomings of many parallel file systems: they do not scale well, and they are very sensitive to certain I/O patterns. In addition, this solution is not as adept at allowing the use of parallel file systems from 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 still be copied to a standard file system before they can be examined by workstation-based tools because most parallel file systems have few facilities for sharing files with workstations.

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 [8]. 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 [19].

### 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 Ref. [17].

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 Fig. 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.

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(*field*, *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.

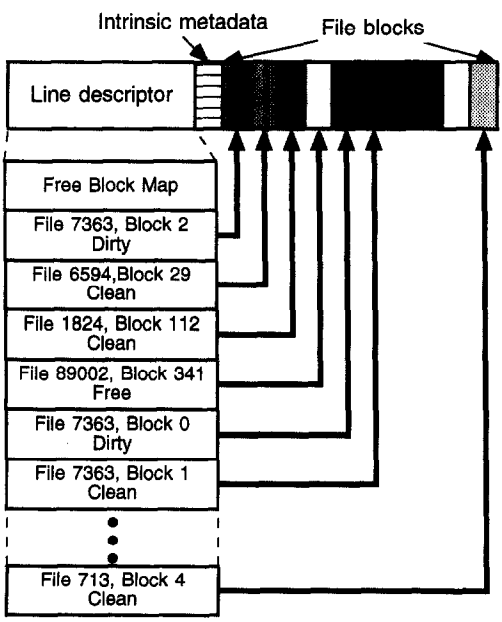


Fig. 1. Layout of a disk line in RAMA.

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 [17]. $S$ is set to 4 for the simulations in this paper.

The hash functions used for the experiments described in this paper are based on the `multiplicativehash` routine from the GNU `libg++` library [14]. This hash function is not likely to be the optimal hash function, but optimizations in the hash function are best explored when RAMA is implemented in hardware. In particular, we hope to explore hash functions that provide even distribution of data to disk lines, perhaps adapting to uneven disk line utilization. There has been research on this topic in the database community, and we hope to use some of those results in improving RAMA's performance. So long as the hash function is known to all nodes, though, the basic operation of RAMA will remain unchanged.

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 have 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 [16].

Because 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 Fig. 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 [8] 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.

Fig. 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.

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

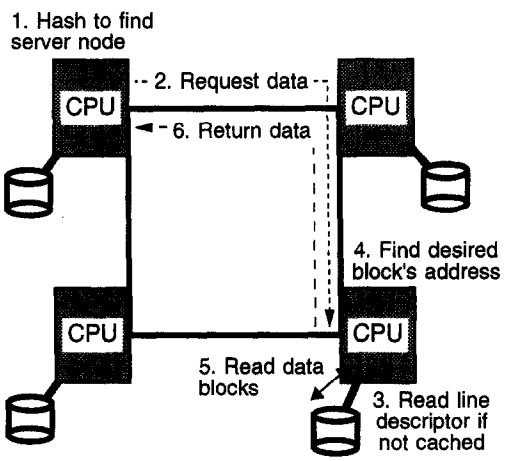


Fig. 2. Steps required to read a file block in RAMA.

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.

Note that this description of file system operation does not mention the caching of file blocks in memory. Certainly, RAMA will take advantage of file caching, particularly for frequently-used blocks such as those containing intrinsic metadata. Since the simulation experiments described in Section 5 do not involve much data reuse, a caching policy was not simulated. Parallel programs with high I/O rates will likely exhibit poor locality, since the high data rate is usually caused by an out-of-core program that must use disk I/O to stage data that will not fit in memory. In such cases, the application has wrung all possible locality from the data by doing its own 'caching.' Nonetheless, it is likely that RAMA will either cache blocks at the node on whose disk the data is stored or use more complex policies such as those discussed in Refs. [1,12], improving performance in situations where many nodes require access to the same few blocks.

### *3.3. Data integrity and availability*

As with most file systems, data integrity and availability are major issues in RAMA. Previous parallel file systems did not address this issue because they were designed purely as working storage for parallel processors. As a result, users rarely stored their files on parallel file systems for any length of time; the loss of data from a file system merely required rereading any lost data from other sources. However, this situation will change as parallel processors become more common and less expensive, allowing users to use their file systems for long-term storage.

File system integrity is especially acute in RAMA, since a single file may be spread over many disks run by many different processors. Similarly, data availability becomes a problem when parts of a single file are stored in many different places, as the file is unavailable if any of those disks or processors is down.

Data integrity is the more important issue, as a file system must never lose data entrusted to it. Additionally, the file system must insure that a block is not 'owned' by the wrong file, as doing so could allow data to be accessed by someone who does not have the proper permission. In addition, a file system must remain consistent, insuring that a crash at an inopportune moment will not corrupt the file system's data structures. After a crash, the file system must insure that every block on disk belongs to exactly one file, or is free.

Since RAMA is designed for MPPs running scientific workloads, it is not so crucial that an application knows exactly when a write is complete on disk. Many long-running programs make checkpoints of their state [22] so they can just use the last complete checkpoint if a crash occurs. So long as a crash does not corrupt existing files, a program can restart using the most recent complete checkpoint. In a scientific computing

environment, losing the last few seconds of file I/O is not fatal if the application is notified of the loss, since the data may be regenerated by rerunning all or part of the application.

One option is to use self-identifying blocks on disk. Each block would reserve a few words to record the file identifier and block number that the data in the block corresponds to. This method has several significant advantages. First, crashes no longer present any problem since the line descriptor can be rebuilt by scanning the disk line. Each node can rebuild the line descriptors on its own disks independently by reading each disk line, reassembling its line descriptor, and writing the descriptor back. Since the process uses large sequential disk reads and writes, rebuilding all of the line descriptors on a disk can be done in little more than the time necessary to read a disk at full speed — about 330 s for a 1 GB disk that can sustain a 3 MB per second transfer rate. To avoid even this small penalty, the file system assumes that all descriptors are correct, and only rebuilds one when it finds a disagreement between a line descriptor and the self-identification for a block in its line. Another advantage for this method is that line descriptors may be written back lazily. This represents a trade off between faster crash recovery time after a crash and improved performance during normal operation. All of these benefits are countered by a few drawbacks, however. One problem is the increased amount of metadata the file system will need. The overhead for metadata would double with a naive implementation that keeps a copy of all metadata in the file block as well. Keeping a full copy is unnecessary, though, and this overhead is only an additional 0.2% in any case. More importantly, though, a file block is no longer the same size as a disk block, and file blocks are no longer a power of two bytes long. Many programs are optimized to take advantage of specific file block sizes, and it is likely that this choice would cause poor performance.

A better option for maintaining consistency is to introduce a fourth state — *reclaimable* — for file blocks. This state explicitly marks those clean blocks that may be reclaimed, and includes a timestamp indicating when they were so marked. Such blocks contain valid data for which copies exist on tertiary storage. However, if a crash has occurred more recently than when the blocks were marked as reclaimable, the blocks are considered free. The fourth state thus allows RAMA to use all of the available disk space as a cache for tertiary storage while still keeping sufficient reallocatable space.

Under this scheme, shown in Fig. 3, all file data is written out before the line descriptor is updated. Clearly, this presents no difficulties if both updates are completed without a system crash. The only difficulty, then, is if the system crashes between the time that the data is written to disk and the time the line descriptor is updated. If a file's blocks are overwritten by new data for the same blocks, the line descriptor still contains the correct description for the blocks. If free blocks are overwritten and a crash occurs before the line descriptor is updated, the blocks are not part of any file and are still marked free. Again, the last few seconds of data are lost, but the file system remains consistent. However, if reclaimable blocks are reused for different files, the line descriptor will still record that they belong to the original files. The file system then uses the rule that reclaimable blocks are invalid if a crash has occurred since the blocks were marked reclaimable. The new data written to the blocks is lost, but the file system remains consistent as the blocks are now marked free. Since the blocks had to be clean

before they could be marked reclaimable, any data in them can be retrieved from tertiary storage. While retrieving the data from tertiary storage may be slow, the amount of such data is likely to be very small, and crashes are infrequent.

Since RAMA does not keep separate free block free lists as used in other file systems, blocks on disk cannot be 'lost.' Each block's state is listed in its block descriptor, so RAMA can quickly rebuild its per-line free block maps after a crash. Additionally, RAMA never needs a file system-wide consistency checking program. This is a necessary criterion for a file system that integrates tertiary storage, since checking a multi-terabyte archive could take days.

File availability is another problem that RAMA must address. Uniprocessor file systems spanning more than one disk may arrange disks in a RAID [9] to keep data available even when a disk has failed. It should be possible to use similar techniques for RAMA. However, it is not clear how they would be integrated into the file system, since each node may rearrange its own disks without notifying other nodes. RAMA can utilize techniques learned from RAID to provide availability even when a disk or processor fails. The precise method for accomplishing this is not addressed by this paper, but remains open for further research.

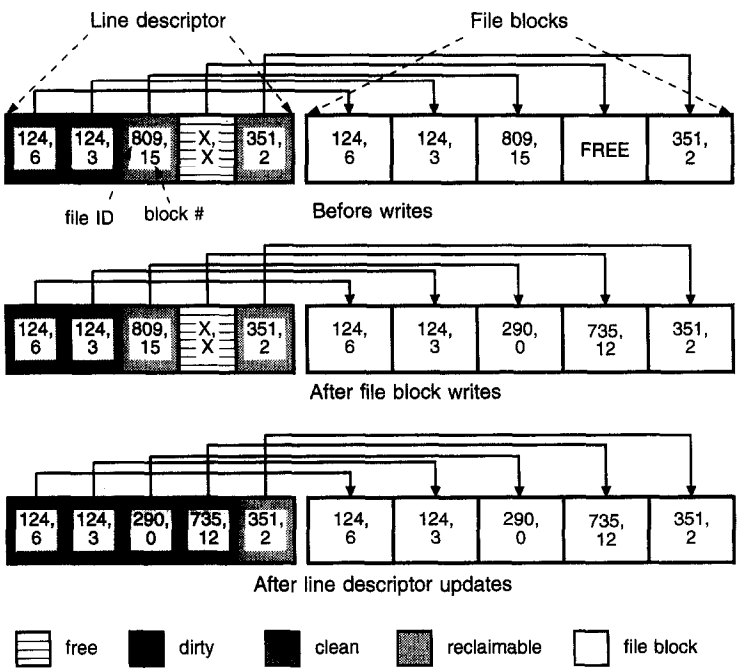


Fig. 3. Scheme for ensuring consistency after a RAMA crash.

### 3.4. Disk storage utilization

Since file blocks in RAMA are assigned to disk lines by a pseudo-random algorithm, some lines in the file system will fill up with valid data while others have plenty of free space. RAMA's use of tertiary storage, can mitigate the problem by using migration to balance the storage load. Unlike traditional file systems that must keep 10–20% of the disk blocks free [16,24], RAMA can fill the entire disk with valid data so long as there are enough clean blocks to reallocate to new files.

A disk line in RAMA is considered full only if all of its blocks are dirty. Free blocks may be reused immediately, and clean blocks can be converted to reclaimable blocks and then reallocated without referencing tertiary storage. If most or all of a line's file blocks are dirty, they must be quickly copied to some other location in the storage hierarchy so future writes can proceed at full speed. This can be done quickly if the MPP running RAMA has one or more relatively large disks running a conventional file system attached to it. This additional storage is considered part of the tertiary storage system, but it has much lower latency and higher bandwidth than tapes or optical disks. Data that must be moved here can be retrieved with latencies on the order of 100 ms or less, as compared to the multiple second penalties that tape drives impose. Since the external storage will be used infrequently — ideally for fewer than 1% of total accesses — its lower performance will have little impact on overall system performance. In this way, RAMA can greatly improve the disk storage utilization of the disks within the MPP at little performance and hardware cost.

One issue that arises with this design is that the disk lines that fill fastest must put their overflow data onto a storage medium that is considerably slower than the disks in RAMA itself. This problem may be mitigated by allowing RAMA to use a portion of each disk as a 'second chance' storage area. Disk lines that overflow before others are full may store their extra blocks here, allowing the data to be retrieved in just twice the time of 'normal' file system requests. Since the simulations in this paper did not run until disk lines became full, we were unable to gauge the effectiveness of this strategy. However, we plan to implement it as part of the RAMA file system, allowing us to measure the performance loss due to unevenness in file block distribution to disk lines.

### 3.5. Tertiary storage and RAMA

RAMA is designed to be used in high-performance computing environments requiring many terabytes of storage. File migration to and from slower, cheaper media must be well integrated into the file system. RAMA's data layout on disk is designed to facilitate such migration. While a few other file systems were designed with tertiary storage in mind [11,20], they are not designed for parallel systems. This limits their performance, and makes them unsuitable for use in a scientific computing environment.

Tertiary storage is integrated into RAMA via one or more user-level storage managers. Whenever a block of data is not found on disk, a tertiary storage manager is queried to find the data. Clearly, this method introduces longer latency than a kernel-based storage manager would. However, latency to tertiary storage is already well over a second; the additional few milliseconds make little difference in overall request latency.

It is likely that RAMA would use prefetching as well as request batching, since disk file blocks are only 8 to 32 KB long, while tertiary storage blocks might be as long as several megabytes or more. In such a case, RAMA might fetch all or much of a file from tertiary when a single block from the file was requested. If this strategy was not optimal — for example, a user might scan the first thousand bytes of each of one hundred one gigabyte files — RAMA would not have to read the entire file. Managing tertiary storage at user level also allows the use of different storage managers, permitting the integration of new devices and new algorithms for managing file migration without recompiling the operating system kernel.

Migration from secondary to tertiary storage is also managed by user-level processes. There may be more than one of these processes, but they will likely be coordinated to avoid duplication of effort. This is not a requirement, however. These processes, called migration managers, direct the copying of files from secondary to tertiary storage. RAMA has special hooks into the file system to allow this, though they are only available to programs run by the superuser. Migration managers are allowed to change the state of a file block, marking dirty blocks as clean. They may also adjust the modification time of a clean block so it will be more or less likely to be written over as more disk space is needed. However, migration managers use the standard file system interface to actually transfer file data between disk and tertiary storage.

A typical migration manager searches through every disk line looking for dirty file blocks older than a certain time. This finds file identifiers that are good candidates for migration to a lower level of the hierarchy. This task is easily parallelizable, using one migration manager for each disk. Each process reads and scans all of the line descriptors on a single disk. This is not a long procedure; a 1 GB disk has less than 4 MB of line descriptors which may be read and scanned in a few seconds. The results from all of these processes are reported to a high-level migration manager. This migration manager decides which files will be sent to tertiary storage, and manages their layout on tertiary media. It also optimizes scheduling for the tertiary media readers, trying to minimize the number of media switches.

Once a file has been written to tertiary storage, its blocks become available for reuse. However, these disk blocks are not immediately freed; instead, they are marked as clean so they may be reclaimed if necessary. There is usually no reason to free blocks once they are safely stored on tertiary media, as they might satisfy a future file request. However, the blocks' modification time might be changed. The migration manager could, for example, decide to preferentially keep blocks from small files on disk. If so, it would mark clean file blocks from large files as being older than blocks of the same age from small files. This will not confuse the file system, as a whole file's modification date remains unchanged, as does the modification date for dirty blocks. Only clean blocks which need not be written to tertiary storage may have their last access dates changed.

This architecture fits well into the Mass Storage Systems Reference Model [3]. RAMA itself acts as a bitfile server and storage server for magnetic disk. The user-level tertiary storage managers are bitfile servers for tertiary storage devices; however, they do not necessarily act as storage servers for these devices.

Tertiary storage devices such as tape libraries and optical disks may be connected to

RAMA in one of two ways. First, a device may be connected directly to a node of the parallel processor. In this case, the migration managers must know the nodes to which each device is attached. Since there will be relatively few tertiary storage devices connected directly to the parallel processor, keeping a static list of them at each migration manager presents little problem. Increasingly, though, storage libraries are being implemented as network-attached peripherals. Here, too, migration manager processes need only keep lists of devices and their network addresses, routing data and messages for them through the parallel processor nodes connected to the external network.

This paper does not address the issue of tertiary storage performance; there have been several papers addressing that issue. We believe, however, that RAMA provides an excellent platform from which to use tertiary storage because of the built-in mechanisms for handling files that are not disk-resident and migrating unused files from disk to slower storage. We plan to experiment with tertiary storage devices when an experimental RAMA file system is built.

## 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 ST3120ON 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 Ref. [13] 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 s 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 [7,26], 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 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. RAMA performance

The RAMA file system is the product of a new way 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 that places a high load on the network links may delay the application's messages, reducing overall performance. RAMA's pseudo-random distribution probabilistically guarantees even distribution of network load while not excessively congesting individual links.

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 because 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.

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 reads 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 Fig. 4: 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 Fig. 5 shows the time required to read the file for the striping configuration with $N$ disked nodes and $D$ disks per node, denoted by $N$n, $D$d 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 Fig. 5 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

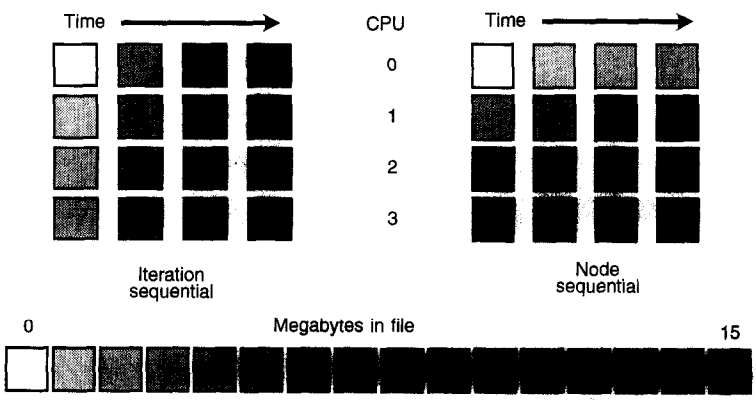


Fig. 4. Transfer ordering for iteration sequential and node sequential access patterns.

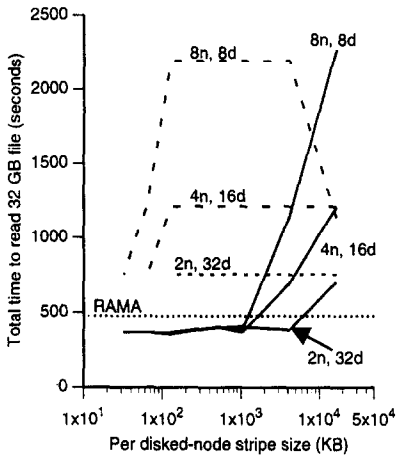


Fig. 5. Time required to read a 32 GB file on an MPP with 64 processors and 64 disks.

constant since it does not 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 Fig. 5 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

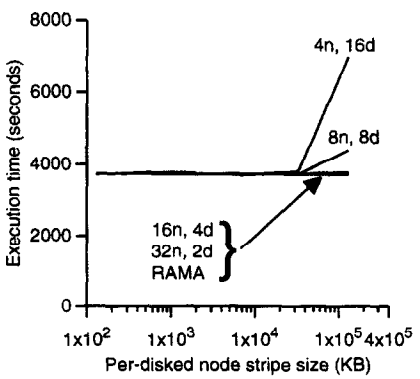


Fig. 6. Execution time for LU decomposition under RAMA and striped file systems on a 64 node MPP with 64 disks.

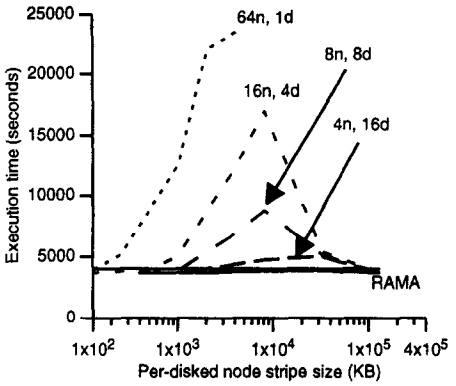


Fig. 7. Execution time for LU decomposition with an alternate data layout.

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 Fig. 6 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 suboptimal. Performance under RAMA matches that of the best striped arrangements, and is better than execution time for the worst data layouts.

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 Fig. 7 were gathered from a simulation of a matrix decomposition code nearly identical to that whose performance is shown in Fig. 6. The sole difference between the two is a single line of code determining the start of each segment in the matrix. A 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

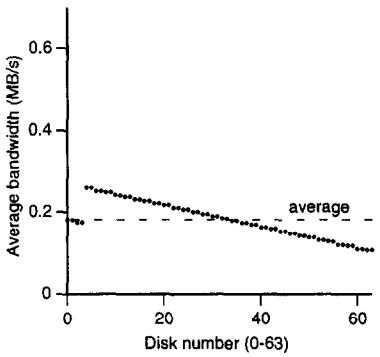


Fig. 8. Disk utilization in a striped file system for a 32K×32K matrix decomposition.

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.

Fig. 8 shows the average bandwidth provided by each of 64 disks in a striped file system during the decomposition of a 32K × 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

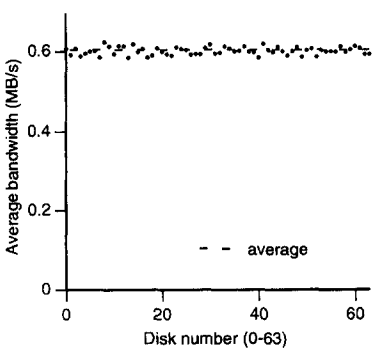


Fig. 9. Disk utilization in RAMA for a 32K×32K matrix decomposition.

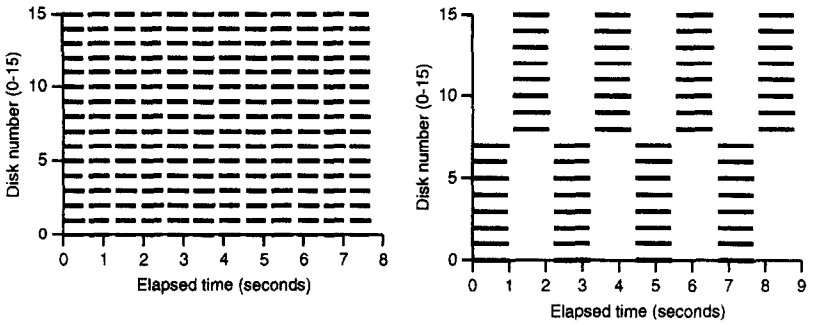


Fig. 10. Disk activity over time for a striped file system during differently-ordered reads of a 1 GB file.

chooses a disk for each 32 KB chunk of the matrix file. The result is the distribution of requests to disks shown in Fig. 9. Each disk delivers between 0.584 and 0.622 MB/s for the 32K × 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 Fig. 10. The left graph shows the ideal situation in which every disk is active all of the time. Often, however, poor disk layout results in the situation shown in the graph on the right. 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.

Here, too, RAMA's pseudo-random distribution avoids the problem. As Fig. 11 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.

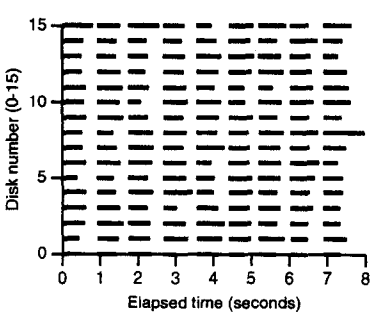


Fig. 11. Disk activity over time for RAMA during a read of a 1 GB file. The access pattern is the same as the right graph of Fig. 10.

### 5.3. Interconnection network utilization

#### 5.3.1. Network load distribution

Many older parallel file systems [21,23] 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 [5] is connected to its 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 Fig. 12 shows, the message traffic created by RAMA does not place high loads on a toroidal mesh 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 Fig. 13 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

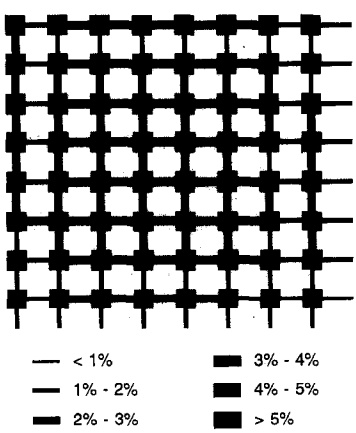


Fig. 12. Interconnection network load under RAMA. Each of the shaded nodes has a single disk attached, and all are being used to read a file at full speed.

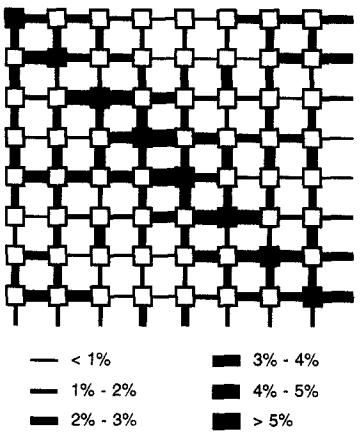


Fig. 13. Interconnection network load under a striped file system with 8 disked nodes, each of which has 8 disks attached. The program run is the same as in Fig. 12.

torus. Thus, a side benefit for RAMA is better interconnection network load leveling from more uniform distribution of file system message traffic.

*5.3.2. Sensitivity to network performance*

Since RAMA distributes data to disks pseudo-randomly, it relies heavily on the MPP interconnection network to move data from where it is stored to where it is needed. As Section 5.3.1 showed, RAMA distributes its network load more evenly than standard striped file systems. However, what happens when network latency is increased or network bandwidth is decreased? We ran simulations of the RAMA file system using two network topologies: a toroidal mesh, as in Section 5.3.1 and a star network with finite-speed links and a hub with infinite bandwidth. Networks of workstations [1] are commonly interconnected using variations of a star network, albeit with a finitely fast hub. Our simulations show that RAMA performs well on star networks, even those with high latency and relatively low bandwidth. Thus, we believe RAMA would be suitable for a network of workstations as well as for more traditional MPPs.

RAMA relies on high network bandwidth to move data between the disk the data is stored on and the MPP node requesting it. Links of 100 MB/s allow RAMA to run at full speed without congesting the interconnection network. Our first simulations tested the effects of varying network bandwidth on the time required to read a large file. While RAMA does not suffer much of a performance loss at lower bandwidths, it does cause more network congestion. Since MPP applications make heavy use of the interconnection network, they may run slower when the file system places a heavy load on the network links. Fig. 14 shows the total time needed to read a 32 GB file and the average load per network link for various link bandwidths on a $16 \times 8$ processor mesh. It takes less than 1% more time to read a file with 10 MB/s links than it does with 200 MB/s links. However, the average link load is, as expected, 20 times higher for the 15 MB/s links than for the 200 MB/s links. If the network links are, on average, 30% loaded by the file system, any applications that use the interconnection network will likely run

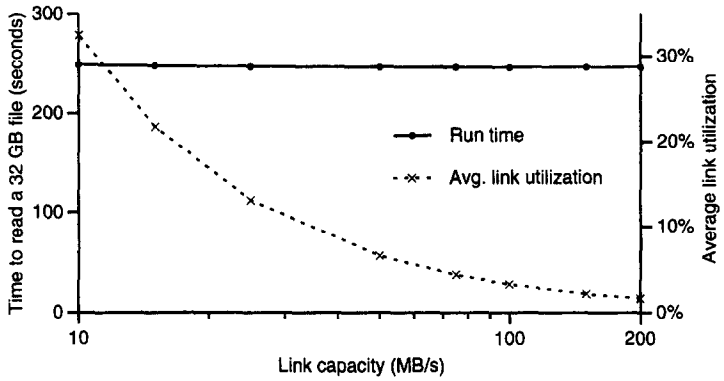


Fig. 14. Effects of varying link bandwidth in a 16×8 mesh interconnection network on RAMA read performance.

slower due to link contention. Thus, low bandwidth links are not acceptable for MPPs running, large scientific applications. However, a 30% load on the network links would be acceptable for workstation file service since file data is normally a large component of network traffic in such systems. Additionally, most workstation programs do not rely heavily on high-bandwidth low-latency network communications and would not be adversely affected by the relatively high network load.

Writing a 32 GB file produces a curve similar to that of the 32 GB read, as Fig. 15 shows. Again, file system bandwidth remains relatively constant while link bandwidth drops from 200 to 10 MB/s. The disks, not the network links, were the bottleneck in both the read and write cases. While a 10 MB/s network takes 100 ms to deliver a megabyte of data, a single disk takes at least half a second to read or write the same data. A single one megabyte file request is split into many smaller requests, each sent to a pseudo-randomly chosen disk. Since each node in the MPP is doing the same thing, each disk receives many small I/O requests. As network speed decreases, some of the

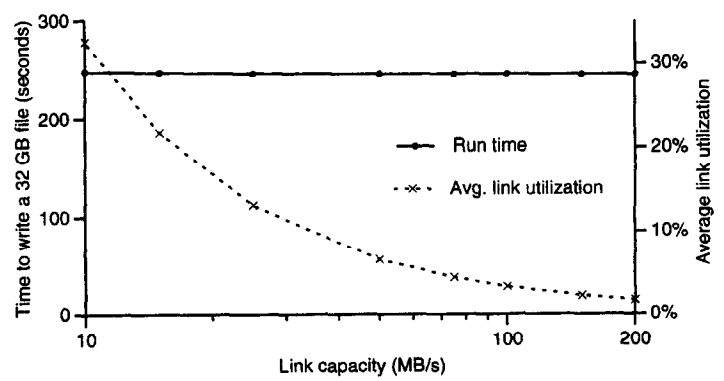


Fig. 15. Effects of varying link bandwidth in a 16×8 toroidal mesh interconnection network on RAMA write performance.

disk I/O requests are delayed. However, those same I/Os must wait even longer for the disk to finish servicing the first few requests. Thus, the file system's speed is limited by disk bandwidth. As long as the interconnection network remains close to an order of magnitude faster than disk, RAMA will perform at full efficiency for large reads and writes.

The results of a simulation using a workstation-style network are shown in Fig. 16. This set of simulations uses a star network (also called a hub-and-spoke network) rather than a toroidal mesh to connect all of the nodes, more closely resembling a network of workstations. The RAMA design provides the same bandwidth in this configuration as in the mesh configuration, but average link utilizations are lower for a star network. In an $x \times y$ mesh network, each message between client and disk must traverse an average of $x/4 + y/4$ links. If the overall file system delivers $b$ MB/s of data, each link must carry $b(x/4 + y/4)/(2n)$ MB/s, where the MPP is composed of $n = xy$ nodes. In a star network with $n$ nodes, however, each message only crosses two links — the link to the client node and the link to the disk node. Since there is only one connection to each node, however, each link carries $2b/n$ MB/s. A star interconnection network will thus run RAMA better than a mesh network with the same number of nodes and the same link bandwidth whenever $(x + y) > 16$. However, mesh networks scale much better than star networks. In a star network, all traffic must pass through a central hub. Building a hub that can support sufficiently high bandwidths between any two of hundreds of nodes is expensive, as the hub may require additional components to support the added bandwidth. On the other hand, adding nodes to a mesh simply involves replicating a basic network interface. Nonetheless, the RAMA design performs sufficiently well on a relatively slow star network to permit its use as a file system for a network of workstations.

Another important network measure is the overhead required to send a packet of information from one node to another. Most of this overhead is due to software, particularly managing the protocol stack and routing messages. Modern computer systems can take anywhere from 1 to 2000 $\mu$s to send out a single message. This delay is largely a software delay caused by a multi-layered network protocol stack. Fortu-

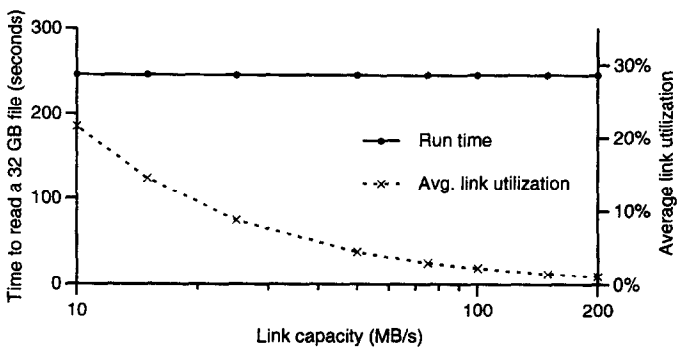


Fig. 16. Effects of varying link bandwidth in a star interconnection network on RAMA performance.

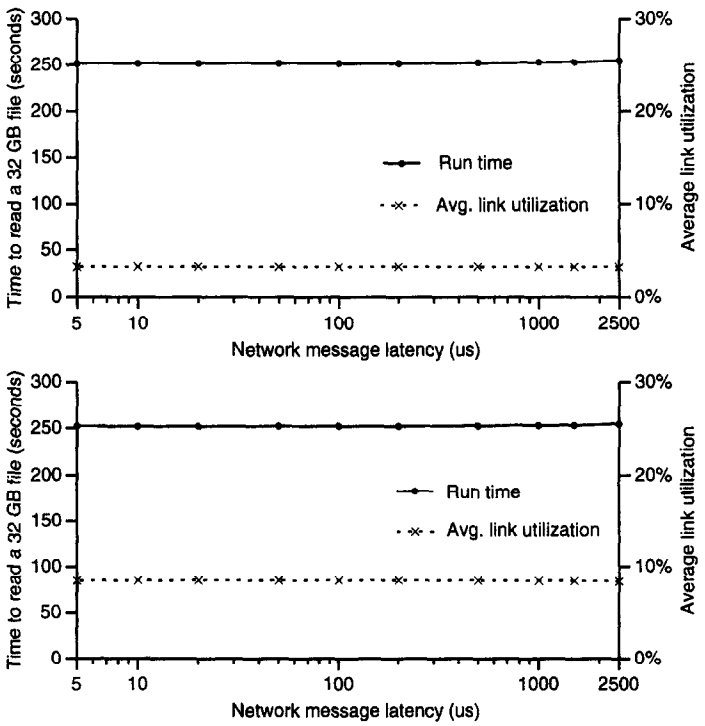


Fig. 17. Effects of varying network message latency on RAMA performance on a 16×8 toroidal mesh (top) and 128 node star network (bottom). The mesh network has 100 MB/s links, while the star network has 25 MB/s links.

nately, this penalty is usually paid only once per packet, even if the data is transmitted as many physical-layer packets and must traverse several network links.

The network model used in the RAMA simulations charges the overhead at the source node before the data is placed into the network. Since this overhead is typically a software delay, the physical network link is not considered in use during this period. Incoming or traversing messages may thus use the link while an outgoing message is in the process of being sent. Long message latencies can potentially lower overall file system performance by increasing the total response time for I/O requests. In particular, a read incurs two delays — one in sending the request, and the other in replying with the requested data.

The RAMA simulations, however, show that message latency has little effect on file system performance on large files, as file system performance is disk-limited. While long message latencies also added to the source node's processor utilization, they did not saturate it in any of the experiments. Fig. 17 shows RAMA's performance as network message latency increases from 5 to 2500 $\mu$s. The increase in message latency causes less than a 2% increase in the time needed to read a 32 GB file. Since the total elapsed time and total bandwidth show little change, link utilization also varies little as message latency increases.

This lack of variation is unexpected, as two delays of 2.5 ms each should add 5 ms to the length of each file request. However, there are two reasons why the variation due to message latency is so small. First, average response time for an individual disk request — in this case 32 KB — is nearly 300 ms. While requests use the disk for only 18.45 ms each, every processor issues 32 requests to various disks to read a total of 1 MB from disk every iteration. For a 128 processor system, this results in 4096 requests being, issued at once, for an average of 32 requests per disk. Since all of the requests are made at the same time, each request will wait, on average, for half of the 32 requests for its disk to finish before it begins service. This incurs a delay of approximately 280 ms which, when added to the average service time of 18 ms, yields a response time of just under 300 ms. An increase of 5 ms in a 300 ms request is less than 2%, in contrast to the 27% increase from a 5 ms increase to an 18 ms request.

Additionally, the disks themselves distribute replies over time. This effect is similar to the one that occurs for slower network links. The entire file system request is delayed by the message latency from the last disk request to complete. However, the latencies for the other messages do not increase the time needed to complete the file system request. While these latencies do make each individual message take longer, only the finishing time of the last disk request determines when the entire file system request finishes. The data from the other disk requests will be available to the requesting node, regardless of message latency, before the arrival of the data from the last disk request to complete.

## 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 Fig. 18 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

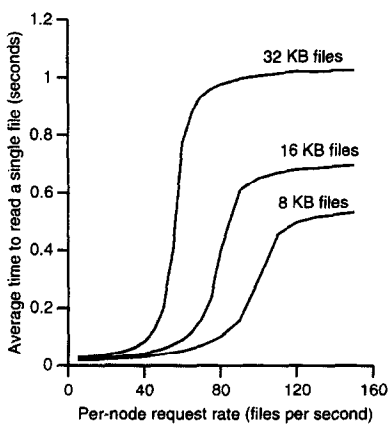


Fig. 18. RAMA read performance for small files.

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.

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. Fig. 18 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 $16 \times 8$ processor mesh in Fig. 18, 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 are exploring file migration algorithms, considering partial file migration and other developments in the fifteen years since Ref. [25].

We are hoping 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] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, D.S. Roselli, R.Y. Wang, Serverless network file systems, in: Proceedings of the 15th Symposium on Operating System Principles, Dec. 1995, pp. 109–126.

[2] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, 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.

[3] S. Coleman, S. Miller, Mass storage system reference model: Version 4, IEEE Technical Committee on Mass Storage Systems and Technology, May 1990.

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

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

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

[7] G.A. Geist, C.H. Romine, LU factorization algorithms on distributed-memory multiprocessor architectures, SIAM J. Sci. Stat. Comput. 9 (4) (1988) 639–649.

[8] R.L. Henderson, 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, 1989, pp. 65–84.

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

[10] R.H. Katz, G.A. Gibson, D.A. Patterson, Disk system architectures for high performance computing, Proc. IEEE 77 (12) (1989) 1842–1858.

[11] J.T. Kohl, C. Staelin, M. Stonebraker, HighLight: using a log-structured file system for tertiary storage management, in: USENIX, Winter 1993, Jan. 1993, pp. 435–448.

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

[13] E.K. Lee, R.H. Katz, An analytic performance model of disk arrays, in: Proceedings of SIGMETRICS, May 1993, pp. 98–109.

[14] The libg++ library for the GNU C++ compiler, version 2.7.2, Available for anonymous ftp from ftp://gatekeeper.dec.com/pub/GNU/libg++-2.7.2.tar.gz.

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

[16] M.K. McKusick, W.N. Joy, S.J. Leffler, R.S. Fabry, A fast file system for UNIX, ACM Trans. Comput. Syst. 2 (3) (1984) 181–197.

[17] E.L. Miller, Storage Hierarchy Management for Scientific Computing, Ph.D. thesis, University of California at Berkeley, Jan. 1995.

[18] E.L. Miller, R.H. Katz, Input/output behavior of supercomputing applications, in: Proceedings of Supercomputing '91, Nov. 1991, pp. 567–576.

[19] E.L. Miller, R.H. Katz, An analysis of file migration in a Unix supercomputing environment, in: USENIX-Winter 1993, Jan. 1993, pp. 421–434.

[20] M.A. Olson, The design and implementation of the Inversion file system, in: USENIX, Winter 1993, Jan. 1993, pp. 205–218.

[21] P. Pierce, A concurrent file system for a highly parallel mass storage system, in: Fourth Conference on Hypercube Concurrent Computers and Applications, 1989, pp. 155–160.

[22] J.S. Plank, Efficient Checkpointing on MIMD Architectures, Ph.D. thesis, Princeton University, June 1993.

[23] T.W. Pratt, J.C. French, P.M. Dickens, 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, 1989, pp. 161–166.

[24] M. Rosenblum, The Design and Implementation of a Log-structured File System, Ph.D. thesis, University of California at Berkeley, 1992.

[25] A.J. Smith, Long term file migration: Development and evaluation of algorithms, Commun. ACM 24 (8) (1981) 521–532.

[26] D. Womble, D. Greenberg, S. Wheat, R. Riesen, Beyond core: Making parallel computer I/O practical, in: Proceedings of the 1993 DAGSIPC Symposium, Hanover, NH, June 1993, pp. 56–63, Dartmouth Institute for Advanced Graduate Studies.