Proceedings of International Conference on Distributed
Systems, Software Engineering and Database Systems

Design and Implementation of a Parallel File Subsystem on TreadMarks

Su-Cheong Mac Ce-Kuen Shieh Bor-Jyh Shieh
Department of Electrical Engineering,
National Cheng Kung University,
Tainan, Taiwan

Li-Ming Tseng
Department of Computer Science
and Information Engineering,
National Central University,
Chung-Li, Taiwan

Abstract

*In TreadMarks and most current Distributed Shared Memory systems, file accesses are usually handled sequentially by one node. A large amount of network traffic is generated between this node and the other nodes to distribute the input data and to collect the resultant data. To reduce network traffic and shorten file access time, we have developed a parallel file subsystem on TreadMarks. A parallel file is partitioned and distributed among all nodes to parallelize file accesses. With our variable data distribution scheme, the network traffic for file accesses is greatly reduced. The total execution time of a 2000*1000 20-iteration Successive Over Relaxation (SOR) program on 8 nodes is reduced from 95 seconds with sequential I/O to 41 seconds with parallel I/O, while that of 1024*1024 Matrix Multiplication on 8 nodes is reduced from 258 seconds with sequential I/O to 236 seconds with parallel I/O.*

1. Introduction

Distributed Shared Memory (DSM) systems [1] provides a shared memory abstraction on a loosely-coupled multiprocessor or network of workstation (NOW). Since data sharing is achieved via network message transfer, one way to improve system performance is to reduce the amount of network traffic. Some DSM systems, such as TreadMarks, has been focused on relaxing memory consistency model to reduce network traffic [2][3][4], thereby improving the performance of DSM applications. However, a time consuming operation in these scientific applications, i.e., file access, has always been neglected in the design of a DSM system and is performed sequentially. This may arise performance problem.

A typical DSM application can be divided into an initialization phase, a computation phase, and a completion phase. The initialization and completion phase are always executed sequentially. In the initialization phase, memory is allocated and the input data is read from disk to memory at the root node, on which DSM system and its application are started. As a result, the input data is accumulated at this node. During computation, the other nodes obtain their required data from the root node via network by page faults. In the completion phase, the root node collects the resultant data from all nodes via network by page faults and stores the data in its disk. The root node may become a bottleneck since input and resultant data has to move via this node to the other nodes. In addition, a large amount of network traffic is produced to transfer the data to the required nodes.

An example is provided here to illustrate the impact of sequential file accesses on DSM system performance. The execution of computation phase of 2000*1000 20-iteration Successive Over Relaxation on 4 nodes under TreadMarks takes 60 seconds, while that of the initialization and completion phases takes 37 seconds. When the same program is executed on 8 nodes, computation phase takes 45 seconds, while initialization and completion phases take 50 seconds. The performance gain in computation phase by doubling number of nodes has been counteracted by the sequential file I/O.

One way to deal with this problem is to parallelize file operations as well as computation. In NOW, every workstation has its own disk. A file can be partitioned into file blocks which are scattered among all nodes to parallelize file accesses. There are already many parallel file systems developed for distributed memory multiprocessor systems [6][7][8]. In these systems, each computation node fetches required file data from the I/O modes since there is no shared memory. These parallel file systems may not be suitable for DSM systems because

This work was supported in part by the NATIONAL SCIENCE COUNCIL, project number 86-2745-E-008-003.

features of DSM systems haven't been considered in their design.

We have designed a parallel file subsystem for page-based software DSM systems that is independent of memory consistency models. A variable data distribution scheme are developed to reduce network traffic by distributing the file blocks among the nodes according to the data access pattern of an application. We provide a file interface similar to UNIX file interface, so that existing DSM applications require little modification to utilize our parallel file services. A prototype has been built on TreadMarks [2] to verify the effectiveness of our design. TreadMarks is a user-level page-based DSM system which supports lazy release consistency models and is developed by Rice University. It is built on UNIX systems, which handle all file accesses.

The structure of this paper is as follows. The design of our parallel file subsystem is present in section 2. In section 3, we describe the implementation of our prototype on TreadMarks. The performance evaluation of our prototype is illustrated in section 4. We conclude in section 5.

2. Design considerations

The target system of our parallel file subsystem is NOW, i.e., a cluster of workstations connected by network. Every workstation has its own disk. In our design, each computation node also acts as a storage node. A parallel file is partitioned into file blocks, which are grouped into file portions according to the data access pattern of the application. The file portions are then assigned to the nodes, so that each node has a disjoint portion of the parallel file, as shown in figure 1. In the following subsection, we will discuss design considerations such as granularity, data distribution scheme, and user interface. The last subsection discusses why file replication or file block replication is not supported in our prototype.

2.1 Granularity

The size of a file block can be a multiple of a memory page. If a file block is larger than a memory page, the reading of the block may require several memory pages. This granule is too large and increases the chance of imbalance in file block distribution. We choose a block size equal to that of a memory page. It allows better integration of the memory management unit and the parallel file system to avoid unnecessary network message

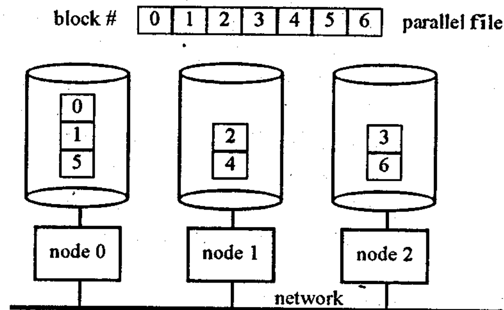

Figure 1. System architecture of our parallel file subsystem

exchange. Moreover, information collection can be done simply by collecting page table entries when file block and memory page have the same size.

2.2 Data distribution scheme

Current parallel file systems usually employ fixed distribution schemes such as interleaving to assign file blocks to disks [7][8]. With fixed distribution scheme, it is easy to locate the disk in which a file block is physically stored. However, in DSM systems with fixed distribution , a node may find that none of the required data are stored on its own disk when the application's data access pattern doesn't match the distribution scheme. A large amount of network traffic is then generated to move the data from the storage nodes to the consumer nodes. To deal with this problem, we have developed a variable distribution scheme, in which file blocks are distributed on disks according to the application's access pattern. The variable distribution scheme is achieved by a metadata file/header and information collection technique.

2.2.1 Metadata file

Since file blocks are distributed on disks according to the access pattern, which is different from application to application, a metadata file for each parallel file on each node is needed to keep track of the global block numbers of the file blocks in the local file portion. Every file block stored in the disk of a node has an entry in the associate metadata file, and this entry records the global block number of that file block in the parallel file. When the parallel file is opened, a request is broadcasted and the

metadata files are opened. When the file read is issued, a read request is broadcasted and each node reads the file blocks from its local disk into the specified shared memory locations according to the metadata file and the starting memory address given in the read function call. When a file is written, the owner node of each memory page of the result data writes that page into its local disk, and the metadata is generating by recording its block number.

2.2.2 Information collection

An information collection technique was developed to record the access pattern of the input files in a single computation phase application. In the first execution of an application, the input file (or files) is placed at the root node. The pages containing the input data will be moved or replicated from the root node to other nodes during computation. By the end of the first execution, the page table entries of the memory pages containing the file blocks are collected by the root node. With this information, the nodes that have accessed a file block can be known. In this way, the application's access pattern of the input file is ascertained and the file blocks are then redistributed accordingly. The redistribution can be achieved with the following rules: (1) If a block is exclusively accessed by a node, it is assigned to that node. (2) If a set of file blocks have been accessed by a set of nodes, they are interleaved among those nodes. The metadata file on each node is then created according to this information. This simple algorithm may cause imbalance in file block distribution and a better algorithm is being constructed. In case of a file write, the owner node of a target page writes that page into the local disk, and each node creates the metadata file accordingly. With this technique, the minimal cost is ensured except in the first execution of the application.

2.2.3 Overhead

Our variable distribution scheme may incur certain overhead. First, the metadata file will surely induce extra disk access cost. However, metadata file is much smaller than the parallel file and its access time is comparatively negligible. Moreover, metadata can be merged with its corresponding parallel file portion to further reduce its access cost. Second, file accesses are serialized in the first execution and data redistribution induces extra overhead. Actually, the metadata file can act as a template. It can be repeatedly used for the same type of programs with similar system configuration and program size but different data. When a data set is used more than one time, e.g., in some image processing applications or during debugging, the data redistribution time can be neglected.

2.3 User interface

A main issue in our design is the usability of our parallel file subsystem. Either we can preserve the UNIX file interface in TreadMarks or provide a new interface for programmers. With a new interface we can employ techniques such as collective I/O to boost the performance of file access. However, this interface is comparatively harder to use than a UNIX one, and existing TreadMarks applications require rewriting or heavy modification. With a UNIX file interface we can preserve the model described in section 1, i.e., only the root node issues all file accesses. Programmers need not coordinate the file accesses or provide data access information with this model. As a result, the existing TreadMarks applications require little modification with this interface. Many TreadMarks applications issue file accesses only in the initialization and completion phases. Accordingly, the whole parallel file must be written or read in a single access, which shortens file access time and simplifies our design. Research in workload characteristic of scientific applications shows that most files are opened as either read-only or write-only [9]. Therefore, files are immutable in our parallel file subsystem.

2.4 Replication

Theoretically, the whole parallel file or some specific file blocks may be replicated on the disks of the requesting nodes to reduce network traffic, just as DSM systems reduce network traffic by replicating memory pages on the requesting nodes. However, it is not provided in our prototype because the reduction of the total amount of network transfer is too small to make file or file block replication a worthwhile effort. Let's assume that a file block is required by two nodes. If the file block is not replicated, a consumer node acquire a copy of the block via page faults from the owner node which is the other consumer. If the block has crossed a page boundary, two page faults are generated. Otherwise, one page fault is produced. If the file block is replicated on the two nodes, these two block-replicas will

be stored into the same memory page, and two page-replicas of this page will be created on this two nodes during file read. The two nodes then report each other that each has a copy of the same memory pages. Two messages are generated if the block doesn't cross page boundary, and four are produced for boundary crossing block. Compared with the one or two page faults in the no replication case, there will be little performance gain.

3. Implementation

In this section, we begin with an introduction of TreadMarks and then describe the implementation of our parallel file subsystem on it.

3.1 TreadMarks

TreadMarks [2] is a user-level page-based DSM system built on a network of workstations running UNIX as operating system. Its developers at Rice University extend the concept of release consistency to propose lazy release consistency [9], which is implemented in TreadMarks.

File accesses in TreadMarks can be handled in sequential or parallel way, but it is not easy for programmers to access file in parallel. According to TreadMarks' manual, UNIX file operations fread() and fwrite() can be used in TreadMarks' applications. Therefore, a user can manually partitions an input file into file portions and manually distributes these portions to the required nodes. During initialization, every node reads its own file portion to shared memory. When computation completes, a programmer writes the resultant data from each node to its own disk and then combines these resultant file portions manually. This is too burdensome since programmers have to do everything manually. In contrast, programmers can access files only via the root node as described in section 1, but their programs will suffer from performance degradation.

3.2 Our prototype on TreadMarks

This section describes the implementation of our design mentioned in section 2. Important data structures and parallel file operations are described here.

3.2.1 Parallel file

As described in section 3, a parallel file is partitioned into file portions according to its data access pattern, a file portion is then transferred to its corresponding node and a metadata header records the global location of each file block in that file portion. Figure 2 shows an example parallel file and the data structure of a metadata header in our subsystem.

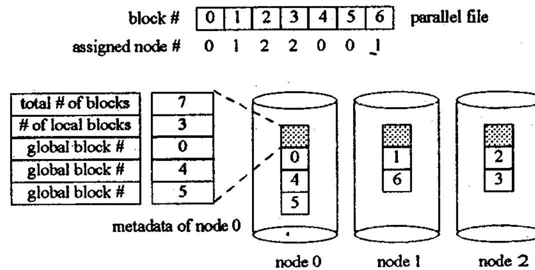

Figure 2. An example of parallel file and data structure of metadata

In this example, a 7-block file is partitioned into 3 file portions. Blocks 0, 4 and 5 are assigned to node 0, blocks 1 and 6 are assigned to node 1 and the remaining blocks are stored in node 2. The metadata header in node 0 is shown in figure 2. The first field records the total number of file blocks in the parallel file, which is 7 in this example. The second field records the number of file blocks in the local file portion, which is 3 in node 1. The following fields record the global block numbers of the file blocks in the local file portion. In the example, the first, second, and third block in node 0 is block 0, 4, and 5 of the parallel file respectively.

3.2.2 Parallel file operations

We provide the following function calls for programmers. This interface resembles the one in UNIX on which TreadMarks is built, so that existing TreadMarks' applications require little modification to utilize our parallel file services.

*int pfd = PIO_open(char *pathname, int flag)*
int PIO_close(int pfd)
*int PIO_read(int pfd, char *buff, int nbytes)*
*int PIO_write(int pfd, char *buff, int nbytes)*
pfd: parallel file descriptor

PIO_open

The root node uses this function call to open a parallel file. A parallel file descriptor (PFD) is assigned to this parallel file and the root node then broadcasts the filename and the PFD of the parallel file. Every node open its local file portion and obtains a local file descriptor (LFD). On each nodes, the PFD and its corresponding LFD are recorded in a table for future use.

PIO_close

When the root node closes a parallel file, it broadcasts the PFD of the parallel file. All nodes then check the tables that map PFD into LFD, find the corresponding LFDs, and close the local file portions.

PIO_read

When a parallel file is read, the root node broadcasts the PFD of the parallel file and the starting memory address of the memory region in which the file will be read in. Every node then calculates the memory location for each file block in the local file portion from the starting memory address and the global block number, i.e., (*the starting memory address of block A*) = (*starting memory address of the file*) + (*global block number of block A*) * (*page size*). Each file block in the local file portion of a node is read to the calculated memory pages. Afterwards, each node sends an acknowledge to the root node. This acknowledge piggybacks the metadata headers of all nodes to the root node, so that the root node knows the exact location of each memory page containing a file block. This information is then broadcast by the root node to inform all nodes about the page locations.

PIO_write

When the root node issues this function call to write a memory region to disk, the starting memory address of the memory region and the PFD of the parallel file is broadcasted. Every node then writes its valid memory pages within that memory region to the file portion of the parallel file. This is made possible with a feature of TreadMarks: when a barrier is completed, all replicas except one of a memory page are invalidated. In our DSM systems, two or replicas of a memory page may be present, and we have to propose a algorithm to decide which node should write the memory page. In TreadMarks, we just make sure that a barrier is placed after the computation phase or before the file writing of the resultant data. Actually, this is already done in most existing TreadMarks applications to ensure that all nodes have completed computation before exiting.

4. Performance evaluation

We have chosen Successive Over Relaxation (SOR) and Matrix Multiplication (MM) to test the effectiveness of our parallel file subsystem compared to sequential file operations. These two programs are the usual DSM benchmarks, possessing relatively high I/O-computation ratio compared to other DSM benchmarks. For each application, we have recorded the initialization time, the computation time, the completion time, and the total execution time of the two cases, one with parallel I/O (PIO) and one with sequential I/O (NPIO). The initialization time is the time taken to complete the initialization phase. The computation time is the elapsed time for the computation phase. The completion time is the time to write the resultant data to disk. The total execution time is the summation of the previous three times.

Our testing environment consists of 8 Pentium-90 PC, each with 32MB RAM and a Seagate ST5850A harddisk (11ms average seek time). The computers are connected by 10Mbps Ethernet. The operating system on each machine is Solaris for x86.

Successive Over Relaxation (SOR)

The input and output data of this application is a 2-dimensional array. In each iteration, the new value of an element is the average of the four neighboring elements. In our experiment, the size of the array is 2000x1000, each element is a floating point number, and there are 20 iterations. The performance result is shown in figure 3.

The initialization phase consists of a file open, a file read and a barrier. Figure 3(a) shows that the initialization time can be reduced by parallelizing file accesses. In NPIO, the initialization time increases with the number of nodes because the barrier takes longer time to complete. In figure 3(b), the computation time in PIO is shorter than that in NPIO. This is because file blocks are assigned to disks according to the data access pattern in our variable distribution scheme, resulting in the reduction of network traffic. From figure 3(c), the reduction of completion time in PIO is impressive

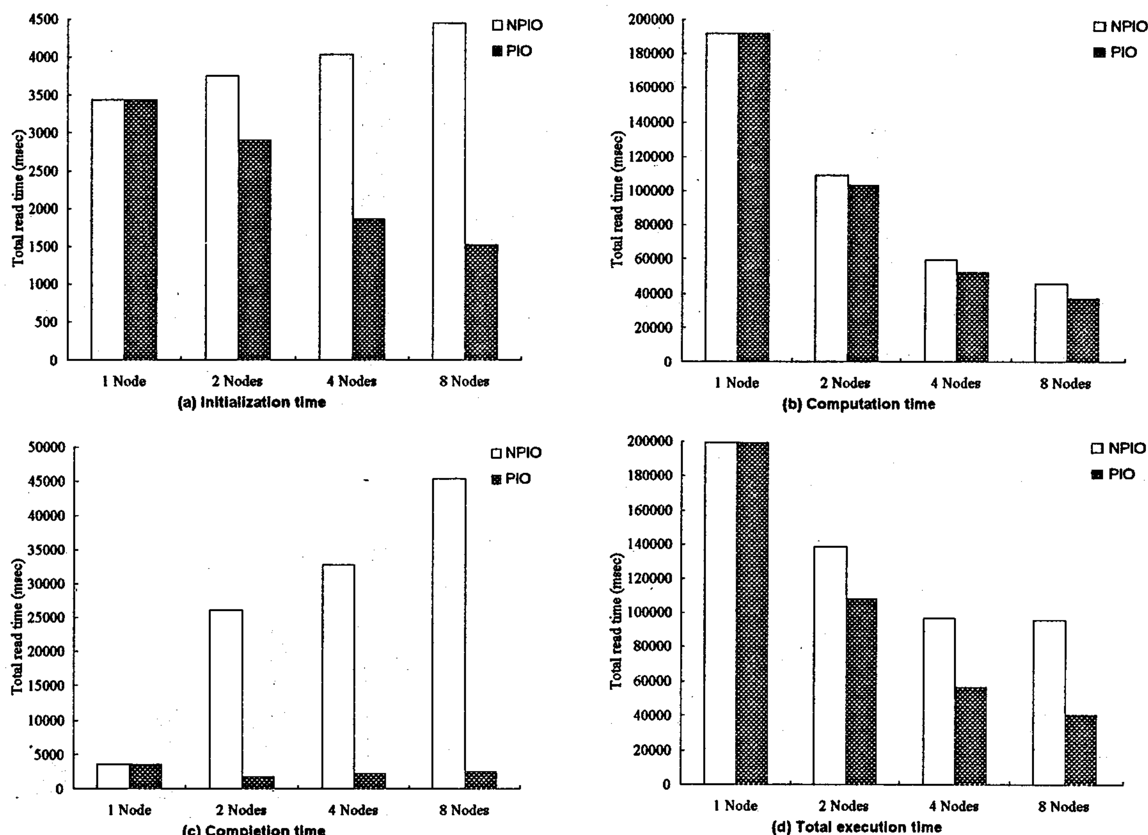

Figure 3. The performance of SOR

compared to NPIO. In NPIO, before the specific memory region is written, the root node have to collect all update information of the memory pages in that region from all nodes and updates those memory pages. This is a time consuming job. In PIO, every node writes its own valid pages to disk. No update from other nodes is required. Figure 3(d) shows the combined effect of the previous three graphs. The total execution time of SOR is significantly reduced in PIO compared to NPIO.

Matrix Multiplication (MM)

The input and output files of MM are three matrices. In our experiment, the size of a matrix is 1024x1024 and each element is a integer. The performance result is shown in figure 4.

From figure 4(a), the initialization time in PIO is only slightly shorter than that in NPIO. This is due to fact that the number of barriers in MM is twice of that in SOR, since there are two input files. Each barrier takes longer time to complete when the number of nodes increases. As a result, the ratio of gain for this part in MM is less than that in SOR. In figure 4(b); the computation time in PIO is better than that in NPIO, but only to a little extent. This is because MM is a massive computational program. The network message processing time is comparatively much less than the total matrix multiplication time. Figure 4(c) shows that our parallel write again significantly shortens the completion time, with reason similar to SOR. Figure 4(d) is the combination of figure 4(a) to 4(c). There is performance gain in PIO, but the ratio of gain is less than that in SOR, due to massive computation in MM.

5. Conclusions

In this paper, we have described the design and implementation of our parallel file subsystem on TreadMarks, a user-level page-based DSM system. Our design employs a variable data distribution scheme to assign file blocks to disks according the data access pattern of an application. Our experiment shows that this parallel file subsystem is feasible since the total execution times of the two

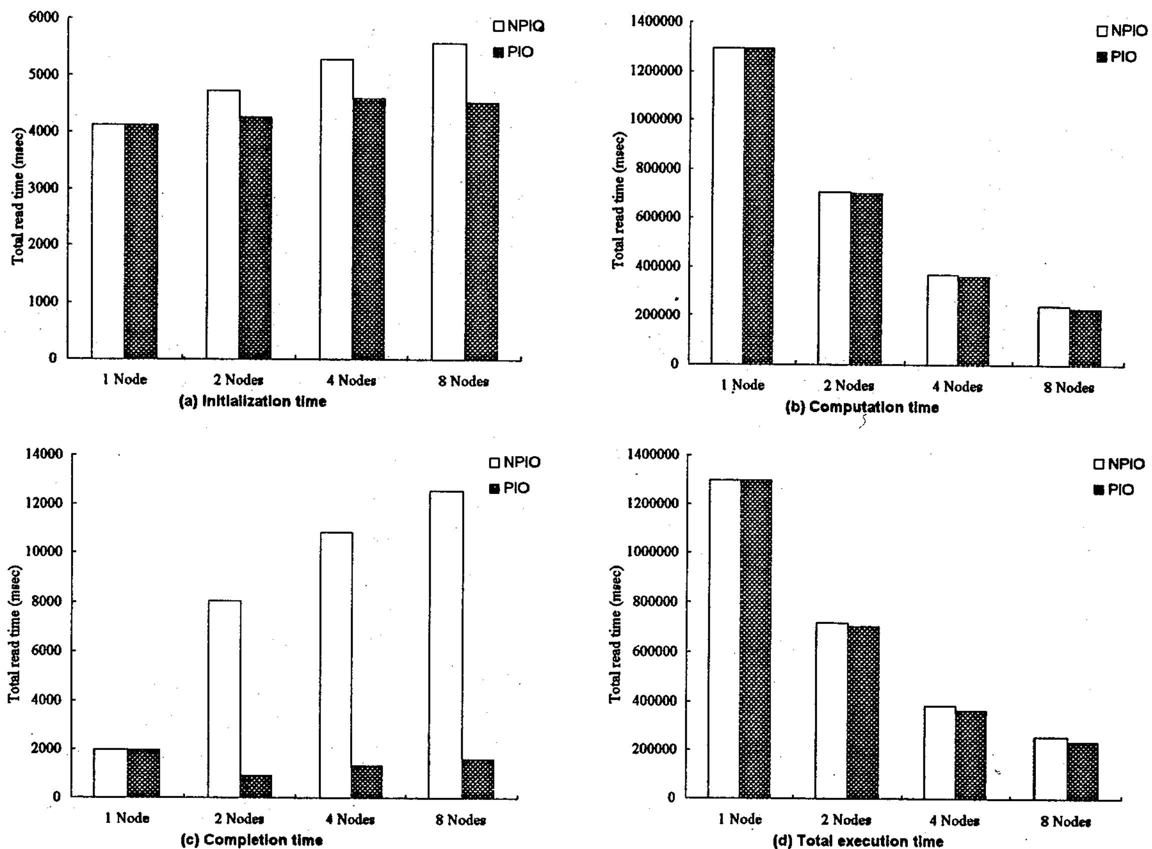

Figure 4. The performance of MM

applications are greatly reduced. In our experiment, the disks have a much faster speed than network. By parallelize the file accesses, surely there will be performance gain and this is shown in the initialization time. The strength of our subsystem is in network traffic reduction in completion phase, thereby significantly improving the performance of DSM systems, as shown in the computation and completion time.

Reference

[1] Kai Li. Shared Virtual Memory on Loosely Coupled Multiprocessors. Ph.D. Thesis, Yale University, TR YALEU-RR-492, September 1986.

[2] P. Keleher, A.L. Cox, S. Dwarkadas, et al. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In Proc. of the 1994 Winter USENIX Conference, pages 115-131, January 1994.

[3] John B. Carter. Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency. Ph.D. thesis, Rice University, September 1993.

[4] C.K. Shieh, A.C. Lai, J.C. Ueng, et al. Cohesion: An Efficient Distributed Shared Memory System Supporting Multiple Memory Consistency Models. In Proc. of Aizu International Symposium on Parallel Algorithms/Architecture Synthesis, pages 146-152, February 1995.

[5] Peter Keleher. Lazy Release Consistency for Distributed Shared Memory. Ph.D. dissertation, Rice University, January 1995.

[6] Rajesh R. Bordawekar. Issues in Software Support for Parallel I/O. Master dissertation, Syracuse University, April 1993.

[7] P.F. Corbett, D.G. Feitelson, J.P. Prost, et al. Parallel Access to Files in the Vesta File System. In Proc. of Supercomputing '93, pages 472-481, November 1993.

[8] S.A. Moyer and V.S. Sunderam. Parallel I/O as a Parallel Application. Technical Report CSTR-941101 Emory University, November 1994.

[9] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, et al. File-Access Characteristics of Parallel Scientific Workloads. Technical report PCS-TR95-263, Dartmouth College, August 1995.