JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 15, 569-583 (1999)

Reducing File-related Network Traffic in TreadMarks via Parallel File Input/Output

CE-KUEN SHIEH+, SU-CHEONG MAC++ AND BOR-JYH SHIEH
Department of Electrical Engineering
National Cheng Kung University
Tainan, Taiwan 701, R.O.C.
+*E-mail: shieh@ee.ncku.edu.tw*
++*E-mail: msc@iii.org.tw*

In this paper, we describe the implementation of a parallel file I/O system on TreadMarks, a page-based software Distributed Shared Memory (DSM) system built on a network of workstations. The main goal of our parallel file I/O system is to reduce file-related network traffic in TreadMarks. This prototype employs our previously proposed variable data distribution scheme, which distributes the file blocks among the nodes according to the application's access pattern, and delayed file access mechanism, which delays the transfer of a requested file block across the network until the block is actually used during computation. Currently, our parallel file I/O system is combined into the user-level library of TreadMarks, with minor modification of TreadMarks' code. Due to our UNIX-like interface, the existing TreadMarks programs require very little modifications. The performance improvement of our prototype on Successive Over Relaxation is quite satisfactory while that on Matrix Multiplication is less significant.

Keywords: distributed shared memory, parallel file I/O, network of workstations, variable data distribution scheme, delayed file access mechanism

1. INTRODUCTION

Distributed Shared Memory (DSM) systems [1-4] provide a shared memory abstraction on a loosely coupled multiprocessor or a network of workstations (NOW). Data are exchanged implicitly via the network between the processors to generate a shared memory abstraction for use by programmers. Therefore, network speed and the amount of network traffic become very important performance issues in a DSM system. Most of the existing DSM systems focus on relaxing memory consistency models to reduce network traffic [2-4], and other research topics include load balancing [5, 6], multithreading [7-10], and fault-tolerance [14]. However, file accesses are usually left sequentially processed by a node for ease of programming, and the overall performance of DSM systems suffers in two ways from this sequential file I/O: (1) high communication overhead due to the large amount of network traffic generated for file data transfer between the I/O node and the other nodes during computation, as shown in our analysis [11]; (2) accumulation of file data at a single node,which may become a bottleneck.

Received February 19, 1998; revised June 2, 1998; accepted July 7, 1998.
Communicated by Wei-Pang Yang.

Although many parallel file systems are available, most of them are designed for distributed/shared memory multiprocessors [12, 13, 15]. In these systems, file data are usually distributed among the nodes in a fixed scheme without knowledge of locality, resulting in a large amount of network traffic. The impact of the lack of locality information on system performance is small since the interconnections in the multiprocessors are fast. However, workstations in a DSM system are usually connected by a slower network, such as Ethernet, and heavy network traffic will induce high communication overhead. Therefore, file data locality has become an important issue in designing a parallel file I/O system for DSM systems.

We have performed an analysis on the performance impact of several file data distribution schemes on DSM systems [11]. The analysis shows that serialization of file accesses on a single node seriously degrades system performance. Moreover, simply interleaving file blocks among the nodes does not yield much improvement in performance. Based on these results, we have designed a parallel file I/O system for page-based software DSM systems. In order to reduce network traffic, we have proposed the variable data distribution scheme and the delayed file access mechanism. The variable data distribution scheme detects the file data access pattern of an application and redistributes the file blocks accordingly. The delayed file access mechanism, formerly called the ROB file access mechanism, delays transfer of a file block across the network until the block is actually used during computation. Our parallel file I/O system provides a sequential-like user interface, so that existing DSM applications require very little modification.

In this paper, a prototype of our parallel file I/O system for TreadMarks [3], a user-level page-based DSM system supporting the lazy release consistency model, will be presented. Since TreadMarks is an user-level library, it is natural to combine our parallel file I/O system with this library, which can be easily linked to TreadMarks programs. The variable data distribution scheme is achieved by monitoring the page fault information of an application during its pre-run execution. The delayed file access mechanism is achieved by means of the memory mapping function of the virtual shared address space provided by TreadMarks. In our implementation, only minor modification of TreadMarks' code is required to support our parallel file I/O system. The performance result of our prototype is quite satisfactory for Successive Over Relaxation (SOR): the total execution time of a 2000*1000 20-iteration SOR program on 8 nodes is reduced from 95 seconds with sequential file I/O to 41 seconds with parallel file I/O. For Matrix Multiplication (MM), the improvement is less significant: the total execution time of a 1024*1024 MM program on 8 nodes is reduced from 258 seconds with sequential file I/O to 236 seconds with parallel file I/O. The massive computation in MM masks the improvement in performance gained by reducing file-related network traffic. Lastly, there is moderate improvement in Raytracing (RT): the total execution time of a 16-balls RT program on 8 nodes is reduced from 71 seconds with sequential file I/O to 53 seconds with parallel file I/O.

The structure of this paper is as follows. Related works are described in section 2. Section 3 gives a brief introduction to TreadMarks. An overview of our parallel file system is presented in section 4. The scheme and mechanism of our parallel file I/O system and their implementation are presented in section 5. In section 6, we describe the modifications we have made in TreadMarks and our user interface. The performance evaluation of our prototype is given in section 7. We conclude in section 8.

2. RELATED WORKS

Munin [2] is a page-based software DSM system built on a network of workstations. Munin employs eager release consistency to eliminate the problem of false sharing, resulting in better performance. However, the overall performance is degraded by the sequentially executed initialization and completion phases of applications. In our design, we parallelize the time-consuming file operations in these two phases and reduce the file-related network traffic in order to improve the overall performance of the DSM applications.

Cohesion [4] is a page-based software DSM system developed by our research group. It is built on a cluster of workstations connected by Ethernet and supports per-node multithreading [8] and multiple consistency models, namely, sequential and eager release consistency. Cohesion is the platform of the first prototype of our parallel file I/O system [16]. The results show that our design is quite efficient for some DSM applications, and that the implementation requires little modification of the DSM system.

Adapt [17] is a subsystem of the Distributed Filaments package [7], which is another DSM system. Adapt tries to minimize the overall completion time of an iterative parallel program by determining the best data placement automatically and dynamically. Adapt monitors the computation and communication time of an iteration, and finds a new data placement if the current one is not optimal. On the other hand, our system seeks the file block placement that reduces the file-related network traffic. In Adapt, there is the overhead of monitoring in every execution of an application. In our system, the overhead of information collection occurs only in the pre-run execution of the application.

PASSION [13] is a parallel I/O system built on Intel Paragon, Touchstone Delta, iPSC/860, and IBM SP2. In the Local Placement Model, which is one of PASSION’s array storing and accessing models, data are redistributed from the I/O nodes and to the local disks of the nodes according to the user-provided distribution pattern in every execution of an application. In our system, redistribution of data is done automatically by the I/O system in the pre-run execution. Users are not required to provide distribution information, and data redistribution is unnecessary in the subsequent executions.

Panda [18] is a parallel I/O library designed for distributed memory architectures, including networks of workstations. There is no dedicated I/O node on a network of workstations. During execution, some compute nodes become I/O nodes at I/O time and return to computation after finishing the I/O operation. Panda minimizes remote data transfer by choosing the optimal number of I/O nodes and optimal nodes, which act as part-time I/O nodes. In our system, every node acts as a part-time I/O node. Remote data transfer is minimized by distributing the data according to the application's access pattern.

The Hurricane File System (HFS) [15] is a file system developed for large-scale shared memory multiprocessors with distributed disks. HFS provides a number of composite Physical Server Objects (PSO) that determine how data are distributed among disks, such as striped-data PSO, replicated-data PSO, application-specific-distribution PSO, etc. Programmers are allowed to choose an appropriate PSO or to combine these PSOs into a new PSO for a file on file creation. In our system, there is no pre-defined distribution object or distribution pattern. The data access pattern of an application is automatically determined in the pre-run execution of the application.

3. TREADMARKS

TreadMarks [3] is a user-level page-based software DSM system built on a network of UNIX workstations. Its developers at Rice University extended the concept of release consistency to propose lazy release consistency and employed an invalidate protocol. Process creation and destruction, synchronization, and shared memory allocation are provided in the TreadMarks application programming interface. However, parallel file I/O services are not included. User programs access files with UNIX file system calls via the root node, the node on which the TreadMarks system and application are started. In the initialization phase, the root node reads the input file from its local disk to the virtual shared address space. All nodes then start computation when the initialization phase is completed. When the computation phase is completed, the root node writes the result data to the disk in the completion phase.

Basically, TreadMarks employs two UNIX signal mechanisms, SIGIO and SIGSEGV. The reliable communication functions in TreadMarks are based on UDP when Ethernet is used. The SIGIO signal handler is responsible for requests and responses from other nodes via the network. Appropriate functions are called according to incoming message types. On the other hand, TreadMarks handles page faults via its SIGSEGV handler and uses the *mprotect* system call to control accesses to shared pages. Whenever a page fault occurs, a SIGSEGV signal is generated. The SIGSEGV handler is then invoked, which examines the nature of the page fault and takes appropriate actions.

4. VIEWS OF PARALLEL FILES IN OUR SYSTEM

Our parallel file I/O system is integrated into TreadMarks running on a cluster of workstations connected by Ethernet. Every workstation has a disk and acts as a part-time I/O node. The system views a parallel file as a collection of file partitions. A parallel file is partitioned into file blocks, which are grouped into disjoint file portions according to the file data access pattern of the application. The file portions are then assigned to nodes, so that each node has a disjoint portion of the parallel file. There is a metadata header for each portion, which records the global location of each file block in a file portion. On the other hand, a user views a parallel file as a continuous stream of data. The partitions and the metadata headers are not known by the user. The system and user views of our system and the data structure of the metadata header are shown in Fig. 1.

Before a file is partitioned, it is placed on the root node (node 0). In the pre-run execution, the root node issues all file accesses, following the usual file access model utilized in TreadMarks applications. The data access pattern of the file is then collected, and the file is partitioned and distributed among all the nodes. Information collection, file partitioning and distribution are performed via our variable data distribution scheme. In subsequent executions, the root node again issues all read accesses. Our delayed file access mechanism ensures that file blocks are placed in the local memory of the consumer nodes instead of the requester node, i.e., the root node. When the computation is completed, write accesses are also issued only by the root node, but our delayed file access mechanism ensures that the data on a node are written to the node's local disk instead of the disk in the root node.

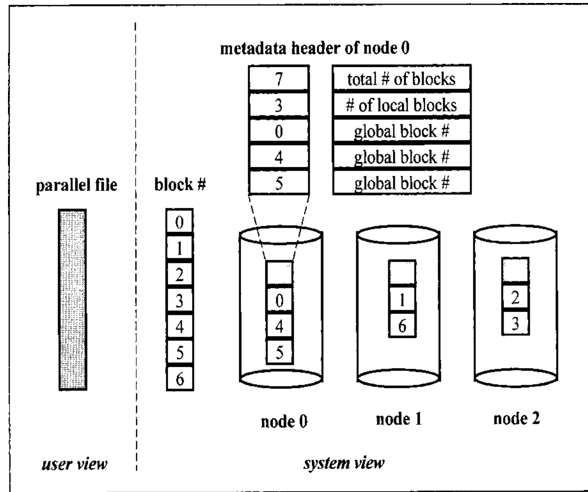


Fig. 1. An example of the parallel file and data structure of metadata.

5. VARIABLE DISTRIBUTION AND DELAYED ACCESS

The two main features of our parallel file I/O system are the variable data distribution scheme and the delayed file access mechanism. The following subsections describe this scheme and mechanism, and their implementation in TreadMarks. In our design, the size of a file block is equal to that of a memory page of the underlying operating system.

5.1 Variable Data Distribution Scheme

The main purpose of our variable data distribution scheme is to distribute file blocks among all nodes according to the application's file data access pattern, thereby reducing network traffic. Fixed distribution schemes, such as round-robin, are easy to implement, but a large amount of network traffic will be generated if the distribution scheme does not match the application's file data access pattern. In our design, the file data access pattern in an application is recorded by means of an information collection technique in the pre-run execution. Using these information, we redistribute the file blocks among the nodes so that a file block will be stored in the disk of a node only if that node needs this file block during computation. In subsequent executions, file blocks are accessed via our delayed file access mechanism, which will be described in the next subsection.

Information collection is achieved in TreadMarks by monitoring page faults. In the pre-run execution of an application, the whole input file is placed in the root node. During execution, the root node reads the file into its local memory pages and maps these pages to their corresponding virtual shared address spaces. When a node accesses a memory page that is not present in the local memory, a page fault is generated, and a request is sent to the current owner node of the faulty page. The owner then sends a copy of the faulty page to the requester so that the requester node can resume execution. By monitoring the page faults of each node, we can determine the set of memory pages containing the file data that

have been accessed by a node, i.e., the file data access pattern of the application. Redistribution of file blocks is done according to this information. Since a file block may be accessed by two or more nodes and file block replication is not currently supported by our design, we define two rules to evenly distribute the file blocks across nodes: (1) If a block is exclusively accessed by a node, it is assigned to that node. (2) If a file block has been accessed by two or more nodes, that block is assigned to the node with the smallest number of blocks. When redistribution is completed, we generate a metadata header for each file partition to keep track of the global block number of each file block, so that a file block will be located at the specified memory address when the block is read in the subsequent executions.

In the case of a file write, each node possesses some memory pages containing the resultant data in local memory. Each node sends a list of local memory pages containing the resultant data to the root node. Using this page information, the root node determines for each node the set of memory pages that should be written to the local disk of a node; each node writes memory pages to local disk according to the decision and generates the corresponding metadata header. It should be noted that the amount of network traffic in a TreadMarks application is not reduced by our variable data distribution scheme in the application's pre-run execution. However, in subsequent executions, network traffic is reduced since most of the file data required by a node can be found in the local disk.

5.2 Delayed File Access Mechanism

Our delayed file access mechanism is specifically designed for our variable data distribution scheme. This mechanism delays the transmission of file data across the network until the data are actually used in computation, thus avoiding unnecessary network traffic. In our parallel file I/O system, we preserve the file access model described in section 3; i.e., all file accesses are issued by the root node in the initialization phase. Using the ordinary file access mechanism, the file data are immediately sent to the requester node when the file is read. However, the file data are not used until the computation phase is started, and the consumer node of a file block may not be the root node. Therefore, sending the file data immediately via the network to the root node during initialization may induce unnecessary network traffic. With the delayed file access mechanism, the file data are not sent across the network to the requester node (always the root node) when a read request is issued. Instead, the owner node of a file partition puts the file data in virtual shared memory. When the data is actually required by its consumer node during computation, the related memory pages are fetched from the owner nodes across the network. File write is done in a similar way. A memory page containing the file data is written to the disk of the owner node of that page, instead of sending the page to the requester node.

We exploit the feature of virtual shared memory address space in TreadMarks to obtain the delayed file access mechanism. When a file read operation is issued by the root node, a file read request, including the virtual shared memory address specified in the read operation, is broadcasted. Each node then checks its metadata header to calculate the corresponding virtual shared memory page for each file block in the local parallel file partition, and the file block is read from the disk and copied to the calculated memory page. When a node has put all the local file blocks in their calculated memory pages, it sends a listing of these pages to the root node. The root node collects these listings and then broadcasts the node locations of all the memory pages containing the file blocks so that other nodes can

locate these pages and ask for their replicas via the network when page faults occur. In this way, the transfer of file data across the network is delayed until the data are actually used in computation. The amount of file data transfer across the network in our delayed mechanism is less than that in ordinary file access mechanisms. However, there is an additional broadcast message in the delayed mechanism, but this is sent only once per parallel file.

File write is handled by a similar mechanism; i.e., the node that actually writes the content of a memory page to its local disk is the one that owns the memory page instead of the requester. When a file write operation is issued at the root node, a message containing the specified memory region is broadcasted. Every node returns a list of local memory pages that are within the specified memory region. The root node then decides the writer node of each page and sends its decision to the corresponding node. Eventually, every node writes the selected local memory pages to the local disk, and a metadata header is generated.

5.3 Overheads

Our adaptive distribution scheme may incur a certain overhead. First, the metadata file will surely induce extra disk access cost. However, the size of the metadata file was always less in our experiment than that of a disk block, which is much smaller than its corresponding parallel file. Actually, the size of the metadata header is approximately 1/2048 of the file size. The overhead of accessing metadata headers is very small compared to the file access time. Second, a pre-run execution of an application is required for different data sizes and system configurations. This pre-run execution is as slow as sequential file I/O because all file data are stored on the root node. However, after the system collects the required file data access pattern information and redistributes the file blocks accordingly, the performance of the following executions of the application will be improved. Moreover, the metadata file can act as a template. It can be repeatedly used for the same types of programs with the same system configuration and program size but different data. In this case, the pre-run execution will be needed only once.

6. MODIFICATION OF TREADMARKS AND APPLICATIONS

Fig. 2 shows the system architecture of our parallel file I/O system. The information collector module collects the data access information of the programs. The file network handler module processes file-related network requests. Modifications required in the user-level library and TreadMarks user programs are described in the following subsections.

6.1 Integrating Parallel File I/O with TreadMarks

Some minor modifications, mainly code addition, are required in TreadMarks to support our parallel file I/O system. Other functions, such as parallel file operations, are just add-on functions added to the user-level library of TreadMarks.

We have modified the SIGSEGV handler in TreadMarks to support information collection in our variable distribution scheme. Some codes are added at the beginning of the page fault handler in order to record the page number of the faulty page. At the end of the

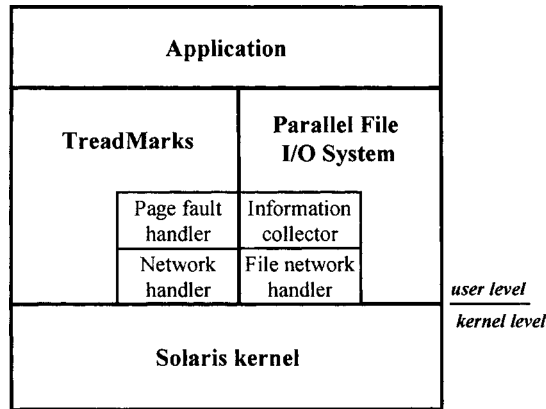


Fig. 2. System architecture of our parallel file I/O system.

pre-run execution, each node will have a list of the page numbers of the memory pages that have generated at least one page fault, i.e., the pages that have been accessed by a node. Our system can find the page numbers of the memory pages that contain file data by checking the starting address of the memory buffer specified in a parallel read request and the size of the buffer. For each memory page containing file data, the list of the nodes that have accessed that page can be found by combining the lists of faulty page numbers from all the nodes. The file blocks can then be redistributed accordingly.

The SIGIO handler in TreadMarks is also modified. We define a new message type for parallel file requests and register our parallel file request handler in this SIGIO handler so that file requests from remote nodes can be redirected to our parallel file request handler. Our network code is based on the network code in TreadMarks.

Since no knowledge of memory consistency protocols is assumed in our design, the implementation of our parallel file I/O system is independent of the memory consistency model employed by TreadMarks. Only basic functions, such as page fault handling and acquiring page table entries, are necessary.

6.2 User Interface

Our parallel file I/O system provides a UNIX-like file interface. With this interface, we can retain the usual file accessing model in TreadMarks applications; i.e., only the root node issues all file accesses. Consequently, the existing TreadMarks applications need very little modification to utilize our parallel file services. Programmers need not coordinate file accesses or provide data access information, and parallelization of file accesses is implicitly achieved by our system. Currently, there are two limitations in our interface: the whole parallel file should be accessed in a single operation, and the files are immutable.

Currently, we provide the following file functions:

```
int pfd = PIO_open(char *pathname, int flag);
int PIO_read(int pfd, char *buff);
int PIO_write(int pfd, char *buff, int nbytes);
int PIO_close(int pfd);
```

pfd: parallel file descriptor.

PIO_open opens a parallel file for subsequent read or write accesses as indicated in *flag*. The root node broadcasts an open notification so that every node opens its local file partition and reads the metadata header in the case of read access, or creates a local file in the case of write access. *PIO_read* stores the content of a parallel file to the virtual shared address space specified in *buff* via our delayed file access mechanism. *PIO_write* stores the content of the virtual shared address space as specified in *buff* to disks again via our delayed file access mechanism. *PIO_close* closes a parallel file. The root node broadcasts a close notification so that every node will close its local file partition.

To utilize the above parallel file I/O services, a programmer simply exchanges the UNIX file functions in his original TreadMarks program with our corresponding parallel file functions; e.g., *fopen()* and *fread()* are replaced by *PIO_open()* and *PIO_read()*, respectively. After linking with our modified TreadMarks library by means of the parallel file access ability, the program is ready for execution.

Fig. 3 shows code fragments of the "no parallel file I/O" version and "parallel file I/O" version of the Successive Over Relaxation (SOR) program. As shown in this example, the two versions are nearly identical. The differences between this two versions are the replacement of UNIX file functions in the "no parallel file I/O" version with parallel file functions in the "parallel file I/O" version, and minor modifications in the file parameters.

```
main(int argc, char *argv[])
{
    Tmk_startup(argc, argv);
    if (Tmk_proc_id == 0) {
        ...
        ...allocate shared memory
        ...
        fpr = fopen(filename,"r");
        buffer = (char *) Tmk_malloc(size);
        Tmk_distribute(&buffer, sizeof(buffer));
        read_ret = fread(buffer, size, 1, fpr);
        Tmk_distribute((char *)&red_, sizeof(red_));
        Tmk_distribute((char *)&blk_, sizeof(blk_));
        ...
        ...continue initialization
        ...
    }
    ...
    ...computation
    ...
    if (Tmk_proc_id == 0)
    {
        fpw = fopen("/export/h3/pfs/sorw.dat", "w");
        for (i = 0; i < M+2 ;i++)
            j = fwrite(red_[i], 4096, 1, fpw);
    }
    fclose(pfr);
    fclose(fpw);
    Tmk_exit(0);
}
```

SOR main function without parallel file I/O

```
main(int argc, *argv[])
{
    Tmk_startup(argc, argv);
    if (Tmk_proc_id == 0) {
        ...
        ...allocate shared memory
        ...
        pfr = PIO_open(filename, "r");
        buffer = (char *) Tmk_malloc(size);
        Tmk_distribute(&buffer, sizeof(buffer));
        read_ret = PIO_read(pfr, buffer);
        Tmk_distribute((char *)&red_, sizeof(red_));
        Tmk_distribute((char *)&blk_, sizeof(blk_));
        ...
        ...continue initialization
        ...
    }
    ...
    ...computation
    ...
    if (Tmk_proc_id == 0)
    {
        fpw =PIO_open("/export/h3/pfs/nod0.dat", "w");
        PIO_write(fpw, red_, M*1024*4);
    }
    PIO_close(pfr);
    PIO_close(fpw);
    Tmk_exit(0);
}
```

SOR main function with parallel file I/O

Fig. 3. SOR code fragments with and without parallel file I/O.

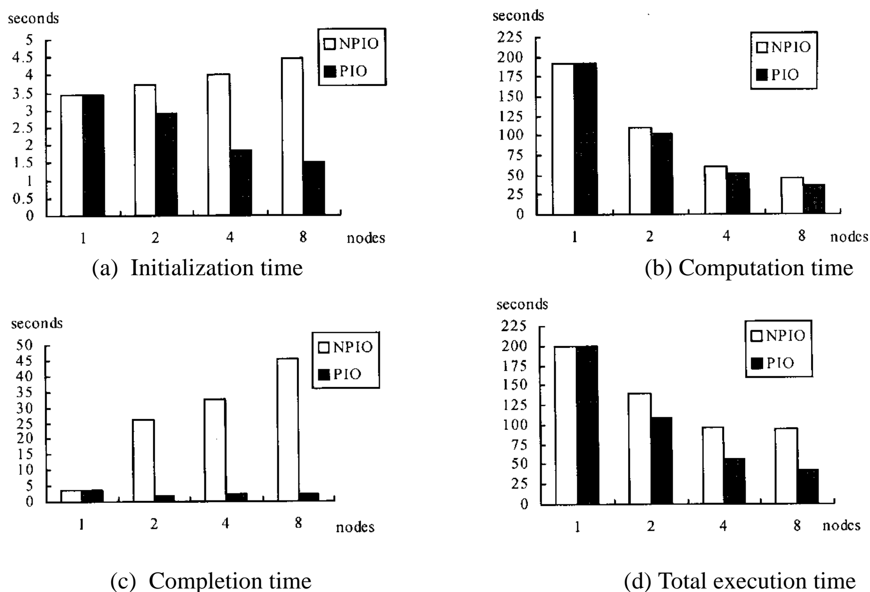
7. PERFORMANCE EVALUATION

We chose Successive Over Relaxation (SOR), Matrix Multiplication (MM), and Raytracing (RT) to test the effectiveness of our parallel file subsystem compared to sequential file operations. These two programs are the usual DSM benchmarks and possess relatively high I/O-computation ratios compared to other DSM benchmarks. For each application, we 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 required to complete the initialization phase. The computation time is the time which elapses during the computation phase. The completion time is the time needed to write the resultant data to disk. The total execution time is the summation of the previous three periods of time.

Our testing environment consisted of 8 Pentium-90 PC, each with 32MB RAM and a Seagate ST5850A harddisk (11ms average seek time). The computers were connected by 10Mbps Ethernet. The operating system on each machine was Solaris for x86. The size of a memory page was 4096 bytes.

7.1 Successive Over Relaxation

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 was 2000x1000, each element was a floating point number, and there were 20 iterations. The performance results are shown in Fig. 4.



(a) Initialization time

(b) Computation time

(c) Completion time

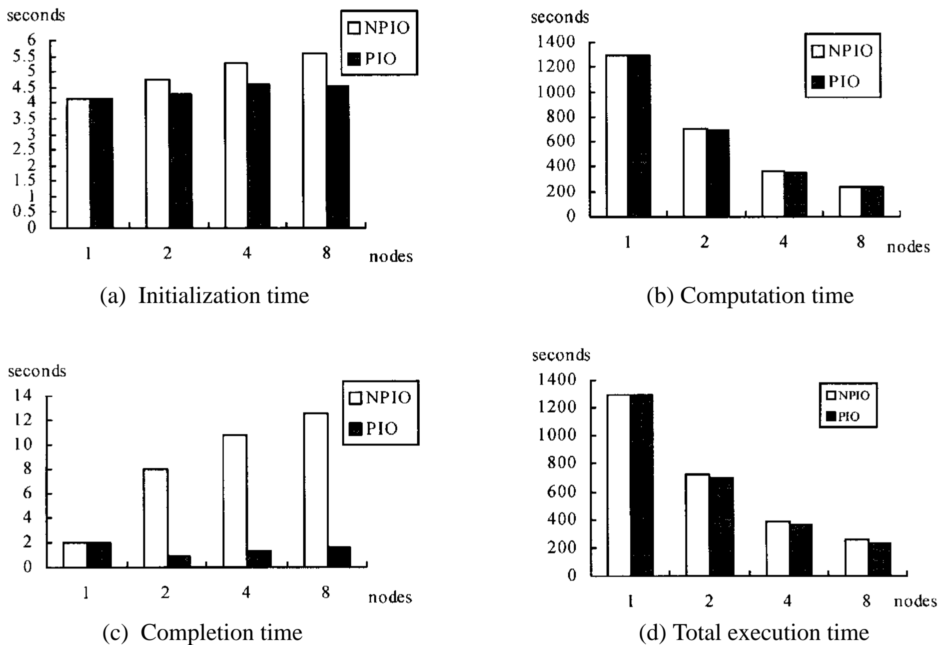
(d) Total execution time

Fig. 4. The performance of SOR.

The initialization phase consists of a file open, a file read and a barrier. Fig. 4(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. As shown in Fig. 4(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 a reduction of network traffic. As shown in Fig. 4(c), the reduction of completion time in PIO is impressive compared to NPIO. In NPIO, before the specific memory region is written, the root node collects all the update information of the memory pages in that region from all the nodes and updates those memory pages, which is a time consuming job. In PIO, every node writes its own valid pages to disk. No updates from other nodes are required. Fig. 4(d) shows the combined effect of the previous three graphs. The total execution time of SOR is significantly reduced in PIO compared to NPIO.

7.2 Matrix Multiplication

The input and output files of MM are three matrices. In our experiment, the size of a matrix was 1024x1024, and each element was an integer. The performance results are shown in Fig. 5.



(a) Initialization time

(b) Computation time

(c) Completion time

(d) Total execution time

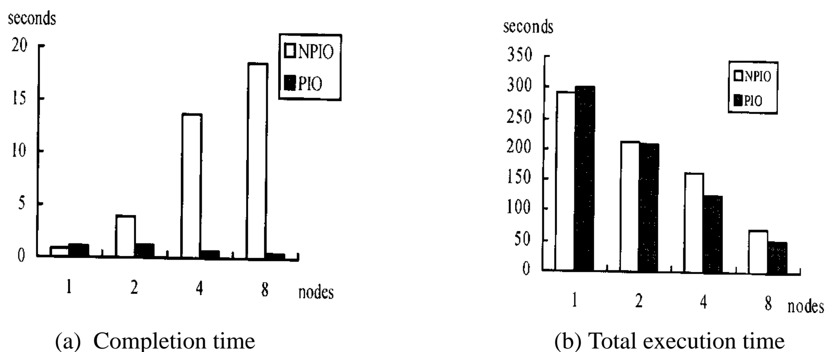
Fig. 5. The performance of MM.

As shown in Fig. 5(a), the initialization time in PIO was only slightly shorter than that in NPIO. This is due to fact that the number of barriers in MM is twice that in SOR since there are two input files. Each barrier takes a longer time to complete when the number of nodes increases. In addition, the multiplier matrix is required by every node. This will induce a large amount of network traffic even after the file is parallelized. Consequently, the ratio of the performance gain for this part in MM is less than that in SOR. As shown in Fig. 5(b), the computation time in PIO is better than that in NPIO, but only by a little amount. This is because MM is a massive computational program. The network message processing time is comparatively much less than the total computation time. Fig. 5(c) shows that our parallel write again significantly shortens the completion time, for reasons similar for the case of SOR. Fig. 5(d) combines the results in Figs. 5(a) to 5(c). There is a performance gain in PIO, but the ratio of gain is less than that in SOR, due to massive computation in MM.

7.3 Raytracing

The input data of RT, which consisted of 16 balls, were less than one block in size and too small to be parallelized. Therefore, the input data were processed sequentially. On the other hand, the output image was parallelized. The size of the output image was 500x350, and each pixel consisted of 4 floating point numbers. The depth of tracing was 5. The performance results are shown in Fig. 6.

Since the input data file were sequentially processed, the initialization time in PIO was the same as that in NPIO. The same situation also occurred with the computation time. For this experiment, we only show the completion time and total execution time of Raytracing. Fig. 6(a) shows the completion time of Raytracing. The performance gain is quite significant in this case. This is because at the end of the computation, the pixels of the output image were distributed among all all nodes. With NPIO, the root node has to collect all the pixels from the other nodes in the completion phase while this is not necessary with PIO. Fig. 6(b) shows that the total execution time of Raytracing in PIO was better than that in NPIO.



(a) Completion time (b) Total execution time

Fig. 6. The performance of RT.

8. CONCLUSIONS

In this paper, we have described the implementation of our parallel file I/O system on TreadMarks, a user-level page-based software DSM system. The prototype shows that only a few modifications are required in TreadMarks. Our system was implemented as several add-on functions in TreadMarks' user-level library, which can be easily linked to user programs. In addition, our user-interface is similar to the UNIX file interface used in TreadMarks applications, so that existing TreadMarks applications need very little modification to utilize our parallel file services.

In the experiments, our system could significantly reduce the total execution time of SOR. Since only the output file in RT is parallelized, the improvement in RT with parallel file I/O is less than that in SOR. There is little improvement in MM with parallel file I/O due to the long computation time compared to the network and I/O times. These results also show that our parallel file I/O system can improve the performance of DSM programs, such as SOR and RT, with little data sharing. The false data sharing induced by sequential file I/O in these programs can be avoided using our system. However, the improvement gained by our system is less noticeable for programs with heavy data sharing, such as the multiplier matrix in MM.

REFERENCES

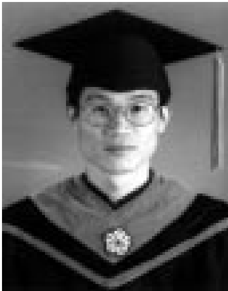
1. Kai Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. Thesis, TR YALEU-RR-492, Department of Computer Science, Yale University, 1986.
2. J. B. Carter, *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*, Ph.D. Thesis, Department of Computer Science, Rice University, 1993.
3. P. Keleher, A. L. Cox, S. Dwarkadas, et al., "TreadMarks: distributed shared memory on standard workstations and operating systems," in *Proceedings of the 1994 Winter USENIX Conference*, 1994, pp. 115-131.
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 *Proceedings of Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, 1995, pp. 146-152.
5. T. Y. Liang, C. K. Shieh and W. P. Zhu, "Task mapping on distributed shared memory systems using hopfield neural network," in *Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference, Western Multi-Conference '97*, 1997, pp. 37-43.
6. A. C. Lai, C. K. Shieh and Y. T. Kok, "Load balancing in distributed shared memory systems," in *Proceedings of the 1997 IEEE International Performance, Computing, and Communications Conference*, 1997, pp. 152-158.
7. D. K. Lowenthal, V. W. Freeh and G. R. Andrews, "Using fine-grain threads and run-time decision making in parallel computing," Technical Report, TR 95-14, Department of Computer Science, University of Arizona, 1995.
8. C. K. Shieh, J. C. Ueng, S. C. Mac, et al., "Multi-threaded design for a distributed shared memory system," in *Proceedings of International Conference on Distributed Systems, Software Engineering, and Database Systems, International Computer*

Symposium, 1996, pp. 248-255.

9. A. Itzkovitz, A. Schuster and L. Wolfovich, "Thread migration and its applications in distribution shared memory systems," Technical Report, LPCR #9603, Department of Computer Science, Technion - Israel Institute of Technology, 1996.
10. K. Thitikamol and P. Keleher, "Per-node multithreading and remote latency," *IEEE Transactions on Computers*, Vol. 47, No. 4, 1998, pp. 414-426.
11. S. C. Mac, C. K. Shieh and J. B. Chang, "Design and analysis of a parallel file system for distributed shared memory systems," *Journal of Systems Architecture*, to appear.
12. P. F. Corbett, D. G. Feitelson, J. P. Prost, et al., "Parallel access to files in the vesta file system," in *Proceedings of Supercomputing '93*, 1993, pp. 472-481.
13. R. Bordawekar, A. Choudhary and R. Thakur, "Data access reorganizations in compiling out-of-core data parallel programs on distributed memory machines," Technical Report, SCCS-622, Northeast Parallel Architectures Center, Syracuse University, 1994.
14. C. Morin and I. Puaut, "A survey of recoverable distributed shared virtual memory systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 9, 1997, pp. 959-969.
15. O. Krieger, "HFS: A flexible file system for shared-memory multiprocessors", Ph.D. Dissertation, Computer Science Research Institute, University of Toronto, 1994.
16. C. K. Shieh, S. C. Mac and J. C. Ueng, "Improving the performance of distributed shared memory systems via parallel file input/output," *Journal of Systems and Software*, to appear.
17. D. K. Lowenthal and G. R. Andrews, "Adaptive data placement for distributed-memory machines," Technical Report, TR 95-13, Department of Computer Science, University of Arizona, 1995.
18. Y. Cho, M. Winslett and M. Subramaniam, "Parallel array I/O on a practical network of workstations," Submitted for publication, Department of Computer Science, University of Illinois.
19. S. A. Moyer and V. S. Sunderam, "Parallel I/O as a parallel application," Technical Report, CSTR-941101, Department of Computer Science, Emory University, 1994.

Ce-Kuen Shieh (Âü úå) is currently an associate professor in the Department of Electrical Engineering, National Cheng Kung University. He received his Ph.D., M.S., and B.S. degrees from National Cheng Kung University, all in electrical engineering. His current research interests include distributed/parallel processing, operating systems, computer networking, and compiler.

Su-Cheong Mac (³Á³¾µ³¾) received his B.S., M.S., and Ph. D. degrees from National Cheng Kung University in 1990, 1992, and 1998, respectively. He is currently a senior engineer in Institute for Information Industry. His current research interests include embedded systems, real time operating systems, and file systems.



Bor-Jyh Shieh (ÁÂ³Õ¹¾) received his M.S. degree from National Cheng Kung University in 1996. His research interests focus on Distributed Shared Memory and parallel I/O.