PART: A Run-Time Library for Tape-Resident Data

Gokhan Memik, Mahmut Taylan Kandemir, Alok Choudhary, and Valerie E. Taylor

Technical Report No. CPDC-TR-9909-014

September 1999

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

PART: A Run-Time Library for Tape-Resident Data

Gokhan Memik, Mahmut Taylan Kandemir, Alok Choudhary, and Valerie E. Taylor
Center for Parallel and Distributed Computing
Northwestern University, Evanston, IL 60202
{memik,mtk,choudhar,taylor}@ece.nwu.edu

Abstract

Over the last decade, processors have made enormous gains in speed. But increase in the speed of the secondary and tertiary storage devices could not cope with these gains. The result is that the secondary and tertiary storage access times dominate execution time of data intensive computations. Therefore, in scientific computations, efficient data access functionality for data stored in secondary and tertiary storage is a must. In this extended abstract, we give an overview of PART, a runtime library that can be used in applications that process tape-resident data. We present user interface and underlying optimization strategy. We also discuss performance improvements provided by the library on the High Performance Storage System (HPSS). The preliminary results reveal that the optimizations can improve response times by up to 97.2%.

1 Introduction

We address the problem of managing the movement of very large data sets between different levels of a hierarchical storage system. It is now widely acknowledged that the data set sizes manipulated by scientific codes are getting larger as programmers have access to faster processors and larger main memories. The data sets whose sizes exceed main memories should be stored in secondary and tertiary storages. Although the prices for secondary storage devices are decreasing, tertiary storage devices are becoming increasingly attractive especially for applications that require vast amount of storage capacity which cannot be satisfied by secondary storage devices and for applications which cannot afford the cost or system complexity of a large number of disk drives. There has been a considerable amount of work in addressing the flow of data to and from secondary storage devices (e.g., magnetic disks) [3, 4, 5, 10, 11, 14, 15, 16, 2]. There has also been a significant amount of work on the management of large scale data in a storage hierarchy involving tertiary storage devices (e.g., tapes devices) [19, 12, 17, 18, 7]. Striping has been studied to improve the response time of tertiary storage devices [13, 6].

The Department of Energy's ASCI plan draws an outline of the expected storage requirements for large-scale computational challenges. According to this plan, a large scientific application today is producing $3 - 30$ terabytes of simulation datasets for a run, requiring 3 petabytes of archive capacity. These sizes are excepted to grow to $100 - 1000$ terabytes per run and to 100 petabytes of archive capacity in the year 2004. Even with the assumptions of aggressive improvements in the evolution of storage devices, the data accesses in these applications will take a significant proportion of the overall execution time [20]. On top of this, aggregate data sizes may require the employment of tertiary storage devices. Many of these applications do not demand the entire datasets to be accessed at a given time. So, having means to access portions of the tape-resident datasets efficiently may decrease the time spent in data accesses significantly.

In this abstract we present PART, a run-time library, that can be used to facilitate the explicit control of data flow for tape-resident data. Our library can be used by application programmers as well as optimizing compilers that manipulate large scale data. The objective is to allow programmers to access data located on tape via a convenient interface expressed in terms of arrays and array portions (regions) rather than files and offsets. In this sense the library can be considered as a natural extension of state-of-the-art run-time libraries that manipulate disk-resident datasets (e.g., [4, 22]). The library implements a data storage model on tapes that enables users to access portions of multi-dimensional data in a fast and simple way. In order to eliminate most of the latency in accessing tape-resident data, we employ a *sub-filing strategy* in which a large multi-dimensional tape-resident *global* array is stored not as a single file but as a number of smaller *sub-files*, whose existence is transparent to the programmer. The main advantage of doing this is that the data requests for relatively small portions of the global array can be satisfied without transferring the entire global array from tape to disk as is customary in many hierarchical storage management systems. In addition to read/write access routines, the library also supports pre-staging and migration capabilities which can prove very useful in environments where the data access patterns are predictable and the amount of disk space is limited.

The main contributions of this abstract are as follows:

- The presentation of a high-level parallel I/O library for tape-resident data. The PART library provides a simple interface to the tape-resident data, which relieves the programmers from orchestrating I/O from tertiary storage devices such as robotic tapes and optical disks.
- The description of the implementation of the library using HPSS [8] and MPI-IO [5]. We show that it is both

simple and elegant to build an I/O library for tape-resident data on top of these two state-of-the-art systems. In this abstract, however, we focus on a single processor performance.

• The presentation of preliminary performance numbers using representative array regions and sub-file sizes. The results demonstrate that the library is quite effective in exploiting the secondary storage – tertiary storage hierarchy without undue programmer effort.

Section 2 gives an overview of the PART library. Section 3 describes sub-filing and its use in the library. Section 4 briefly explains the implementation and Section 5 presents the user interface. Section 6 gives preliminary experimental results and Section 7 concludes the abstract with a summary and an outline of future work.

2 Library Overview

The library provides routines to efficiently perform I/O required in sequential and parallel applications. It can be used for both in-core and out-of-core applications. It uses a high-level interface which can be used by application programmers and compilers. For example, an application programmer can specify what section of an array she wants to read in terms of lower and upper bounds in each dimension, and the library will fetch it in an efficient manner, first from tape to disk and then from disk to main memory. It provides a portable interface on top of HPSS [8] and MPI-IO [5]. It can also be used by an optimizing compiler that targets programs whose data sets require transfers between secondary storage and tertiary storage. It might even be possible to employ the library within a database management system for multi-dimensional data.

At the heart of the library is an optimization technique called *sub-filing*, which is explained in greater detail in the next section. It also uses collective I/O using a two-phase method, data pre-staging, pre-fetching, and data migration. The main advantage of sub-filing is that it provides low-overhead random access image for the tape-resident data. Sub-filing is invisible to the user and helps to efficiently manage the storage hierarchy which can consist of a tape sub-system, a disk sub-system and a main memory. The main advantage of the collective I/O, on the other hand, is that it results in high-granularity data transfers between processors and disks, and it also makes use of the higher bandwidth of the processor interconnection network.

In general a processor has to wait while a requested tape-resident data set is being read from tape. The time taken by the program can be reduced if the computation and tape I/O can be overlapped somehow. The pre-staging achieves this by bringing the required data ahead of the time it will be used. It issues asynchronous read calls to the tape sub-system, which help to overlap the reading of the next data portion with the computation being performed on the current data set. The data pre-fetching is similar except that it overlaps the disk I/O time with the computation time.

3 Sub-filing

Each *global tape-resident* array is divided into *chunks,* each of which is stored in a *separate sub-file* on tape. The chunks are of equal sizes in most cases. Figure 1(a) shows a two-dimensional global array divided into 64 chunks. Each chunk is assigned a unique *chunk coordinate* (x_1, x_2), the first (upper-leftmost) chunk having $(0, 0)$ as its coordinate. For the sake of ensuing discussion we assume that the sub-files corresponding to the chunks are stored in row-major as depicted in the figure by horizontal arrows.

A typical access pattern is shown in Figure 1(b). In this access a small two-dimensional portion of the global array is requested. In receiving such a request, the library performs three important tasks:

• Determining the sub-files that collectively contain the requested portion,

• Transferring the sub-files that are *not* already on disk from tape to disk, and

• Extracting the required data items (array elements) from the relevant sub-files from disk and copying the requested portion to a buffer in memory provided by the user call.

In the first step, the set of sub-files that collectively contain the requested portion is called *cover*. In Figure 1(b), the cover contains the sub-files $(1, 2)$, $(1, 3)$, $(1, 4)$, $(2, 2)$, $(2, 3)$, $(2, 4)$, $(3, 2)$, $(3, 3)$, and $(3, 4)$. Assuming for now that all of these sub-files are currently residing on tape, in the second step, the library brings these sub-files to disk. In the third step, the required portion is extracted from each sub-file and returned to the user buffer. Note that the last step involves some computational overhead incurred for each sub-file. Instead, had we used just one file per global array this computational overhead would be incurred only once. Therefore, the performance gain obtained by dividing the global array into sub-files should be carefully weighed against the extra computational overhead incurred in extracting the requested portions from each sub-file. Our preliminary experiments show that this computational overhead is not too much.

Parallel reads by multiple processors pose additional problems. Consider now Figure 1(c) where four processors are requesting four different sub-columns of a region. The underlying cover contains 28 sub-files. After bringing these sub-files from tape to disk, we have a problem of reading the required sub-portions (sub-columns) for each processor. As stated by del Rosario et al. [9], *collective I/O* is a technique in which processors perform I/O

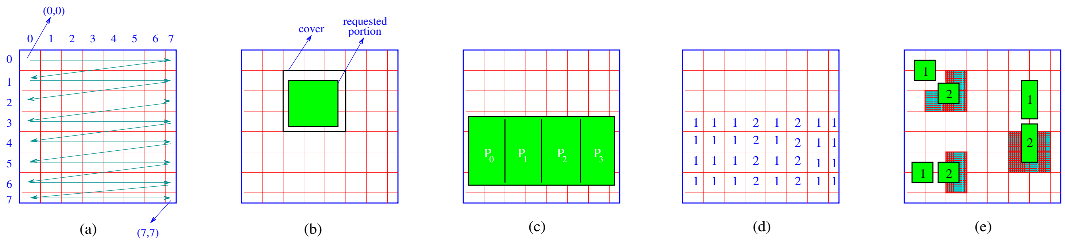

Figure 1: (a) A global tape-resident array divided into 64 chunks. (b) An access pattern (shaded portion) and its cover. (c) An access pattern involving four processors. (d) The global array with each sub-file marked with the number of processors that share it. (e) Successive array accesses.

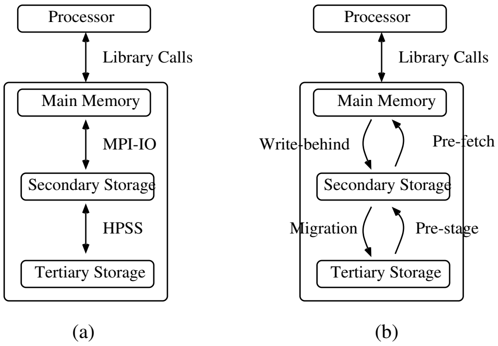

Figure 2: (a) Library architecture. (b) Pre-fetching, pre-staging, and migration.

on behalf of each other in order to reduce the time spent in disk-I/O at the expense of some extra communication. In two-phase I/O, a specific implementation of collective I/O, taking advantage of knowing the access and storage pattern of an accessed array, we can perform I/O in two phases. In the first phase, the processors access the data in a *layout conformant* way (to exploit spatial locality on disk as much as possible) and in the second phase they re-distribute the data in memory among themselves such that the desired access pattern is obtained. While it is quite straightforward how to use collective I/O when we have a single file, in our multiple file case it is not clear how to utilize it. One simple approach might be to use collective I/O for each sub-file on disk. In our example, that would mean calling a collective I/O routine 28 times. A better alternative might be to read the data from disk to memory in two steps. In the first step, the processors that have exclusive access to some sub-files perform these independent accesses. In the second step, for each of the remaining sub-files, we can perform collective I/O using only the processors that request some data from the sub-file in question. Considering Figure 1(d), this collective I/O scheme corresponds to first reading the sub-files marked '1' and then collectively (using two processors) reading the sub-files marked '2'. We plan to implement this last collective I/O strategy in the future.

During successive reads from the same global file it might happen that the same sub-file can be required by two different reads. Assuming that the sub-file in question still resides on the disk after the first read, it is unnecessary to read it again from tape in the second read. In such a case only the other (additional) sub-files required by the current access are read from tape. The situation is shown in Figure 1(e) for three scenarios. In each case, the first portion read is marked '1' and the second portion read is marked '2'. The shaded parts around the second portions correspond to additional sub-files that are needed for the second read. In other words, the library effectively uses caching [3].

4 Implementation

We are implementing the PART library on top of HPSS [8] and MPI-IO [5]. The connections between different components are shown in Figure 2(a). In a read call, the sub-files are first read from tape to disk using HPSS and then from disk to main memory using MPI-IO. As mentioned earlier, we employ collective I/O between disk and memory. In a write call the direction of data-flow is reversed. Figure 2(b) shows the corresponding storage levels for each action described in the following sections.

To store the information about the file and the chunks, we are currently using the Postgres95 database [23]. When a new file is created, the user may enter the necessary information about the chunks. The detailed information about the file creation is given in Section 5. Then this meta-data is stored in the database for later usage. When a user opens a previously created file, the corresponding meta-data about the file and the chunks are read from the database and cached in the memory. Then the following accesses uses this meta-data. The database is informed about the changes when the file is closed. In other words, the database is accessed only in file open and file close.

Table 1: Some of the library routines.

Initialization/Finalization Routines	
Routine	Functionality
T_INITIALIZE	Initializes the library structures
T_FINALIZE	Finalizes the library structures
File Manipulation Routines	
Routine	Functionality
T_OPEN	Opens a tape-resident global file for read/write
T_CLOSE	Closes a tape-resident global file
T_REMOVE	Removes both the sub-files of the file and the corresponding info.
Array Access Routines	
Routine	Functionality
T_READ_SECTION	Reads a rectilinear section
T_WRITE_SECTION	Writes a rectilinear section
Stage/Migration Routines	
Routine	Functionality
T_STAGE_SECTION	Stages a rectilinear file section from tape to disk
T_STAGE_WAIT	Waits for a Stage to complete
T_PREFETCH_SECTION	Pre-fetchs a rectilinear file section from tape (or disk) to memory
T_PREFETCH_WAIT	Waits for a Pre-fetch to complete
T_MIGRATE_SECTION	Migrates a rectilinear file section from disk to tape

5 User Interface

The routines in the library can be divided into four major groups based on their functionality – Initialization/Finalization Routines, File Manipulation Routines, Array Access Routines, and Stage/Migration Routines. Table 1 lists some of the basic routines in the library and their functionalities.

Initialization/Finalization Routines: These routines are used to initialize the library buffers and meta-data structures and finalize them when all the work is done. The routine to initialize the system has the format

int T_INITIALIZE ().

This routine initializes the connections to the HPSS and the database. It returns a positive number upon successful completion. Similarly, int T_FINALIZE () closes the above mentioned connections.

File Manipulation Routines: These routines are used for creating files, opening existing files, closing open files and removing all the chunks and the information related to a global file. T_OPEN is used for creating new files and for opening existing files. It returns a file handle for later referral to the file. The synopsis of T_OPEN is as follows:

T_FILE T_OPEN (char *filename, char *mode, T_INFO *tapeinfo).

'Filename' stands for the name of the file to be opened. 'Mode' indicates whether the file is opened for read, write, or read/write. 'Tapeinfo' is the structure used for entering the necessary information about the file and chunks. It has fields for the elementsize, number of dimensions, the size of each dimension of the chunk and the size of each dimension of the global file.

Array Access Routines: These routines handle the movement of data to and from the tape subsystem. An arbitrary rectilinear portion of a tape-resident array can be read or written using these access routines. Let us focus now on T_READ_SECTION. The signature of this routine is

int T_READ_SECTION (T_FILE *fd, void *buffer, int *start_coordinate, int *end_coordinate)

'fd' is the file descriptor returned by T_OPEN. 'Start_coordinate' and 'end_coordinate' are arrays that hold the boundary coordinates for the section to be read. There are as many elements as the dimensionality of the associated tape-resident global array. This command reads the corresponding elements and stores them in 'buffer'. As discussed earlier, what actually happens here is that the relevant sub-files are read from tape to disk (if they are not on the disk already), and the required sections are read from these sub-files on disk and forwarded to the corresponding positions in the buffer in main memory. The syntax for the T_WRITE_SECTION routine is almost the same except that the direction of the transfer is reversed.

Stage/Migration Routines: These routines are used to stage and migrate the data between the tapes and the disk sub-system. The command

int T_STAGE_SECTION (T_FILE *fd, int *start_coordinate, int *end_coordinate)

immediately returns and starts a data staging operation in the background from tape to disk. It returns an integer to the application which can be interpreted as a descriptor for the associated pre-stage operation. Note that what is actually performed here is to bring the relevant *sub-files* from tape to disk. Note also that there is no 'buffer' parameter in the signature. The routine

int T_STAGE_WAIT (int pre-stage_descriptor)

can be used to wait for a previously initiated pre-stage operation to complete.

int T_PREFETCH_SECTION (T_FILE *fd, void *buffer, int *start_coordinate, int *end_coordinate)

is used to start a pre-fetch operation from disk to memory. The parameters are the same as for T_READ_SECTION. It returns an integer which can be used as a pre-fetch descriptor in a later T_PREFETCH_WAIT call.

int T_MIGRATE_SECTION (T_FILE *fd, int *start_coordinate, int *end_coordinate)

starts to migrate the relevant sub-files (i.e., those corresponding to the section described in the signature) from disk to tape. It should be used with care as these sub-files may contain portions of data that will be requested by a later library call.

Table 2: Access patterns, execution times, and percentage gains.

	Pattern Information			Write Operations			Read Operations		
Acc. Ptr.	Start Coor.	End Coor.	Total float. points	Times w/o chunking	Small Chunk Gain (%)	Large Chunk Gain (%)	Times w/o chunking	Small Chunk Gain (%)	Large Chunk Gain (%)
A	$(0, 0)$	$(1000, 1000)$	$1 * 10^6$	2774.0	96.1	94.5	784.7	85.2	77.1
B	$(0, 0)$	$(4000, 1000)$	$4 * 10^6$	2805.9	83.8	84.9	810.1	43.2	55.6
C	$(0, 0)$	$(24000, 1000)$	$24 * 10^6$	2960.3	8.8	37.9	793.3	-240.5	-172.4
D	$(5000, 5000)$	$(6000, 6000)$	$1 * 10^6$	3321.2	96.7	95.4	798.4	84.1	79.7
E	$(0, 0)$	$(50000, 80)$	$4 * 10^6$	151.7	-3525.1	-2437.6	165.2	-3229.3	-2623.9
F	$(0, 0)$	$(80, 50000)$	$4 * 10^6$	138723.3	96.0	97.2	39214.1	85.9	88.5
G	$(0, 0)$	$(1000, 4000)$	$4 * 10^6$	11096.3	95.9	96.4	3242.9	88.3	88.6
H	$(6000, 6000)$	$(8000, 8000)$	$4 * 10^6$	5095.2	91.2	96.5	1612.9	76.6	89.9

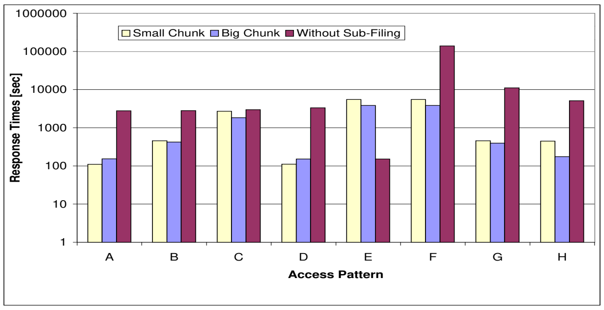

Figure 3: Execution times for **write** operations.

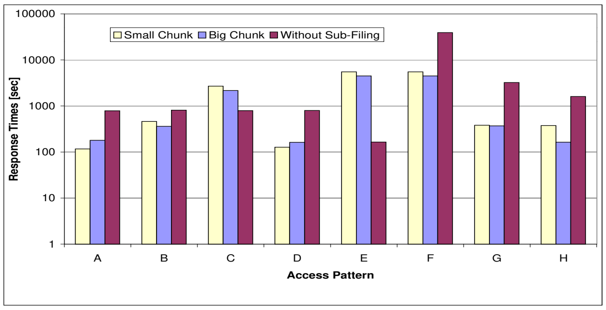

Figure 4: Execution times for **read** operations.

6 Experiments

The experiments are performed using the HPPS at the San Diego Supercomputing Center (SDSC). We have used the low level routines of the SDSC Storage Resource Broker (SRB) to access the HPSS files. SRB is a client-server middleware that provides a uniform interface for connecting to heterogeneous data resources over a network and accessing replicated data sets [1].

We experimented with different *access patterns* in order to evaluate the benefits of the library. Table 2 gives the start and end coordinates (on a two dimensional global array) as well as the number elements read/written for each access pattern (A through H). Note that the coordinate $(0, 0)$ corresponds to the upper-left corner of the array. In each case, the accessed array consists of 50000×50000 floating point elements (10 GB total data). We used two different sub-file (chunk) sizes: small (1000×1000 elements) and large (2000×2000 elements).

The columns 5 through 10 of Table 2 show the performance results obtained. For each operation (read or write) we give the response times (in *seconds*) for a naive access strategy and the gains obtained against it using our library which employs sub-filing. The naive strategy reads/writes the required portion from/to the array directly, i.e., it does not use sub-filing and the entire 50000×50000 array is stored as a single large file. For the sub-filing cases we show the *percentage reduction* in response time of the naive scheme. For example, in access pattern A, the sub-filing with small chunk size improved (reduced) the response time for the read operation by 85.2%. Figures 3 and 4 show the results obtained in graphical form. Note that the y-axes on the figures are *logarithmically scaled.*

In the patterns A and D, where a 4MB square chunk is accessed on the left corner and around the middle, respectively, the small chunk size outperforms the large chunk size as the latter accesses extra data elements that do not belong to the required portion. In the pattern H, on the other hand, increasing the chunk size reduces the number of I/O calls which in turn results in the best response time. In B and G, 16MB of data are accessed in orthogonal directions. In G, since we access a sub-column portion of a row-major array, we need to issue 4000 I/O calls in the naive case. In B, the naive strategy issues only 1000 I/O calls to access the same volume of data. Consequently, the impact of sub-filing is more pronounced in G. In E, the naive strategy outperforms the sub-filing as the entire portion can be accessed in a single I/O call. Note that the

HPSS allows the users to access portions of the data residing in tape. The response time of the naive solution for the access pattern E will increase dramatically for the tape architectures, where the granularity of access is a file. By comparing the response times of A, B, C, and E, we note that the response times are dominated by the number of I/O calls (in the naive version) and of chunks (in the sub-filed versions–that also corresponds to I/O calls–) rather than by the volume of data accessed. Finally, in the pattern F (whose response time in the naive case was calculated using interpolation from A and G), the sub-filing strategy has the best performance of all and brings a 97.2% improvement in write calls.

An important aspect of our library is its handling of random I/O accesses. When we compare the times for the access patterns A and D, we see that there is a 20% increase in the response time of the naive strategy for the write operation. On the other hand, the times for sub-filed versions remains the same.

Overall, the sub-filing strategy performs very well compared to the naive strategy which performs individual accesses to a large file, except for the cases where the access pattern and the storage pattern of the array match exactly. Even in that case a suitable chunk shape can be chosen to match the response time of the naive strategy. However, automatic selection of optimal chunk sizes (considering the future access patterns) is beyond the scope of this work.[1]

7 Conclusions and Future Work

We presented a portable interface to the tape-resident data. It makes it easier for the user to specify the tape I/O required in sequential and parallel applications. The experience gained during its design and development will, hopefully, also help in reaching a set of standard routines for accessing the tape-resident data. We are in the process of implementing the library. We completed the read, write, pre-stage, and pre-fetch routines and made some initial experiments with them. We are currently implementing different migration routines and collective I/O strategies and will later experiment with I/O-intensive applications that manipulate tape-resident data.

References

[1] S. Baru. Storage Resource Broker (SRB) Reference Manual. Enabling Technologies Group, San Diego Supercomputer Center, La Jolla, CA, August 1997.

[2] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. *Proceedings of the ACM Symp. on Prin. and Prac. of Paral. Prg.*, pages 1–10, July 1995.

[1] In the final version of this abstract we will report results obtained using several chunk shapes/sizes that satisfy a given access pattern.

[3] P. Cao, E. Felten, and K. Li. Application-controlled file caching policies. In *Proc. the 1994 Summer USENIX Technical Conference,* pages 171–182, June 1994.

[4] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: parallel and scalable software for input-output. *NPAC Technical Report SCCS-636,* 1994.

[5] P. Corbett, D. Fietelson, S. Fineberg, Y. Hsu, B. Nitzberg, J. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface, In *Proc. Third Workshop on I/O in Paral. and Distr. Sys.*, IPPS'95, Santa Barbara, CA, April 1995.

[6] L. Golubchik, R. R. Muntz, and R. W. Watson. Analysis of Striping Techniques in Robotic Storage Libraries. In *Proc. of the 14th IEEE Mass Storage Sys. Symp.*, pages 225–238, 1995.

[7] B. Kobler, J. Berbert, P. Caulk and P. C. Hariharan. Architecture and Design of Storage and Data Management for the NASA Earth Observing System Data and Information System (EOSDIS). In *Proc. of the 14th IEEE Mass Storage Systems Symposium*, pages 65–78, 1995.

[8] R. A. Coyne, H. Hulen, and R. Watson. The high performance storage system. In *Proc. Supercomputing 93*, Portland, OR, November 1993.

[9] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proc. the 1993 IPPS Workshop on Input/Output in Paral. Comp. Sys.,* April 1993.

[10] J. del Rosario and A. Choudhary. High performance I/O for parallel computers: problems and prospects. *IEEE Computer,* March 1994.

[11] C. S. Ellis and D. Kotz. Prefetching in file systems for MIMD multiprocessors. In *Proc. of the 1989 Intl. Conf. on Paral. Proc.,* pages I:306–314, St. Charles, IL, August 1989.

[12] B. Hillyer and A. Silberschatz. Random I/O scheduling in Online Tertiary Storage Systems. In *Proc. of the 1996 ACM SIGMOD Conference*, pages 195–204, 1996.

[13] A. L. Drapeau and R. H. Katz. Analysis of Striped Tape Systems. In *Proc. of the 12th IEEE Mass Storage Symposium*, Monterey, CA, March 1993.

[14] J. F. Karpovich, A. S. Grimshaw, and J. C. French. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proc. the Ninth Annual Conference on Object-Oriented Prog. Sys., Lang., and Appl.,* pp. 191–204, Oct 1994.

[15] D. Kotz. Multiprocessor file system interfaces. In *Proc. the Second Intl. Conf. on Paral. and Distr. Info. Sys.,* pages 194–201. IEEE Computer Society Press, 1993.

[16] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, V 27(2), pp 21–34, April 1993.

[17] T. Johnson and E. L. Miller. Performance measurements of Tertiary Storage Devices. In *Proc. of 24rd International Conference on Very Large Data Bases*, pages 50–61, August 1998.

[18] T. Johnson and E. L. Miller. Benchmarking Tape System Performance. In *Proc. of the 15th IEEE Mass Storage Systems Symposium*, 1996.

[19] S. Sarawagi: Execution reordering for tertiary memory access. *Data Engineering Bulletin* 20(3): 46–54, 1997.

[20] P. H. Smith and J. van Rosendale. Data and visualization corridors. *Technical Report CACR-164,* CACR, Caltech, Sept 1998.

[21] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. To appear in *Proc. the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999.

[22] S. Toledo and F. G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations, In *Proc. Workshop on I/O in Paral. and Distr. Sys.,* May 1996.

[23] A. Yu, and J. Chen. *The POSTGRES95 User Manual.* Dept. of EECS, University of California at Berkeley, July 1995.