MTIO
A MULTI-THREADED PARALLEL I/O SYSTEM *

Sachin More Alok Choudhary
Dept.of Electrical and Computer Engineering
Northwestern University, Evanston, IL 60201 USA

Ian Foster
Mathematics and Computer Science Division,
Argonne National Laboratory, Argonne, IL 60439 USA

Ming Q. Xu
Platform Computing, Inc.,
North York, Ontario M2N 6P6 Canada

Abstract

This paper presents the design and evaluation of a multi-threaded runtime library for parallel I/O. We extend the multi-threading concept to separate the compute and I/O tasks in two separate threads of control. Multi-threading in our design permits a) asynchronous I/O even if the underlying file system does not support asynchronous I/O; b) copy avoidance from the I/O thread to the compute thread by sharing address space; and c) a capability to perform collective I/O asynchronously without blocking the compute threads. Further, this paper presents techniques for collective I/O which maximize load balance and concurrency while reducing communication overhead in an integrated fashion. Performance results on IBM SP2 for various data distributions and access patterns are presented. The results show that there is a tradeoff between the amount of concurrency in I/O and the buffer size designated for I/O; and there is an optimal buffer size beyond which benefits of larger requests diminish due to large communication overheads.

1. Introduction

Parallelism in I/O can be achieved by storing the data across multiple disks so that it can be read/written in parallel. Based on this idea, known as *data striping*, a number of *parallel file systems* were designed ([1], [7], [4], [8]). These parallel file systems support a UNIX-like file system interface with a limited amount of control over the layout of the data over the disks. While these parallel file systems did provide an improvement in I/O performance over their sequential counter-parts, it was found that they did not live up to expectations under a typical production workload on a parallel computer([5], [2]).

Some of the performance problems associated with parallel file systems can be dealt effectively with if the user application can be *tuned* to the individual characteristics of the parallel file systems. Efforts in this direction resulted in number of runtime libraries that provide a parallel application with a consistent and easy to use I/O interface([10] [9]). However, many such I/O libraries uses the traditional heavyweight UNIX processes to model both compute and I/O tasks. The major drawbacks of such an approach are low processor utilization, absence of asynchronous I/O and overhead of context switching. Thus, a low cost mechanism is needed to co-locate both compute and I/O tasks on the same physical processor to maximize utilization of parallel computing resources as well as to reduce the overhead of context switching.

This paper presents the design of a runtime library that incorporates several important features for improving performance as well as facilitating optimizations. First, the library is based on a multi-threaded approach. Multi-threading is a well-known technique for masking communication latencies in parallel programs. We extend this concept to separate compute and I/O tasks in two separate threads of control. Multi-threading in I/O in our design permits a) asynchronous I/O even if the underlying file system does not support asynchronous I/O; b) copy avoidance from the I/O thread to the compute thread by sharing address space; and c) a capability to perform collective I/O asynchronously.

Further, we present techniques for collective I/O which are implemented as part of this library. These techniques attempt to maximize load balance, concurrency and request size for I/O while reducing communication overhead of collective I/O.

Section 2 presents the overall approach to the MTIO design. Details of the system design, techniques for collective I/O and other optimizations are presented in Section 3. Sec-

*This work was supported in part by NSF Young Investigator Award CCR-9357840, NSF CCR-9509143 and in part by the Scalable I/O Initiative, contract number DABT63-94-C-0049 from Defense Advanced Research Projects Agency(DARPA) administered by US Army at Fort Huachuca.

1063 7133/97 $10.00 © 1997 IEEE

tion 4 presents performance results on IBM SP2. Conclusions are presented in Section 5.

2. Our Approach

In our programming model, we assume a set of compute nodes and and an I/O subsystem. The I/O subsystem is managed by file system software (external to MTIO) which is responsible for data management. The user application runs on multiple nodes. Although the collective model proposed below tends to encourage a loosely synchronous SPMD style programming, MTIO does *not* stipulate it. A program consists of two components, compute blocks and I/O blocks. Compute and I/O blocks occur alternately in the program. MTIO uses two threads of control to manage these two components. A *compute thread (CT)* is used to carry out the compute blocks. An I/O thread (IOT) is used to service the I/O blocks. In this paper, we describe two I/O models under this framework :

Individual Model The compute thread issues an I/O request to the co-located I/O thread. This model essentially permits the overlapping of I/O and computation. However, I/O is performed independently of other compute threads' I/O requests.

Collective Model This model assumes that *all* the compute threads issue I/O request(s) at the same logical step in the program. The I/O threads then consult each other to arrive at a common I/O strategy. The Collective model is useful when a file is shared by all the nodes in the program and each node reads (possibly overlapping) parts of the file. Note that I/O under collective model implies an implicit global synchronization point in the program. Though a powerful (and yet simple) technique by itself, collective I/O cannot provide the expected level of performance when used with the standard UNIX file system interface. A multiple request interface coupled with collective I/O can provide consistent performance improvements.

3. Algorithms

This section provides an overview of the system design and some of the algorithms used in MTIO . For a detailed description please refer to [6].

3.1. Threads Management and I/O Request Communication

The functions `mtio_init` and `mtio_shutdown` are used to initialize and shutdown MTIO . Among other things, they initialize internal data structures like MTIO *file table* and manage MPI (which is the message passing library used by MTIO for inter-processor communication). The program starts as a compute thread on each node. The MTIO initialization function creates the I/O thread on the same processor. This I/O thread acts as the *I/O server* and serves requests from the compute thread. Since compute thread and the I/O thread share the same address space, the communication between the two is performed using shared memory.

When a compute thread invokes an I/O operation, its I/O request is sent to the I/O thread. Compute thread then gives up control of the processor (yields) explicitly so that the I/O thread can get scheduled. After the I/O thread gets scheduled, it starts processing the I/O request. This processing may or may not involve I/O threads running on other processors depending on the type of I/O model. I/O thread then prepares a response which is subsequently communicated back to the compute thread. I/O thread then yields the processor to the compute thread, which examines the response and carries on with the computation. The exact sequence of I/O processing depends on the type of the I/O request and is explained in the following sections.

3.2. File Operations

Two functions, `mtio_open` and `mtio_close` are provided to open and close a file respectively. Note that a file should be opened by *all* processors irrespective of whether that processor uses the file or not. This restriction is placed in order to ensure correct operation of collective model functions. Additionally, the operation of opening a file implies a global synchronization point in the program. However closing a file does *not* imply a synchronization point. `mtio_read` and `mtio_write` are the generic functions used to read from and write to a file.

3.3. Asynchronous I/O

Asynchronous operations are implemented on top of their synchronous counterparts. This allows the asynchronous capability to be orthogonal to other I/O optimizations done by MTIO . The key idea is to choreograph the scheduling of compute and I/O threads to allow maximum overlap between I/O and computation. Upon receiving an asynchronous I/O request, the I/O thread prepares and sends a *wait handle* to the compute thread. Note that the wait handle merely sits in the shared memory and is not received by the compute thread till the compute thread gets scheduled. Now the I/O thread can either 1) yield the processor to the compute thread explicitly or 2) it can start the I/O operation which will eventually block the I/O thread, enabling the compute thread to proceed. In either case, the compute thread will receive the wait handle and start computation. The I/O operation is being carried out by the file system in the background while the compute thread is doing computation. When the I/O operation is over, the I/O thread gets unblocked. It starts executing when it is scheduled by the threads scheduler. The compute thread executes *mtio_wait* call when it needs the data from the I/O operation. This results in the compute thread being blocked. The blocking of the compute thread allows the I/O thread to be scheduled as soon as the I/O operation is over. It is important to note that MTIO provides asynchronous I/O capabilities even if the underlying file system only supports synchronous I/O.

Proceedings of the 11th International Parallel Processing Symposium (IPPS '97)
1063-7133/97 $10.00 © 1997 IEEE

3.4. Collective I/O with Multiple Request Specification

The multiple I/O request specification interface (see [6] for details) allows an application program to specify multiple I/O requests in one call. This permits optimizations by MTIO which otherwise will not be possible if the requests are generated in different calls. The interface puts some constraints on the collective I/O requests. All I/O requests should belong to the same file descriptor. Also, all requests should be of same type, that is they belong to the same model and are either synchronous or asynchronous. Major decisions/steps involved in collective I/O are *I/O request communication, I/O request reconfiguration, I/O scheduling* and *I/O execution and data redistribution*. The collective I/O algorithm is executed by all processors. The other alternative is to ask a distinguished processor to execute the algorithm which then sends to other processors their respective I/O schedules. The first strategy was chosen because it 1) entails no communication and 2) processors other than the distinguished processor sit idle while the distinguished processor computes the I/O schedule.

In the first step all I/O requests are exchanged resulting in a list containing I/O requests of all processors. Each item in the list contains processor that issued this request, processor that will serve this request (This is assigned at a later stage), buffer space for the data, offset in file and length of the request. Thus at the end of this step every processor knows every other processor's requests.

The next step is to combine the I/O requests so that there are fewer and larger resultant I/O requests. An important question here is : how much *extra* memory does a processor have which can be used for the operation. The user has to specify how much extra memory is available at runtime. Also the size of this extra memory available should be greater than or equal to the biggest I/O request issued by any processor. Note that this is justifiable because in a traditional model, an I/O request requires the memory be available to store the data to be read/written.

The algorithm first tries to combine I/O requests that are overlapping or consecutive. It then tries to combine I/O requests that result from the previous step. The reason to carry out the combine operation in two phases is : Combining overlapping/consecutive requests is more profitable than combining requests that have a *gap* between them because the latter ends up reading/writing more data than the combined size of the requests. Note that the algorithm never breaks any I/O request across two resultant I/O requests.

One of the most important decision in collective I/O is to decide which processor processes which I/O requests. The following points need to be considered while assigning the I/O requests to the processors.

Load Balancing Considerations Each processor should process approximately the same number of I/O requests.

Minimization of Inter-Processor Communication (MIPC) Since inter-processor communication is an overhead in collective I/O, the algorithm attempts to assign I/O request in such a way that there is minimum amount of inter-processor communication involved.

The above goals are sometimes conflicting. The load balancing consideration calls for spreading the I/O requests among all processors. While the MIPC criteria tries to lump together I/O requests belonging to the same processor and tries to assign them to that processor.

Another consideration here is : what happens to the requests that are overlapping but that cannot be combined due to memory constraints? Obviously *data reuse* technique can be used here to optimize the cost of I/O. To overcome this problem we mark a request as *to be served by the same processor* if it overlaps with the previous request but can not be combined with it due to memory constraint.

It is obvious that it is not possible to fully satisfy both load balancing criteria and MIPC criteria all the time. Hence the algorithm uses a heuristic where it first tries to optimize on the MIPC criteria by assigning a request whose data belongs to a single processor to the same processor. These are the requests that will definitely benefit from the MIPC optimization. It then uses the load balancing criteria for I/O assignment for the rest of the requests. It first checks if the request is marked *to be served by the same processor*. If so, it assigns it the same processor that the previous request in the list has. If not, it chooses a processor that has minimum number of I/O requests assigned to it.

The algorithm then determines the processor that carries out maximum number of requests. Let this number be n_{max}. The I/O execution and data communication phase is carried out in n_{max} iterations. In each iteration the processor does I/O (if any). It is followed by a *partial complete exchange* where each processor sends any data that belongs to some other processor(s) fetched by it to its recipient(s) and receives any data that belongs its own I/O request(s) fetched by some other processor(s).[1] Additionally, it copies any data that belongs to its own I/O request(s) that it fetched, in appropriate buffers.

4. Performance Results

The development and performance evaluation of MTIO was carried out on a IBM Scalable POWERparallel System (SP) located at Argonne National Laboratory. The MTIO library is built on top of several layers of software and hardware. The MPI layer provides inter-process communication which is used by the compute threads to perform application specific message passing. The Nexus [3] layer provides the low level support for lightweight threads. The

[1] This is what happens in a read operation. Write operation is exactly symmetric. First the data exchange takes place followed by I/O execution. Note that in write operation the processor may have to read some data in cases where combined requests have a *gap* between them.

Proceedings of the 11th International Parallel Processing Symposium (IPPS '97)
1063-7133/97 $10.00 © 1997 IEEE

Buffer Size (MB)	Processors							
	1				2			
	sync.		async.		sync.		async.	
	Time (sec)	B/W (MB/s)	Time (sec)	B/W (MB/s)	Time (sec)	B/W (MB/s)	Time (sec)	B/W (MB/s)
1	304	0.84	347	0.73	304	0.84	337	0.75
2	311	0.82	258	0.99	285	0.89	253	1.01
4	278	0.92	231	1.10	286	0.89	228	1.12
8	272	0.94	207	1.23	278	0.92	216	1.18
16	275	0.93	209	1.22	278	0.92	214	1.19
	Processors							
	4				8			
1	306	0.83	345	0.74	334	0.76	368	0.69
2	298	0.85	258	0.99	321	0.79	297	0.86
4	286	0.89	237	1.08	308	0.83	259	0.98
8	285	0.89	220	1.16	298	0.85	240	1.06
16	283	0.90	216	1.18	327	0.78	260	0.98

Table 1. Asynchronous I/O Results : The total size of the file is 256MB. The bandwidth shown here is per node and each node reads the entire file. For example, for the eight processor combination the bandwidth for the 8MB buffer case would be 8.48 MB/s.

file system layer provides support for I/O operations. UNITREE and IBM Parallel File System (PIOFS) were used as secondary storage for the experiments. UNITREE is a Hierarchical Storage Manager (HSM). The hierarchy consists of two levels. At the higher level RAID disks are used and at the lower level a tape robot system is used. The UNITREE software takes care of migration and staging of the files between the two levels. PIOFS [1] installed on Argonne National Laboratory's IBM SP uses eight nodes of the machine as server nodes.

4.1. Asynchronous I/O

The experiments presented here (using UNITREE HSM as secondary storage) involved reading a 256MB file by each node. Each node reads the whole file one block at a time. It processes the data before reading the next block. The results of the experiment are shown in Table 1. Except for a few exceptions (discussed below) the asynchronous algorithm outperforms the synchronous algorithm.

For smaller buffer sizes the performance of the asynchronous algorithm is worse than the synchronous counterpart. The reason is that the amount of I/O and computation overlap achieved in that case is not sufficient to offset the extra overhead associated with asynchronous I/O.

The success of asynchronous I/O depends on how much overlap can be achieved between I/O and computation keeping the overheads low. Since I/O is more expensive than computation, the maximum amount of overlap that can be achieved depends on computation time as well as the overhead associated with asynchronous I/O. Figure 1 compares

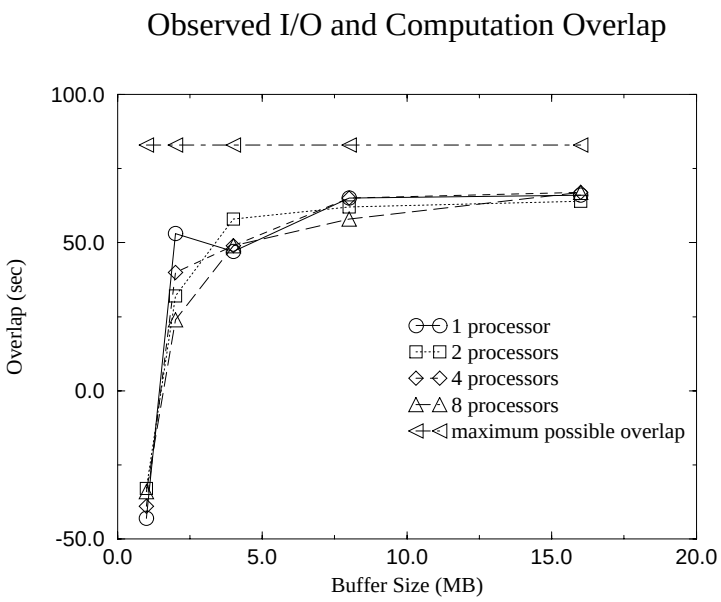

Figure 1.

the maximum theoretically possible overlap with the one actually achieved.

As can be seen from the figure, except for a few cases (very small buffer size) MTIO can successfully provide up to 80 percent overlap between I/O and computation. We performed experiments with various configurations and obtained similar results. Furthermore, we experimented with various scheduling policies for performing asynchronous I/O (e.g. FIFO vs. time-shared). Due to lack of space these results are not presented here.

4.2. Collective I/O

In a distributed memory machine, data structures such as arrays are distributed across processors. Different distributions of data presents different access patterns to the file system. Based on the distribution, each processor may require several I/O calls to fetch (or write) non-contiguous data from a file. In the next set of experiments we present the performance results for collective I/O using different data distributions. The experiments (using IBM PIOFS as the secondary storage) use a $1024X1024$ array whose each element is a double precision floating point number (total file size = 8MB). Similar results were obtained for other data sizes.

I/O under collective model involves three major components; namely, communication overhead, computation overhead and the time to carry out the rescheduled I/O (will be called I/O time henceforth). Table 2 shows the time taken by direct I/O and I/O under collective model for different buffer sizes for 32 processors when the array is distributed in *column-block* fashion and is stored in a row-major order. Each processor makes 1024 noncontiguous read requests of size 256 bytes for a total of 256KB.

Direct I/O	Total Time 19.56			
Collective Model				
Buffer Size	Total Time	I/O Time	Comm. Time	Processor Concurrency
8MB	4.57	0.85	2.42	1
4MB	3.51	0.45	1.76	2
2MB	3.25	0.25	1.66	4
1MB	3.07	0.15	1.53	8
512KB	3.14	0.14	1.63	16
256KB	3.31	0.12	1.74	32
128KB	4.20	0.09	2.49	32
64KB	4.30	0.12	2.24	32
32KB	6.96	0.13	3.89	32
16KB	7.51	0.37	3.37	32
8KB	12.43	0.80	5.03	32

Table 2. Total time (and its components) taken by collective read operation (array size=1024 X 1024, column-block distribution, 32 processors). Processor concurrency refers to the number of processors involved in the I/O operation.

At one extreme we have 8MB of extra buffer on each node. In this case only one processor reads the data in a *single* read operation and distributes it among the rest of the processors. As the buffer size decreases more processors are involved in the I/O so that processor concurrency for I/O increases. In other words, the product of the number of processors involved in I/O and buffer size per processor remains constant. In this particular case, the maximum processor concurrency for I/O is 32, which is reached when the buffer size per processor is 256KB. Note that we get more and more I/O parallelism as the number of processors doing I/O increases resulting in smaller and more I/O requests (from a single 8MB request to thirty two 256KB requests). At 256KB buffer size *all* the processors participate in the I/O operation. Each node executes exactly one read request of size 256KB and then exchanges data with its peers. As the buffer size dips below 256KB threshold, each processor still reads 256KB of data but the I/O is carried out in *multiple phases*. From buffer sizes 256KB to 1KB the I/O parallelism does not increase but the number of requests increases and size of each request decreases (from thirty two 256KB requests to 8192 1KB requests). This results in poorer I/O performance.

When the buffer size is maximum (8MB), all the data is read by one processor and then distributed among the rest. This results in high communication overhead (one processor sends 31K messages of size 256 bytes). As the number of processors doing I/O increases the communication overhead decreases due to better distribution of communication responsibility among the processors. This trend continues until we reach buffer size of 256KB. At this point

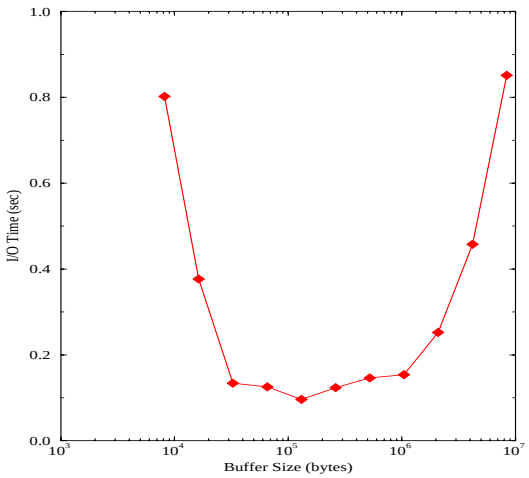

Figure 2. I/O time taken by collective read operation (Array size=1024 X 1024, Distribution=column-block,Processors=32)

the communication is still carried out in a single phase (that is all the data exchanged by processor *x* and processor *y* is sent/received in a single step, as a results each node sends 992 messages of size 256 bytes). As the buffer size reduces further, communication has to be carried out in a *multiple* phases which incur extra overhead.

Hence at very small buffer sizes both the I/O and communication performance is not good. As the buffer size increases the performance also improves. This trend continues until we approach an optimal buffer size. Beyond this point the performance starts degrading again. The size of the optimal buffer size was found to be dependent on the I/O access pattern. Performance of direct I/O was found to depend heavily on the access patterns. In contrast the performance of collective I/O was found to be stable against changes in the access patterns. Table 3 summarizes results for different number of processors and three different data distributions. The table demonstrates that collective I/O design presented in this paper provides consistent and much better performance than direct (traditional) I/O.

Due to lack of space, other experiments and features of MTIO are omitted from this paper. In summary, those features include : 1) Independent model of I/O in which an I/O thread seeks services of other I/O threads to perform I/O on its behalf (note that other compute threads are not involved in this request); 2) Experiments with different thread scheduling techniques (e.g. FIFO, time-slicing); and 3) Thread yielding strategies employed by the I/O thread in asynchronous I/O.

Distribution	Processors											
	4			8			16			32		
	Direct	Collective		Direct	Collective		Direct	Collective		Direct	Collective	
	Time (sec)	Buffer Size	Time (sec)	Time (sec)	Buffer Size	Time (sec)	Time (sec)	Buffer Size	Time (sec)	Time (sec)	Buffer Size	Time (sec)
Row-Block	0.480	2MB	0.363	0.273	1MB	0.330	0.168	512KB	0.292	0.146	256KB	0.290
Column-Block	5.935	1MB	1.096	4.792	1MB	0.782	11.619	512KB	0.631	8.987	1MB	3.072
Block-Block	4.216	2MB	0.950	4.792	1MB	0.782	4.881	512KB	1.740	19.564	1MB	1.399

Table 3. The table compares time taken to read the array by direct I/O and collective I/O (at optimal buffer size) for different data distributions.

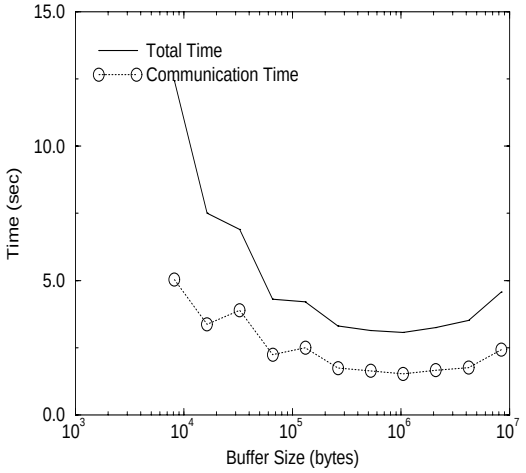

Figure 3. Communication time and Total time taken by collective read operation (Array size=1024 X 1024, Distribution=column-block, Processors=32)

5. Conclusions

This paper focused primarily on the techniques asynchronous I/O and collective I/O. A threads based approach was used to build MTIO and it was shown that a judicious choice of I/O optimization techniques (or a combination of them) can provide significant performance improvement. The main contributions include a runtime implementation of various I/O optimization techniques, providing asynchronous I/O capability when the underlying file system does not provide one and asynchronous I/O under collective model which is not provided by other runtime libraries since collective I/O is inherently synchronous.

References

[1] F. E. Bassow. Installing, managing, and using the IBM AIX Parallel I/O File System. IBM Document Number SH34-6065-00, February 1995. IBM Kingston, NY.

[2] R. Bordawekar, A. Choudhary, and J. M. D. Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–376, 1993.

[3] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating communication and multithreading. *Journal of Parallel and Distributed Computing*, 1996.

[4] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal. PPFS: A High Performance Portable Parallel File System. Technical Report UIUCDCS-R-95-1903, University of Illinois at Urbana Champaign, January 1995.

[5] D. Kotz and N. Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, November 1994.

[6] S. More. MTIO. Master's thesis, Dept. of Electrical and Computer Engineering, Syracuse University, August 1996.

[7] S. Moyer and V. Sunderam. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[8] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996.

[9] K. E. Seamons and M. Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, pages 218–227, September 1994.

[10] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and K. Sivaramakrishna. Passion: Optimized i/o for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.