Source Level Transformations to Improve I/O Data Partitioning

Yijian Wang and David Kaeli
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA 02115
yiwang, kaeli@ece.neu.edu

Abstract

The main goal for parallel I/O is to increase I/O parallelism by providing multiple, independent data channels between processors and disks. To realize this goal, I/O streams need to be parallelized and partitioned at multiple system layers. Contention at any level can dramatically decrease performance and limit scalability. To address this disk contention bottleneck, it is important to carefully study disk access patterns.

From our previous work on I/O profiling, we found that I/O access patterns of parallel scientific applications are usually very regular and highly predictable. Thus it is possible to detect I/O access patterns statically during compiler time. Large datasets are logically linearized in file space on disk, and these intensive data accesses follow a linear space traversal. In this paper, we present our recent work on compiler-directed I/O partitioning, based on Linear Disk Access Descriptors (LDAD). We use the SUIF compiler infrastructure to perform data-flow analysis and recognize LDADs. We then use these LDADs to guide our I/O data partitioning that utilizes multiple disks to significantly increase I/O throughput.

1 Introduction

In the area of high performance computing, we are seeing a growing need to work with very large datasets, which usually reside on secondary storage systems. As the gap between processor and disk speeds continues to grow, when multiple processes access data stored on the same storage device simultaneously, high disk latencies will be incurred. Access to disk-based data can be a barrier to achieving scalable parallel performance.

The main goal for parallel I/O is to increase I/O parallelism by providing multiple, independent data channels between CPUs and disks. To realize this goal, I/O streams need to be parallelized at multiple layers (i.e., at the process level [6, 11], file level [4], and disk level [8, 15]). Contention at any level can dramatically decrease performance and limit scalability. Ideally, each I/O process only needs to access a file partition located on its associated local disk and thus no disk contention will occur. Unfortunately, for real applications, data sharing exists when multiple processes access the same file space. We need an effective method to recognize I/O access patterns in order to effectively parallelize I/O streams.

Previous studies characterizing parallel I/O accesses have shown that I/O intensive parallel applications often access disks using a large number of non-contiguous, small, data chunks [2, 13]. These I/O characteristics prevent disk bandwidth to be fully utilized and impact I/O performance. Access patterns can be detected both statically (at compile time) [3, 7, 12, 16] and dynamically (at runtime) [17]. In [7, 16], Paek et al. use Linear Memory Access Descriptors to detect memory array access patterns within loop nests. In our previous studies [1, 17], we have used profile-directed optimization to improve both memory and disk I/O accesses. In [17], we found that disk I/O accesses exhibit very regular and highly predictable patterns. In this paper, we present our recent work on a static (i.e., compile time) approach to performing data partitioning.

The rest of this paper is organized as follows. Section 2 will describe our static I/O partitioning approach. Section 3 will describe the experimental environment that we are using to do this work. Section 4 will discuss four I/O intensive workloads and evaluate how our compile-time driven approach can obtain significant speedup. In section 5, we will summarize this paper and talk about our future work.

2 Compiler-directed Optimization

2.1 Linear Disk Access Descriptor

Data intensive applications often access a very large dataset (i.e., a multi-dimensional array) on disk. When a large dataset is allocated on disk, it is logically linearized and laid out in file space. Hence, accessing the array on disk should be equivalent to traversing a linear space. I/O access patterns can characterized using three values (shown in Figure 1):

1. span
2. stride
3. size

The stride records the distance between two contiguous data chunks; the span captures the total distance accessed by the program; the size denotes the size of every contiguous data chunk.

A Linear Disk Access Descriptor consists of these three values and is denoted as $LDAD(span, stride, size)$.

Figure 2 shows typical code implementing parallel IO using MPI-IO [14]. In this example, multiple processes perform interleaved writes to a shared file, all within a loop. After MPI completes initialization and the shared file is opened in the loop nest, each process writes a small data chunk starting from an address defined by a MPI function call. In this example, the stride of each process is $nprocs*SIZE$ and the span is computed as $K * nprocs * SIZE$, where $SIZE$ is the size of the contiguous data chunks.

Next, we define two different classes of $LDADs$. An LDAD can be:

- an *interleaved LDAD*, if $stride/size$ is equal with number of processes (See Figure 3(a)), or
- an *overlapped LDAD*, if $stride/size$ is less than the number of processes(See Figure 3(b)).

An LDAD L1(span1,stride1,size1) is said to have a finer granularity than LDAD L2(span2, stride2, size2) if:

- L1 and L2 are interleaved LDADs;
- $span1 = span2$;
- $stride1 < stride2$ (see Figure 3(c)).

2.2 SUIF2 Compiler Infrastructure

In order to capture LDAD patterns exhibited within an application, we perform data flow analysis using the SUIF2 compiler infrastructure [5]. SUIF is a compiler infrastructure designed to support research and development of a range of compilation techniques based on a program representation called SUIF [5]. Many previous studies have used SUIF to study loop transformation optimizations as well as characterize data parallelism [9, 10]. In [10], Wolf and Lam presented a loop-level transformation algorithm to improve data locality within a loop nest. In [9], Hall et al. proposed to automatically parallelize and optimize sequential programs for shared-memory multiprocessors using SUIF.

The goal of our data flow analysis is to predict values of LDADs statically i.e., at compile time). Variables can be classified into four categories based on their charateristics:

- variables that are initialized early in the source code (i.e., through static analysis) and never modified;
- variables that are initialized at the beginning of the dynamic execution and never modified;
- variables that are initialized and then modified later, and it is possible to resolve these values statically;
- variables whose values are impossible to detect statically due to complicated control flow.

We are using the SUIF2 compiler to analyze values of LDADs. We input source code to SUIF2, which performs front-end transformations and data flow analysis, and outputs any predicted values of LDADs.

2.3 Weighted I/O Control Flow Graph

Data-intensive applications often access disk many times during their execution. Every access may exhibit a different access pattern over different executions. We capture file operations (reads and writes) for every basic I/O code block, as well as the LDAD access pattern. We then construct a *weighted I/O control flow graph* for each dataset/file.

In a I/O control flow graph, every node denotes a basic I/O code block, which is usually a loop nest in the source code. The value on each node denotes a read or write, capturing the particular file operation. The weighted directed edge between each pair

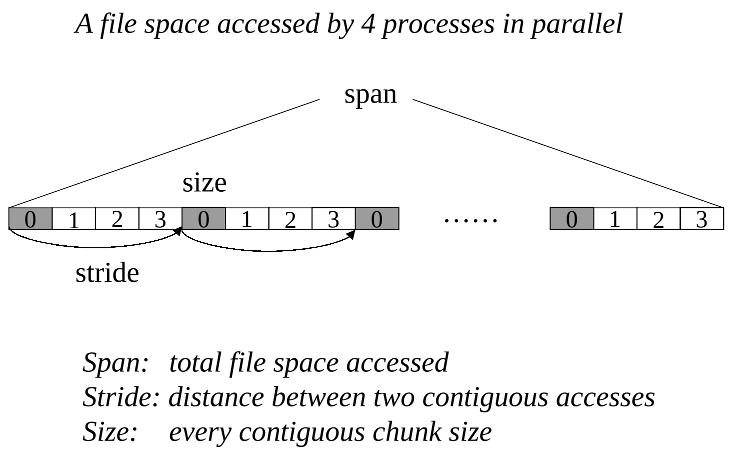
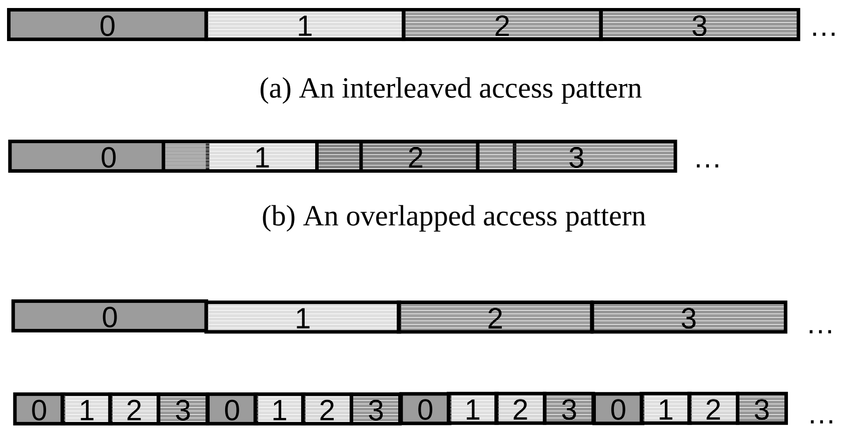

A file space accessed by 4 processes in parallel
span
size
0 1 2 3 0 1 2 3 0 …… 0 1 2 3
stride
Span: total file space accessed
Stride: distance between two contiguous accesses
Size: every contiguous chunk size

Figure 1: Linear I/O access patterns.

```
…
MPI_Init( );
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &mynod);
…
MPI_File_open( MPI_COMM_WORLD, “filename”,
                    MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
…
…
For( i=0; i<K; i++) {
            …
            MPI_File_write_at( fh, i*nprocs*SIZE+mynod*SIZE,
                                          buff, SIZE,
                                        MPI_DOUBLE_PRECISION,
                                          mstatus, ierr);
            …
}
```

Figure 2: Typical parallel I/O code.



(a) An interleaved access pattern

(b) An overlapped access pattern

(c) access patterns with different granularity

Figure 3: Access patterns

of read and write nodes is labeled with the number of times the dataset is read or written after it has been written to, or read from, disk. From our profiling study [17], I/O control flow patterns contained in parallel scientific applications typically fall into one of three categories shown in Figure 4:

- some applications first generate a large data file and then read the data back a number of times (Figure 4(a));
- some applications periodically write to and read from disk (Figure 4(b));
- some applications read data from an input data source and generate files dynamically during execution (Figure 4(c)).

Although some applications may exhibit a hybrid combination of these control flow patterns, we only focus on these three I/O control flow graphs described above.

After building the weighted I/O control flow graph, we traverse each write-read pair in every graph to find the access pattern which possesses the heaviest weight. We then use this access pattern as a template to guide source codes transformations. For example, in Figure 4(a), access pattern LDAD1,2 has the heaviest weight, which means that LDAD1,2 is traversed most frequently, and thus we need to modify the write access pattern and the other two read access patterns to LDAD1,2. In Figure 4(b), LDAD2,1 and LDAD 2,2 have the same weight and so we can choose either of them as a template. However, in Figure 4(c), there is no dominant write-read pair.

2.4 Source Level Transformation

To effectively guide I/O partitioning, our goal is to allocate datasets across multiple disks in an attempt to maximize data parallelism. Having determined an optimized access pattern using the I/O control flow graph, we partition local dataset of each process onto its local disk and keep shared datasets resident on centralized disks. We substitute the original file operations with our partitioned file operations, which will localize file accesses to local disks.

The steps in our algorithm are shown in Figure 5. We use the SUIF compiler infrastructure [5] to perform all dataflow anlysis and I/O control flow analysis in order to generate the initial LDADs and optimized LDADs, respectively. The optimization information is used to guide I/O partitioning and source level transformations. We then compare the runtimes of the optimized code and the original source.

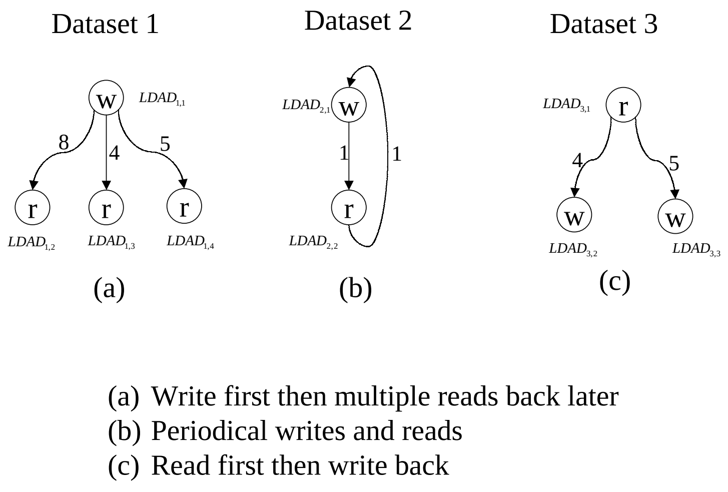


Figure 4: Weighted I/O Control Flow Graphs

3 Experimental Environment

We have performed a set of experiments on a Beowulf cluster with a centralized SCSI RAID device and multiple local IDE disks, as shown in Figure 6. The Beowulf Cluster used in this work has 32 nodes; each node has a local 8.4 GB IDE disk and there are shared SCSI RAID devices directly attached to four of these nodes.

Table 1 provides detailed configuration information about the Beowulf Cluster. In the results presented, all runs used standard nodes and one RAID-connected node. The SMP node is not used in this work.

Table 2 provides raw bandwidth rates for a local access to an IDE disk, a non-local access to the SCSI disk, and a local access to a SCSI disk. I/O rates are measured for different chunk sizes. Non-local access assumes that the I/O must communicate across the 100Mb switched ethernet network to transfer data, as well as to read/write to the SCSI disk.

4 Experiments

4.1 Parallel I/O workloads

We study four parallel I/O-intensive workloads. Some characteristics of these workloads is shown in Table 3.

- The Perf benchmark is a parallel I/O test program provided with the MPICH standard distribution. The code is written in C. Every process writes a 1 MB chunk at a location determined by its rank, and then reads it back later. The chunk size is user-defined. There is no overlap between chunks. Two parallel I/O schemes are studied with this benchmark: MPI I/O (provided in the benchmark source) and partitioned I/O (our own implementation).

- Mandelbrot is an image processing application that generates a Mandelbrot image file. In this benchmark, a Mandelbrot image data file (256MB) is generated by multiple processes and then read back for visualization. The code is written in C. The code is computationally intensive, I/O intensive, and visualization intensive. The size of each contiguous file access depends on the number of processes. Two parallel I/O schemes are studied with this benchmark: MPI-IO (provided in the benchmark source) and partitioned I/O (our own implementation).

- Jacobi is a file-based out-of-core jacobi application from University of Georgia. In this program, the initial out-of-core data is stored in a file. The

Table 1: Hardware specifics of the Beowulf Cluster used in this work

Number of nodes	32 (27 standard nodes, 4 RAID device host nodes and 1 SMP node)
Processor Type	Intel Pentium II 350 (standard nodes and RAID nodes)
Memory	256MB SDRAM, PC100, ECC (standard nodes and RAID nodes), 2GB (SMP nodes)
Disk adapters	
IDE	Onboard Intel PCI (PIIX4) dual ultra DMA/33
SCSI	UltraWide SCSI
RAID device	Morstor TF200 with 6-9GB Seagate SCSI disks, 7200rpm
RAID capacity	36GB usable, one hot spare
IDE disk	IBM UltraATA, 8.4GB, 5400rpm
Parallel file system	NFS 3
Network	10/100 Ethernet Cisco Catalyst 2924 Switch
NIC	Intel 82558 10/100Mb

Table 2: Raw bandwidth rates in MBs per second for our Beowulf Cluster.

Disk/Operation	128	512	1K	2K	4K	32K	64K	128	512	1MB
IDE read	7.1	11.9	13.0	10.1	9.8	7.6	8.5	8.2	8.1	9.0
SCSI read non-local	0.4	1.1	2.1	3.9	5.6	7.0	7.5	10.7	9.9	8.9
SCSI read local	9.8	13.8	14.6	16.7	16.4	19.5	16.1	17.9	17.1	17.9
IDE write	2.6	3.2	3.7	4.6	4.5	4.1	4.1	4.4	4.9	4.1
SCSI write non-local	0.2	0.6	0.8	1.7	2.8	2.1	2.8	2.8	3.3	3.3
SCSI write local	4.6	8.2	9.9	12.5	11.6	12.9	11.2	10.0	11.3	12.1

Table 3: Parallel I/O workloads used in this work

Name	Source	Language	Parallel I/O implementation
perf	MPICH	C	MPI-IO
Mandelbrot	Synthetic	C	MPI-IO
ooc-jacobi	Univ. of Georgia	C	MPI-IO
ooc-FFT	MPI-SIM	C	MPI-IO

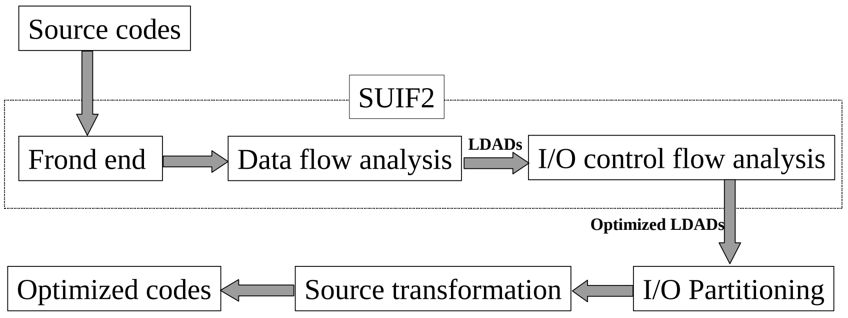


Figure 5: Procedure of Compiler-directed I/O partitioning.

code is written in C. The result of the jacobi iteration is written to a second file. The size of each file is 64 MB. The size of every contiguous data chunk is a function of the number of processes. Two parallel I/O schemes are studied with this benchmark: MPI-IO and partitioned I/O.

- FFT is another file-based out-of-core application from MPI-SIM. It performs a Fast Fourier Transform on a disk-resident array of 1M complex numbers. The size of each dataset depends on the number of processes. The input file size is 8MB and two temporary files and the final result file are generately dynamically, each being 8MB in size. The code is written in C. We compare the performance of MPI-IO and partitioned I/O for cluster sizes of 4, 8 and 16 nodes. Since the the number of nodes had to be a power of 2, we were not able to configure a system with 32 nodes.

4.2 Experimental Results

Figure 7 shows the overall execution time for the four file-IO oriented applications studies as run on 24 nodes of our cluster. The runtime results have shown that I/O throughput has been significantly improved by compiler-directed partitioned I/O. Overall execution time has been reduced by 81.2%, 27.8%, 87.1% 72.8% for perf, Mandelbrot, Jacobi and FFT, respectively. We have achieved significant speedup because that we have created multiple independent data channels between CPUs and disks by providing local access to local disk; we have reduced the network contention to centralized disk devices; we have also reduced disk seek time by improving spatial locality.

5 Summary and Future Work

To achieve high performance in file data dominated applications, I/O streams must be parallelized at both the application level and the disk level. Multiple disks should be employed to create multiple independent data channels between CPUs and underlying storage systems. In this paper, we have been able to demonstrate how to perform source-to-source code transformations, assigning I/O chunks to partitions in an attempt to produce parallel streams of access. Experimental results show that we have reduced execution time by 28-87% for 4 I/O-dominated applications.

For more complicated applications, it is not always possible to detect file access patterns statically because of their intricate control flow. In past work, we have been able to characterize complicated I/O access patterns dynamically using a profile-guided ap-

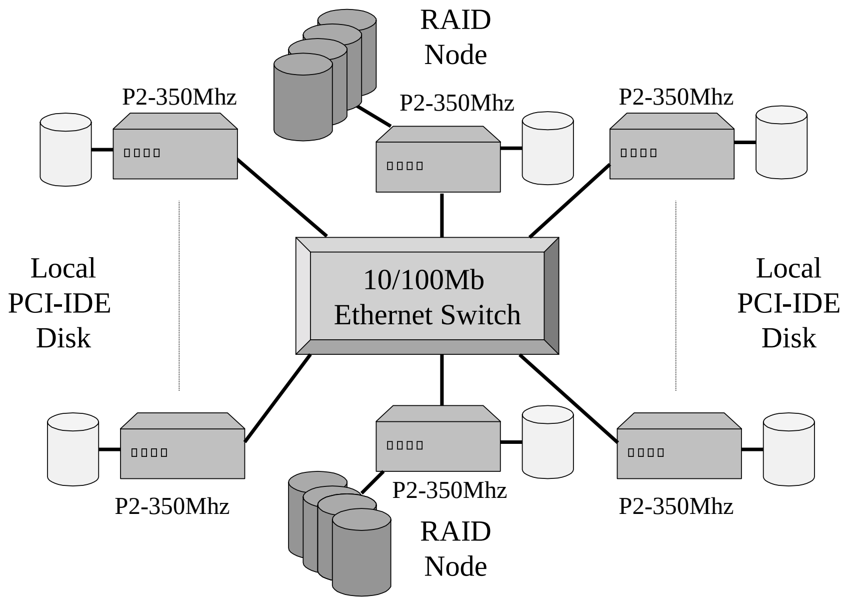


Figure 6: Structure of the Beowulf Cluster used in this work.

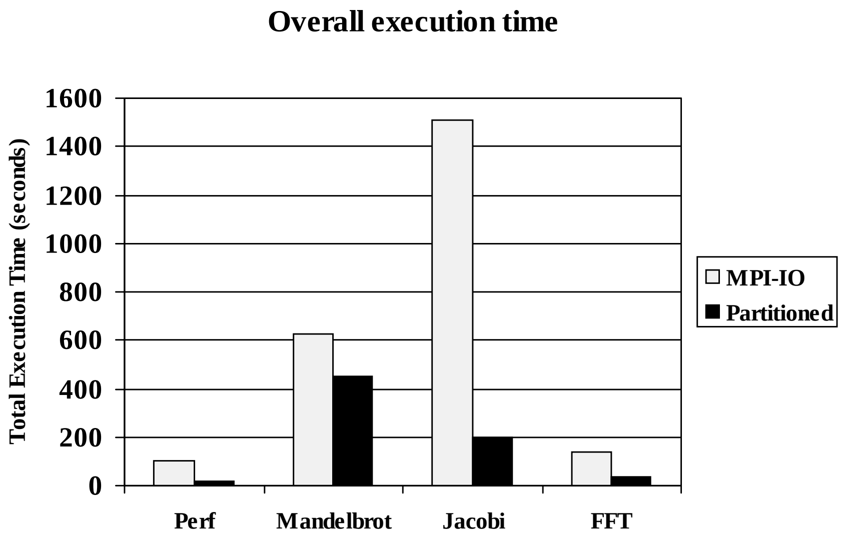


Figure 7: Performance of the entire applications comparing MPI I/O and compiler-directed partitioned I/O.

proach [17]. In our future work, we will focus on using a more sophisticated data flow and I/O control flow analysis to guide data partitioning, and we will also study how better to use profiling information to guide our compiler analysis to deliver high I/O performance.

This work was funded by CenSSIS, the Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the NSF (Award Number EEC-9986821), and by an NSF Major Research Instrumentation Grant (Award Number MRI-9871022).

References

[1] D. Kaeli, L. Fong, D. Renfrew, K. Imming and R. Booth. Performance of a CC-NUMA Prototype. *IBM Journal of Research and Development*, 41(3), 1997.

[2] E. Smirni and D. Reed. Workload Characterization of Input/Output Intensive Parallel Applications. In *Proceedings of 9th International Conference on Computer Performance Evaluation*, June 1997.

[3] G. Memik, M. Kandemir and A. Choudhary. Design and Evaluation of a Compiler-directed Collective I/O technique. In *Submitted to Euro-Par 2000*, 2000.

[4] G. Memik, M. Kandemir,and A. Choudhary. Exploiting Inter-File Access Patterns Using Multi-Collective I/O. In *Proceedings of the 1st Conference on File and Storage Technologies(FAST)*, January 2002.

[5] http://suif.stanford.edu/. The SUIF Compiler System.

[6] http://www.mpi forum.org/. Message Passing Interface Forum.

[7] J. Hoeflinger and Y. Paek. The Access Region Test. In *Proceedings of the 12th Annual Workshop on Languages and Compilers of Parallel Computing*, 1999.

[8] K. Hwang, H. Jin, and Roy. S.C. Orthogonal Striping and Mirroring in Distributed RAID for I/O-Centric Cluster Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(1), 2002.

[9] M. hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Transactions on Computers*, 29(12), 1996.

[10] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), 1991.

[11] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on Frontiers of Massively Parallel Computation*, February 1999.

[12] S. Hiranandani, K. Kennedy and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Supercomputing*, pages 86–100, 1991.

[13] T. Madhyastha and D. Reed. Learning to Classify Parallel Input/Output Access Patterns. *IEEE Transactions on Parallel and Distributed Systems*, 13(8), 2002.

[14] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel a nd Distributed Systems*, pages 23–32, 1999.

[15] Y. Hu, Q. Yang, and T. Nightingale. RAPID-Cache — A Reliable and Inexpensive Write Cache for Disk I/O Systems. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, January 1999.

[16] Y. Paek, J. Hoeflinger, and D. Padua. Simplication of Array Access Patterns for Compiler Optimizations. In *Proceedings of ACM SIGPLAN'98*, 1998.

[17] Y. Wang and D. Kaeli. Profile-Guided I/O Partitioning. In *Proceedings of the 17th ACM International Conference on Supercomputing*, June 2003.