Compiler-Directed I/O Optimization*

Mahmut Kandemir
Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802, USA
kandemir@cse.psu.edu

Alok Choudhary
Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208, USA
choudhar@ece.nwu.edu

Abstract

Despite continued innovations in design of I/O systems, I/O performance has not kept pace with the progress in processor and communication technology. This paper addresses this I/O problem from a compiler's perspective, and presents an I/O optimization strategy based on access pattern and storage form (file layout) detection. The objective of our optimization strategy is to determine storage forms for array-based data sets taking into account future use of data (future access patterns). To tackle this problem, we present a three-step strategy: (i) determining all I/O access patterns to the array, and among them, selecting the most dominant (i.e., the most beneficial) access pattern; (ii) determining the most suitable storage form for the array taking into account the most dominant access pattern detected in the previous step; and (iii) optimizing the non-dominant access patterns using collective I/O, an optimization that allows each processor to do I/O on behalf of others if doing so improves overall performance.

Keywords: Parallel I/O, Optimizing Compilers, I/O-Intensive Codes, File Layouts, Access Patterns.

1. Introduction

Despite continued innovations in design of I/O systems, I/O performance has not kept pace with the progress in processor and communication technology. For example, although disk transfer times have reduced in recent years (as a result of the increases in magnetic-media densities), the overall improvement in disk access times has been much less than the corresponding improvement in CPU cycle times and memory access times. In addition to this, while recent years have witnessed a growth in I/O software (particularly in the areas of runtime libraries and file systems), the performance of I/O-intensive applications is still far behind the performance of CPU-intensive codes. This is mainly due to lack of concrete automatic optimization techniques for I/O [11].

This paper addresses this I/O problem from a compiler's perspective and presents an I/O optimization strategy based on access pattern and storage form (file layout) detection and program transformations. The main idea is to make access patterns and storage forms compatible for as many disk-resident arrays as possible. To achieve this objective, our solution proceeds in three complementary steps for each disk-resident array: (i) determining all I/O access patterns to the array, and among them, selecting the most dominant (i.e., the most beneficial) access pattern; (ii) determining the most suitable storage form for the array taking into account the most dominant access pattern detected in the previous step; and (iii) optimizing the non-dominant access patterns using collective I/O, an optimization that allows each processor to do I/O on behalf of others if doing so improves overall performance. Our objective is to select a storage form (for each disk-resident array), which is the best from the perspective of future I/O accesses to the array. However, the proposed approach also works with the file layouts that have already been determined (e.g., by a previous application).

The remainder of this paper is organized as follows. Next section describes the I/O problem that we address in this work. Section 3 presents our approach in detail. Section 4 concludes the paper by summarizing our major contributions.

2. I/O Problem

There is a broadening gap between performances of I/O-intensive applications and CPU-bound applications. This is due to two major factors. First, there exists an increasing disparity in the performance of I/O devices and the performance of processors and communication links on parallel platforms. In other words, I/O devices are inherently slow

*This work was funded (in part) by the National Science Foundation through the grant CCR #0097998.

1530-2075/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

compared to CPU devices and communication links [11]. Second, the current software support for optimizing I/O-intensive codes is not adequate. This second point manifests itself in a number of ways. First, at the compiler level, very few techniques (e.g., [20, 4, 3, 12]) target specifically I/O statements; rather, many data dependence and data reuse analysis techniques employed by state-of-the-art optimizing compilers fail when, during compilation, an I/O statement (e.g., within a loop) is encountered. Second, at the runtime system, libraries, and operating system (OS) levels, no single, standard API for I/O exists. Instead, each vendor has its own API, which makes code portability very difficult. Different APIs are used in Unix, in commercial parallel file systems like Intel PFS [21], IBM PIOFS [8], in research file systems such as PPFS [13], Galley [18], in I/O libraries such as PASSION [22], Panda [6], ChemIO [19], and in MPI-IO [7, 23, 24], a recent attempt for I/O-interface standardization. Finally, mainly, due to the first two factors, there are not many solid programming strategies available for I/O-intensive applications.

Despite this grim picture, specific application domains present better optimization opportunities for I/O. One of these domains is scientific computing where multi-dimensional disk (or tape) resident arrays are manipulated using multiple processors. While many scientific environments process vast amounts of data, in most cases, the regularity in data access patterns enable code and data structuring for better I/O behavior. Therefore, previous work in I/O compilation [3, 20, 15, 4] targeted scientific applications that manipulate out-of-core data sets. Note that a compiler is in a good position for optimizing I/O in a program-wide fashion as it can capture the global (program-wide) data access pattern.

Many parallel I/O-intensive scientific codes from different fields such as astrophysics and computational chemistry access their data from files stored on (multiple) disks. Consequently, it is of utmost importance that disk access patterns exhibited by these applications are compatible with the storage form of data (e.g., its striping style across parallel disks). Working at the disk storage level, however, makes a compiler-based optimization process very complex and non-portable as it requires access to low-level disk system-related parameters. Instead, we show in this paper that an optimizing compiler can work with file layouts which directly represents the programmer's view of the data set in question. Note that while this high-level abstraction makes compiler's job easier, it also brings an inaccuracy. Nevertheless, our results reported in this paper show that setting the abstraction level to file layouts (file storage forms) is a reasonable approach. Therefore, an optimization process can target improving file access patterns.

It should be noted that the suitability of a file access pattern is directly linked to the storage form (layout) of the file. In other words, whether a file access pattern is good or not depends on the file layout. If, for example, a two-dimensional array stored in file as row-major is accessed by multiple processors using a column-wise access pattern, we cannot expect a good performance in general. This is because the column-wise access pattern is not compatible with the row-major storage form. In contrast, a row-wise access pattern (whereby each processor accesses a row-block portion of the said array) is compatible with the row-major storage form, and can be considered good. While for a given storage form, it is possible to determine the most suitable file access pattern(s), for applications which create their own disk-resident arrays before using them in subsequent computation, we can go one step further, and determine the storage forms as well.

Based on the discussion above, we can define our problem as follows: Given an I/O-intensive application which manipulates multi-dimensional arrays stored in disk-resident files, what is the best file storage form for each disk-resident array and what is (are) the most suitable access pattern(s) considering the best storage pattern.

In the ideal case, we would like to store a given array in such a form that most of the future accesses to the array will have the best possible I/O performance. While this problem definition implies that for the best performance both the most suitable access pattern and the storage form should be determined, it does not say whether the access pattern should be determined before the storage form, or vice versa. There are several issues here that need to be addressed. First, there may not be a single access pattern for a given array. Instead, each reference in the code might result in (define) an access pattern, and it is very likely that two different references (to the same disk-resident array) might demand different access patterns. Consequently, a conflict resolution mechanism is needed to distinguish the most suitable access pattern from the others. Second, we may not be able to find a storage form which is compatible with all access patterns. Because of this, we need to adopt a strategy for handling the access patterns not compatible with the storage form of the array. Third, as hinted above, access patterns and storage forms are not independent attributes; rather, the relation between them should be taken into account in determining the best combination. The following section explains our solution to the problem defined above and addresses some of the issues mentioned.

3. Our Approach

Our objective is to select for each disk-resident data set (array) a file layout and an access pattern (compatible with the file layout). One might come up with different strategies for this. One alternative would be first fixing the layout at a specific form, and then determining the access patterns.

IEEE COMPUTER SOCIETY

Alternatively, we can first determine the access pattern, and then select the storage form. Our strategy is a combination of both and is as follows for each disk-resident array:

(1) Determine all access patterns to the array. These access patterns are defined by the references in the code to that array. Then, among these patterns, select the *most dominant* (or *most beneficial*) one.[1]

(2) Taking into account the most dominant access pattern found in the previous step, determine a storage form (layout) for the array. Since the storage form is determined based on the most dominant access pattern, we can expect that it is the most beneficial one as far as I/O performance is concerned.

(3) Since the selected storage form may not satisfy all access patterns to the array (as there would be patterns which are different from the most dominant one), optimize these non-dominant access patterns as much as possible. For this purpose, we can use optimization techniques such as collective I/O [24, 10].

Below we describe these three steps in detail.

3.1. Access Pattern Detection

In a given code, each array might be accessed using different patterns. Consider the code fragment on the left part of Figure 1 which consists of two separate doubly-nested loops that access a disk-resident array `u`. Assuming that in the first nest the `i` loop is parallelized, and in the second nest the `k` loop is parallelized, the access pattern for (the reference in) the first nest is row-wise (i.e., each processor accesses a row-block), whereas the references in the second nest exhibit column-wise and row-wise access patterns as depicted on right in Figure 1. Since all loops in this fragment have the same number of iterations (`N`), all three references have the equal saying in the final performance of the code (assuming that no conditional flow of control exists within the loops). Since two of three access patterns are row-wise, it might be reasonable to set the dominant access pattern for array `u` to row-wise.

Let us now explain our dominant access pattern detection strategy for the general case. We start by a formal definition of the access pattern concept. An access pattern of a given array `u`, $\mathcal{P}_{\mathtt{u}}$ is a function of the parallelization strategy and specifies how each dimension of the array is divided (distributed, or decomposed) across parallel processors. To simplify discussion, let us focus on an m-dimensional case. An m-dimensional array `u` can be accessed using, say, p processors in a number of ways. One approach might be

[1] This 'most dominant access pattern' concept will be formalized later in the paper.

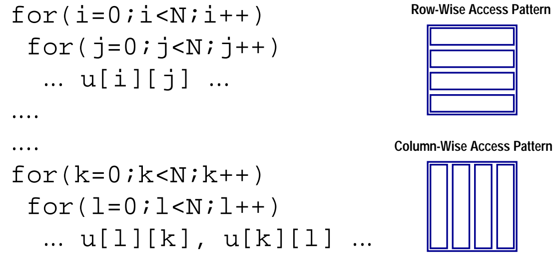

Figure 1. Left: A code fragment which consists of two separate nested loops; Right: Two different access patterns.

to distribute only a specific dimension d ($1 \leq d \leq m$) across all p processors; the remaining dimensions are not distributed. We can represent such an access pattern using the notation $[*][*][*]....[*][p][*]....[*]$, where $[*]$ denotes an undistributed dimension and $[p]$ (which appears in the dth dimension) indicates that the dimension is distributed across p processors. Alternatively, p processors can be divided into m groups, with p_i being the number of processors in the ith group. Here, we have $p = p_1 + p_2 + ... + p_m$. Then, an I/O access pattern such as $[p_1][p_2]....[p_i]....[p_m]$ indicates that the ith dimension is shared by p_i processors. It is also possible to adopt hybrid access patterns such as $[*][*]....[*][p_i][p_{i+1}][p_{i+2}]....[p_{m-1}][p_m]$, where only the dimensions i and higher are distributed across processors. For a two-dimensional array case with p processors, $[p][*]$, $[*][p]$, and $[p_1][p_2]$ denote row-wise, column-wise, and block-wise access patterns, respectively, where $p = p_1 + p_2$.

It should be noted that these access pattern representations come directly from parallelization. Consider, for example, the code fragment in Figure 1 once more, asssuming that we have p processors. If the `i` loop is parallelized using all p processors and the `j` loop is run sequentially, we obtain an access pattern of $[p][*]$ for the reference in the first nest. In the second nest, on the other hand, we have two access patterns, $[*][p]$ and $[p][*]$, assuming that only the `k` loop is parallelized.

Our strategy for determining the most dominant access pattern is based on the concept of *pattern weight.* The weight of a pattern is the number of times it occurs during execution. Note that different references can exhibit the same pattern, and can, therefore, contribute to the weight of the same pattern. To find the contribution of each reference, we first estimate the number of times each array reference will be touched in a typical run. To perform this estimation, our current approach uses profile data (where available) and also static analysis. Then, for each reference, we determine the access pattern that it contributes to, and finally, for each pattern, find the weight. We obtain the profile data used in this study as follows. The program being analyzed is in-

IEEE COMPUTER SOCIETY

strumented such that when it is run a file is created which contains the number of times each reference in the code is accessed during execution. When profile data is not available, we use a simple static analysis strategy which employs some form of symbolic manipulation. Specifically, our current strategy computes the number of loop iterations for each loop in the code and predicts the direction of 'if conditionals' within loops. These two pieces of information, namely, loop count estimation and if-direction prediction allows the compiler to symbolically (if the loop bounds and array sizes are not constant) calculate the number of times each reference will be touched during a typical execution. One point should be made clear. For each array, there might be more than one write statements in the code. These write statements are also treated as (normal) array accesses; that is, they contribute to access pattern.

3.2. Storage Form Detection

As mentioned earlier in the paper, to achieve the best I/O performance, the file layout and the file access pattern should be compatible. To see why this is so, consider the column-wise access pattern shown on the right portion of Figure 1. Assuming that we have p processors and that the array in question is $\mathtt{N} \times \mathtt{N}$ and is stored in file as row-major, each processor accesses an $\mathtt{N} \times \mathtt{N}/p$ subarray (column-block) of the array. Since the file layout is row-major, this access pattern means that each processor needs to access small subrows of size $\mathtt{N}/p$ (and $\mathtt{N}$ of them). It should be noted that each subrow should be accessed using a separate I/O call.[2] Previous research (e.g., [2, 15]) shows that initiating I/O call is the most costly part of a file access, consequently, such an access pattern will incur this most costly part $\mathtt{N}$ times. Let us now concentrate on a row-wise access pattern for the same row-major array. In this case, each processor accesses a $\mathtt{N}/p \times \mathtt{N}$ row-block portion of the array. Note the array elements accessed by a given processor are consucutive in the file space (as the array is row-major). So, each processor can read its elements using minimum number of I/O calls (restricted only by the number of maximum elements that can be read by a single I/O call; this is typically a file system-defined parameter). This small example illustrates the importance of compatibility between the access pattern and the storage form.

Since our access pattern detection strategy determines the most dominant access pattern, we set the storage form to the most dominant access pattern. Specifically, we define different storage forms analogous to access pattern forms defined in the previous subsection. If we consider only linear layouts, an m-dimensional disk-resident array can be stored in $m!$ forms, each corresponding to nested traversal of axes in a pre-determined order. For instance, a three-dimensional array can be stored in six different forms, the well-know row-major and column-major layouts being only two of them. Similarly, a two-dimensional array can have only two forms, row-major and column-major. Non-linear layouts (e.g., blocked layouts), on the other hand, divide a given array into chunks. The elements that belong to the same chunk are stored in file consecutively. Note that, for a given chunk, we can adopt any linear layout for storing its elements. Similarly, the relative order of chunks with respect to each other might be row-major, column-major, or higher-dimensional equivalents of them, depending on the dimensionality of the array.

For a given m-dimensional array, linear layouts can be represented using permutation vectors in which each dimension is given a unique number i ($1 \leq i \leq m$). If, for two dimensions with their numbers i and j, $i < j$, this indicates that the elements in the second dimension are visited more frequently than those of the first dimension. For example, 1234 represents a four-dimensional row-major array whereas 321 represents a three-dimensional column-major array. The non-linear layouts, on the other hand, are represented using a notation very similar to the access pattern representation. Specifically, a layout form such as $[p_1][p_2]....[p_i]....[p_m]$ indicates that the first dimension is divided into p_1 portions, the second dimension is divided into p_2 portion, and so on. Such a division generates $p_1 \times p_2 \times .... \times p_{m-1} \times p_m$ chunks. It should also be noted that our non-linear layout representation form subsumes linear layouts. For example, an access pattern of the form $[*][*][*]....[*][k][*]....[*]$, where $[k]$ appears in the dth dimension, indicates that the dth dimension is divided into k chunks, while the other dimensions are not divided. As a simpler example, $[\mathtt{N}][*]$ (for an $\mathtt{N}$-dimensional array) indicates a row-major layout. Since there is one-to-one correspondence between this (non-linear) storage form representation and the access pattern representation presented earlier, our compiler employes this representation. Let us assume that the dominant access pattern is $[p_1][p_2]....[p_i]....[p_{m-1}][p_m]$, where each p_i can be a $[*]$ corresponding to a dimension not distributed, or a positive integer value which gives the number of processors across which the dimension is distributed. We set the storage form of the array to this access pattern. For example, a dominant access pattern such as $[p][*]$ (row-wise) leads to a storage form $[p][*]$. This storage form implies that the second dimension is not divided, but the first dimension is divided into p chunks. Note that if the elements belonging to the same chunk are stored in row-major form, this storage form is the same as $[k][*]$, where $k \geq p$ and p divides k evenly. Note that, if the storage form is determined as explained above, all array references with the dominant access pattern

[2] An alternative strategy which reads the entire array and sieves out unwanted elements is not a very reasonable approach in I/O-intensive applications due to large array sizes.

IEEE COMPUTER SOCIETY

access their data (from the array in question) using minimum number of I/O calls. However, as discussed below, this may not be the case for remaining references.

3.3. Optimizing Non-dominant Patterns

Since, for a given array, there might be access patterns different from the most dominant one, we need to handle these (non-dominant) patterns as well. To do this, our compiler employs an I/O optimization technique, called *collective I/O* [10, 24]. As discussed in [24], in many parallel applications, although each process may need to access several noncontiguous parts of a given file (which holds an array), the requests coming from different processors are often interleaved, and may together span a large contiguous chunk. If each processor accesses its portion of data (i.e., data required to perform its computation) directly, then numerous I/O calls, each for a small sized contiguous portion need to be issued. For example, accessing a column-block of a row-major array results in a such a scenario. Instead, if the processors co-operate and negatiate a global file access pattern, a much better I/O performance can be attained. The idea is then to merge the requests from different processors and to service (collectively) the merged request. While collective I/O can be implemented at the disk [16], server [6], and client [10] levels, in this work, we employ a client-side implementation called *two-phase I/O* [10].

In two-phase I/O, the processors first access the file using an access pattern which is most compatible with the storage form of the file. This is the first phase and is called layout-efficient I/O. After that, in the second phase (called data redistribution), the processors engage in many-to-many communication so that each processor receives its portion (of the array). Consider, as an example, a two-dimensional array stored in a file in row-major fashion (say $[4][*]$) and accessed (e.g., read) using an access pattern of $[*][4]$ (i.e., a column-wise access). To prevent each processor from issuing many I/O calls, the two-phase I/O proceeds as depicted in Figure 2. In the first-phase, the processors read the file using the access pattern $[4][*]$. In the second phase, they perform a collective communication to re-distribute the data between them so as to obtain the final (desired) access pattern $[*][4]$. While collective I/O incurs an extra communication cost (whose magnitude depends on the number of processors involved, on the amount of data that needs to be communicated, and on the existence of collective communication primitives in the underlying file system), since the per item I/O cost is, in general, much higher than the per item communication cost in today's parallel machines, we can expect large performance gains.

Although we have discussed the overhead due to collective I/O in terms of inter-processor communication, in shared-memory parallel architectures, the data re-

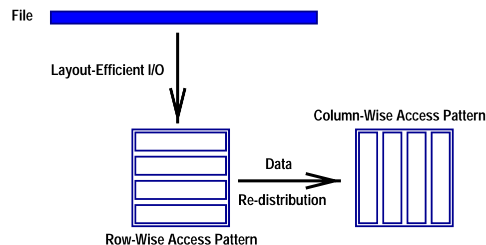

Figure 2. Two-phase I/O.

distribution step (the second phase) can be implemented using non-local memory accesses. To optimize these accesses, data layout transformations can be employed.

4. Conclusions

Optimizing I/O performance of scientific codes is an important problem as data set sizes keep getting larger and larger. While continued improvements in I/O hardware (e.g., disks with fast access times) are very important for achieving large reductions in time spent in I/O, one also needs to consider addressing the I/O problem at the software level. This paper has presented a compiler-directed I/O optimization strategy based on access pattern and storage pattern detection. By using collective I/O only when necessary, our three-step approach implements the rule that the most beneficial storage layout is the one that minimizes the I/O latency of future data accesses.

References

[1] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: a framework for optimizing parallel I/O. In Proc. *Scalable Parallel Libraries Conference,* 1994.

[2] R. Bordawekar. *Techniques for Compiling I/O Intensive Parallel Programs*. Ph.D. Dissertation, ECE Department, Syracuse University, Syracuse, NY, May 1996.

[3] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In Proc. *the ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–10, July 1995.

[4] P. Brezany, T. A. Muck, and E. Schikuta. Language, compiler and parallel database support for I/O intensive applications, In Proc. *High Performance Computing and Networking*, Milano, Italy, May 1995.

[5] L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani. Efficient organization and access of multidimensional datasets on tertiary storage systems. *Information Systems Journal,* 20(2): 155–183 (1995).

[6] Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel I/O performance optimization in panda. In Proc.

1530-2075/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

the 10th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 108–118, Puerto Vallarta, Mexico, June 1998.

[7] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: a parallel file I/O interface for MPI, *IBM Yorktown Heights Research Report, RC 19841,* November 21, 1994.

[8] P. F. Corbett, D. G. Feitelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. D. Herr, J. Kavaky, T.R. Morgan, and A. Zlotek. Parallel file Systems for the IBM SP computers. *IBM Systems Journal,* Vol.34, No.2 (June 1995), pp. 222–248.

[9] T. Cormen and A. Colvin. ViC*: a preprocessor for virtual-memory C*. *Dartmouth College Computer Science Technical Report PCS-TR94-243,* November 1994.

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

[11] J. del Rosario and A. Choudhary. High performance I/O for parallel computers: problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[12] R. Ferreira, G. Agrawal, and J. Saltz. Compiling object-oriented data intensive applications. In Proc. *the International Conference on Supercomputing,* pp. 11–21, May 2000.

[13] J. V. Huber, Jr., C. L. Elford, D. A. Reed, A. A. Chien, and David S. Blumenthal. PPFS: a high-performance portable parallel file system. In Proc. *International Conference on Supercomputing,* July 1995.

[14] M. Kandemir. A collective I/O scheme based on compiler analysis. In Proc. *the Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers,* University of Rochester, Rochester, NY, USA, May 25–27, 2000.

[15] M. Kandemir, A. Choudhary, and R. Bordawekar. Data access reorganization in compiling out-of-core data parallel programs on distributed memory machines. In Proc. *International Parallel Processing Symposium,* Geneva, Switzerland, April 1997.

[16] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In Proc. *the 1994 Symposium on Operating Systems Design and Implementation,* pages 61–74, November, 1994.

[17] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In Proc. *the Second Symposium on Operating Systems Design and Implementation,* pages 3–17, October 1996.

[18] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In Proc. *10th ACM International Conference on Supercomputing,* 1996.

[19] J. Nieplocha, I. Foster, and R. A. Kendall. ChemIO: high-performance parallel I/O for computational chemistry applications. *International Journal of Supercomputer Applications and High Performance Computing,* 1998.

[20] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. In Proc. *the IEEE Symposium on The Frontiers of Massively Parallel Computation*, pages 110–118, February 1995.

[21] B. Rullman. Paragon Parallel File System. *External Product Specification,* Intel Supercomputer Systems Division.

[22] R. Thakur, A. N. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. PASSION: optimized I/O for parallel applications. *IEEE Computer,* 29(6): 70–78, June 1996.

[23] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In Proc. *the Sixth Workshop on I/O in Parallel and Distributed Systems,* May 1999, pp. 23–32.

[24] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In Proc. *the 7th Symposium on the Frontiers of Massively Parallel Computation,* February 1999, pp. 182–189.

[25] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, CA, 1996.

IEEE COMPUTER SOCIETY