I/O in Parallel Applications: The Weakest Link

Rajeev Thakur *Ewing Lusk* *William Gropp*
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{thakur, lusk, gropp} @mcs.anl.gov

Abstract

Parallel computers are increasingly being used to run large-scale applications that also have huge I/O requirements. However, many applications obtain poor I/O performance on modern parallel machines. This special issue of IJSA contains papers that describe the I/O requirements and the techniques used to perform I/O in real parallel applications. We first explain how the I/O application program interface (API) plays a critical role in enabling such applications to achieve high I/O performance. We describe how the commonly used Unix I/O interface is inappropriate for parallel I/O and how an explicitly parallel API with support for collective I/O can help the underlying I/O hardware and software perform I/O efficiently. We then describe MPI-IO, a recently defined, standard, portable API specifically designed for high-performance parallel I/O. We conclude with an overview of the papers in this special issue.

To appear in *The International Journal of Supercomputer Applications and High Performance Computing*, Special Issue on I/O in Parallel Applications, 1998.

1 Introduction

In recent years great advances have been made in the CPU and the communication performance of parallel computers. Similar advances have not been made, however, in the input/output (I/O) performance of parallel machines, as I/O has long been a neglected area. Although parallel computers with peak performance of 1 Tflops/sec or more are available, real applications running on parallel machines usually achieve I/O bandwidths of at most a few hundred Mbytes/sec. In fact, many applications achieve less than 10 Mbytes/sec [4].

As parallel computers get faster, scientists are increasingly using parallel computers to solve problems that require a large amount of computing power. Most of these applications also need to perform I/O for a number of reasons, such as reading initial data, writing the results, checkpointing, out-of-core data sets, scratch files for temporary storage, and visualization. (See [14, 1, 6] for a list of many such applications.) Since I/O is slow, the I/O speed, and not the CPU or communication speed, is often the bottleneck for such applications. For parallel computers to be truly usable for solving real, large-scale problems, the I/O performance must be scalable and balanced with respect to the CPU and communication performance of the system.

In this article, we first briefly discuss the hardware and software support for parallel I/O on modern machines. We explain how an inappropriate application program interface (API) is often the cause of poor I/O performance in applications and how an explicitly parallel API with support for collective I/O can help the underlying I/O hardware and software perform I/O efficiently. We then describe MPI-IO, a recently defined, standard, portable API specifically designed for high-performance parallel I/O. Finally, we give an overview of each paper in this special issue. For the most part, this article as well as the other papers in this issue deal with the type of parallel I/O commonly seen in high-end scientific computing. In the commercial area, where parallelism often comes about through processing of independent transactions, the issues may be different from the ones addressed here.

2 Parallel I/O Infrastructure

Parallel I/O can be defined as multiple processes of a parallel program making concurrent I/O requests to the file system. Most parallel machines are provided with some hardware and software support for parallel I/O.

Distributed-memory parallel machines, such as the IBM SP and Intel Paragon, have a set of I/O nodes, each connected to one or more disks or RAIDs. Usually, although not always, the I/O nodes are dedicated for I/O, and no compute jobs are run on them. The I/O nodes function as

servers for the parallel file system. The parallel file system typically stripes files across multiple I/O nodes or disks: A file is divided into a number of smaller units called striping units, and the striping units are assigned to disks in a round-robin manner. File striping provides higher bandwidth and enables multiple processes to access distinct portions of a file concurrently. In some machines, the compute nodes also have local disks of their own that are not directly accessible from other nodes. These disks are used to store files local to each processor.

Shared-memory parallel machines, such as the SGI Origin 2000 and HP/Convex Exemplar, do not have separate I/O nodes. The operating system schedules the file-system server on the compute nodes. However, these machines do have multiple disks and a file system that stripes files across the disks.

For a good discussion of issues related to parallel I/O systems, see [7, 15].

3 Application Program Interface

The application program interface (API) plays a critical role in enabling the user to express I/O operations conveniently and also in conveying sufficient information about user-level access patterns to the I/O system so that the system can perform I/O efficiently.

The Unix I/O interface is the most commonly used interface for parallel I/O at present. However, it was designed mainly for uniprocessor file systems and for access patterns commonly found in uniprocess programs. Accordingly, it allows the user to access only a single chunk of data at a time. It has no notion of collective I/O requests from multiple processes. (POSIX [12] does define a function called `lio_listio` that accepts a list of requests, but it is not supported on all machines, and it is not collective.) One can use the Unix I/O interface in parallel programs; each process can make Unix I/O calls on its own, independent of other processes. However, as we explain below, using this interface often turns out to be very inefficient. The main reason is that the access patterns in parallel programs are quite different from those in uniprocess programs [21, 4, 2, 26]. In parallel programs, each process may need to access a noncontiguous data set. In many cases, the accesses of different processes may be interleaved in the file, and together they may span large, contiguous portions of the file. With the Unix I/O interface, the programmer has no means of conveying the "big picture" of the access pattern to the I/O system. Each process must seek to a particular location in the file, read or write a small portion of data, then seek to the next noncontiguous location, read or write a small portion of data, and so on. The result is that each process makes hundreds or thousands of requests for small pieces of data to the file system.

The example in Figure 1 illustrates this point. The figure shows an access pattern commonly

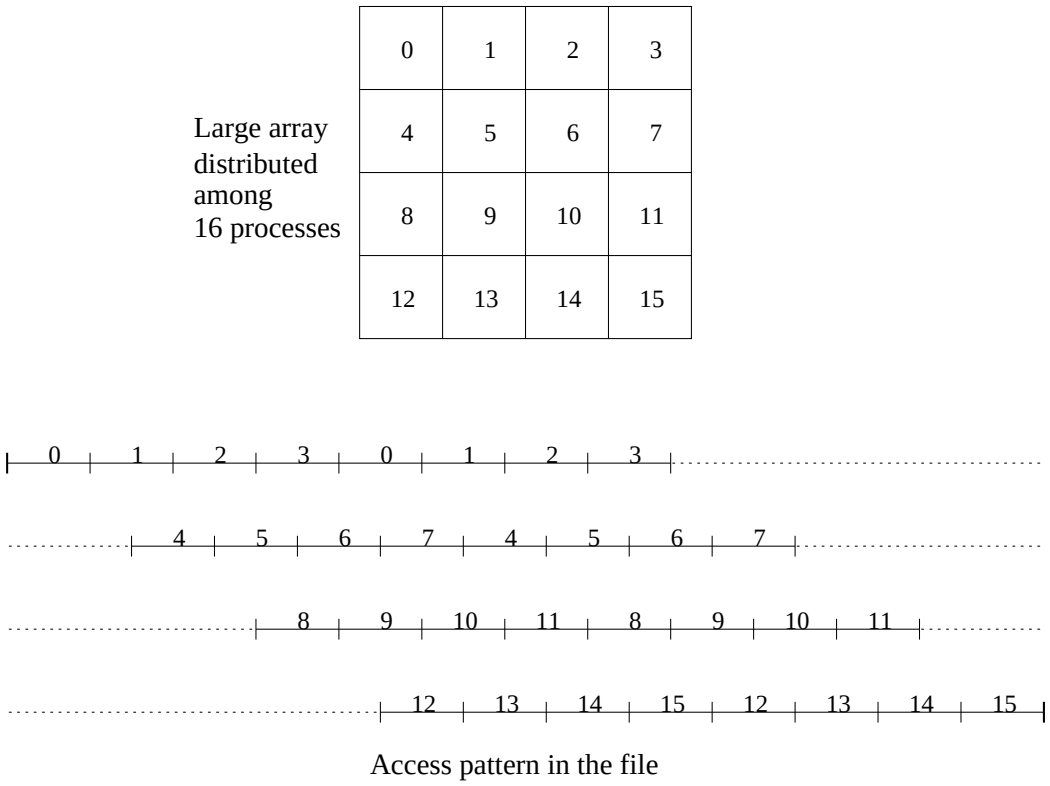

Figure 1: Example access pattern in parallel applications

found in parallel applications. A two-dimensional array is distributed among 16 processes in a (block, block) fashion. The array is stored in a file corresponding to the global array in row-major order, and each process needs to read its local array from the file. The data distribution among processes and the array storage order in the file are such that the file contains the first row of the local array of process 0, followed by the first row of the local array of process 1, the first row of the local array of process 2, the first row of the local array of process 3, then the second row of the local array of process 0, the second row of the local array of process 1, and so on. In other words, the local array of each process is not located contiguously in the file. To read its local array by using a Unix-like API, each process must seek to the appropriate location in the file, read one row, seek to the next row, read that row, and so on. Each process must make as many read requests as the number of rows in its local array. If the array is large, the file system may receive thousands of read requests.

Parallel file systems are designed to handle large I/O requests, and they perform poorly when bombarded with numerous small requests. If the API is able to specify the noncontiguous accesses of each process as well as the group of processes making such requests, the I/O system can access data efficiently by using a technique called collective I/O [5, 27, 16, 24]. A collective I/O implementation

tries to combine the noncontiguous requests of multiple processes into larger contiguous requests and access data in large chunks. Numerous studies have shown that collective I/O can improve performance significantly [5, 27, 16, 24]. However, collective I/O cannot be done with the Unix API.

Over the past few years, many research parallel file systems and I/O libraries have been developed that perform various optimizations, including collective I/O [28, 11, 20, 17, 3, 10, 25, 9, 19]. Each of these, however, has a different API with varying degrees of portability and generality. The only standard, portable API that has been available on all machines is the Unix API. Therefore, most users write applications for the Unix API and get bad performance for reasons explained above. Clearly, a single, standard, portable API designed specifically for parallel I/O is needed, together with high-performance implementations of it on all machines.

Fortunately, there is now such an API, namely MPI-IO, the I/O chapter in MPI-2 [18]. MPI-IO has been designed based on experience with various existing APIs as well as knowledge of the I/O access patterns in parallel applications. MPI-IO can be considered as Unix I/O plus many features specifically intended for portable, high-performance parallel I/O. These additional features include support for noncontiguous accesses in memory and file, collective I/O, nonblocking I/O, standard file data representation, and user-defined file data representation. MPI-IO provides a mechanism for users to provide hints applicable to a particular implementation or I/O environment (e.g., number of disks, striping unit, access pattern). MPI-IO also allows the user to select either an atomic or a nonatomic file consistency mode. The default consistency mode is nonatomic, which enables an implementation to perform certain optimizations that are not possible with the more stringent atomic mode. The nonatomic mode is intended for the most common case where the accesses of different processes do not overlap. If accesses overlap, the atomic mode should be used.

Porting applications from Unix I/O to MPI-IO is easy, because MPI-IO provides functions that are equivalent to those in Unix I/O. For better performance, however, the special features of MPI-IO must be used. Many implementations of MPI-IO are in progress [30, 8, 13, 23, 22], and most vendors of parallel machines plan to provide MPI-IO as part of their MPI-2 product.

We ourselves are developing a portable MPI-IO implementation called ROMIO. ROMIO 1.0.0 is freely available from `http://www.mcs.anl.gov/home/thakur/romio` and works on most parallel computers and networks of workstations. ROMIO is optimized for noncontiguous accesses and collective I/O. We have seen impressive performance results with ROMIO's collective I/O implementation. For example, Figure 2 shows the I/O bandwidths obtained on the Intel Paragon at Caltech for an astrophysics application template, which has an access pattern similar to that in Figure 1. (This application is described in detail in [29].) A three-dimensional array is block-distributed in

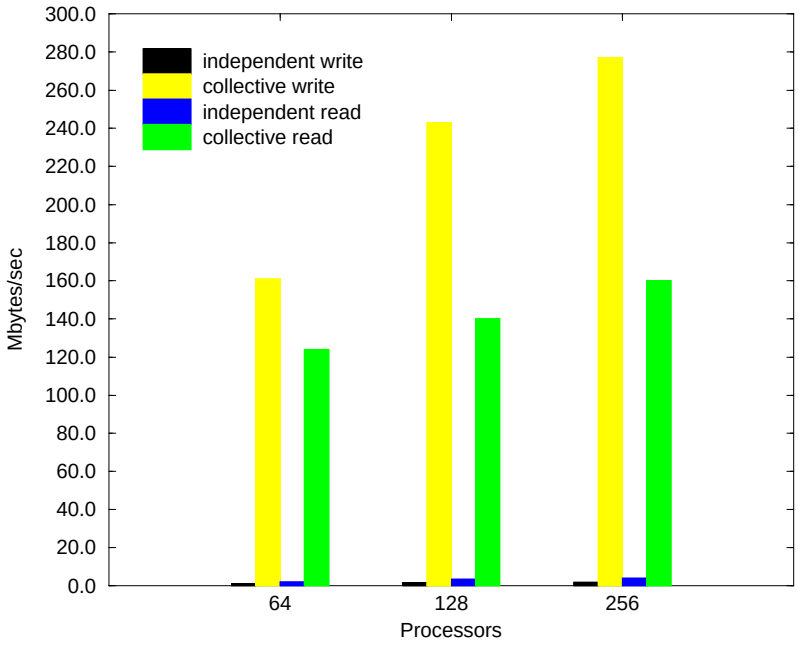

Figure 2: Independent versus collective I/O with MPI-IO on the Intel Paragon at Caltech for a 3D astrophysics application template

all three dimensions and must be accessed (read/written) from a file containing the global array in column-major order. The results shown are for an array of size $512 \times 512 \times 1024$. With Unix-like independent I/O, the bandwidths obtained were less than 5 Mbytes/sec in all cases, whereas with collective I/O, the bandwidths were in the range of 120–280 Mbytes/sec. The scalability of the collective I/O performance is limited by the amount of I/O hardware on the machine.

4 This Special Issue

The efficacy of any parallel I/O system can be meaningfully judged only by its performance with real applications. Although many papers exist in the literature about parallel I/O systems, comparatively little information exists about applications that use parallel I/O. This special issue aims to increase the understanding of the nature of I/O in real parallel applications.

The authors of the papers in this issue have used various methods to meet their I/O needs. Some have used Unix I/O, others have used existing I/O libraries, and some have developed their own libraries and techniques. Note that the MPI-IO interface was still being defined at the time

most of this work was done, and MPI-IO implementations were not readily available.

Below is a brief overview of each paper in this issue.

- "Efficient Parallel I/O in Seismic Imaging" by Ron Oldfield, David Womble, and Curtis Ober. This paper describes a seismic-imaging application called Salvo. Salvo is written with MPI and runs on many different machines. It uses a separate I/O partition, consisting of a set of extra compute nodes allocated at run-time, to perform all I/O. I/O requests from the compute partition are communicated to the I/O partition, and the I/O partition accesses data from the file system. The authors have developed an analytical model for estimating the I/O and compute times, and they present performance results on an Intel Paragon.

- "Characterization of I/O Requirements in a Massively Parallel Shelf Sea Model" by P. Lockey, R. Proctor, and I. James. This paper describes the I/O requirements of an application that models continental shelf sea regions. The application is written in Fortran and uses MPI. All parallel I/O detail is hidden inside high-level routines developed by the authors. These routines provide a single, portable I/O interface across machines with different I/O architectures and file systems. The I/O in the application consists of reads and writes of three-dimensional distributed arrays. The authors present an analytical cost model and performance results on a Cray T3D.

- "An Experimental Study to Analyze and Optimize Hartree-Fock Application's I/O with PASSION" by Meenakshi Kandaswamy, Mahmut Kandemir, Alok Choudhary, and David Bernholdt. This paper describes in detail the I/O behavior of a computational chemistry application that uses the Hartree-Fock method. The authors first used the original code with its Fortran I/O calls. They then studied the I/O behavior by replacing the Fortran I/O calls with calls to the PASSION I/O library. They discuss the impact of PASSION's I/O optimizations and various application-related and system-related parameters on the performance of this application. All experiments were performed on an Intel Paragon.

- "A Comparison of Logical and Physical Parallel I/O Patterns" by Huseyin Simitci and Daniel Reed. This paper compares the logical I/O requests of an application with the corresponding physical I/O that actually takes place at the disk level. The authors use some synthetic benchmarks and one computational chemistry application called MESSKIT. They performed all experiments on an Intel Paragon. They found that the physical I/O patterns induced by application requests are affected greatly by the file striping mechanisms, file system policies, and disk hardware attributes.

- "A Study of I/O in a Parallel Finite Element Groundwater Transport Code" by David Mackay, G. Mahinthakumar, and Ed D'Azevedo. This paper describes the I/O in a parallel finite-element groundwater transport code. The authors compare three different strategies for performing I/O in this code: having a single processor do all the I/O, using variations of vendor-specific I/O extensions, and using a library they developed called EDONIO. They present performance results on multiple machines: Intel Paragon, IBM SP, HP/Convex Exemplar, SGI/Cray Origin 2000, and SGI Power Challenge. They also performed some preliminary experiments using MPI-IO and report that the performance with MPI-IO was comparable to that with the EDONIO library.

- "ChemIO: High-Performance Parallel I/O for Computational Chemistry Applications" by Jarek Nieplocha, Ian Foster, and Rick Kendall. This paper describes the I/O requirements of some computational chemistry applications, such as Hartree-Fock, MRCI, RI-SCF, and RI-MP2. The authors have developed an I/O library, called ChemIO, that is intended specifically to meet the I/O requirements of computational chemistry applications. ChemIO supports three different I/O models: disk resident arrays, exclusive access files, and shared files. The authors present performance results with ChemIO on an Intel Paragon and an IBM SP.

- "Parallel Run-Length Encoding (RLE) Compression—Reducing I/O in Dynamic Environmental Simulations" by G. Davis, L. Lau, R. Young, F. Duncalfe, and L. Brebber. This paper describes a climate-modeling application that generates so much data ($\approx$60 Tbytes) that data reduction via compression is essential. The authors have developed a compression algorithm, based on run-length encoding, that exploits the special gather-scatter hardware and multiple processors on Cray parallel vector machines.

5 Conclusions

The papers in this special issue represent a small sample, even within scientific computing, of the applications that need large-scale I/O capabilities. They exhibit a variety of approaches to meeting their I/O needs. MPI-IO and the implementations of it that we expect to see in the next few years show promise in contributing to solve I/O performance and portability problems. However, the effectiveness of MPI-IO, or any parallel I/O system, must be based on its performance with real applications. We hope that the application studies in this special issue provide a useful step in the long-term evaluation of scalable I/O solutions.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; and by the Scalable I/O Initiative, a multiagency project funded by the Defense Advanced Research Projects Agency (contract number DABT63-94-C-0049), the Department of Energy, the National Aeronautics and Space Administration, and the National Science Foundation.

References

[1] Applications Working Group of the Scalable I/O Initiative. Preliminary Survey of I/O Intensive Applications. Scalable I/O Initiative Working Paper Number 1. On the World-Wide Web at `http://www.cacr.caltech.edu/SIO/SIO_apps.ps`, 1994.

[2] S. Baylor and C. Wu. Parallel I/O Workload Characteristics Using Vesta. In R. Jain, J. Werth, and J. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 7, pages 167–185. Kluwer Academic Publishers, 1996.

[3] P. Corbett et al. Proposal for a Common Parallel File System Programming Interface, Version 0.60. On the World-Wide Web at `http://www.cs.princeton.edu/sio`, June 1996.

[4] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input-Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[5] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, April 1993. Also published in Computer Architecture News, 21(5):31–38, December 1993.

[6] J. del Rosario and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *Computer*, 27(3):59–68, March 1994.

[7] D. Feitelson, P. Corbett, S. Baylor, and Y. Hsu. Parallel I/O Subsystems in Massively Parallel Supercomputers. *IEEE Parallel and Distributed Technology*, 3(3):33–47, Fall 1995.

[8] S. Fineberg, P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO—A Portable Implementation of MPI-IO. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 188–195. IEEE Computer Society Press, October 1996.

[9] N. Galbreath, W. Gropp, and D. Levine. Applications-Driven Parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471. IEEE Computer Society Press, November 1993.

[10] G. Gibson, D. Stodolsky, P. Chang, W. Courtwright II, C. Demetriou, E. Ginting, M. Holland, Q. Ma, L. Neal, R. Patterson, J. Su, R. Youssef, and J. Zelenka. The Scotch Parallel Storage Systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410. IEEE Computer Society Press, Spring 1995.

[11] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.

[12] IEEE/ANSI Std. 1003.1. Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) [C Language], 1996 edition.

[13] T. Jones, R. Mark, J. Martin, J. May, E. Pierce, and L. Stanberry. An MPI-IO interface to HPSS. In *Proceedings of the Fifth NASA Goddard conference on Mass Storage Systems*, pages I:37–50, September 1996.

[14] D. Kotz. Applications of Parallel I/O. Technical Report PCS-TR96-297, Dept. of Computer Science, Dartmouth College, October 1996. Release 1. Available on the World-Wide Web at `http://www.cs.dartmouth.edu/reports/abstracts/TR96-297`.

[15] D. Kotz. Introduction to Multiprocessor I/O Architecture. In R. Jain, J. Werth, and J. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 4, pages 97–123. Kluwer Academic Publishers, 1996.

[16] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[17] O. Krieger and M. Stumm. HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions. In *Proceedings of Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 95–108. ACM Press, May 1996.

[18] Message-Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. July 1997. On the World-Wide Web at `http://www.mpi-forum.org/docs/docs.html`.

[19] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O Library for Out-of-Core Computations. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 196–204. IEEE Computer Society Press, October 1996.

[20] N. Nieuwejaar and D. Kotz. The Galley Parallel File System. *Parallel Computing*, 23(4):447–476, June 1997.

[21] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[22] J. Prost. MPI-IO/PIOFS. World-Wide Web page at `http://www.research.ibm.com/people/p/prost/sections/mpiio.html`, 1996.

[23] Darren Sanders, Yoonho Park, and Maciej Brodowicz. Implementation and Performance of MPI-IO File Access Using MPI Datatypes. Technical Report UH-CS-96-12, University of Houston, November 1996.

[24] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[25] K. Seamons and M. Winslett. Multidimensional Array I/O in Panda 1.0. *The Journal of Supercomputing*, 10(2):191–211, 1996.

[26] E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 49–59. IEEE Computer Society Press, 1996.

[27] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[28] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for Parallel Applications. *Computer*, 29(6):70–78, June 1996.

[29] R. Thakur, W. Gropp, and E. Lusk. An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application. In *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation (ACPC) with Special Emphasis on Parallel Databases and Parallel I/O*, pages 24–35. Lecture Notes in Computer Science 1127. Springer-Verlag., September 1996.

[30] R. Thakur, E. Lusk, and W. Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.