

Optimizing Collective I/O Performance on Parallel Computers: A Multisystem Study

Ying Chen

Department of Computer Science
University of Illinois
Urbana, IL 61801
ying@cs.uiuc.edu

Jarek Nieplocha

Environmental Molecular Sciences Laboratory
Pacific Northwest National Laboratory
Richland, WA 99352
j_nieplocha@pnl.gov

Ian Foster

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
foster@mcs.anl.gov

Marianne Winslett

Department of Computer Science
University of Illinois
Urbana, IL 61801
winslett@cs.uiuc.edu

Abstract

While individual parallel I/O systems can incorporate sophisticated techniques and achieve impressive performance in particular situations, researchers as yet have only limited understanding of the impact of various design decisions or of the techniques required for performance robustness. One remedy is to perform detailed comparative studies of different I/O libraries. In this paper, we describe such a study for the Disk Resident Array and Panda libraries, both designed to support high-performance I/O for arrays. While the two systems have many similarities, their designs and implementations are based on different assumptions and target different applications. We base our study on two I/O structures commonly encountered in scientific applications: the collective read/write of an entire array and the collective read/write of an arbitrary array section. Experiments are performed on two parallel file systems (IBM PIOFS and Intel PFS) and one commodity Unix file system (AIX JFS). Our results yield insights into

Research at UIUC was supported by by NASA under grant NAGW 4244 and NCC5 106. Work at ANL 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 multi-agency project funded by DARPA, DOE, NASA, and NSF. Work at PNNL was supported by the High Performance Computing and Communications Program of the Office of Computational and Technology Research, U.S. Department of Energy under contract DE-AC066-76RLO1830 with Battelle Memorial Institute which operates the Pacific Northwest National Laboratory. Computing facilities were provided by NASA Ames Numerical Aeronautic Simulation (NAS) Division, Argonne National Laboratory, and California Institute of Technology.

the major parameters determining the performance of array-oriented collective I/O.

1 Introduction

Poor I/O performance is a significant obstacle to the widespread use of parallel computers. Parallel I/O libraries represent one important direction for research intended to overcome this obstacle. A parallel I/O library implements a mapping from application-level I/O requests to low-level requests to the underlying file system. High-performance execution requires the library to be able to generate efficient mappings with little runtime overhead. Portability and usability demand that the strategies used to determine these mappings be effective over diverse platforms and I/O patterns. That is, we want the I/O library to be *performance robust*.

Scientific applications often operate primarily on array-oriented data. Hence, many parallel I/O libraries focus on array-oriented I/O operations, in which a program reads or writes a contiguous section of a multidimensional array (e.g., [4, 11, 13, 14]). Despite the apparent commonality of purpose, however, we see considerable variation in the techniques used to structure I/O operations, and only limited understanding of which techniques are superior. These problems arise in part because the developers of different libraries focus on different applications and execution environments. The lack of standard benchmarks and comparative studies is another obstacle to understanding.

To improve understanding of parallel I/O issues, we have conducted a detailed study and comparison of two array-oriented parallel I/O systems, Disk Resident Arrays [10] (DRA: developed at Pacific Northwest National Laboratory and Argonne National Laboratory) and Panda [14] (from the University of Illinois). Each system has been optimized for applications and execution environments of interest. Hence, a careful study of their performance on appropriately chosen benchmark problems can reveal how library design decisions affect

performance.

Our comparative study focuses on collective I/O operations, in which all processes in a parallel computation participate in a high-level I/O operation to transfer all or part of a distributed array between main memory and secondary storage. In Section 2, we compare the implementation techniques used in DRA and Panda and identify key performance parameters. Then, in Section 3 we study the effectiveness of these implementation techniques and the impact of different performance parameter settings. Our focus is on two common I/O patterns: reading/writing an entire array and reading/writing an array section. Experiments are performed on three platforms and file systems: the Parallel File System (PFS) on the Intel Paragon, the PIOFS parallel file system on the IBM SP2, and the AIX JFS commodity Unix file system also on the IBM SP2. Section 4 discusses related work, and Section 5 concludes the paper and outlines the future work.

2 DRA and Panda Design

Scientific applications frequently have a single-program multiple-data structure, in which each processor executes more or less the same program. These applications often operate on regular, multidimensional arrays distributed regularly over multiple processors. Often, many processors collectively engage in a conceptually simple request, such as creating an entire distributed array. If care is not taken, a conceptually simple request can result in a long and inefficient sequence of low-level operations.

The DRA and Panda libraries overcome this problem by providing high-level interfaces that allow programs to request that entire distributed arrays (or sections of such arrays) be transferred between memory and disk. These interfaces are easy to use and can facilitate optimization, since system developers can use specialized techniques below the level of the interface. High-level interfaces also enhance portability. The DRA and Panda interfaces differ primarily in terms of the types of data structures on which they operate. DRA provides functions for reading and writing data structures called global arrays, constructed with the Global Array (GA) library [11, 12]. DRA maintains array objects (called “disk resident arrays”) on disk and supports data transfer between global arrays and disk resident arrays. Panda provides functions for reading and writing user-constructed distributed arrays with HPF-style BLOCK, CYCLIC and * distributions [14].

We identify key commonalities and differences in the DRA and Panda library implementations below.

2.1 DRA and Panda Commonalities

In addition to collective, array-oriented interfaces, DRA and Panda share the following basic implementation strategies.

Chunked array layout. Rather than store array data on disk in a row-major or column-major order, both libraries decompose an array into chunks (subarrays) and store the array on disks in the chunked format.

These layouts increase data locality across multiple dimensions, typically reducing the number of disk accesses required to obtain a working set of data in memory.

Long sequential file accesses. To minimize startup time for file transfer, both DRA and Panda allocate large fixed-size buffers for I/O usage and, whenever possible, accumulate I/O requests in this buffer before issuing a file system call. The I/O buffer is typically a megabyte or more, many times larger than a disk block. In DRA, disk array chunks are sized to fit in available I/O buffer space. In Panda, the read or write unit is the disk array chunk, except that if a disk chunk is larger than the I/O buffer, the disk chunk is broken into smaller subchunks that can fit into the buffer. The buffer size can be set by user applications dynamically.

Read and write by chunk unit. Read and write operations that involve entire arrays are straightforward. Read and write operations on array sections, however, are more challenging. These can be classified as either aligned or nonaligned (Figure 1). In an aligned request, the boundaries of the requested section align exactly with disk array chunk boundaries. Otherwise, the request is nonaligned (Figure 1 (b)). Aligned accesses are handled in a similar fashion to operations on entire arrays. Nonaligned write requests are implemented via the strategy used in PASSION [15]. An entire disk array chunk is always written. When the requested section covers only parts of the disk array chunks, each I/O node brings the entire disk array chunk into its I/O buffers, overwrites the buffer with the overlapping sections, and then writes the entire disk array chunk back to disk. The same strategy is employed in reverse for reads. Hence, nonaligned requests must read and write more data than was requested. In addition, nonaligned write operations require extra reads to bring nonaligned disk array chunks into memory.

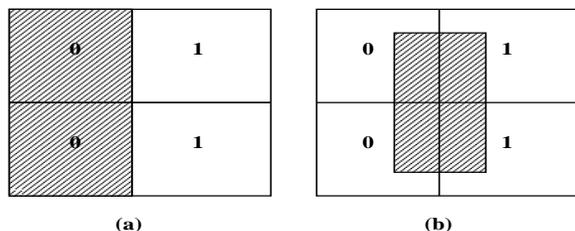


Figure 1: The difference between aligned and nonaligned requests. A 2-D array is broken into 4 disk array chunks and assigned to 2 I/O nodes. Shaded areas represent the requested array section. In (a), an aligned request asks for an array section that overlaps with 2 disk array chunks and the boundaries of the requested section align with the disk array chunk boundaries. In (b), a nonaligned section is requested, and the requested section overlaps with 4 disk array chunks. Thus, 4 disk array chunks have to read into memory, even though only portions of the 4 disk array chunks are needed by the application.

2.2 DRA and Panda Differences

Disk array distribution interface. Collective I/O operations transfer data from a distributed disk array to a distributed in-memory array. If the disk array and in-memory array are distributed in the same manner, an I/O library can avoid potentially expensive in-memory reorganizations.

DRA and Panda both provide programmer control of in-memory array distribution. In DRA, distribution is managed by the GA library and hence need not be specified in DRA calls. Panda I/O calls, however, must specify the in-memory array distribution.

DRA and Panda adopt different philosophies regarding programmer control of disk array distribution. DRA users provide hints that guide but do not direct disk array distribution. DRA uses these hints (which can indicate, for example, the shape and size of a typical I/O request) as well as information about the internal I/O buffer size and file system characteristics to choose a disk array distribution. In contrast, Panda users can control disk array distribution directly. When creating an array, the user can specify how the array should be decomposed into “chunks” in memory and on disk. The default is termed “natural chunking,” in which the disk array distribution is identical to the in-memory distribution.

The effectiveness of the DRA and Panda approaches depends on the relative sophistication of the DRA heuristics and the Panda user. In principle, DRA heuristics should be able to select distributions that provide good performance for the expected range of access patterns. On the other hand, the Panda interface allows the programmer to select a distribution that is known to be optimal in a particular situation.

Communication schemes. DRA uses the GA library’s array-oriented shared-memory operations [11], while Panda uses message passing for its internal communications.

2.3 Performance Parameters

For a particular workload, high performance can depend on appropriate choices being made for two classes of key performance parameters: (1). *Library configuration parameters*, including the number of I/O nodes used, the I/O buffer size on each I/O node, the communication mechanisms used, the underlying file system used, and the specific file system facilities used. (2). *Disk array distribution parameters*.

3 Performance Studies

To quantify the performance impact of various DRA and Panda implementation techniques and key performance parameters, we conducted a wide variety of experiments on three different file systems: PFS, PIOFS, state-of-the-art parallel file systems, supported on the Intel Paragon and IBM SP2, respectively; and AIX JFS, a standard sequential file system, which in the system configuration used for our experiments was supported on each node of an IBM SP2.

3.1 Methodology

Experiments were conducted on entire array and non-aligned array benchmarks. The entire array benchmark reads and writes arrays of at least 200 MB; the non-aligned array benchmark reads and writes a nonaligned section of a double-precision square array of at least 200 MB. We report the fraction of peak file system performance achieved by DRA and Panda for each set of experiments. On JFS and PFS, we report times averaged across ten runs. The experimental results are fairly consistent on these two platforms. We calculated the standard deviation for each experiment and the maximum standard deviation is 0.029 for JFS results and 0.036 for PFS results. For PIOFS, we report the maximum of ten experiments because PIOFS results had large variations. We are not certain of the reason for these variations, which arose despite considerable care to eliminate interference from other users in this shared file system.

We also conducted a separate set of experiments, discussed in Section 3.2, to determine the peak performance of the underlying file systems. Because of space limitations, we present only write results; read results were similar.

3.2 Validating the Effectiveness of DRA and Panda I/O Strategies

On each platform, we describe the experimental configuration, the techniques used to measure peak performance, and the results obtained (for the best settings tried) for DRA and Panda.

3.2.1 JFS Experiments

We performed our JFS experiments on the 144-node IBM SP2 at the Numerical Aeronautics Simulation (NAS) division of NASA Ames Research Center. Each node of this system is an RS/6000 Model 590 with a 66.7 MHz POWER2 multichip RISC processor, with a 256 KB data cache, and 128 MB memory. Each node also has a local disk of about 2 GB with 3 MB/s disk bandwidth and hence can serve as an I/O node. At the time of our experiments, each node ran version 4.1.1.3 of the AIX operating system and supported the JFS sequential file system. Nodes are connected via a high-performance switch that can deliver 40 MB/s peak bidirectional bandwidth. We used a local disk partition of approximately 1.2 GB on each I/O node to store the array data in all our tests; at the time of our experiments, this partition was 16–22% full. We used a 200 MB array in our experiments on this platform.

JFS Peak Performance. We measured peak performance by sequentially reading and writing a 512 MB file to disk on a single node with different request sizes (from 64 KB to 4 MB). The measured peak write and read rates are 2.756 MB/s and 2.85 MB/s per node, respectively. And the performance did not differ much with different request sizes.

JFS Results. The JFS implementations of DRA and Panda use identical disk layouts, storing local chunks in

a separate file on each I/O node. Both implementations use the same JFS functions, in particular calling the `fsync` function to flush file data to disks before a write request returns to the user application.

Figure 2 shows the performance of DRA and Panda for the entire array and nonaligned array benchmarks using a 512 KB buffer size. We also varied the number of compute nodes from 16 to 32 in our experiments. Both DRA and Panda used a (10×10) disk array distribution in the entire array benchmark and a (50×50) disk array distribution for nonaligned benchmarks¹. Both DRA and Panda are able to achieve more than 90% of the peak JFS throughput for the entire array benchmark and above 80% for the nonaligned array benchmark. Performance depends on the shape and size of the requested array section, the disk array distribution used, and the number of I/O nodes used (see Section 3.3). In Figure 2, the nonaligned array section chosen is the same for both DRA and Panda, and it overlaps with half of the entire array.

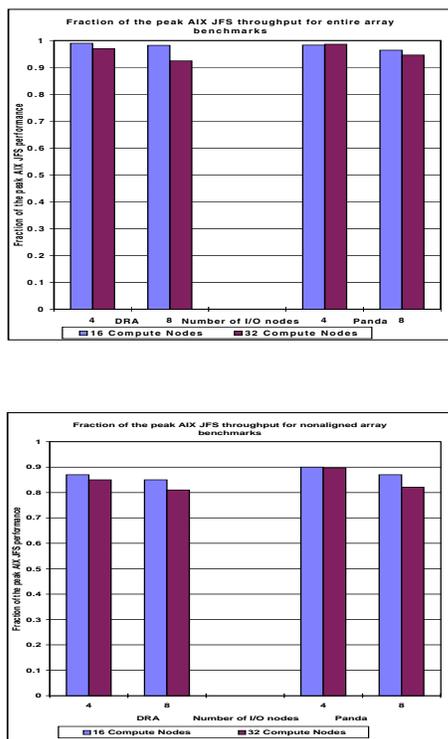


Figure 2: Performance of DRA and Panda on JFS for 200 MB entire (top) and nonaligned (bottom) array benchmarks using different numbers of compute nodes and different numbers of I/O nodes, and with a 512 KB buffer on each I/O node.

¹An $(a \times b)$ distribution breaks a two-dimensional array into a , roughly equal sized, chunks along one dimension, and into b chunks along the second dimension. The 50×50 distribution used in our experiments yielded chunks of approximately 320 KB.

3.2.2 Performance on IBM PIOFS

We performed our PIOFS experiments on the 128-node IBM SP2 at Argonne National Laboratory. All nodes of this system run AIX 3.2.4. Eight nodes are RS/6000 Model 970s with 256 MB main memory and serve as dedicated PIOFS servers; the others are RS/6000 Model 370 with 128 MB main memory. Each PIOFS server has 3 GB local SCSI disk with 8 MB/s bandwidth; hence, total PIOFS capacity is 24 GB.

PIOFS distributes files across multiple PIOFS servers [1]. Each file consists of a set of cells; each cell is stored on a particular server node, and the default number of cells is the number of PIOFS servers. Cells are striped across the PIOFS servers in a round-robin fashion. A file is divided into basic striping units (BSUs), which are assigned to cells in a round-robin fashion. The default BSU size is 32 KB. PIOFS also supports dynamic file partitioning. A PIOFS file can be divided into subfiles by specifying a logical view for the file before accessing the file data. Logical views allow user applications to access noncontiguous data in a single request.

PIOFS Peak Performance. We measured PIOFS peak performance using two different PIOFS configurations: essentially those configurations used by DRA and Panda. We measured peak PIOFS write rates of 30.4 MB/s when writing a 2 GB file using the DRA strategy and 33.7 MB/s when using the Panda strategy.

PIOFS Results. Figure 3 shows DRA and Panda performance for the entire array and nonaligned array benchmarks. Both DRA and Panda are able to achieve above 88% of peak PIOFS performance for the entire array benchmark and above 75% of peak for the nonaligned array benchmark.

3.2.3 Performance on Intel PFS

We performed our PFS experiments on the Intel Paragon at Caltech, which has 512 compute nodes and 92 dedicated I/O nodes. Each compute node has 32 MB main memory. The 92 I/O nodes are divided into three partitions, of size 12, 16, and 64. Nodes in the 12-node partition have 32 MB main memory and a 2 GB Maxtor RAID disk with 4 MB/s peak performance. Nodes in the 16-node and 64-node partitions have 64 MB main memory and a 4 GB Seagate disk with 8 MB/s peak performance.

Files maintained by PFS are striped across a set of stripe directories [8]. In our experiments, we used the 16 node partition, 16 stripe directories (one per node), and the default stripe unit of 64 KB.

PFS Peak Performance. We measured peak PFS performance by using a strategy similar to that used by DRA on PIOFS: asynchronous I/O with interleaved file layout strategy. Both DRA and Panda use this strategy on PFS. The only difference is that we use PFS asynchronous calls instead of POSIX asynchronous I/O calls, and the common PFS file is opened using `M_ASYNC` mode so that each node obtains an independent file pointer. Each I/O node waits for the requests to finish

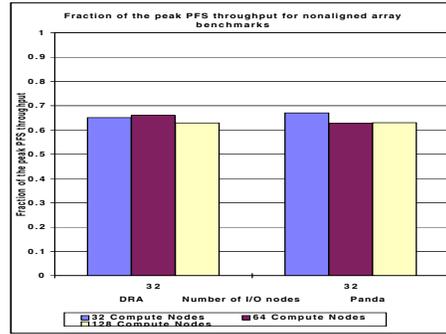
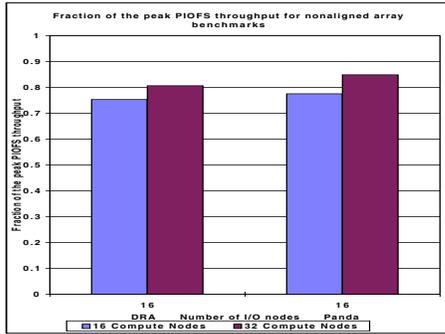
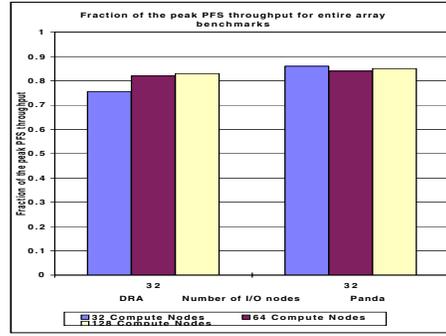
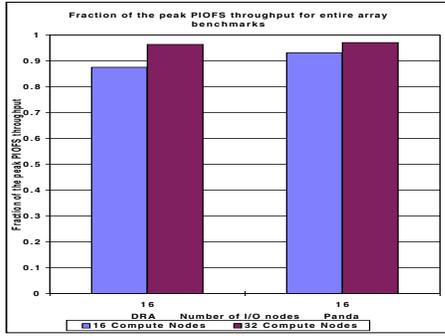


Figure 3: DRA and Panda performance on PIOFS for 800 MB entire (top) array and nonaligned (bottom) array benchmarks using a 2 MB buffer on each I/O node; the nonaligned array section overlaps half of the entire array. 16 I/O nodes were used in these experiments. The disk array distribution used for the entire array benchmark in DRA is (10×10) and (4×4) in Panda, and the disk array distribution used for the nonaligned array benchmark was (10×10) in both DRA and Panda.

Figure 4: DRA and Panda performance on PFS for 800 MB entire (top) and nonaligned (bottom) benchmarks using a 2 MB buffer on each I/O node; the non-aligned array section overlaps half of the entire array. We varied the number of compute nodes from 32 to 128, and used 32 I/O nodes in our experiments.

by calling the `iowait` function. A 2 GB file is written and a peak PFS performance of 84 MB/s is measured.

PFS Results. Figure 4 shows DRA and Panda performance for entire array and nonaligned array benchmarks. Both DRA and Panda achieve above 75% of the peak PFS performance for the entire array benchmark and above 64% of the peak for the nonaligned benchmark.

3.3 Performance Impact of Parameter Settings

We focus here on a small subset of the parameters at a time while fixing the values for all other parameters, and we compare the performance results obtained by using different parameter values.

3.3.1 Number of I/O Nodes

To determine the performance impact of the number of I/O nodes, we studied the performance of DRA and

Panda independently without an interlibrary comparison. We present Panda results only; DRA results are similar.

The leftmost group of bars in Figure 5 shows the performance of Panda on JFS using different numbers of I/O nodes while writing a 200 MB array using 512 KB buffer on each I/O node. Performance drops by a small amount when the number of I/O nodes increases. This is because the amount of data handled by each I/O node decreases, and there is a fixed startup cost that includes the time for Panda compute nodes to pass the I/O request information to the Panda I/O nodes, and the time for Panda I/O nodes to process the information before any disk array chunks can be communicated.

The middle group of bars in Figure 5 shows aggregate throughputs for writing an entire array using a 2 MB buffer on each I/O node on PIOFS. Panda achieves 91% of measured peak PIOFS performance when 8 or 16 I/O nodes are used. With 4 I/O nodes, the overall performance degrades because the array chunks on one I/O node are mapped to one subfile in one cell on one PIOFS server using the dynamic file partitioning (as described in Section 3.2.2), so only half of the 8 PIOFS servers are involved in handling I/O requests, and

the effective PIOFS utilization degrades.

The rightmost group of bars in Figure 5 shows the aggregate throughputs for Panda writing a 200 MB array on PFS. On this platform, small numbers of I/O nodes cannot keep the PFS servers busy. This is because there are 16 PFS servers who can serve I/O requests from Panda I/O nodes, and peak performance was measured using all 16 servers.

In summary, the number of I/O nodes can have a significant impact on DRA and Panda performance on these platforms, with the actual impact depending on the underlying platform characteristics and file system configuration. On JFS, the performance impact is relatively small, but on PIOFS and PFS it can be substantial.

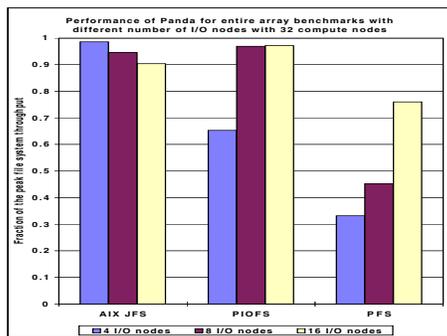


Figure 5: Panda write performance for entire array requests on JFS, PIOFS, and PFS using different numbers of I/O nodes and 32 compute nodes.

3.3.2 Disk Array Distribution

Figure 6 shows Panda performance for the nonaligned array benchmarks on a 200 MB array using different disk array distributions on JFS. The default disk array distribution (natural chunking) results in the worst performance for nonaligned requests. This phenomenon arises because with natural chunking, chunks tend to be very large. Thus, when a read request is received, the disk array chunks corresponding to the requested subarray will often belong to just a subset of the I/O nodes (Figure 7). Hence, available I/O bandwidth is decreased. This so-called load-imbalance problem can be healed by choosing a different number of I/O nodes such that the workload is roughly evenly distributed across I/O nodes instead of changing the disk array distribution. This suggests that the optimal parameter setting could also be determined by combinations of several parameter values rather than individual parameter values independently.

Both Figure 6 and Figure 7 suggest that this load-imbalance problem can be eliminated by using fine-grained chunking disk array distributions such that none of the I/O nodes would have a significantly larger amount of data to work on than others. However, the fine-grained disk array distribution could also introduce excessive cost in array reorganization when array data

is moved between application memory and I/O server memory. For instance, in Figure 7, if the array in-memory distribution is (4×4) , using a (4×4) disk array distribution does not require array reorganization when forming disk array chunks. However, if a (4×8) disk array distribution were used, it would be necessary for Panda to extract a section of array data from memory in order to form a disk array chunk. This array reorganization cost typically depends on the underlying communication network and file system characteristics, array in-memory distribution, and the number of I/O nodes used. The cut-off point between the savings gained by fine-grained chunking distribution and the cost introduced by array data reorganization is not always easily identified.

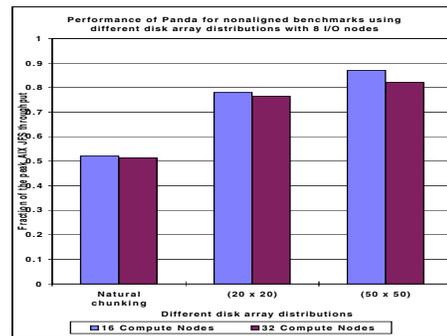


Figure 6: Panda write performance for the nonaligned array requests on JFS using different disk array distributions and 32 compute nodes.

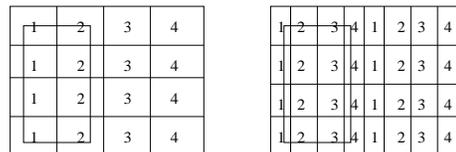


Figure 7: The impact of disk array distributions on Panda performance. A 2-D array is distributed across 4 I/O nodes. The smaller rectangle denotes the requested (nonaligned) array section. On the left, a 4×4 disk array distribution results in the request being processed by just 2 I/O nodes (1 and 2). On the right, a 4×8 distribution allows all 4 I/O nodes to be involved, enhancing parallelism.

Disk array distribution also impacts performance on PIOFS and PFS. However, the quantification of the performance impact on these parallel file systems for different disk array distributions depends on the details of the parallel file system data striping and file layout policies. On PIOFS and PFS, even though DRA and Panda have chosen a disk array distribution, an array chunk on a particular DRA or Panda I/O node could be split up into smaller pieces by the parallel file system servers and striped across multiple PIOFS or PFS servers depending on the file layout policies chosen. This is not a

problem on JFS because each I/O node has full control over its own local disk, and array chunks are written directly into its local disk. To illustrate the effects of disk array distributions, we consider the JFS example sufficient.

This study implies that choosing optimal disk array distributions can be a difficult task, and the performance of the system could depend on the disk array distribution or the combination of the number of I/O nodes used and the disk array distribution chosen. With effective internal heuristics in selecting optimal parameter settings, DRA’s approach of asking for hints from users could be more robust (and less dangerous) than Panda’s direct approach of user-controlled parameter selection. However, DRA’s heuristics need to be tuned to provide high performance.

3.3.3 Buffer Size

Studies of the effects of buffer size on DRA and Panda performance [10, 14] showed that for JFS, a relatively small buffer (512 KB) offered the best performance. On JFS, read and write operations do not block the calling process, so communication and computation performed by DRA and Panda when gathering one disk array chunk can be overlapped with disk I/O associated with previous disk array chunks. Using small chunks increases this overlap and improves overall performance. In general, as long as the internal communication and computation overhead is smaller than the underlying file system overhead, the performance of these libraries is bounded by the peak file system performance.

In contrast, better performance was achieved on PIOFS and PFS when using a larger buffer (e.g., 4 MB). On these parallel file systems, multiple PIOFS or PFS servers can handle an I/O request simultaneously. Since an I/O request can be striped across all the file system servers, with small buffers we may not utilize the available server resources well, and the overall parallel file system utilization can be low.

3.3.4 File System Policies and File Layouts

To study the performance impact of the underlying file system policies and file layout, we performed an inter-library performance study on PIOFS. DRA and Panda used different file layouts, PIOFS file system facilities (Section 3.2.2), and communication mechanisms (Section 2.2).

Figure 8 shows DRA and Panda performance on PIOFS. Our experiments show that the internal communication and computation overheads in DRA and Panda are relatively small compared with the overall cost for handling each I/O request (about 30% for DRA, and 25% for Panda), and the performance difference due to the use of different communication mechanisms is small. Thus, most of the difference came from the use of different file system policies and file layouts. Clearly, with a small number of I/O nodes, Panda’s choice of file system policy and file layout is better than DRA’s. However, the Panda strategy also has its limitations. Because the data on each I/O node is mapped to one PIOFS server, the number of I/O nodes must be a multiple of the number of PIOFS servers; otherwise, load imbalances occur.

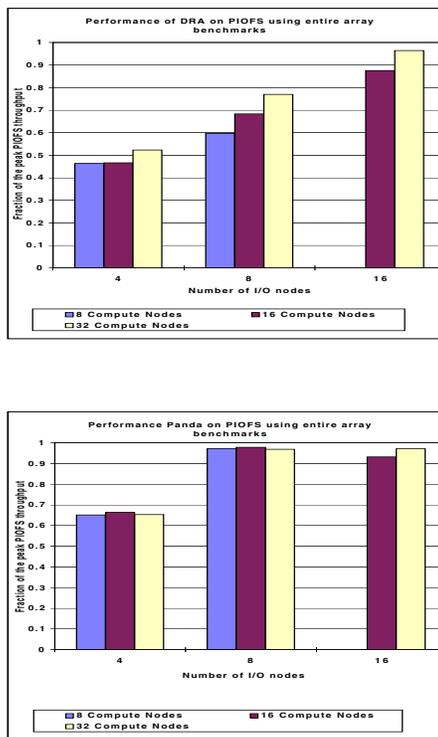


Figure 8: DRA (top) and Panda (bottom) performance on PIOFS for entire array requests, when writing a 200 MB file and using a 2 MB buffer with different file system policies and file layouts.

3.4 Performance Discussion

Both DRA and Panda achieve close to peak file system various parameters can impact performance significantly in certain situations. We list some general observations from our performance study.

- Selecting optimal parameter settings requires detailed knowledge of the individual components of the library itself, such as the internal algorithms, the underlying file system, and the underlying communication systems.
- Selecting optimal parameter settings requires a comprehensive understanding of interdependencies among library parameters.
- Selecting optimal parameter settings requires knowledge of the target workload characteristics.
- The importance of a particular performance parameter in affecting the overall performance is dependent on the runtime system.

These observations suggest that leaving control to users to decide on the parameter settings can result in suboptimal parameter settings. We believe that the solution is to incorporate automatic performance optimization techniques in an I/O library, an approach that we propose to investigate in future research.

4 Related Work

Poor I/O performance on parallel computers has inspired numerous proposals for improved parallel file systems and I/O libraries. Vesta [4] and Galley [13] provide new parallel file system interfaces that conceal the parallelism within the file system. These file systems also support different file layouts to facilitate fast accesses. Experience suggests that while parallel file systems can be useful, they do not avoid the need for careful parameter selection.

Many specialized parallel I/O libraries have been developed with the goals of providing portability, ease of use and high-performance support to parallel applications. PPFs [7] focuses on efficient caching and prefetching support for parallel applications. MPI-IO [3] provides a portable I/O interface to MPI programs; it also supports collective I/O interfaces. Parallel I/O techniques such as two-phase I/O [2] and disk-directed I/O [9] for collective I/O operations have been adopted in many of these libraries. However, no published studies examine the major performance factors of these systems for a wide range of I/O patterns, problem sizes, and execution environments.

Little work has been done on automatic performance optimization in parallel I/O community, however, such techniques have been explored by other communities. For example, Golding et al. [6] have proposed an attribute-managed storage system approach to the management of large-scale storage systems. Foster and Worley [5] have used performance modeling techniques to guide algorithm selection in parallel climate models.

5 Conclusions

Our results show that both the DRA and Panda parallel I/O libraries achieve high I/O performance for interesting problems. However, high performance often requires that the user choose optimal values for system parameters such as disk array distribution and the number of I/O nodes. Unfortunately, proper parameter selection often requires deep knowledge of the parallel I/O library, the underlying file system features, and the application I/O request characteristics. We therefore propose to investigate techniques for the automatic optimization of key parameters, such as array on-disk layouts in the near future.

References

- [1] Fern E. Bassow. *IBM AIX Parallel I/O File System: Installation, Administration, and Use*. IBM, Kingston, N.Y., May 1995. Document Number SH34-6065-00.
- [2] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel i/o. In *Proceedings of Supercomputing '93*, pages 452–461, Portland, Oregon, November 1993.
- [3] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: A parallel file i/o interface for MPI. Technical Report NAS-95-002, NASA Ames Research Center, January 1995. Version 0.3.
- [4] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Overview of the vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993.
- [5] I. Foster and P. Worley. Parallel algorithms for the spectral transform method. *SIAM Journal of Scientific and Statistical Computation*, 18(3), 1997. To appear.
- [6] Richard Golding, Elizabeth Shriver, Tim Sullivan, and John Wilkes. Attribute-managed storage. In *Proceedings of the Workshop on Modeling and Specification of I/O*, San Antonio, Texas, October 1995.
- [7] James V. Huber, Jr, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFs: A high performance portable parallel file system. In *Proceedings of the International Conference on Supercomputing*, June 1995.
- [8] Intel Corporate. *Paragon User's Guide*, 1993.
- [9] David Kotz. Disk-directed i/o for MIMD multiprocessors. Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994. Revised November 8, 1994.
- [10] J. Nieplocha and I. Foster. Disk resident arrays: An array-oriented i/o library for out-of-core computations. In *Proceedings of Frontiers '96 of Massively Parallel Computing Symposium*, September 1996.
- [11] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. Global arrays: A portable “shared-memory” programming model for distributed memory computers. In *Proceedings of Supercomputing '94*, pages 340–349, 1994.
- [12] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.
- [13] N. Nieuwejaar and D. Kotz. Performance of the galley file system. In *Proceedings of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems*, Philadelphia, PA, May 1996.
- [14] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server directed collective i/o in panda. In *Proceedings of Supercomputing '95*, San Diego, California, December 1995.
- [15] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized i/o for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.