

Server-Directed Collective I/O in Panda

K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett

Computer Science Department

University of Illinois, Urbana, Illinois 61801

{seamons,ying,pecj,jozwiak,winslett}@cs.uiuc.edu

Keywords: Collective I/O, Parallel I/O

Abstract

We present the architecture and implementation results for Panda 2.0, a library for input and output of multidimensional arrays on parallel and sequential platforms. Panda achieves remarkable performance levels on the IBM SP2, showing excellent scalability as data size increases and as the number of nodes increases, and provides throughputs close to the full capacity of the AIX file system on the SP2 we used. We argue that this good performance can be traced to Panda's use of server-directed i/o (a logical-level version of disk-directed i/o [Kotz94b]) to perform array i/o using sequential disk reads and writes, a very high level interface for collective i/o requests, and built-in facilities for arbitrary rearrangements of arrays during i/o. Other advantages of Panda's approach are ease of use, easy application portability, and a reliance on commodity system software.

1 Introduction

In the past few years, researchers in the HPC community have suggested many approaches to improve i/o performance for scientific applications on massively parallel platforms. In the work described in this paper, we focus on the problem of combining three proposed techniques that we thought offered exceptional promise, and show that they are in fact a fortuitous combination:

1. storage of arrays by subarray chunks in main memory and on disk
2. high-level interfaces to i/o subsystems
3. use of 'disk-directed i/o' to make more efficient use of disk bandwidth

The first technique, array chunking, has been in use in main memory for decades as a means of handling out-of-core computations, and more recently as a way of decomposing arrays for computation on parallel platforms. Array chunking in memory improves performance by increasing the locality of computation on a processor. Array chunking on disk has recently been proposed by a number of researchers as a way of allowing fast strided access to disk data on parallel platforms; most researchers assume, however, that eventually array data must be placed in 'traditional' row-major or column-major order for consumption by other applications, e.g., visualizers on sequential platforms. Our work is somewhat unusual in that we see intrinsic value in chunked arrangements ('schemas') of array data on disk: such schemas will in general improve performance for data consumers even on sequential platforms, because they increase the locality of data across multiple dimensions, thus typically reducing the number of disk accesses that an application must do to obtain a working set of data in memory.

This research was supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland, by NSF under PYI grant IRI 89 58582, and by NASA under NAGW 4244 and NCC5 106.

A penchant for high-level i/o interfaces is our legacy from our group's community of origin, the database world. High-level interfaces give us the flexibility to optimize everything beneath the level of the interface. High-level interfaces also give a cleaner view of many situations, just as a printout of the source code of a program is more meaningful to a human than a hexadecimal dump of the same code. The cleaner view means we have a higher level semantic understanding of i/o requests, and we can use the extra semantic information to make more intelligent optimization decisions. In addition, higher-level interfaces are easier for users to master, and give their applications better portability.

Disk-directed i/o [Kotz94b], as described in the following sections, uses information about a collective i/o request that is being issued by a group of compute nodes. The description of the entire request can be digested by the i/o nodes, so that they can plan how they would like to receive data from the compute nodes in order to minimize disk seeks. Then the i/o nodes can request pieces of data, in the order they want them, from the compute nodes. (One can think of this as an illustration of the advantage of having higher-level semantic knowledge.) Disk-directed i/o has, to the best of our knowledge, not been implemented in real systems, but results from simulations look very promising.

We have combined the three approaches discussed above in a new version of the Panda array i/o library (URL <http://bunny.cs.uiuc.edu/CADR/panda.html>), and tested it on sequential Unix workstations and on an IBM SP2. Panda 2.0 ('Panda' hereafter) runs on top of ordinary Unix file systems, uses MPI for all communication, is written in C++, and is very compact. Panda combines our fondness for high-level interfaces with our interest in disk-directed i/o, and thus Panda implements a perversion of disk-directed i/o that we call *server-directed i/o*. The details of this architecture will be discussed in Section 2.

While perhaps of academic interest, Panda would be of little value if it did not offer significant performance advantages. In performance tests of Panda on the IBM SP2, we found that Panda provided excellent i/o bandwidth under a wide variety of conditions, providing throughputs close to the full capacity of the underlying AIX file system. We also found that Panda's performance scaled very well under increasing numbers of nodes and array data size. These performance results will be discussed in Section 3, followed by related work in Section 4 and a summary and conclusion in Section 5.

2 Panda internal architecture

Panda is designed to support synchronized collective i/o of SPMD-style application programs on ordinary workstations, distributed memory parallel architectures, and networks of workstations. The overall philosophy behind Panda is to maximize i/o performance by doing sequential reads and writes whenever possible. Since high-performance scientific applications typically spend most of their i/o time reading and writing entire arrays, theoretically sequential reads and writes are possible; however, many current systems that support parallel i/o do not offer a sufficiently high-level interface for the i/o system to realize that the random-seeming pattern of read and write requests arriving at i/o nodes can be viewed as a sequential pattern overall. Panda's high-level API, described below, avoids this pitfall.

A second difficulty in realizing sequential reads and writes is that i/o requests from compute nodes may arrive at i/o nodes in a manner that discourages sequential access, even when the overall pattern of sequential access is known to the compute nodes and/or i/o nodes. Panda avoids this problem by supporting collective i/o operations using a server-directed i/o strategy, under which the i/o nodes, rather than the compute nodes, direct the flow of i/o requests. Since the i/o nodes understand both the layouts of array data in files on disk and also in memory, the i/o nodes can sequentially read data from files, scattering the data to compute nodes as it arrives from disk, and employing the reverse strategy for writes. Under this approach, normal Unix file system prefetching and caching will work perfectly, avoiding the myriad buffering errors that caused poor performance in many early parallel i/o systems. If files are laid out more-or-less sequentially on disk, as is generally the case with the huge data files employed by our target applications, then sequential file reads will translate to inexpensive sequential disk reads in most cases. While server-directed i/o is not the only possible means of achieving sequential i/o, we will show that it performs well for closely synchronized sets of compute nodes, while removing any reliance on costly custom system software.

The system architecture of Panda is shown in Figure 1, which shows how Panda is distributed across compute nodes (Panda clients) and i/o nodes (Panda servers). Under server-directed i/o, the application program running on the compute nodes communicates with the client via Panda's high-level collective i/o

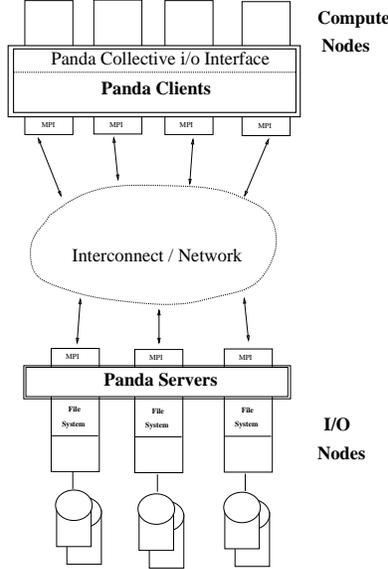


Figure 1: Panda server-directed i/o architecture

interface for multidimensional arrays (we anticipate the current architecture is extensible to additional data types). Panda currently supports applications that distribute arrays across all compute nodes using HPC-style BLOCK- and *-based array schemas. By default, for performance and convenience, Panda uses a disk schema that is identical to the memory schema. We refer to this as “natural chunking” [Seamons94b]. However, users may override the default by declaring any BLOCK- and *-based schema for disk; this is useful when users know how the data will be accessed in the future and wish to optimize for the future. The schemas on disk and in memory may decompose the arrays in radically different ways, as illustrated by the example in Figure 2, discussed below.

Panda assumes all clients will participate in the collective i/o at approximately the same time, although the application need not synchronize prior to the collective i/o call. Panda’s high-level interface for array i/o supports collective i/o at the level of requests to checkpoint computation, restart computation from checkpoint files, and save the output from timestep computations, as well as requests to write out a single array or a group of arrays. Figure 2 contains sample C++ application code that uses Panda’s classes and methods to output timestep data; note how easy the interface is to use.

When an application makes a collective i/o request to Panda, a selected client (the master client) sends to a selected server (the master server) a short very-high-level description of the two schemas for the arrays, in memory and on disk. Note the clients and servers play a different role than in traditional client/server architectures where the clients make requests of the server. After the initial request to the servers for a collective i/o operation, Panda’s servers then make data requests of the clients. The master server then informs all the other servers of the schema information, and each server plans how it will request or send its chunks of the array data to or from the relevant clients.

In the case of write operations, the servers all know that array data is to be chunked on disk as described in the disk schema, and the chunks are implicitly assigned in a round-robin fashion across all the servers. (Note here that our penchant for high level interfaces has led us to stripe data at the chunk level rather than at a disk block level. This from others’ work, where the arrays were striped across i/o nodes in traditional (row-major or column-major) order according to disk block size [Kotz94b, Kotz95b, del Rosario94].) Once a server has made its plan, it assembles its first assigned chunk by sending a request for a logical sub-chunk to all the clients that hold a part of its first assigned chunk. Depending on the memory schema and disk schema, a client can receive a request for a sub-chunk of array data that is not contiguous in its memory (e.g., the client may hold $A[1, 1, 1]..A[100, 200, 300]$ and receive a request for $A[20, 30, 40]..A[50, 60, 70]$). The client is responsible for any reorganization required to assemble the requested sub-chunk when this happens.

Our emphasis on high-level interfaces, even within Panda itself, gives easy support for ‘strided’ requests since clients and servers send logical requests for sub-chunks to each other. This allows clients and servers to choose their own local optimization strategies when appropriate. The server is also responsible for reorganizing the received sub-chunks so that they form the entire chunk in traditional array order. Then the server sends its first assigned chunk to disk, and begins to assemble the second assigned chunk. The reverse procedure holds for reads.

Panda’s strategy assumes clients and servers have sufficient memory to buffer a requested array sub-chunk in memory if needed. To limit buffer space requirements and also maximize i/o performance, Panda uses a form of sub-chunking on disk (i.e., the internal subdivision of chunks into smaller chunks) to break large disk chunks into more manageable units on-the-fly when performing a collective i/o. (After experimentation, we chose a subchunk size of 1 MB for all experiments in this paper.) This happens transparently to the user and the Panda client, and does not change the memory schema, disk schema, or round-robin assignment of chunks in any way. In the future we will also allow users to explicitly request sub-chunked schemas in memory and on disk, as that will be advantageous for some applications.

The servers do not communicate with one another during plan formation or while array data is being gathered or scattered to the clients; nor do the clients. Upon completion of a collective i/o request, the servers inform the master server who informs the master client that the collective i/o operation is completed. The master client in turn informs the other clients.

3 Panda performance results

We tested the performance of Panda’s server-directed i/o architecture on the IBM SP2 at NASA Ames Research Center. Table 1 details the characteristics of the IBM SP2 at NAS. More detailed information can be found at <http://lovelace.nas.nasa.gov/Parallel/SP2>. NAS’s SP2 does not currently have any parallel i/o facilities; each processor has its own AIX file system. Panda uses the AIX file system directly on each i/o node for storing array data. The partitions we used to store array data were approximately 15-30% full at the time of our experiments.

Our experiments measure Panda’s performance to read and write a single array and multiple arrays. These read and write operations are primitive operations in Panda that underlie Panda’s timestep, checkpoint, and restart operations. Our runs vary the number of compute nodes, the number of i/o nodes, the size of arrays, and the disk schemas. All of our runs use 40 or fewer nodes. We would have preferred to do more runs with large numbers of nodes, but have not yet been able to get the large blocks of machine time necessary for the larger runs. We flush the file system buffer before each read operation by writing a large temporary file and then deleting the file. We flush the data to disk using `fsync` for each write operation.

For performance data, we repeated each test 5 times and use the average elapsed time. The elapsed time is the maximum time spent by any compute node on the collective i/o request. As a basis for a normalized throughput percentage, we measured the underlying AIX file system performance on single nodes of the SP2 by measuring the throughput in MB/s obtained reading or writing a file using AIX file system calls. Our tests accessed files of total size 32 MB and 64 MB using 1 MB i/o requests, as is done in Panda when array chunks are larger than 1 MB. The peak throughputs obtained are included in Table 1. Panda’s normalized throughput for array reads and writes is calculated by dividing Panda’s throughput per i/o node by the peak AIX throughput.

We conducted tests to read and write a single 3D array of size 16-512 MB using natural chunking, with the same BLOCK, BLOCK, BLOCK data schema in memory and on disk. In this configuration, array chunks can be transferred between Panda clients and servers with very little processing overhead in Panda. The peak message passing throughput of 34 MB/s for MPI-F on the SP2 far exceeds the peak measured AIX file system throughput for reads and writes, which is below 3 MB/s on each disk. Thus, we expect Panda’s throughput at each i/o node to be very close to the maximum throughput of the AIX file system.

Panda scales well for both reads and writes while varying the size of the array, the number of compute nodes, and the number of i/o nodes in the tests we conducted using natural chunking. We tested nine combinations of compute nodes and i/o nodes, where the number of compute nodes was 8 ($2 \times 2 \times 2$ mesh), 16 ($4 \times 2 \times 2$ mesh), or 32 ($4 \times 4 \times 2$ mesh) and the number of i/o nodes was 2, 4, or 8. In these tests, the

Total number of nodes	160 nodes
Each node	RS6000/590 workstation
Each processor	66.7 MHz, POWER2 multi-chip RISC
Node operating system	AIX operating system
Languages supported	Fortran 77, Fortran 90, C, C++
Total memory	23.9 GB
Total memory per node	128 MB
Total disk space	485 GB
Total disk space per node	2 GB
Total memory bandwidth	342 GB/s
Peak performance	42.8 Gflops
High-performance switch bandwidth (hardware)	40 MB/s, bidirectional
Disk peak transfer rate	3.0 MB/s
I/O bus	SCSI
I/O bus peak transfer rate	10 MB/s
Node file system block size	4 KB
Measured peak throughput for AIX file system reads	2.85 MB/s
Measured peak throughput for AIX file system writes	2.23 MB/s
NAS-measured message passing latency	43 microseconds
NAS-measured message passing bandwidth	34 MB/s

Table 1: The system characteristics of the NAS IBM SP2.

amount of data per processor (array chunk size) on the compute nodes ranges from 0.5 MB per processor up to 64 MB per processor, depending on the array size and number of compute nodes.

Figures 3 and 4 show the aggregate and normalized throughput for reads and writes using 8 compute nodes. The throughputs for 16 and 32 compute nodes (not shown here, but implied by subsequent figures) are similar, because Panda scales well as the number of compute nodes increases so long as the array chunks remain large enough that MPI latency does not dominate the time to transfer an array chunk. As predicted, the throughputs are high for both reads and writes, from 85-98% of peak AIX performance at each i/o node. As array size increases, the sequential reads and writes reach a maximum of 1 MB in size, since that is the maximum chunk size that Panda transfers between clients and servers. Larger chunks are broken into 1 MB subchunks to reduce buffer space requirements on the i/o nodes. As the chunk size per processor decreases, the throughput declines since the underlying AIX file system throughput declines when writing a small file with write size less than 1 MB.

The bottleneck on the NAS IBM SP2 for the tests we conducted using natural chunking is the relatively slow 3 MB/s peak disk transfer rate. To show that Panda is able to make efficient use of the available message passing bandwidth, we simulated an infinitely fast disk by commenting out the actual file system open/close/write/read commands in the Panda server code. When simulating fast disks, the next potential system bottleneck is the peak MPI message passing bandwidth and latency on the IBM SP2. We repeated the same tests for writing and reading arrays using natural chunking with simulated disks. In this case, the *normalized throughput per i/o node* is the throughput per i/o node divided by the peak message traffic bandwidth of 34 MB/s for both writes and reads. The throughputs will be similar for both reads and writes, since the gathering and scattering of array data between the Panda servers and clients are essentially identical with respect to total number of messages and message sizes. Figures 5 and 6 show the throughput for 32 compute nodes using natural chunking and simulating an infinitely fast disk (results are similar for 8 and 16 compute nodes). As predicted, the throughputs for reads and writes are very similar, near 90% of peak MPI performance in most cases. As array size decreases so that total elapsed time is very small due to the high throughput, the startup overhead for Panda (measured as approximately .013 seconds) begins to dominate the elapsed time, causing the normalized throughput to decline since startup costs are included in the elapsed time.

In Panda’s server-directed i/o architecture, array data is automatically reorganized whenever the in-memory schema and the on-disk schema differ. Reorganization occurs during the i/o operation as array data is transferred between Panda clients and servers. For example, suppose a 512 MB array of size $512 \times 512 \times 512$ is distributed as BLOCK, BLOCK, BLOCK across 32 processors in a $4 \times 4 \times 2$ mesh. Declaring a BLOCK, *, * disk schema will place the array in traditional order across several disks, so that the data can be migrated to a sequential machine with the array in a single file in traditional order by simply concatenating all the files on the i/o nodes together. Since traditional order is particularly important for today’s applications, we conducted experiments reading and writing array data with a BLOCK, *, * disk schema and a BLOCK, BLOCK, BLOCK memory schema. We conducted experiments with 12 combinations of compute nodes and i/o nodes, with the number of compute nodes being 8 ($2 \times 2 \times 2$ mesh), 16 ($4 \times 2 \times 2$ mesh), 24 ($6 \times 2 \times 2$ mesh), or 32 ($4 \times 4 \times 2$ mesh) and the number of i/o nodes being 2, 4, 6, or 8. The logical i/o node mesh for the BLOCK, *, * distribution on disk can be thought of as $(n \times 1 \times 1)$ where n is the number of i/o nodes.

Figures 7 and 8 show the aggregate and normalized throughput for reading and writing arrays in traditional order. The throughputs are high, from 68-95% of peak AIX performance at each i/o node, and are slightly lower than those obtained using natural chunking. Compared to natural chunking, extra messages and extra MPI overhead are required to handle strided requests and to reorganize the data between the memory and disk schemas. Since the memory and network bandwidth are high relative to the disk bandwidth on the SP2, the overheads for reorganization compared to natural chunking are not significant in these tests.

As with the normalized tests, we conducted tests simulating an infinitely fast disk to observe how Panda utilizes the network bandwidth. Figure 9 shows the aggregate and normalized throughput for writing arrays in traditional order while simulating an infinitely fast disk (reads are similar). Compared to the natural chunking simulations, the cost for reorganization is now visible, as the throughput for both reads and writes ranges from 38-86% of peak MPI performance. We believe that these throughputs can be improved by using non-blocking communication when performing data rearrangement.

All the experiments described previously in this paper are load balanced with respect to the amount of data per processor and the amount of data read or written by each i/o node. Using natural chunking, array chunks may be unevenly distributed across i/o nodes when the number of i/o nodes does not evenly divide the number of compute nodes. Fortunately, as the number of compute nodes increases, load imbalance becomes less significant for a fixed number of i/o nodes. In addition, a schema such as the traditional order schemas just presented can be chosen which distributes the data evenly across all the i/o nodes. For these reasons, we do not consider load imbalance to be a significant issue in Panda.

We conducted additional experiments reading and writing multiple arrays, which we do not present here due to space limitations. Panda achieves high throughputs reading and writing multiple arrays, similar to the throughput for single arrays, when the size of array chunks is large enough so that MPI latency is not a bottleneck.

4 Related work

Space does not permit a proper discussion of all the work related to Panda; we will limit our discussion to papers from the HPC community that *appeared in 1993 or later and that focus on parallel implementations of collective i/o*.

Recent studies characterizing dynamic i/o patterns in scientific applications [Kotz94a, Purakayastha94, Pasquale94] show that many scientific applications have regular patterns of i/o behavior, such as physical periodicity in strided access to multidimensional arrays, or temporal periodicity in checkpoint and restart operations. In both cases all application compute nodes participate in i/o operations to access array data.

The observation of these i/o behaviors led to interest in supporting efficient collective i/o operations in parallel file systems or parallel i/o libraries. Through a collective i/o interface, the application’s compute nodes can cooperately issue a small number of requests to gather a large amount of data, passing along any semantic information associated with the i/o requests. With a collective i/o implementation, there are more opportunities for disk i/o optimization, as one can form large orderly contiguous disk i/o requests rather than servicing disk i/o requests as they arrive in random order. Several parallel file systems and multidimensional array libraries have provided a collective i/o interface [Bennett94, Brezany95, Corbett94b,

Corbett95, del Rosario94, Karpovich94, Seamons94b, Seligman94]; some of them have shown that collective i/o provides high performance given appropriate language, compiler and run-time system support.

Three approaches to i/o optimization are discussed in [Kotz94b]: traditional caching, two-phase i/o, and disk-directed i/o. Traditional caching does not support a collective i/o interface; i/o requests are served as they arrive. Each i/o node has a file cache and prefetches, thus optimizing sequential file accesses. However, in a multiprocessor environment, although file accesses may be sequential when viewed at an abstract level, they typically look strided when viewed at the level of an individual compute node. Without a high level semantic view of the collective i/o requests, the file system is not able to predict whether sequential prefetching will be useful or when to flush the file cache. Intel CFS [Pierce93] uses traditional caching, and [Kotz93b] shows that CFS only uses half of the raw disk bandwidth.

[Bordawekar93] considers a ‘two-phase’ access strategy for collective i/o. In this approach, for read operations, the compute nodes cooperate to bring all the data into memory in a way that minimizes the total number of disk accesses by having the data layout in memory conform to the data layout on disk. In the second phase the compute nodes permute the data in memory until each node has the data that it needs.

Disk-directed i/o for collective i/o operations is proposed in [Kotz94b]. Under this approach, compute nodes tell the i/o nodes about a collective i/o request they plan to perform. Based on this semantic information, the i/o nodes determine which compute nodes they should ask for data or send data to. The i/o nodes plan their data accesses so that they are able to form a large contiguous disk i/o request rather than many smaller requests. The simulation in [Kotz94b] promises good performance. [Kotz95a] compares the simulated performance using disk-directed i/o with that of traditional caching on irregular distributions of data, and shows that disk-directed i/o is never slower in the test cases. [Kotz95b] uses this approach to implement an out-of-core LU decomposition problem and shows that it is much better than the traditional caching scheme.

Among other approaches to collective i/o, [Galbreath93] buffers several i/o requests before issuing disk i/o requests. In this approach, each processor contributes its requests to a buffer, then the buffers are gathered together and written into or read from by a master processor.

[Bennett94] gives a strategy to minimize the number of i/o requests by coalescing a large number of requests from compute nodes into one big i/o request. As in all of the work discussed above, under this approach the compute nodes are responsible for figuring out where in each file to get or put the data, and then pass this information on to the i/o nodes, which actually do the necessary disk accesses. Conceptually, the question of where to read or write in a file is a low-level task properly belonging to specialized i/o routines, and is not something a normal scientific programmer wants to deal with; we have endeavored to relieve the application code of this task in Panda, as shown in Figure 2.

Of the implementation approaches described above, disk-directed i/o seems to promise the best performance. However, there are some disadvantages in implementing disk-directed i/o at the physical level, as intended by its creators. Chief among these is the introduction of operating system and file system dependencies into implementations of disk-directed i/o, making ports difficult, and a requirement for custom system software, making use difficult on architectures such as a network of workstations. With these problems in mind, we chose to apply the idea of disk-directed i/o at a logical level, incorporating it into Panda on top of the native file system. The performance study in the previous section showed that logical level disk-directed i/o still offers good performance advantages.

5 Conclusions

In this paper we have described the architecture and performance of Panda 2.0, which offers a high-level interface for array i/o on parallel and sequential machines. Panda’s application program interface supports an i/o model at the abstract level of an array rather than at the traditional Unix file system level. The physical details of mapping array data to files and the implementation algorithm to perform that mapping are entirely invisible to the application programmer. While one might expect the price of encapsulating physical storage details to be a drop in performance, this paper demonstrates that high performance array i/o is attainable with a high-level interface; in fact, combining portability and high performance may only be possible through a high-level interface. We expect high-level interfaces such as Panda’s to become the

interfaces of choice for scientific applications in the future. They provide simplicity, portability, and, most importantly, allow for an efficient underlying implementation.

The Panda server-directed i/o architecture, as described in this paper, is a prime example of an efficient underlying implementation for array i/o. The server-directed i/o architecture is targeted for SPMD-style applications on distributed memory parallel machines and network of workstation environments that read and write in-memory arrays that are distributed across the processors in the style of HPF. It is intended for applications that are closely synchronized during i/o (sometimes referred to as collective i/o). A high-level interface is instrumental to the success of server-directed i/o, since it provides Panda with a global view of an upcoming collective i/o operation that Panda can use to plan how to perform the i/o operation using sequential reads and writes to disk. Panda is novel in its use of concepts from disk-directed i/o at the logical level to achieve sequential reads and writes. To the best of our knowledge, Panda is the first implementation of concepts from disk-directed i/o.

The most important feature of Panda is its excellent performance on the IBM SP2 at NAS. In all our experiments, Panda achieves throughputs close to the full capacity of the underlying AIX file system, using a variety of array sizes, number of nodes, and disk schemas. The results are particularly exciting in light of Panda's small size (less than 5K lines of C++ code), clean class hierarchy, and easy portability (due to use of MPI), as well as the fact that Panda uses a commodity file system (AIX on the SP2) and requires no custom system software. Panda's excellent performance on the SP2 argues for the viability of server-directed i/o, thus removing the complaint of operating system and file system dependence sometimes directed against disk-directed i/o. More generally, our experience with Panda suggests that it may be possible to achieve good i/o performance on massively parallel platforms without costly custom operating system and file system software; we will be able to run Panda on a network of ordinary workstations without changing any code.

In conclusion, we are very pleased with the performance of Panda on the SP2, and see a bright future for server-directed i/o as a way of providing ease-of-use, application program portability, and most importantly, high performance array i/o. In the near future we plan an extensive performance study of Panda's rearrangement facilities and are developing a cost model to predict Panda's performance given an in-memory and on-disk schema. We also look forward to examining the performance of mixed workloads on the SP2; as Panda makes it possible for each application on the SP2 to have its own dedicated set of i/o nodes, we are curious about the impact of i/o node sharing on i/o-intensive applications.

References

- [Bennett94] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz, Jovian: A framework for optimizing parallel I/O, Proceedings of the 1994 Scalable Parallel Libraries Conference, pages 10-20. IEEE Computer Society Press, October 1994.
- [Bordawekar93] R. Bordawekar, J. Miguel del Rosario, and A. Choudary, Design and Evaluation of Primitives for Parallel I/O, Proceedings of Supercomputing '93, pages 452-461, 1993.
- [Brezany95] P. Brezany, T. Mueck, and E. Schikuta, Language, compiler and parallel database support for I/O intensive applications, Proceedings of the High Performance Computing and Networking 1995 Europe Conference, Milano, Italy, May 1995, Springer-Verlag.
- [Corbett94b] P. F. Corbett and D. G. Feitelson, Vesta file system programmer's reference. Technical Report Research Report RC 19898 (88058), IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, October 1994. Version 1.01.
- [Corbett95] P. Corbett, D. Feitelson, Y. Hsu, J. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A Parallel File I/O Interface for MPI, Technical Report NAS-95-002, NASA Ames Research Center, January 1995.
- [del Rosario94] J. M. del Rosario, M. Harry, A. Choudhary, The Design of VIP-FS: A Virtual, Parallel File System for High Performance Parallel and Distributed Computing, Technical Report SCCS-628, NPAC, Syracuse, NY, May 1994.

- [Galbreath93] N. Galbreath, W. Gropp, and D. Levine, Applications-Driven Parallel I/O, Proceedings of Supercomputing '93, pages 462-471, 1993.
- [Karpovich94] J. F. Karpovich, A. S. Grimshaw, J. C. French, Extensible File Systems (ELFS): An Object-Oriented Approach to High Performance File I/O, Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, Portland OR, August 1994.
- [Kotz93b] D. Kotz, Throughput of Existing Multiprocessor File Systems (An Informal Study), Dartmouth PCS-TR93-190, 1993.
- [Kotz94a] D. Kotz, and N. Nieuwejaar, Dynamic file-access characteristics of a production parallel scientific workload. In Proceedings of Supercomputing '94, pages 640-649, November 1994.
- [Kotz94b] D. Kotz, Disk-Directed I/O for MIMD Multiprocessors, First Symposium on Operating Systems Design and Implementation, November 1994.
- [Kotz95a] D. Kotz, Expanding Potential for Disk-Directed I/O. Dartmouth PCS-TR95-254, submitted to SPDP '95.
- [Kotz95b] D. Kotz, Disk-Directed I/O for Out-of-Core Computation. Dartmouth TR PCS-TR95-251, submitted to HPDC '95.
- [Pasquale94] B. Pasquale, and G. Polyzos, Dynamic I/O Characterization of I/O intensive Scientific Applications, Technical Report No. CS94-364, University of California, San Diego, April 1994.
- [Pierce93] P. Pierce, A Concurrent File System for a Highly Parallel Mass Storage Subsystem, Proceedings of the 4th Conference on Hypercube Computers and Applications, Monterey, March 1989. , pp. 155-160.
- [Purakayastha94] A. Purakayastha, C. Ellis, D. Kotz, N. Nieuwejaar, and M. Best, Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor, Duke University Technical Report CS-1994-33, October 1994.
- [Seamons94a] K. E. Seamons and M. Winslett, Physical Schemas for Large Multidimensional Arrays in Scientific Computing Applications, Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management, Charlottesville, Virginia, September 1994.
- [Seamons94b] K. E. Seamons and M. Winslett, An Efficient Abstract Interface for Multidimensional Array I/O, Proceedings of Supercomputing '94, Washington D.C., November 1994.
- [Seligman94] E. Seligman, and A. Beguelin, High-Level Fault Tolerance in Distributed Programs, School of Computer Science, Carnegie Mellon University, USA. Technical report CMU-CS-94-223, December 1994.

```

// Array schema information
int array_rank          = 3;
int temperature_array_size[] = {512, 512, 512};
int density_array_size[]   = {256, 256, 256};
int pressure_array_size[]  = {512, 512, 512};
int int_size_in_bytes     = sizeof(int);
int double_size_in_bytes  = sizeof(double);
int memory_rank           = 2;                // Distribute array on 64 processors
int memory_layout[]       = {8,8};
int disk_rank             = 2;                // Store on disk in traditional order
int disk_layout[]         = {8,1};
Distribution memory_dist[] = {BLOCK,BLOCK,NONE}; // HPF BLOCK,* directives
Distribution disk_dist[]  = {BLOCK,BLOCK,NONE};

// Schema object for arrays when in memory
ArrayLayout *memory = new ArrayLayout("memory layout", memory_rank, memory_layout);

// Schema object for arrays when on disk
ArrayLayout *disk    = new ArrayLayout("disk layout", disk_rank, disk_layout);

// Array objects
Array *temperature = new Array("temperature", array_rank, temperature_array_size,
                               int_size_in_bytes, memory, memory_dist, disk, disk_dist);

Array *pressure    = new Array("pressure", array_rank, pressure_array_size,
                               double_size_in_bytes, memory, memory_dist, disk, disk_dist);

Array *density     = new Array("density", array_rank, density_array_size,
                               double_size_in_bytes, memory, memory_dist, disk, disk_dist);

// ArrayGroup object - specify a name and a file to hold schema information
ArrayGroup *simulation = new ArrayGroup ("Sim2", "simulation2.schema");

// Tell which arrays are to be part of the logical array group
simulation->include(temperature);
simulation->include(pressure);
simulation->include(density);

// Run the simulation for 100 timesteps
for (int i=0; i<100; i++)
{
    // Compute the next timestep
    compute_next_timestep();

    // Collective i/o - the timestep method may be called repeatedly during the computation
    // All three arrays are output with this single collective i/o request
    simulation->timestep();

    // Collective i/o - take a checkpoint
    if (i == 50) simulation->checkpoint();
}

```

Figure 2: Sample C++ application code using Panda’s high-level collective i/o interface to take checkpoints and output data from timestep computations

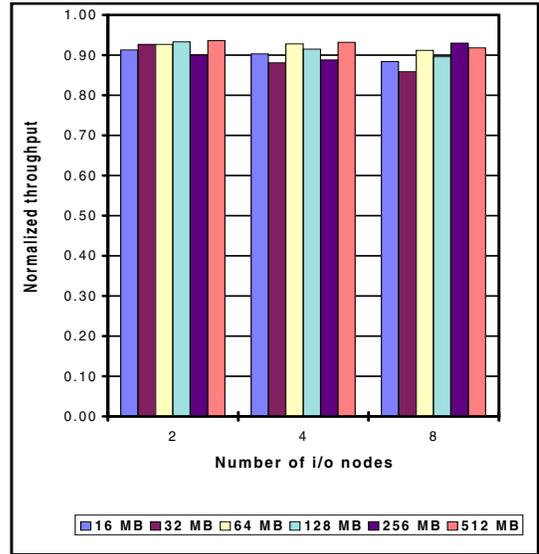
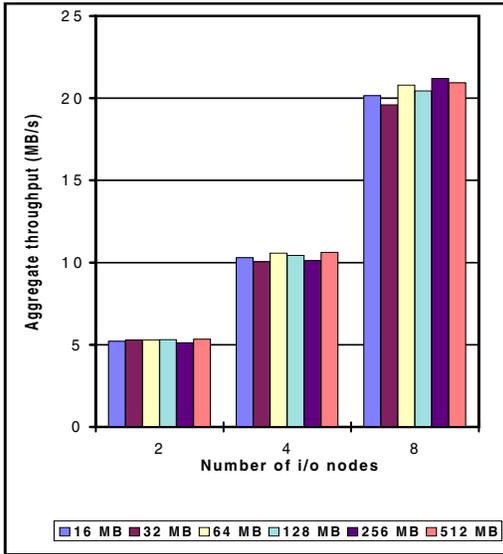


Figure 3: Aggregate and normalized throughput for reading arrays of 16 MB to 512 MB from 8 compute nodes as a function of the number of i/o nodes, using natural chunking.

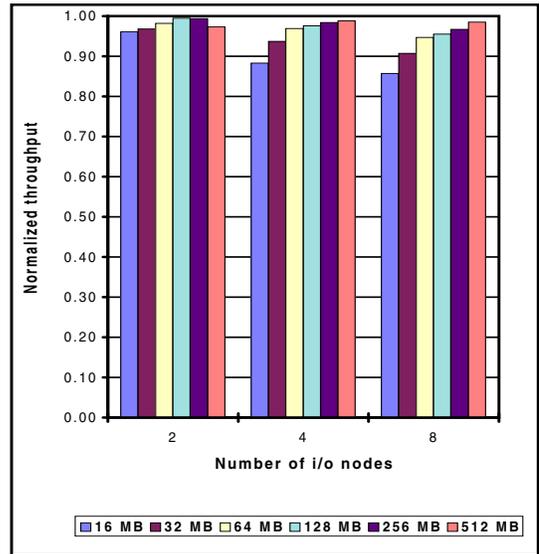
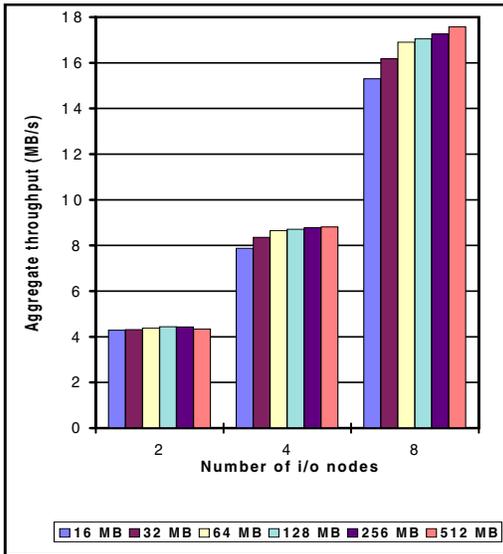


Figure 4: Aggregate and normalized throughput for writing arrays of size 16 MB to 512 MB from 8 compute nodes as a function of the number of i/o nodes, using natural chunking.

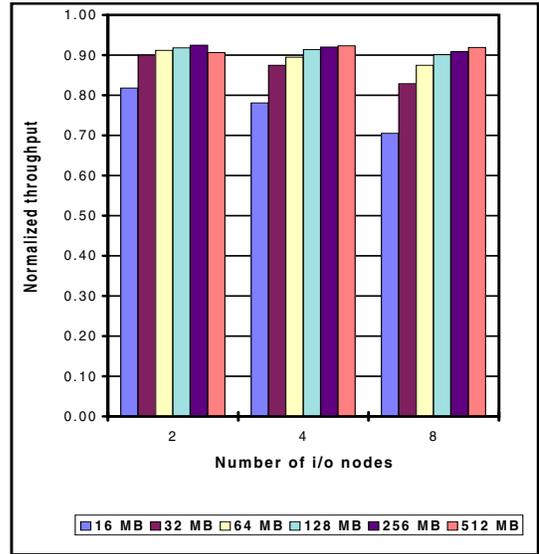
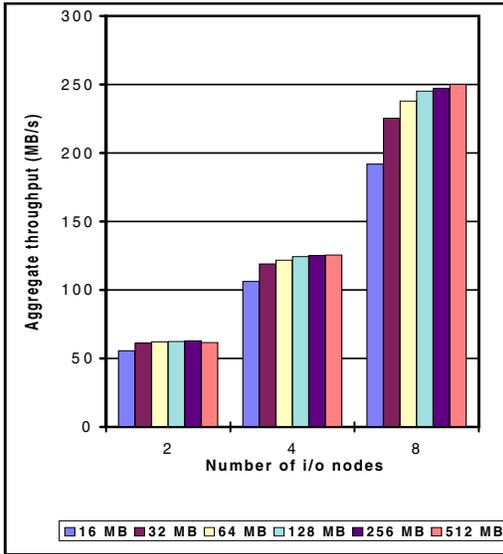


Figure 5: Aggregate and normalized throughput for reading arrays of size 16 MB to 512 MB from 32 compute nodes as a function of the number of i/o nodes using natural chunking and simulating an infinitely fast disk.

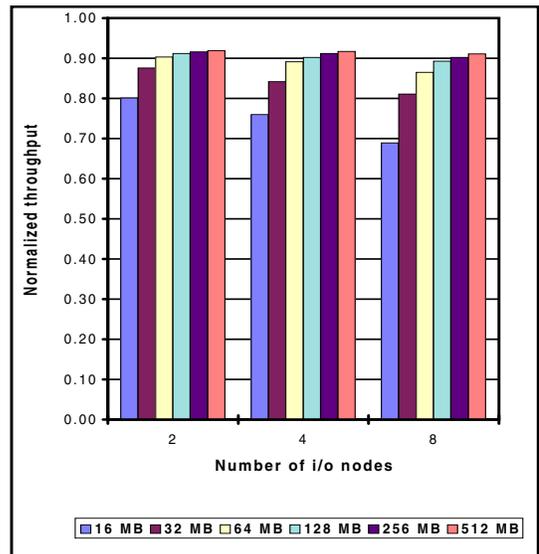
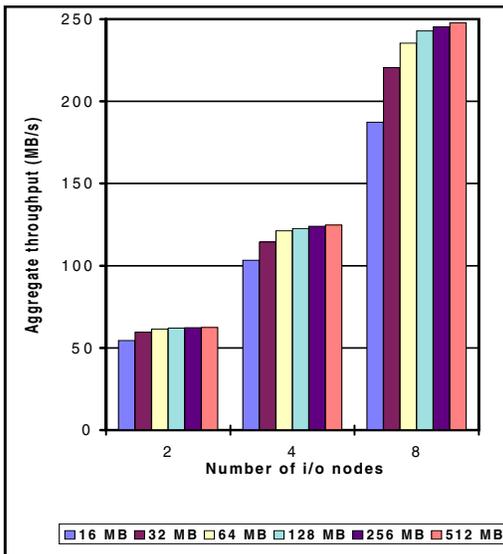


Figure 6: Aggregate and normalized throughput for writing arrays of size 16 MB to 512 MB from 32 compute nodes as a function of the number of i/o nodes using natural chunking and simulating an infinitely fast disk.

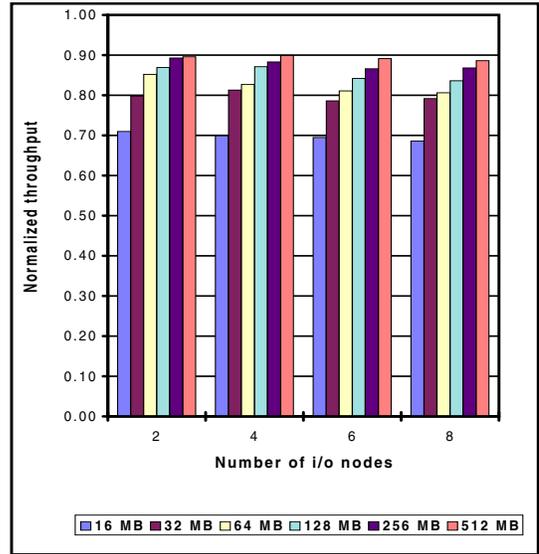
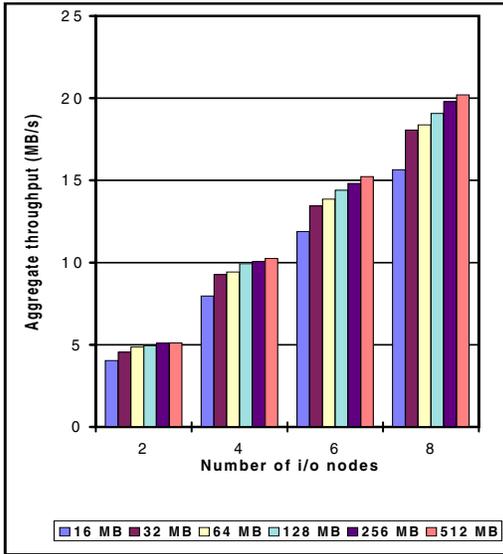


Figure 7: Aggregate and normalized throughput for reading arrays of size 16 MB to 512 MB from 32 compute nodes as a function of the number of i/o nodes, using traditional order on disk.

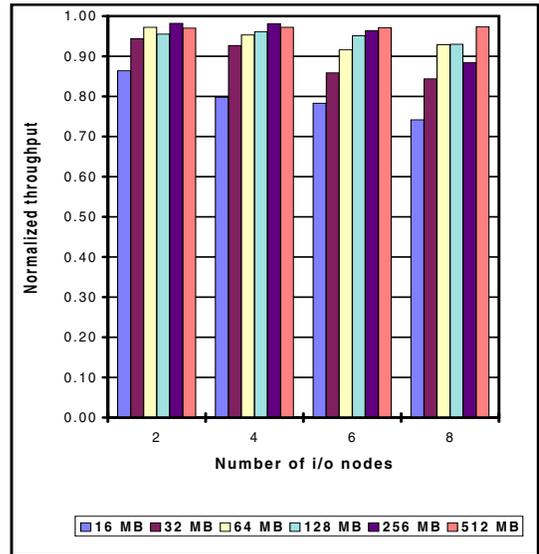
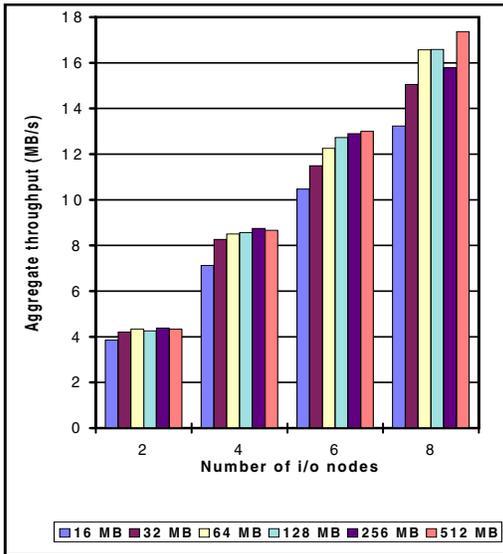


Figure 8: Aggregate and normalized throughput for writing arrays of size 16 MB to 512 MB from 32 compute nodes as a function of the number of i/o nodes, using traditional order on disk.

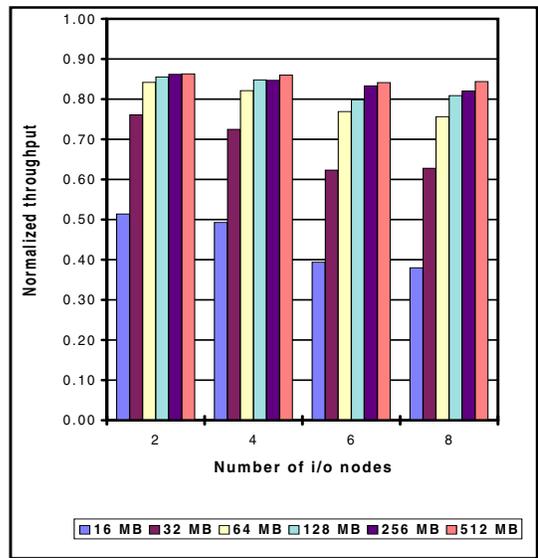
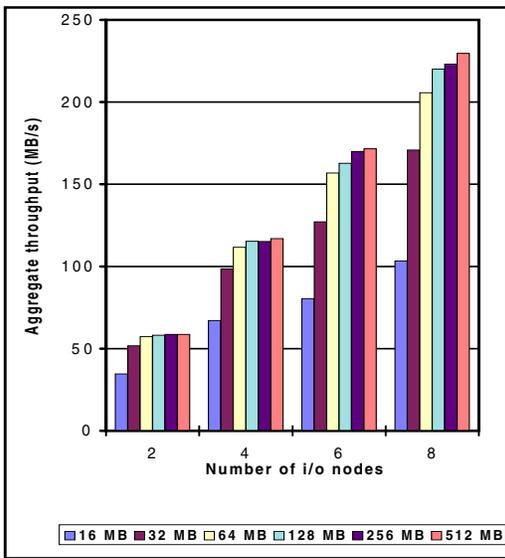


Figure 9: Aggregate and normalized throughput for writing arrays of size 16 MB to 512 MB from 16 compute nodes as a function of the number of i/o nodes, using traditional order on disk and simulating an infinitely fast disk.