6-5-1996

The Panda Array I/O Library on the Galley Parallel File System

Joel T. Thomas
Dartmouth College

The Panda Array I/O library on the Galley Parallel File System

Joel T. Thomas

Dartmouth Computer Science Technical Report PCS-TR96-288
Senior Honors Thesis
Dept. of Computer Science
Dartmouth College
Joel.T.Thomas@dartmouth.edu

June 5, 1996

Abstract

The Panda Array I/O library, created at the University of Illinois, Urbana-Champaign, was built especially to address the needs of high-performance scientific applications. I/O has been one of the most frustrating bottlenecks to high performance for quite some time, and the Panda project is an attempt to ameliorate this problem while still providing the user with a simple, high-level interface. The Galley File System, with its hierarchical structure of files and strided requests, is another attempt at addressing the performance problem. My project was to redesign the Panda Array library for use on the Galley file system. This project involved porting Panda's three main functions: a *checkpoint* function for writing a large array periodically for 'safekeeping,' a *restart* function that would allow a checkpointed file to be read back in, and finally a *timestep* function that would allow the user to write a group of large arrays several times in a sequence. Panda supports several different distributions in both the compute-node memories and I/O-node disks.

We have found that the Galley File System provides a good environment on which to build high-performance libraries, and that the mesh of Panda and Galley was a successful combination.

1 Introduction

The Galley Parallel File System [1, 2, 3] is a powerful system for building parallel I/O applications. To demonstrate the viability of the file system, it is useful to implement software that can take advantage of its features and show true performance boosts compared to other more 'traditional' parallel file systems. My project consisted in taking the Panda Array I/O library [4, 5, 6] from the University of Illinois as a base and recreating part of their work on the Galley File System.

2 Background

The goal of Panda is to provide a high-level, intuitive interface for dealing with the large arrays that are normally used in scientific computing. At the same time, the implementation of the various distributions should be as efficient as possible so that the user does not have to suffer poor performance. The user creates *Array* objects by specifying rank, element size, and dimension sizes to an Array constructor. The user can then *insert* one or more arrays into *ArrayGroups*. There are three kinds of ArrayGroup

objects that Panda supports: the TimeStep object, which is used to periodically write the group, the CheckPoint object, which stores a snapshot of a group of arrays, and a Restart object, which rereads in the information from a CheckPoint file. When a user calls the `timestep` method on a TimeStep object, every Array in that object is collectively written to the end of the TimeStep file. The filename is a parameter to the TimeStep constructor. As further timesteps are called, they are appended to the same TimeStep file. Similarly, the `checkpoint` method of a CheckPoint object collectively writes all Arrays in the CheckPoint object. Further calls to the checkpoint method do not write new data in separate places as a TimeStep would, but instead store only one valid checkpoint at a time. Finally, the `restart` method of a Restart object rereads a series of checkpointed Arrays into the compute-nodes' memories.

In addition to supporting these various ArrayGroup objects, Panda allows different data distributions to be used in the compute-node memories and I/O-node disks. For instance, using the HPF terminology [7], one could have a (BLOCK, *) distribution on the compute nodes, but for whatever reason, use a (*, BLOCK) distribution when writing to disk.

The purpose of this project was to retain most of Panda's high level interfaces with dealing with large arrays, but to change some of the actual implementation to take advantage of the Galley File System.

The Galley File System is a parallel file system that tries to maximize flexibility and I/O performance. Instead of using the conventional file-system interface that is similar to UNIX, Galley uses a hierarchical file structure that can give the library designer much more power when dealing with I/O. Instead of seeing files as a long stream of bytes, files are split into *subfiles*, usually one per disk. These subfiles can then be subdivided into several segments, called *forks*, in which actual data can be written. Subfiles are referred to by numbers that Galley assigns, and forks are referred to by names that the user provides when they are created. By careful use of this hierarchical format, a library builder can take advantage of efficient I/O by using strided calls instead of a large number of relatively smaller-sized traditional I/O calls.

3 Project Description

Panda 2.0 uses a method called *server-directed I/O*. This method in effect leaves the I/O processors (IOPs) in charge of the flow of information. After a master server has been informed by the master client node that a collective write or read is required, the process is then directed from the IOP's point of view. Let us examine a collective write. Each IOP contacts the various compute nodes that carry parts of the chunk it requires. This contacting would consist of several MPI messages sent to the various compute nodes. Once the appropriate information is collected from all of the compute nodes, the IOP has enough data to complete the write. A reverse operation would occur for a collective read.

To use Panda successfully on Galley, we had to redesign the flow of information. The IOPs would not be able to make requests of the compute nodes in the fashion that the original Panda project wished, so we had to reconfigure the existing code so that the compute nodes were able to request (or send) precisely the data they needed from (or to) the IOPs. We no longer used Panda's I/O servers, but instead used Galley's interface

to its IOPs. By forming concise Galley calls for large chunks, we made large reads and writes using strided calls.

The final implementation made many changes to Panda's original design. First and foremost were the changes needed to the file structure. Let us first discuss the changing of the TimeStep setup. Originally, Panda had several files associated with an I/O server's portion of the file. One described the *schema* information and the others were the actual data. For instance, consider a timestep object called "z1timestep." Then the files created would be:

z1timestep.2	schema file for 2nd timestep
z1timestep.2.0	data file for 2nd timestep, I/O node 0
z1timestep.2.1	data file for 2nd timestep, I/O node 1
z1timestep.2.2	data file for 2nd timestep, I/O node 2
z1timestep.3	schema file for 3rd timestep
z1timestep.3.0	data file for 3rd timestep, I/O node 0
z1timestep.3.1	data file for 3rd timestep, I/O node 1
z1timestep.3.2	data file for 3rd timestep, I/O node 2
etc.	

In the Galley setup, we had only one Galley file, named the same as the timestep object. All schema files in the old system are stored in subfile 0 of the Galley file as separate forks, with names of the form:

schema<#>

where <#> is timestep number.

To store the actual data, appropriate forks were created in each subfile of the form:

data<#1>_<#2>

where

<#1> is timestep number and

<#2> is a 'copy number,' explained below with regard to CheckPoint files.

In the case of TimeStep, the copy number is always 0, since only one copy is kept of each timestep. Thus, all forks of the form "data0_0" are associated with one timestep, all the forks of the form "data1_0" are associated with the next timestep, and so on. Below is a picture of the current setup using Galley.

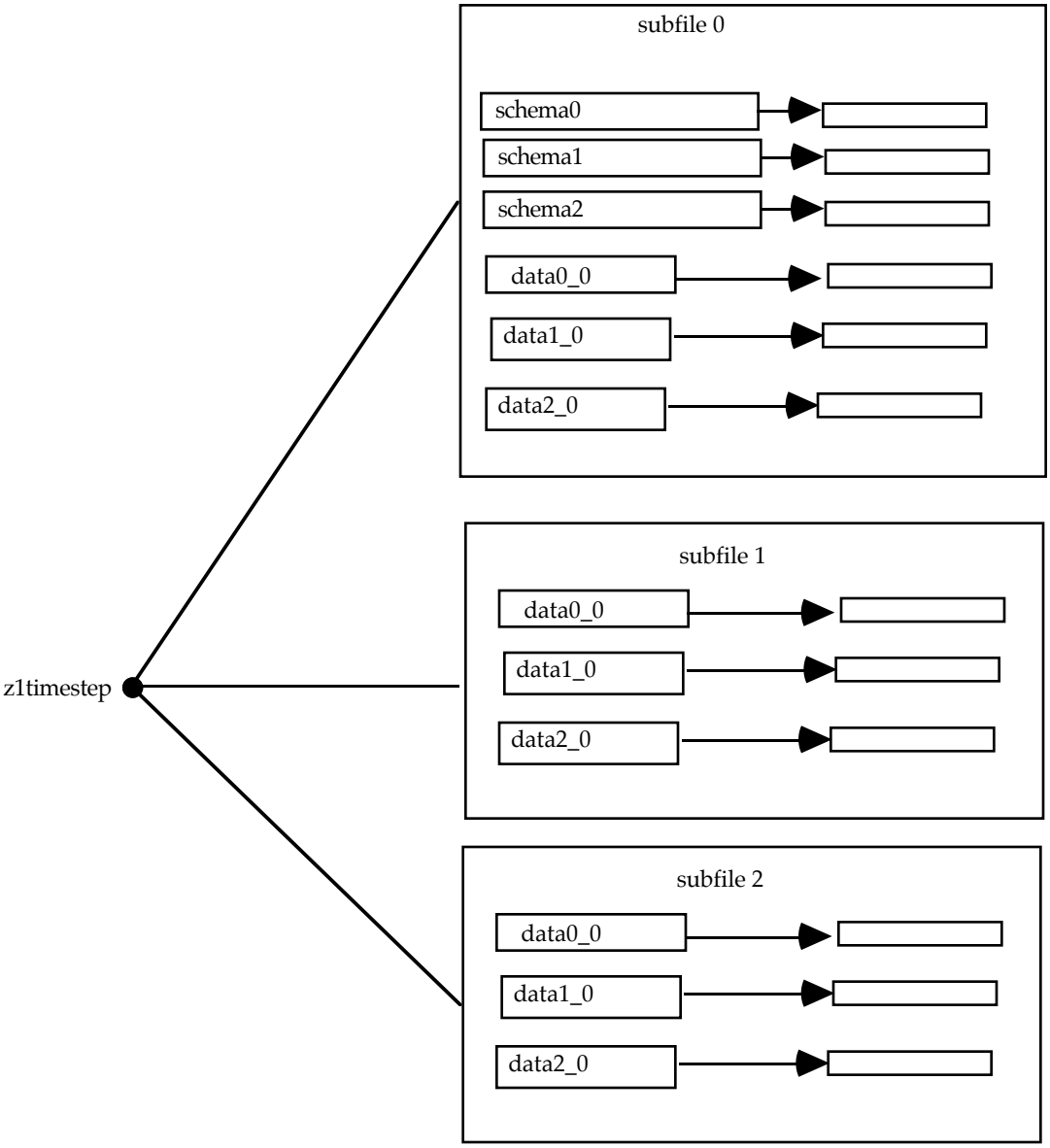

Figure 1: TimeStep file

The CheckPoint file structure is shown below:

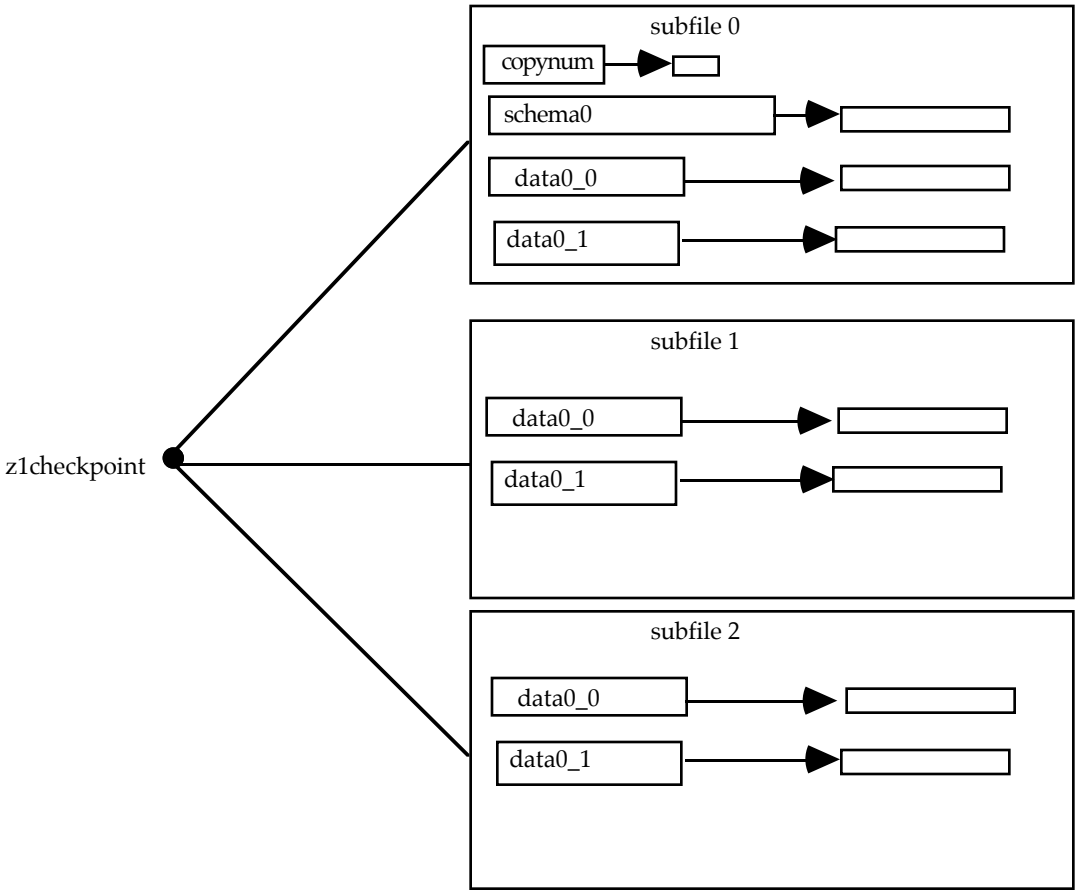

Figure 2: CheckPoint file

A CheckPoint file was used to periodically save a snapshot of a large array in case of a system crash, so that the user could restart from that snapshot and continue with the simulation. All the schemas were stored in subfile 0 as is done in the TimeStep file. Unlike a TimeStep file, each data fork in a CheckPoint file had two counterparts. We alternated which set of forks we write, so that in case a crash occurred in the middle of a checkpoint, we did not destroy good data.

We clearly must have some way of knowing which set of forks in the CheckPoint file are actually the valid data when we want to do a Restart. In subfile 0, there is a special fork called "copynum." Within this fork, we store the copynumber (0 or 1) of the forks that are valid. This fork is only written after a new checkpoint operation has been completed. By combining this copynumber with "data0_" as we did when we wrote the CheckPoint, we access the appropriate forks during a Restart operation. For instance, assume copynum contained 1, and the name of the checkpoint file was z1checkpoint. Then we must access fork data0_1 for each subfile of z1checkpoint to reread all the data.

As we altered the various ArrayGroup objects to take advantage of Galley, we also had to keep in mind that we were supporting Panda's different distributions on the compute-node memories and I/O-node disks. We decided to keep within the most basic distributions for this project, while allowing space for growth later. We currently

support (*,*), (BLOCK,*), (*,BLOCK) and (BLOCK,BLOCK) in the compute-node memories, and (*,*) and (BLOCK,*) on the I/O node disks.

The following is an example of how we used Galley's non-blocking strided calls to write these distributions appropriately to the forks.

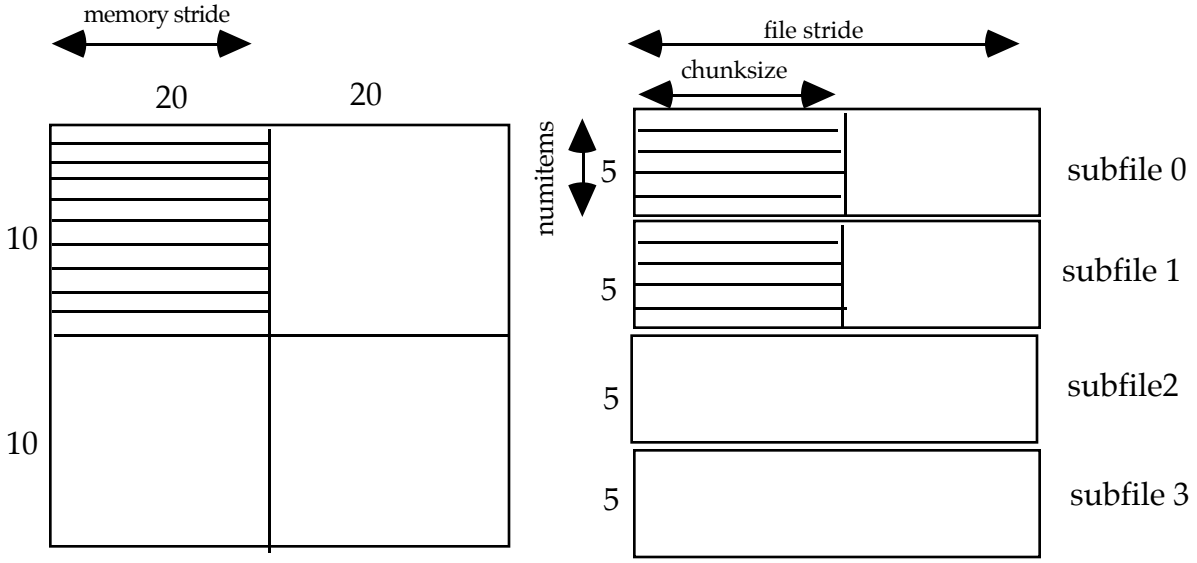

Figure 3: (BLOCK, BLOCK) to (BLOCK,*) distribution

The figure on the left shows a (BLOCK, BLOCK) distribution of a 20x40 byte array. The array is to be distributed among the disks in a (BLOCK, *) format, illustrated on the right. In this example, we are following what CPnum 0 does. To write all of CPnum 0's local data, we need two separate Galley strided calls. One strided call will write 5 items of size *chunksize* to subfile 0, while the next call will write 5 items of size *chunksize* to subfile 1. The strided call skips the appropriate stride in both memory (20 bytes) and the file (40 bytes) after each write. Thus, with appropriate calculations, one can make several concise Galley strided calls and efficiently write large data sets.

An example of the Panda objects in a C++ program is shown in Figure 5 at the end of this paper. As shown, the user specifies the size of the global array, and the distribution layouts on both the compute-node memories and the IOP disks. They create a `Panda` object to initialize Panda and MPI. They then create `ArrayLayouts` for both memory and disks. Galley also needs an initialization call. At this point, we can create an `Array` object and fill it with test data using `background_fill`. Testing is thus easy, because each record of the array is assigned a unique value. After we create a `TimeStep` and `CheckPoint` object, we can insert our newly created `Array` into those two `ArrayGroups`, and then call timestep and checkpoint as needed in our computations.

To support (BLOCK, *) using Galley's strided calls was an intriguing matter. The problem was that strided calls can only be made one fork at a time. One may initially think that we could make one strided call per fork to write all the data in our local chunk. However, it is possible for one chunk of data to be split among forks such that it begins on the end of fork n and ends somewhere in the beginning of fork n+1. Figure 4 illustrates a possible situation:

Figure 4 : Illustration of a chunk crossing two subfiles

The black chunks in Figure 4 are the parts of the fork we actually want to write to, while the white parts are the parts of the file we are skipping over. As shown, the last data chunk to be written is partly written on subfile n and partly on subfile n+1. A strided call by itself cannot handle this situation if it arose. Thus, we had to accompany our strided calls with two simple standard Galley write calls that do not use the strided interface, but merely write a specified number of bytes started at a given offset in a fork. Thus, our pseudocode representing the process may look something like this.

For all subfiles:
1) If we have leftover data from the last fork, do a simple write of that data.
2) Calculate how many chunks of data can fit within this fork.
Call a strided write for that many chunks of data.
3)If there is leftover data to be written in this fork that was not handled by the strided write, call a simple write to complete the fork.

This complicated our code a little more than we would have liked. While the Galley strided interface is convenient and powerful, not every pattern is conveniently strided. (Note that Galley's nested-batched interface [4] could express the three writes in one call, although we did not use that call.)

4 Experimental Setup

To test Panda's performance on Galley, we wanted to run a series of tests that would push the library to the limit. The FLEET lab at Dartmouth College has 8 RS/6000 computers (64 MB RAM each) running AIX 4.1. There is a total of 24 gigabytes of hard drive space, with 3 gigabytes attached to each processor. Each RS/6000 is also connected to an Ethernet and FDDI network. Our tests were run over the FDDI network.

Each test consisted of one timestep of varying array sizes. We decided to use a fairly wide range of array sizes. We used array sizes of 16, 32, 64 and 128 total MB for all distributions of (*,*), (*,BLOCK), (BLOCK,*), (BLOCK,BLOCK) in the compute-node memories. The distribution on the IOP disks were always (BLOCK,*). In all the tests we used 4 processors as compute nodes, distinct from the nodes used as IOPs. To add another dimension to the testing for writing, we also ran these combinations with either 2,3 or 4 IOPs, which in effect made us use 2,3, or 4 subfiles for each test that we ran. For reading, tests were only run with 4 IOPs.

During the tests, each compute node filled its portion of the array with test data depending on its location in the global array. Then a barrier was called before the I/O operation. After the I/O operation completed, the compute nodes called another barrier. We measured the elapsed time between barriers. After each test, we flushed Galley's cache. We ran three tests for each data set, and computed the average time. We

computed the average throughput to be the number of bytes in the global array divided by the average elapsed time. To do some further analysis on each of the three tests, we also computed a coefficient for each triple. This coefficient consisted of (maximum_time - minimum_time)/average time. These coefficients can be used to give a better understanding of how reliable the results are.

5 Results and Discussion

Figures 6-9 at the end of this paper show the graphs for writing` for each combination on the compute-node memories. The data sets for reading are shown in figures 10-13, although tests were only run for 4 IOPs for reading. There is one graph for each of the distributions. In each graph, the various array sizes used are clustered by the number of IOPs used in the testing. Each value represents the throughput in MB/s for that array size and number of IOPs. The range of coefficients calculated (as discussed in section 4) is also displayed with each graph.

We have found that Panda performs fairly well on Galley with various data sets. Since the original Panda library's performance tests were tested with quite a different configuration, and since the FLEET lab limited the number of compute nodes and IOPs we could use, it is somewhat hard to compare to their original results. However, we feel that for a 4 compute node configuration that the results look promising if one were to scale to a higher level.

More IOPs (and correspondingly, more disks) appear to in general lead to increased throughput. A maximum of approximately 4.8 MB/s for writing was achieved with 4 IOPs. Overall, the (BLOCK,*) to (BLOCK,*) distribution achieved the best results. This could possibly be the result of the fact that this is in essence "natural chunking," where the compute-node memories and the IOPs have the same distributions. In essence, we can do four straight writes of a contiguous chunk on 4 separate disks at the exact same time, which should likely lead to greater possible throughput.

Similar results were achieved with regard to reading tests. There did seem to be some improvement in throughput for reading, with a maximum of approximately 8 MB/s. Some of the reading results may be the result of possible leftover caching effects, although every effort was made to test in such a fashion as to counter those effects.

On average, we achieved approximately 1.5 to 2.5 MB/s per IOP, which is in the range of what Galley is capable of. Thus, the results found for both reading and writing seem reasonable.

We feel that the combination of Panda and Galley is a promising one. Since their goals are very closely linked, one would hope that their marriage would bring them both even closer to the goal of efficient I/O. Since we were able to keep Panda's simple interface and take advantage of Galley's powerful features, we are hopeful that the same can be done for later Panda versions fairly straightforwardly.

Another lesson to be learned from this project is the usefulness of Galley as an environment for building libraries. While the filesystem itself is limited in some respects with regard to file management, its interface is fairly easy to understand and can be useful if your program is designed appropriately. The hierarchical nature of the filesystem can be put to a great advantage.

While there were some fairly great initial efforts to redesign the original Panda concept to work on Galley, the port itself seemed straightforward in some respects after it was completed. Perhaps this is the advantage of hindsight. In any case, we feel that future ports of later and improved versions of Panda will likely go more smoothly than this initial attempt.

6 Summary

We have discussed the combination of the Panda Array I/O Library and the Galley File System. Both have a goal of efficient I/O and both strive to maximize performance. While the initial makeup of the Panda library could not be used with Galley, we were able to make the appropriate redesigns to keep much of the functionality of the original Panda, as well as its interfaces, and yet take advantage of Galley's strengths.

We have found that while altering Panda to mesh with the Galley File System took some initial redesigning time, afterwards the combination seemed to work well together. Galley's strided interface allowed simple, concise calls to get a large amount of I/O work done with good performance. However, because of the "leftover data" problem discussed in section 3, we had to implement some non-intuitive workarounds for Panda to perform correctly. Thus, the combination, while perhaps not a perfect match, seems like a workable and viable amalgam.

7 Future Work

Panda version 2.1 [1] should be coming out soon after release of this paper. Version 2.1 will apparently be quite an overhaul of 2.0. Since 2.1 is only available internally at the University of Illinois, we do not know for certain how different it will be from 2.0. Our impression is that the current code will probably not be directly transferable from 2.0 to 2.1. The main structure of the added code, however, should provide a good basis for someone interested in working with Panda 2.1 and Galley.

Because of the structure of Panda 2.0, there is no access to arrays that you are manipulating (writing and reading). As Ying Chen from the University of Illinois said, "That is not our purpose, either. The way that Panda stores/read arrays should have nothing to do with how users manipulate array elements." Thus, there is currently no convenient function to access the arrays, since Panda itself is doing the memory allocation and does not provide methods of its own for doing this. The main function of Panda 2.0 is to illustrate the advantages of being able to distribute differently on the compute nodes and I/O nodes, the advantages of server-directed I/O, and the convenient interface. In Panda 2.1, however, the user should be able to allocate memory at the application level, pass a pointer to this allocated space to the Panda library calls, and then manipulate the arrays using normal array notation.

Acknowledgments
Thanks to my thesis advisor, Prof. David Kotz, for his time and help on this project. Being able to take advantage of the FLEET lab of Dartmouth College was an exciting opportunity, and I am glad to have had the chance. Since the whole project is built upon

Galley, I am obviously greatly indebted to Nils Nieuwejaar. Thanks also to Nils for his help and advice when working with Galley. Thanks of course to the Panda team at the University of Illinois for the use of their source code, as well as their helpful responses via e-mail.

```
// Time Step with 4 compute nodes and 3 IOPs
int main(int argc, char **argv)
{
    TimeStep *t1;
    CheckPoint *c1;
    Array *array1;
    Array *array2;
    ArrayLayout *mem1;
    ArrayLayout *mem2;
    ArrayLayout *disk1;
    ArrayLayout *disk2;
    int arrayrank = 2;
    int arraysize[] = {3000,4000};
    int esize = 4;            // element size
    int mrank = 2;
    int mlayout[] = {2,2};      // memory layout, block_block
    int drank = 2;
    int dlayout[] = {3,1};    // disk layout,block_none
    int io_nodes = 3;

    Distribution mem_dist[] = {BLOCK,BLOCK};
    Distribution disk_dist[] = {BLOCK,NONE};

    Panda * bear = new Panda(GalleyMPIFS,io_nodes,argc,argv);

    // Set up memory and disk layouts
    mem1 = new ArrayLayout ("MemArrayLayout" ,mrank,mlayout,
                                        bear->me_in_group());
    disk1 = new ArrayLayout("DiskArrayLayout",drank,dlayout,
                                        bear->me_in_group());

    // init Galley File System
    gfs_init(argc, argv, NULL);

    // Array
    array1 = new Array("z1Array",arrayrank,arraysize,esize,
                      mem1,mem_dist,disk1,disk_dist,mstrides,
                dstrides,io_nodes);
    // Fill our part of the array with appropriate test data
    array1->background_fill();


    // Time Step
    t1 = new TimeStep("z3TimeStep",  "z3timestep" );
    t1->insert(array1);

    c1 = new CheckPoint("zsCheckPoint", "zscheckpoint");
    c1->insert(array1);

    // Perform several timestep, checkpointing periodically

    for (int idx=0; idx<10; idx++)
        {
            t1->timestep();
            if (idx==5)
                c1->checkpoint();
        }

    // delete all objects created
    delete t1;
    delete array1;
    delete disk1;
    delete mem1;
    delete bear;

    gfs_finalize();
    return(0);
}
```

Figure 5: Sample Panda program on Galley

Graphs for Writing

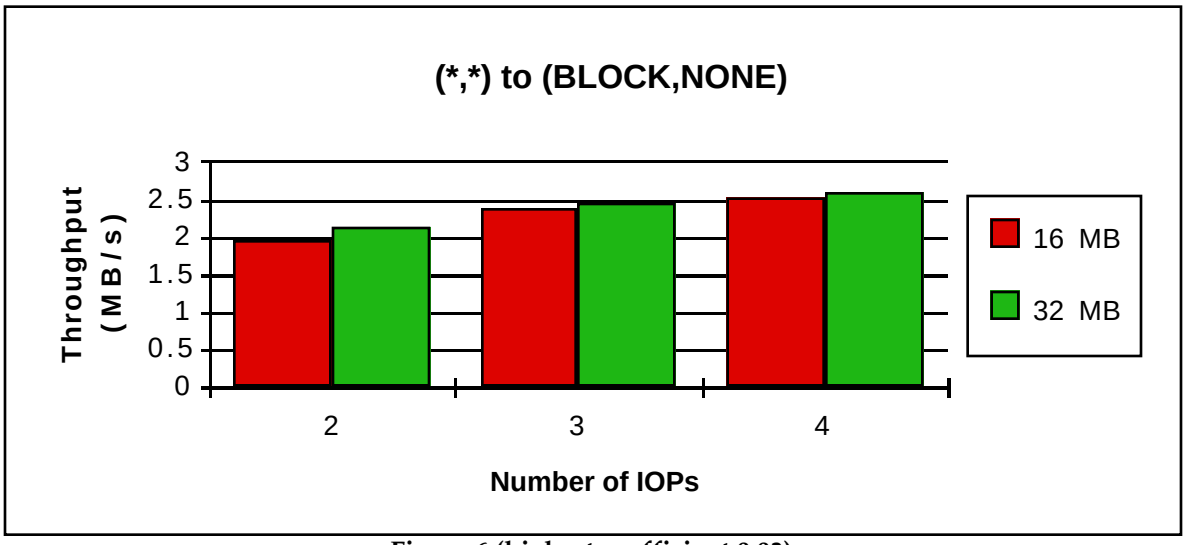

Figure 6 (highest coefficient 0.03)

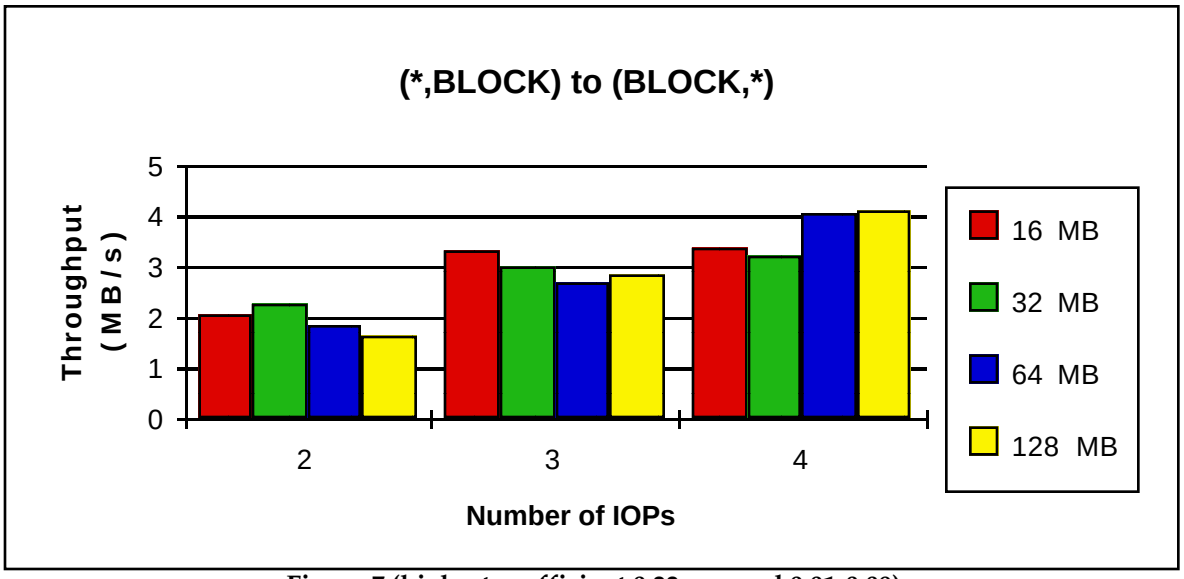

Figure 7 (highest coefficient 0.22, general 0.01-0.09)

Graphs for Writing

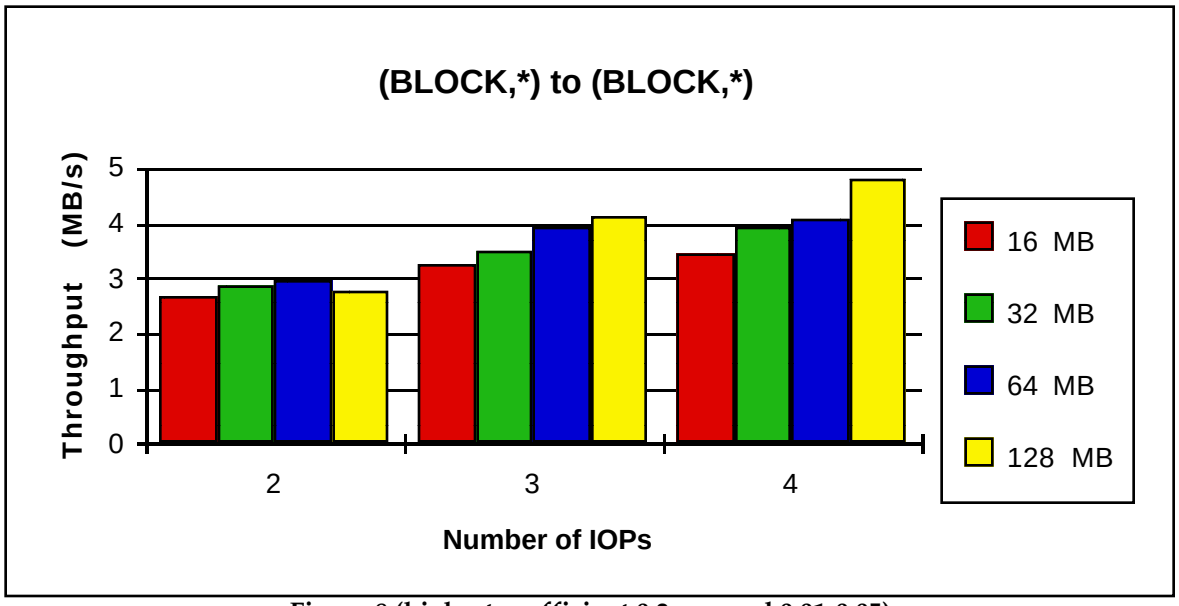

Figure 8 (highest coefficient 0.2, general 0.01-0.05)

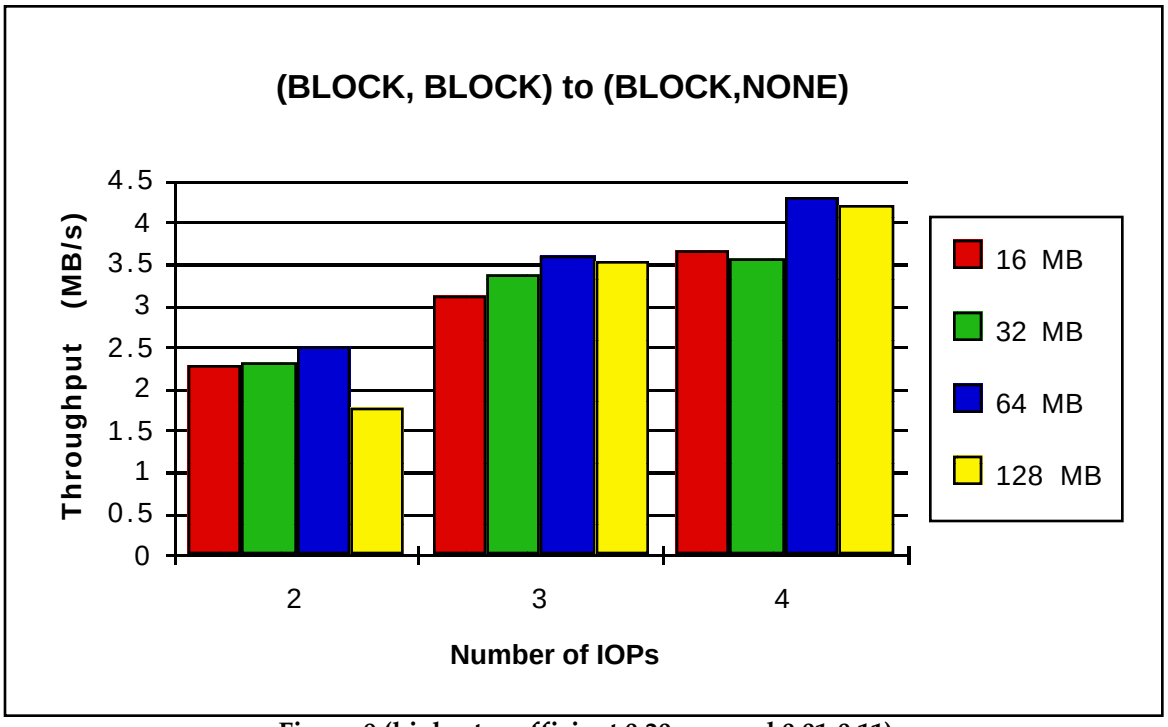

Figure 9 (highest coefficient 0.20, general 0.01-0.11)

Graphs for Reading

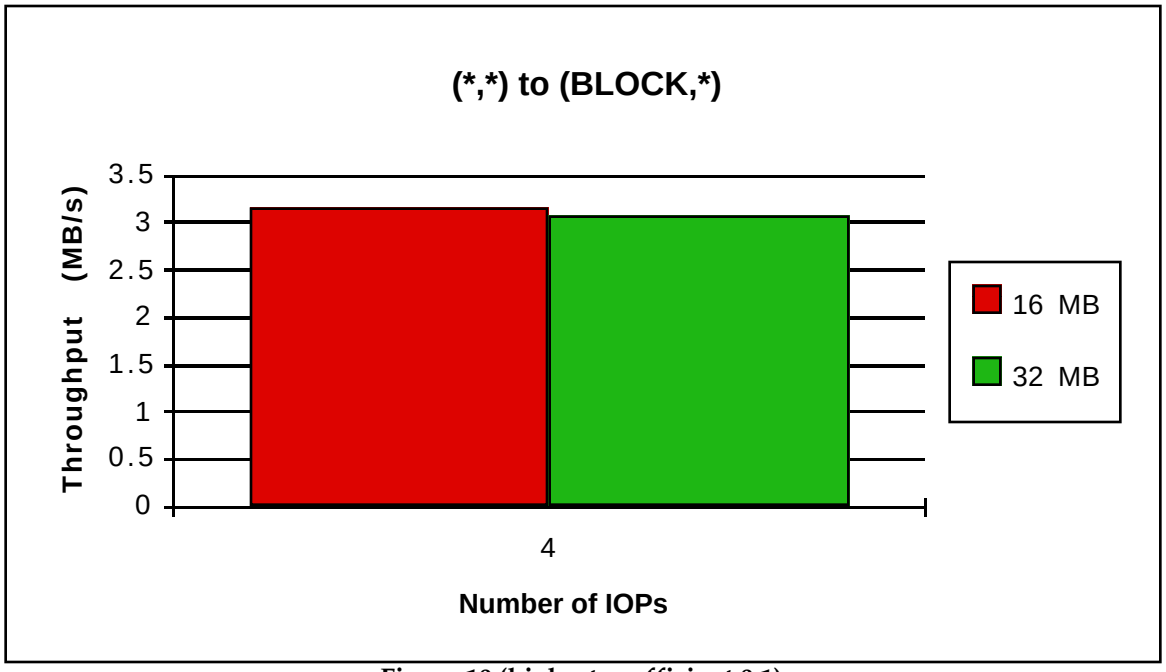

Figure 10 (highest coefficient 0.1)

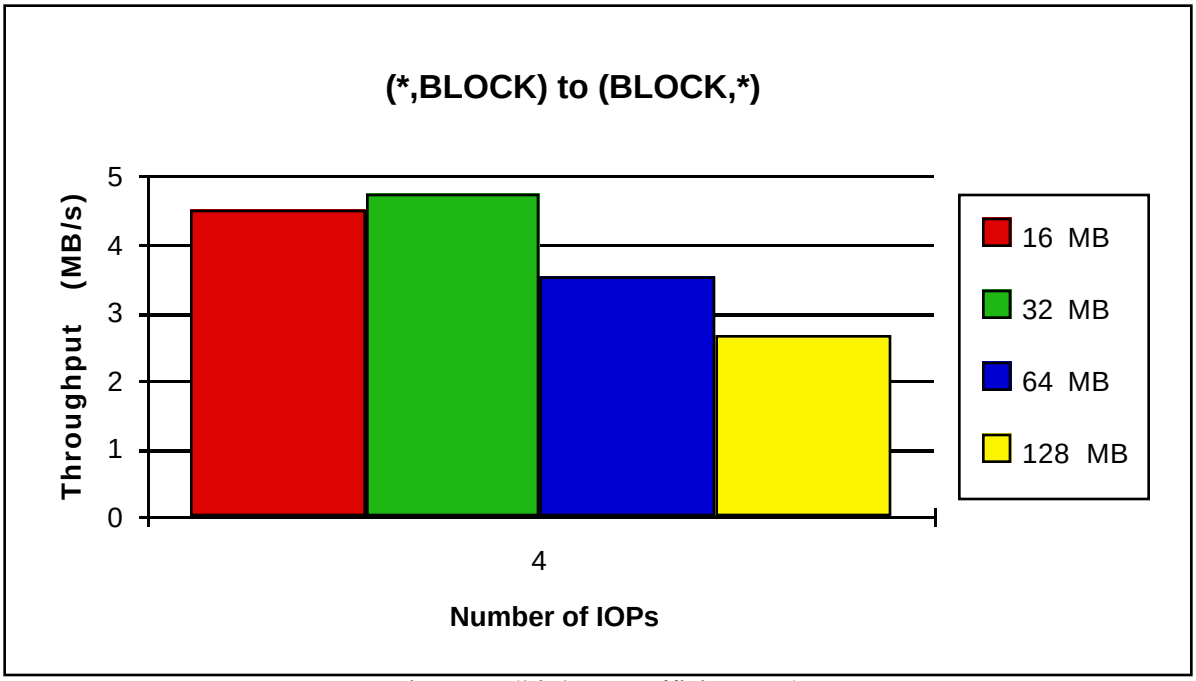

Figure 11 (highest coefficient 0.1)

Graphs for Reading

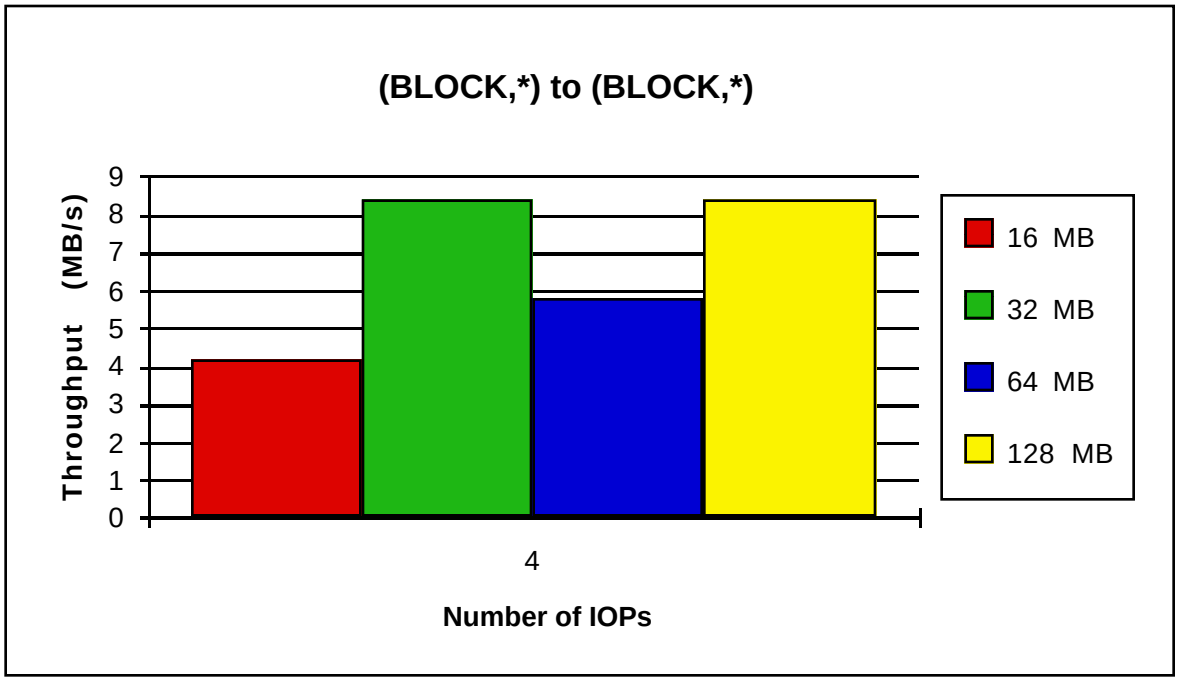

Figure 12 (highest coefficient 0.1)

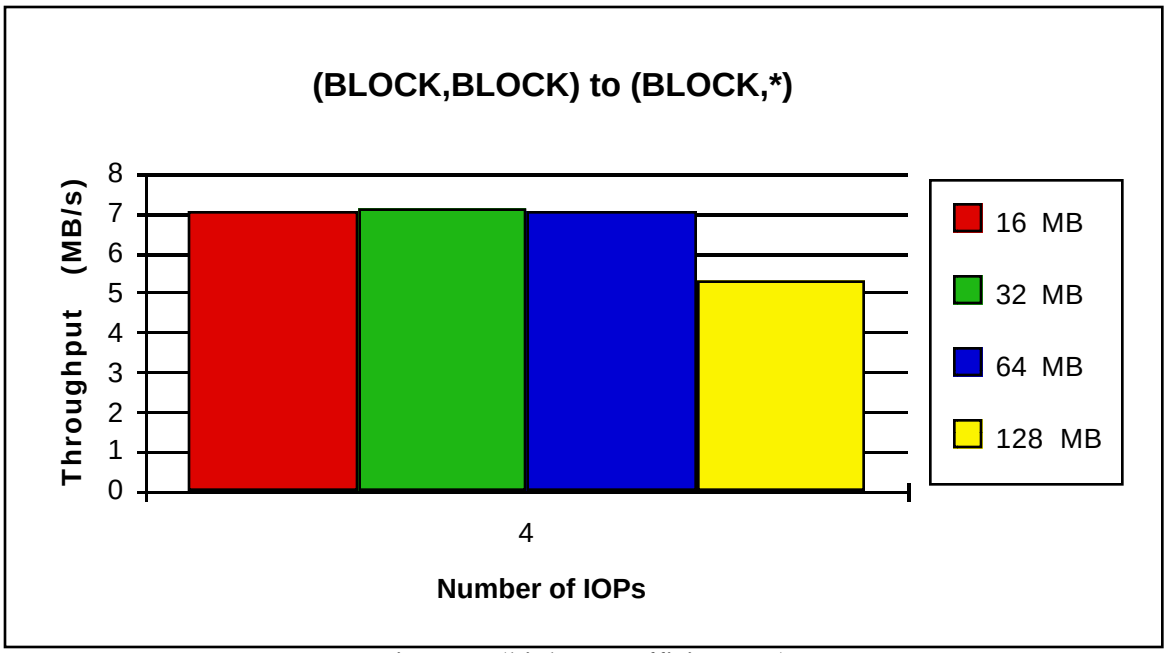

Figure 13 (highest coefficient 0.1)

References

[1] Y. Chen, M. Winslett, K. E. Seamons, S. Kuo, Y. Cho, and M. Subramaniam. Scalable message passing in Panda. In Fourth Workshop on Input/Output in Parallel and Distributed Systems, pages 109-121, Philadelphia, May 1996.

[2] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In Proceedings of Supercomputing '95, December 1995.

[3] K. E. Seamons, Y. Chen, M. Winslett, Y. Cho, S. Kuo, P. Jones, J. Jozwiak, and M. Subramanian. Fast and easy I/O for arrays in large-scale applications, October 1995. At SPDP'95.

[4] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems, pages 47-62, April 1995.

[5] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In Proceedings of the 10th ACM International Conference on Supercomputing, pages 374-381, May 1996.

[6] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In Fourth Workshop on Input/Output in Parallel and Distributed Systems, pages 83-94, May 1996.

[7] David B. Loveman. High Performance Fortran. IEEE Parallel and Distributed Technology, 1(1):25-42, February 1993.