Parallel Input/Output with Heterogeneous Disks

S. Kuo, M. Winslett, Y. Chen, Y. Cho
Computer Science Department
University of Illinois
Urbana IL 61801
s-kuo,winslett,ying,ycho@uiuc.edu

M. Subramaniam
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
masubram@us.oracle.com

K. Seamons
Transarc Corporation
The Gulf Tower
Pittsburgh, PA 15219
seamons@transarc.com

Abstract

Panda is a high-performance library for accessing large multidimensional array data on secondary storage of parallel platforms and networks of workstations. When using Panda as the I/O component of a scientific application, H3expresso, on the IBM SP2 at Cornell Theory Center, we found that some nodes are more powerful with respect to I/O than others, requiring the introduction of load balancing techniques to maintain high performance. We expect that heterogeneity will also be a big issue for DBMSs or parallel I/O libraries designed for scientific applications running on networks of workstations, and the methods of allocating data to servers in these environments will need to be upgraded to take heterogeneity into account, while still allowing users to exert control over data layout. We propose such an approach to load balancing, under which we respect the user's choice of high-level disk layout, but introduce automatic subchunking. *The use of subchunks allows us to divide the very large chunks typically specified by the user's disk layout into more manageable-size units that can be allocated to I/O nodes in a manner that fairly distributes the load. We also present two techniques for allocating subchunks to nodes, static and dynamic, and evaluate their performance on the SP2.*

1. Introduction

Scientific applications often center around large multidimensional arrays [1] and require the use of some form of persistent data storage. For example, they need to periodically output the current state of computation to files which outlive the invocation of the programs that created them, or read array data from files which existed before the execution of the programs. Often these applications are I/O bottlenecked [19], i.e., I/O time will occupy a significant fraction of total run time, because of the mismatch in development in CPU and secondary storage devices. When running on a parallel platform, the I/O bottleneck problem is worsened as computation time is further reduced by using multiple processors in parallel to solve the application problem. Declustering has been used to alleviate this problem by using multiple disks in parallel, where each disk is controlled by a processor (I/O node), to increase I/O bandwidth for outputting data to secondary storage. However, without special care for organizing the data on multiple disks and optimizing the movement of data to/from secondary storage, the aggregate bandwidth of the multiple disks cannot be fully utilized. Therefore, it is crucial to application scientists to have an easy and efficient way of accessing persistent data storage, provided by a database management system (DBMS) or an I/O library.

Panda (http://drl.cs.uiuc.edu/panda/), a DBMS-style I/O library developed at the University of Illinois, is designed for this need — facilitating storage management for scientists by providing an easy-to-use high-level interface, making applications portable to different sequential platforms, parallel platforms, and networks of workstations without applying machine-dependent I/O optimizations, and, most importantly, offering high performance. In performance tests on the NASA Ames Research Center's IBM SP2, Panda achieved excellent I/O performance for input and output of very large arrays under a wide variety of conditions [18], using dedicated I/O nodes ("full-time I/O nodes"). Thus we felt ready to try Panda with a "real" scientific application, H3expresso, supplied by the International Numerical Relativity Group at the National Center for Supercomputing Applications at the University of Illinois, headed by Prof. E. Seidel.

Three issues were discovered when we ran Panda with H3expresso on the IBM SP2 at Cornell Theory Center,

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 58552, and by NASA under NAGW 4244 and NCC5 106. Computing facilities were provided by NASA Ames Numerical Aeronautic Simulation (NAS) Division, Cornell Theory Center, and the National Center for Supercomputing Application (NCSA) at the University of Illinois.

where our scientists like to do their runs. First, unlike the NASA Ames SP2 that we used earlier, the Cornell I/O nodes ran at different speeds. This led us to add new features to Panda to guarantee high performance when I/O nodes are heterogeneous. Second, H3expresso users need data to be migrated to other platforms for post-processing; without special care, offloading might easily be the most expensive part of a run. This led us to study alternatives for offloading data and add that functionality to Panda. Finally, H3expresso users are reluctant to dedicate any processors as I/O nodes; this led us to add functionality to Panda to allow a processor to serve both as a compute node and an I/O node ("part-time I/O nodes").

We discussed the solutions to the data migration problem and presented the overall performance of combining Panda with H3expresso in a previous paper [13]; the current paper will focus on handling heterogeneity. The heterogeneity problem has not been explored by existing DBMSs or parallel I/O libraries designed for scientific applications. Most of these works are targeted solely for a homogeneous environment where all I/O nodes are equally capable of performing I/O. However, the I/O heterogeneity problem has been a big issue on the Cornell SP2, because Panda tends to run as slowly as the slowest I/O node. Networks of workstations, with their likely heterogeneity and the possibility of sharing the I/O resources with interactive users, will cause similar I/O performance problems for any scientific DBMS or parallel I/O library.

In the remainder of the paper, we begin by presenting Panda's internal architecture both for part-time I/O nodes and full-time I/O nodes. Section 3 focuses on the approaches we developed for the heterogeneity problem. Performance results and analyses are shown in Section 4. Related works will be discussed in Section 5 and Section 6 concludes this paper.

2. System Architecture of Panda 2.1.1

Panda is designed to support SPMD-style application programs running on ordinary workstations, distributed memory parallel architectures, and networks of workstations. Panda 2.1 [5] is designed for an environment where processors are designated as compute nodes (Panda clients) or full-time I/O nodes (Panda servers). Compute nodes are dedicated to applications' computation and will issue a collective I/O operation provided by Panda when they want to input or output a set of arrays[1]. I/O nodes are normal processors that use local disks to do input/output. As all compute nodes cooperate in issuing the I/O request and I/O nodes direct the flow of reading/writing data from/to secondary storage (as described below), no concurrency control mechanism is required to ensure serializability in accessing disk.

[1]Panda only supports I/O of multidimensional arrays, the most important data type for high-performance scientific computing.

Panda provides users with high-level interfaces to request I/O of a set of arrays directly. For each I/O operation, users describe the arrays to be read/written: number of arrays, their ranks, sizes and locations, size of each array element, the layout of the arrays in memory (compute layout) and on disk (disk layout). A layout is composed of two parts: distribution and mesh. A mesh can be thought of as the arrangement of logical compute nodes or logical I/O nodes into a multidimensional array that determines each processor's logical neighbors. For example, 8 compute nodes can be arranged into a $2 \times 2 \times 2$, 2×4, 4×2, 8×1, or 1×8 mesh. Given a mesh, an array distribution determines how the array data is partitioned into disk chunks or memory chunks across the logical processors. Panda 2.1 supports the HPF BLOCK, CYCLIC(k)[2] and * distributions along each dimension of an array. The layouts for the arrays in memory and on disk may be completely different, in which case Panda automatically rearranges the data from one layout to the other during I/O. "Natural chunking" [17] occurs when the disk and memory layouts are identical. The disk chunks determined by a layout are mapped in a round-robin fashion to the physical I/O nodes present at run-time. Figure 1 gives example layouts.

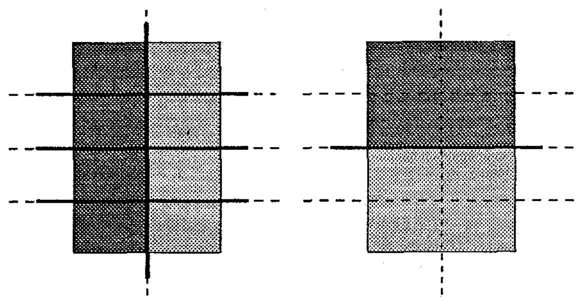


Figure 1. A 2-D array is distributed across 8 compute nodes and 2 I/O nodes. In memory, a 4x2 compute mesh is used; on disk, the left-hand side uses the natural chunking distribution (8 logical and 2 physical I/O nodes) and the right-hand side stores the array in traditional row-major order, split across two I/O nodes (a (BLOCK, *) HPF distribution). Thick lines represent the disk mesh and dotted lines represent the compute mesh. Array chunks filled with different colors are assigned to different I/O nodes.

[2]Panda supports the CYCLIC(k) distribution in memory but not on disk (except as part of natural chunking, described below), because we do not know of any need for cyclic disk distributions. This paper discusses only BLOCK and * disk distributions.

In a heterogeneous environment, one potential concern for I/O is the possibility that different compute nodes would arrive at a collective I/O call at significantly different times. If this occurred, then collective I/O implementations would need to take special steps to handle the staggered arrival times of different compute nodes. However, from discussions with applicaiton scientists, we learned that this type of load imbalance would be an issue only for embarassingly parallel applications, which are far outnumbered by applications in which compute nodes must regularly communicate with their logical neighbors. In the latter type of application, total run time is eventually limited by the speed of the slowest compute node. Thus to take full advantage of a parallel platform, scientists would distribute different amounts of data to different compute nodes, and redistribute at run-time as needed, if either data-dependent or hardware-dependent considerations made load imbalance on the compute nodes a serious consideration. Thus in this paper we assume that all compute nodes reach a collective I/O call at roughly the same time[3].

Thus at roughly the same time, the application processes running on the compute nodes issue a request to each local client to input or output a set of arrays. A selected client sends to a selected server a high-level description of the request (schema message). Once a server receives the schema message, it informs all other servers of the schema information. Based on the schema information, each server can compute the disk layout and determine which disk chunks are its responsibility. Then it plans how to request (for a write operation) chunk data from clients and write gathered data to disk, with the goal to optimize disk accesses. More precisely, to gather a disk chunk, the server determines the indices of the overlapping portion of the assigned chunk on each client, sends a request message to each overlapping client, and then waits for the clients to send the requested data back. After a chunk is gathered, it is written to disk. Read operations follow the same overall strategy, in reverse. We call this architecture "server-directed I/O", and details about its use in Panda 2.1's system architecture, and the results of previous Panda performance studies, can be found in [5, 18].

While collaborating with H3expresso users, we found that application scientists often want to retain as many nodes as possible to act as compute nodes. This motivated us to alter Panda 2.1 (creating version 2.1.1) to support **part-time I/O nodes.** With this approach, any node can act as both a client and a server with respect to I/O. All nodes do computations together, and at I/O time, some of them become I/O nodes. This approach is appropriate for SPMD programs that do parallel I/O on platforms where many nodes have disks, such as the SP2 or clusters of workstations. All nodes work on their local memory chunks during the computation phase. When compute nodes synchronize to do collective I/O, the I/O phase begins and some nodes become I/O nodes, gathering data from clients[4] and writing out one chunk at a time to disk.

[3]We have investigated means of handling staggered arrivals, but will not include that feature in a Panda release until we are convinced of its utility. Also, we are investigating the problem of I/O load imbalance caused by different amounts of data on different compute nodes and/or I/O nodes.

3. Supporting Heterogeneous Disks

To support high-performance I/O with heterogeneous servers, a load-balancing approach is needed to distribute data fairly across servers. The load-balancing approach described in this section is designed for writes, because Panda can control the disk layout for write operations requested by the application. For read operations, either the data was previously written by the application, as for an out-of-core application[5], in which case Panda also controls the layout; or else the data is new input to the application, in which case it is coming directly from outside the SP2 and is not an issue for load balancing (though specialized facilities for bulk data loading may be of benefit). Thus the key is to provide good load balancing in Panda for writes. Complicating any effort at load balancing, however, is the degree of control over data placement traditionally held by application programmers in Panda. Normally users select a disk layout to decompose array data into disk chunks, and Panda internally distributes disk chunks roughly evenly among the I/O servers in a round-robin order. To provide good load balancing, both the decomposition and allocation strategies should be changed.

3.1. Decomposition

With the use of HPF distribution directives in specifying disk layouts, a programmer would expect each disk chunk specified by the disk layout to exist as a single entity on some I/O node. And the obvious approach to allocating chunks to servers with the goal of achieving good load balancing is to assign extra disk chunks to some servers. This direct approach will not help because disk chunks tend to be extremely large (e.g. 1/8 of the entire array, typically hundreds of megabytes), so servers with extra chunks will tend to be overloaded, while others have no chunks at all.

The same load-imbalance problem was encountered in previous Panda out-of-core experiments, which read and write subarrays rather than entire arrays [6]. We found that

[4]As each I/O node has two roles, it might happen that a server needs data from its local client. When this occurs, neither request messages nor data transfer are required. Instead, data are copied locally from the client buffer to the server buffer.

[5]For out-of-core applications, a collective write operation will be later followed by a collective read of the same data.

typical I/O node meshes lead to very large disk chunks. Thus, the disk chunks corresponding to a subarray being read or written often belong to just a subset of the I/O nodes. Hence, the load is imbalanced and available I/O bandwidth is decreased. The approach taken in [6] was to change the I/O mesh by hand to decrease the granularity, and the paper concluded that choosing an optimal disk array distribution is a difficult task and performance would not be robust if users had to select the disk layout themselves. We believe that a different approach, described below, can have the same beneficial effect for applications reading/writing subarrays without requiring user intervention, while maintaining good performance for applications that read and write entire arrays.

While it is tempting simply to ignore the user's request for a particular disk layout, such a move would be rash. Users are often the only source of information regarding future access patterns for the data. For example, most typically the data will be moved from the parallel environment to a row-major or column-major file on a sequential workstation for postprocessing. Alternatively, the data may be read in again as the next step in an out-of-core computation. In either case, the user will have very useful information about the appropriate distribution and appropriate meshes. However, usually the user will not have the low-level understanding necessary to suggest a specific size for disk chunks, as dictated by the specification of a single mesh for each write operation in Panda 2.1. In a heterogeneous environment, for good load balance the layout should conform to each node's capability, which is not known to users.

Thus it is unreasonable to expect users to be able to pick an appropriate disk mesh, but unwise to ignore their suggestions. One possible approach is to enlarge the mesh they request, e.g., by multiplying by a factor of two in selected dimensions until chunks are deemed sufficiently small. This would allow good load balance as well as conforming to applications' needs, i.e. taking future access patterns into concern. However, users might object to their choice of mesh being overridden. A better approach is possible if we assume that users do not care about details of the allocation strategy to be used, as long as the strategy can fully utilize the aggregate I/O bandwidth. That is, for write operations, Panda can hide the details of the layout of disk chunks, provided that Panda can still read the data back. More precisely, we can:

- Decompose the array into disk chunks based on the requested layout.
- Divide each disk chunk of the array into multiple smaller subchunks.
- Assign subchunks to I/O nodes with the goal to achieve good utilization of the aggregate I/O bandwidth.

Given the use of subchunks, the next issue is to determine their size and shape. There are two obvious choices for subchunk shape: use the same distribution in each dimension as was used for the chunks themselves, or use a (BLOCK,*,...,*) distribution, in which case the subchunks can simply be concatenated to form the original chunk. The (BLOCK,*,...,*) approach to subchunking has long been used in Panda, in a limited form: all subchunks of a chunk were assigned to the same I/O node. Under this approach, the final arrangement of data on disk is identical whether or not subchunking has been used. Although this limited form of subchunking does not help in balancing the load on I/O nodes, it does keep I/O node buffer requirements low, and careful choice of subchunk size will also ensure peak file system throughput. From past experience, we know that a maximum subchunk size of 1 MB gives peak file system performance on most target platforms, and so the same maximum size was used for the new subchunking facility.

At write time, Panda will introduce subchunking if the initial decomposition into chunks is too coarse to allow good performance. To determine subchunk size, we use an algorithm that loops through the dimensions of the disk chunk and repeatedly divides the chunk in half along selected dimensions until the size of the subchunks is below the predetermined threshold, currently set to 1 MB. If using (BLOCK,*,...,*)-style subchunking, only the first dimension will be subdivided. Otherwise, the same distribution chosen for the array overall is applied to the chunk itself, using the following algorithm (illustrated in Figure 2):

```
disk_distribution[ ] = the user's selected
  disk distribution, with each dimension
  either ``*'' or ``BLOCK'' distributed
size = number of bytes in first disk chunk
  of the array (no other chunk can be larger
  in any dimension than the first chunk,
  with an HPF-style distribution)
submesh[ ] = [1, ..., 1]  (submesh has as
  many elements as the array has dimensions)
threshold = maximum permissible number of
  bytes in a subchunk

begin FIND_submesh(size, disk_distribution,
  submesh, threshold)

  if disk_distribution = [``*'', ..., ``*'']
    return submesh (we respect the user's
        decision to have only a single chunk)
  i = 1
  while size > threshold
    if disk_distribution[i] is not ``*''
      submesh[i] = submesh[i] * 2
      size = ceiling(size / 2)
    i = ((i + 1)(mod n)) + 1

  return submesh
end FIND_submesh
```

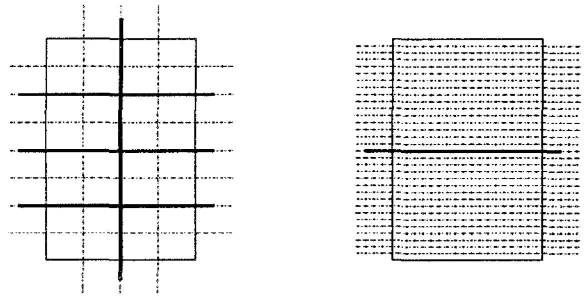


Figure 2. A simple example illustrates the automatic subchunking algorithm, using the same distribution for chunks as for the overall array. Assume a 32 MB array is distributed across 8 compute nodes (4x2 mesh) with a (BLOCK, BLOCK) in-memory distribution. Thick lines represent the disk mesh specified by the user. Dotted lines represent the results of applying automatic subchunking. The left-hand side uses natural chunking. That is, the disk mesh is 4x2 and the on-disk distribution (BLOCK, BLOCK). In this case, the size of each disk chunk is 4 MB, larger than the predefined threshold (1 MB). The algorithm cuts each chunk in half along both dimensions and generates 32 subchunks of size 1 MB. The right-hand side stores the array in traditional row-major order. The disk mesh is 2 and the on-disk distribution (BLOCK, *). In this case, the size of each disk chunk is 16 MB, larger than 1 MB. The algorithm subdivides the chunk along the first dimension (according to the distribution) four times and generates 32 (1 MB) subchunks.

3.2. Allocation

Once the array is decomposed and subchunked, the next problem is to evenly allocate the subchunks to the I/O servers. To balance the load across I/O nodes with different capabilities, the intuitive way is to get the capability of each node first, then distribute the load across the nodes according to their relative speeds. One can imagine a spectrum of possible strategies for workload distribution, based on how fine-grained are the allocations of work to nodes. For example, at one extreme, I/O node capabilities may be computed a priori from the hardware possessed by each I/O node, and used to allocate all upcoming tasks at the beginning of each I/O call. At the other extreme, the load could be rebalanced across I/O nodes after each subchunk is output. We have chosen to investigate two points on the spectrum, one close to the purely static end and another close to the purely dynamic end.

Our *static allocation* approach performs an empirical test of the local file system at each I/O node at the beginning of a run, to determine the capability of the node. Each I/O node writes three 1 MB chunks to disk to estimate its performance, then broadcasts the result to all other I/O nodes. On receiving the messages from all other I/O nodes, each I/O node will independently determine which subchunks resulting from a decomposition are its responsibility. For example, given subchunks 0 to 39 (numbered by a standard-order traversal of the subarray mesh) and two homogeneous I/O nodes in Panda 2.1.1, I/O node 0 will be responsible for outputting even-numbered subchunks; while in Panda 2.1, I/O node 0 would write out subchunks 0 to 19. Assigning subchunks interleavedly among the I/O nodes has the advantage that when a subarray instead of an array is read/written, the disk chunks of the subarray are likely to be spread across multiple I/O nodes, giving better load balance.

From the subchunk size and the throughput of an I/O node, we can estimate the time it will take to write a subchunk. Then, a subchunk will be assigned to the node which can finish writing it in the nearest future. For example, if there are 128 subchunks (1 MB each) and 2 I/O nodes, with speed 5 MB/sec and 2 MB/sec respectively, subchunk 0 will be assigned to I/O node 0 as it has an earlier finishing time (0 + 0.2 sec) than I/O node 1 (0 + 0.5 sec). Subchunk 1 will be assigned to I/O node 0 ($0.2 + 0.2 < 0.5$) as well. Subchunk 2 will be assigned to I/O node 1 ($0.4 + 0.2 > 0.5$) accordingly. Continuing in this manner, we can assign all the subchunks. Figure 3 shows the results of applying the decomposition strategy and the static allocation approach to the same configuration as in Figure 1, except that I/O nodes are heterogeneous.

The measurements acquired from the empirical test are used to assign subchunks to I/O nodes for the first I/O operation. For example, H3expresso simulates the behavior of black holes over time, calling Panda to output the current state of the simulation once every ten time steps, for later postprocessing and visualization. At each call, the workload will be assigned based on the most recent information on file system performance of each I/O node in order to adapt to dynamic load fluctuations. More precisely, during each collective I/O operation, each I/O node will measure the time it spent in writing out the data and the size of the data. Based on the measurements, an I/O node can calculate its current local file system throughput and will broadcast the result to all other I/O nodes at the beginning of the next collective I/O operation.

The static allocation approach should work well if file system performance obtained by the application is fairly stable on each I/O node. For example, each I/O node on the SP2 is dedicated to a particular job; if I/O nodes were shared, performance might fluctuate due to concurrent col-

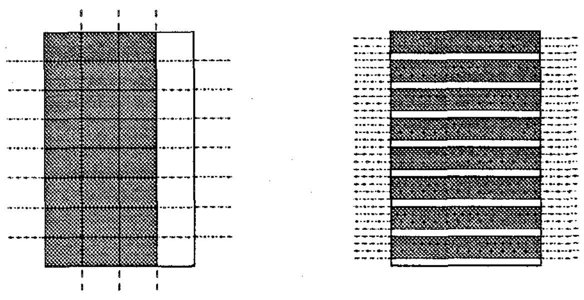


Figure 3. Static allocation strategy: Of the two I/O nodes, one (assigned with dark-colored array chunks) is three times as fast as the other (assigned with light-colored array chunks). Thick lines represent the disk mesh specified by the user. Dotted lines represent the results of applying subchunking. The left-hand side uses natural chunking and the right-hand side stores the array in traditional row-major order.

lective I/O operations. For such a situation, one might consider a pure dynamic allocation approach, using a manager/worker strategy to dynamically balance the load across the I/O nodes. An extra node is used as a manager to assign work to I/O nodes on a first come, first served basis. Each I/O node gathers a single subchunk at a time and when a node finishes writing its current subchunk, it asks the manager for another subchunk. This process repeats until all subchunks are written out. This pure dynamic allocation approach suffers from two problems. First, the master might become a hot spot in the system if the interconnect is slow[6], as all workers have to talk to the master for the job assignment. Second, one factor in Panda's high performance is its use of a write-behind strategy[7] to overlap communication, computation and disk activities. Therefore, normal write operations are non-blocking and the time for a *fwrite* call is not the time to write data to disk but to buffer the data in file system caches, which would normally be the same across all I/O servers even though some of them are connected to slow disks. In order to get the capability of each I/O node to be used as a basis for scheduling future subchunks, *fsync* calls have to be issued in the middle of an I/O operation and the degree of overlapping of communication and file I/O is reduced.

Due to these two problems, the dynamic allocation approach we implemented is a variation of the above pure approach. Instead of planning an entire I/O operation at the beginning or assigning a subchunk at a time to a worker, N subchunks are assigned to a worker during each communication with the master. A large value for N will minimize the overhead of the dynamic allocation strategy, which a small value for N may produce a better load balance. While many strategies are possible, we have chosen to have the value of N depend both on the capability of the worker and the size of the array data left to be written. At first, N is fairly large, to keep manager-worker communication overhead low. N decreases as the collective I/O call proceeds, allowing any unevenness resulting from earlier assignments to be smoothed over. As in the static allocation approach, the first time the application makes a collective I/O call, an empirical test is executed to determine the capability of each I/O node. The result is used to assign the first set of subchunks. Performance data from writing out the first set of subchunks is then used to assign the following set of subchunks, allowing the manager to react to dynamic system changes.

The algorithm we adopted sets the maximum number of subchunks assigned on each call to the master to $N \times total\_number\_of\_I/O\_nodes$, until half of the data has been written out. The experiments in this paper used $N = 20$, so if all I/O nodes are equally capable, each I/O node can be assigned 20 MB of data on its first call to the master. If between $1/m$th and $1/(m+1)$th of the data remains to be written out, for some integer m, then at most $\max(1, \lfloor N/m \rfloor)$ subchunks will be assigned to each worker in the next round of calls to the master. For example, consider the example given earlier for the static approach, which has 128 subchunks (1 MB each) and 2 I/O nodes (with speeds of 5 MB/sec and 3 MB/sec initially). Forty subchunks are assigned in the first round of calls to the master: 25 for I/O node 0 and 15 for I/O node 1. In the second round of calls to the master, 88 subchunks of the original array remain to be written out, over half of the total array data. This round will also assign 40 subchunks. Assuming the relative speed of the I/O nodes is the same as before, I/O node 0 will be assigned 25 subchunks while I/O node 1 will be assigned 15 subchunks. In the third run, 48 subchunks, around 1/3 of the total array data, are left to be written. In this round 20 subchunks are assigned ($10 \times$ the number of I/O nodes) to the I/O nodes based on the capabilities gathered from the previous round. Continuing in this manner, eventually all subchunks are assigned.

[6]On many parallel platforms, e.g., the SP2, the Origin 2000, or a Myrinet-based workstation cluster, the interconnect is so fast that disk speed is the main bottleneck for parallel I/O. In such cases, the extra communication overhead required by the manager/worker strategy is not a serious problem as the time required for a worker to talk to the master is very small compared to the time required to read or write a subchunk. However, in an Ethernet-based or FDDI-based cluster environment, message passing would become the bottleneck of the system as the interconnect would not be able to keep the disks busy. In those cases, the extra communication overhead required by the manager/worker strategy would be a concern.

[7]To increase write performance, for a write operation, the data is copied to the file cache first and will be written back to disk later when the number of dirty pages in memory exceeds a predefined threshold or Panda executes an *fsync* to force data to be written to disk.

4. Performance results

We experimented with the static and dynamic allocation approaches on the SP2 at the Cornell Theory Center. The Cornell SP2 has 512 nodes, divided into 48 wide nodes and 464 thin nodes. Wide nodes have better performance than thin nodes because of differences in memory buses and cache sizes. Table 1 is a brief summary of the current configuration of the Cornell SP2. The JFS performance numbers were determined empirically following the methodology of [18]. More details on the SP2 installation can be found on line at http://www.tc.cornell.edu/UserDoc/Hardware/SP.

General information		
	Thin nodes	Wide nodes
Node type	RS/6000 model 390	RS/6000 model 590
Processor	66.7 MHz, POWER2	66.7 MHz, POWER2
Main memory	128-256 MB	256-2048 MB
Scratch space	662 MB or 1.8 GB	1.8 GB
Disk drives	Starfire	Starfire
Measured file system peak performance		
JFS writes	6.39 MB/sec	
JFS reads	6.58 MB/sec	
Message passing performance		
Latency	50 microseconds	
Bandwidth	35 MB/sec	

Table 1. The current hardware and software configurations of the Cornell IBM SP2

When we began experimentation, within each type of node, different nodes had different amounts of memory and different kinds of disk drives. Most nodes had fast Starfire drives, but some had older, slower SCSI drives, a feature that was originally undocumented. Some time after experimentation began, an upgrade took place, after which all nodes had the same Starfire disk drives. To finish the evaluation, heterogeneity was simulated by writing different amounts of data to disk on different nodes. We let some I/O nodes run at full speed, and made the remainder run twice as slow by calling JFS to write out each chunk twice in a row. In the experiments described in this section, we called Panda from a small test program, rather than from inside H3expresso, in order to shorten the total execution time, and determined how long each call took to complete. A 512×512×512 single-precision array (512 MB) with a (BLOCK, BLOCK, BLOCK) memory distribution was written out in each call to Panda, and the average throughput of five calls was computed. We ran all our experiments with both subchunking strategies: a (BLOCK,*,...,*) distribution applied to each chunk, or the same distribution for each chunk as for the entire array. Both approaches gave similar and good results and we will show only the results of using the same distribution for chunks as for the entire array. Further experience with actual applications is needed before we will know which approach is advantageous for applications that later read the data.

Figures 4 and 5 show example aggregate throughputs and the fraction of peak potential throughput obtained while varying the number of total and slow I/O nodes. The fraction of peak throughput is computed from aggregate throughput by dividing by the best aggregate throughput of the I/O subsystem, which is the sum of the available throughput of the AIX file systems on each I/O node[8]. We also computed the error bars corresponding to a 95 percent confidence interval for the mean and found that they are quite small; the error bars for the fraction of peak throughput extend less than .055 in each direction in all cases, and extend at most .02 in each direction when fewer than 16 I/O nodes are used. Neither figure counts the extra master node that is used in the dynamic approach, or the extra startup overhead of empirical performance determination that is part of the static approach.

Panda always exhibits certain general performance trends, which we will describe here before discussing the trends specific to the heterogeneous environment. First, for a fixed array size, Panda's utilization of the available throughput on each I/O node drops slightly as the number of I/O nodes is increased, because each I/O node receives less data, hence spends less time accessing disk, and so Panda's internal communication overhead is more visible. Similarly, Panda's utilization of each I/O node is higher with slower I/O nodes, as the extra disk time hides internal Panda communication overheads, which can often be overlapped with disk activity. This can be seen in Figures 4 and 5, because with a fixed number of I/O nodes, utilization rises slightly as the number of slow I/O nodes increases. Finally, when the memory and disk layouts are different, requiring data reorganization during transfer, performance drops slightly due to the increased cost of MPI's scatter/gather operations; thus the performance in Figure 4 is overall slightly higher than in Figure 5. Details about these trends can be found in [18].

Let's now look at the new trends found in the figures. First, when using Panda's default round-robin assignment of chunks to a fixed number of I/O nodes (2, 4, 8, or 16), aggregate throughput is almost the same no matter how many slow I/O nodes are used. For example, in the cases of using 8 I/O nodes in Figure 4, the aggregate throughput when using the round-robin assignment drops to 23 MB/sec when a slow I/O node is introduced to the system and remains the same when more slow nodes are added. That is, the overall performance is determined by the slowest I/O node, and the fraction of peak throughput drops as low as 48%. By using either the static or dynamic approach, the fraction of peak throughput increases to above 85%, a respectable uti-

[8]The simulated slow nodes will have lower available peak throughput as they do not run at full speed.

lization.

Second, the static approach performs a little better than the dynamic approach for most cases. This is because on the SP2, the extra communication overhead for the dynamic approach is not significant compared to the time required for disk accesses. For a very few cases where the static approach achieved lower performance, the load was not as well balanced as it might have been. The static approach needs very precise estimates of the relative capabilities of each I/O node, as the estimates are used to assign subchunks for the whole I/O operation; sometimes the empirical test of file system speed obtained a result somewhat different from the peak throughput of 6.4 MB/sec that we have found to hold for the SP2 overall. The dynamic approach is more flexible in that if there is a little error in the initial capability measurement, run-time adaptations are possible.

Perhaps, then, the static approach might benefit from a revised empirical performance test. One possible remedy is to increase the amount of data written out in the test; however, this would increase the startup time for an application. Further, when used with a real application, the error would be confined to the first collective I/O call to Panda, as subsequent calls would use the more accurate estimates obtained during the large writes performed during the previous call.

As the number of I/O nodes becomes very large, one would expect the master node to eventually become a bottleneck under the dynamic approach. This is starting to happen with 16 I/O nodes, as can be seen in the 0/16 case in Figure 5, which measures the overhead of the dynamic strategy. A bottleneck can be avoided by slightly staggering the amount of work distributed to different nodes, so that I/O nodes will complete their current round of work at slightly different times. The master node can keep track of the amounts of stagger and factor that into the assignments given in the last rounds of communication with the master, so that all I/O nodes complete their last round at approximately the same time, thereby minimizing total time spent in the call to Panda.

The above experiments evaluated the performance of both allocation approaches in a static heterogeneous environment, i.e., a system dedicated to a single application; neither interference from the outside world nor change in the system's workload happens at run-time. In a networked computing environment, dynamic load fluctuation may happen at run-time when interactive jobs generate disk accesses to load/store files. The distribution of the load among the I/O servers should adapt to run-time changes as well in order to minimize the I/O time. To see how well these two approaches can adapt to dynamic load changes, we simulated a scenario where an application makes 8 collective I/O calls to Panda, with no computation between calls to Panda. All I/O nodes run at full speed at the beginning (from iteration 1 to iteration 4), but a competing load is added to one of the nodes in the middle (iteration 5) such that the node runs twice as slow. The competing job leaves the system in the following I/O operation (iteration 6). The results are shown in Figures 6 and 7.

As can be seen from the figures, when the load of the system is static, the dynamic allocation approach performs worse than the static approach because of the communication and *fsync* overheads explained above. However, at iteration 4, when a slow node is introduced into the system, the dynamic allocation approach can react to load changes and can still utilize 85% or more of peak throughput. On the other hand, performance of the static approach, which plans the entire I/O operation at the beginning based on the capability of each node obtained from the previous I/O operation, drops to as low as 52%, and gradually recovers back to high-performance after two iterations. That is, the static allocation approach is not as sensitive to run-time system changes as the dynamic approach, where subchunks are assigned to the most appropriate node based on the current situation.

5. Related Work

This work is related to research on providing efficient access to persistent storage for scientific applications, which is of great current interest to scientific database researchers and parallel I/O researchers; and handling heterogeneity by balancing the workload among nodes, which has been an active research topic in parallel processing. Heterogeneity has been also a big issue for federated and distributed database research; however, the problem they are looking at is different from our scenario.

5.1. Database system

Database researchers have emphasized the importance of providing an efficient organization of files on disk in order to speed up applications' accesses to data. They argue that the traditional method of storing an array in row-major or column-major order will lead to disastrous performance when access patterns are different from storage patterns. Most of these works have been applied to real applications. For example, [2] designed a specialized data management system (DMS) for particle physics codes. [11] used a PLOP file structure for the array storage of radio astronomy applications at the NRAO. [16] enhanced the POSTGRES DBMS to support multidimensional arrays with chunked schemas (one chunk per disk block). They chose an optimal chunk layout based on the actual access patterns of the arrays when used by global change scientists in the Sequoia project [20]. Paradise [8] used a client-server architecture and provided an extended-relational data model for modeling GIS applications, with support for 2D chunked arrays.

[14] provided a query language which can directly manipulate scientific data stored in any format (e.g. NetCDF or HDF), as long as a driver is created to mediate between the query language and the storage format. [17] proposed 'natural chunking' (also used in this paper) for storing the data of computational fluid dynamics applications. The latter effort was targeted for write-intensive applications, requiring the output of solution files and periodic checkpointing; the others were optimized for read-intensive applications, performing query-like I/O. We have not encountered any read-intensive application on massively parallel platforms, except for out-of-core applications. None of these research projects addressed problems of performance in a heterogeneous environment.

5.2. Parallel I/O

Parallel I/O researchers have emphasized the design of parallel file systems or parallel I/O libraries suitable for large scale scientific applications. The collective I/O approach (e.g. [3, 12, 18]) was shown to produce much higher performance than that attainable if each processor independently performs I/O operations. Most of these works were applied to out-of-core applications. These are I/O-intensive because memory cannot hold the entire problem, so data needs to be moved to and from secondary storage and each processor's main memory periodically. [12] applied his disk-directed I/O approach to an out-of-core LU-decomposition program. [3] used the PASSION runtime library with three out-of-core applications: a Laplace equation solver using the Jacobi iteration method, LU factorization with pivoting, and three dimensional red-black relaxation. None of the parallel I/O approaches described in the literature addresses the load balancing issues that arise when heterogeneity is introduced.

PIOUS [15] and VIP-FS [10] are parallel file systems designed for accessing permanent storage in networked computing environments, where heterogeneity is a big issue. However, PIOUS focused on supporting concurrent and fault-tolerant access and did not address the load balancing issue. VIP-FS emphasized another heterogeneity problem: dealing with the low throughput caused by multiple users on shared-media networks.

5.3. Load balancing

Load balancing in a heterogeneous environment of workstation clusters is a research topic in the parallel processing community. An application problem is decomposed into individual subtasks, which are distributed across available processors in order to be executed concurrently. As processors' capabilities might not be the same and may change dynamically, load balancing is the key issue for providing high performance in such an environment. [7] designed a partitioning system for heterogeneous networked data-parallel processing. [4] proposed a heterogeneous partitioning strategy based on the load situation at start and dynamically balanced the load throughout the entire computation. Basically, these and our static allocation strategies follow the same problem-solving flow: perform an empirical test at the beginning of a run to get the capability of each (compute or I/O) node, distribute the workload based on the initial information, and monitor and dynamically balance the workload at run-time. Load balancing is also an issue for job scheduling in distributed systems [9]. However, the approaches used to combat the problem do not carry over well to the I/O arena, because the former is a scheduling problem (the number of jobs is not known in advance and jobs are scheduled as they enter the system); while the latter is an assignment problem (the amount of work is known and can be assigned at the beginning of a run).

6. Summary and Discussion

This paper discussed the heterogeneity problem which arose when combining Panda, a DBMS-style I/O library designed for scientific applications, with a real application, H3expresso, that simulates black holes. We had very high expectations both for Panda's ease of use and high-performance, based on laboratory experiments conducted earlier without a real application. As usual when a new piece of software moves from the laboratory to deployment, some surprises were encountered and new issues were raised that we think will be important for many laboratory-grown packages.

We encountered some obstacles initially to good performance on the Cornell SP2, a machine we had not used previously. The heterogeneity of the I/O nodes on the Cornell SP2 at the time we began experimentation caused poor performance, as Panda tended to run as slow as the slowest I/O node. We devised an automatic chunk decomposition technique and two allocation strategies for handling the heterogeneity, and found that in a static heterogeneous environment, like the SP2, where nodes in the system are dedicated to a single application, a static allocation of work to I/O nodes, based on a quick run-time test of the speed of the local file system, resulted in the best performance. However, for a dynamic heterogeneous environment, which may occur in workstation clusters when nodes are shared among interactive and parallel jobs, the dynamic allocation approach performed much better in adapting to run-time system changes.

We expect that heterogeneity will be a big issue for DBMSs designed for scientific applications running on networks of workstations, and the methods of allocating data to I/O nodes in those environments will need to be upgraded to take heterogeneity into account, as Panda 2.1.1 does. To generalize our approaches, we plan to extend our work and evaluate both allocation strategies in a networked environment to see how well these two approaches can perform when interconnects are slower. Also, in such an environment, compute nodes may also be heterogeneous, causing different nodes to contain different size memory chunks and making load balancing for I/O even more important and more difficult. In our future work, we will be addressing this type of heterogeneity as well.

References

[1] J. L. Bell, G. S. Patterson, Jr., Data Organization in Large Numerical Computations, The Journal of Supercomputing, vol. 1, no. 1, 1987.

[2] J. L. Bell, A Specialized Data Management System for Parallel Execution of Particle Physics Codes, ACM SIGMOD International Conference on Management of Data, 1988.

[3] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel and M. Paleczny, A Model and Compilation Strategy for Out-of-Core Data Parallel Programs, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1995.

[4] C. H. Cao and V. Strumpen, Efficient Parallel Computing in Distributed Workstation Environments, Parallel Computing, vol. 19, no. 11, 1993.

[5] Y. Chen, M. Winslett, K. Seamons, S. Kuo, Y. Cho, M. Subramaniam, Scalable Message Passing in Panda, Fourth Annual Workshop on I/O in Parallel and Distributed Systems, 1996.

[6] Y. Chen, I. Foster, J. Nieplocha, M. Winslett, Optimizing Collective I/O Performance on Parallel Computers: A Multisystem Study, ACM International Conference on Supercomputing, 1997.

[7] P. E. Crandall and M. J. Quinn, A Partitioning Advisory System for Networked Data-Parallel Processing, Concurrency: Practice and Experience, vol. 7, no. 5, 1995.

[8] D. DeWitt, N. Kabra, J. Luo, J. Patel and J. Yu, Client-Server Paradise, Proceedings of the 29th VLDB Conference, 1994.

[9] D. L. Eager, E. D. Lazowska and J. Zahorjan, Adaptive Load Sharing in Homogeneous Distributed Systems, IEEE Transactions on Software Engineering, vol. 12, no. 5, 1986.

[10] M. Harry, J. Miguel, del Rosario and A. Choudhary, VIP-FS: A Virtual, Parallel File System for High Performance Parallel and Distributed Computing, Proceedings of the Ninth International Parallel Processing Symposium, 1995.

[11] J. F. Karpovich, J. C. French and A. S. Grimshaw, High Performance Access to Radio Astronomy Data: A Case Study, Conference on Scientific and Statistical Database Management, 1994.

[12] D. Kotz, Disk-directed I/O for an Out-of-Core Computation, Proceedings of the Symposium on High Performance Distributed Computing, 1995.

[13] S. Kuo, M. Winslett, K. E. Seamons, Y. Chen, Y. Cho, and M. Subramaniam, Application Experience with Parallel Input/Output: Panda and the H3expresso Black Hole Simulation on the SP2, To appear in Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[14] L. Libkin, R. Machlin and Limsoon Wong, A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques, SIGMOD Proceedings, 1996.

[15] S. A. Moyer and V. S. Sunderam, PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments, Proceedings of the Scalable High-Performance Computing Conference, 1994.

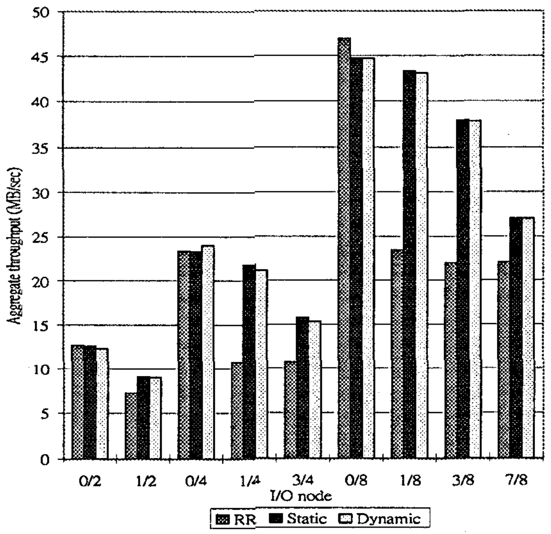
[16] S. Sarawagi and M. Stonebraker, Efficient Organization of Large Multidimensional Arrays, Proceedings of the 10th International Conference on Data Engineering, Feb. 1994.

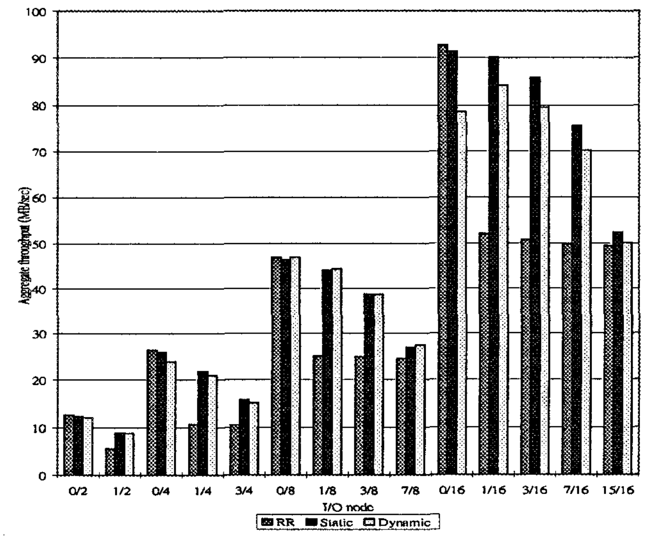
[17] K. E. Seamons and M. Winslett, Physical Schemas for Large Multidimensional Arrays in Scientific Computing Applications, Conference on Scientific and Statistical Database Management, 1994.

[18] K. E. Seamons, Y. Chen, M. Winslett, Y. Cho, S. Kuo, M. Subramaniam, Persistent Array Access Using Server-Directed I/O, Conference on Scientific and Statistical Database Management, 1996.

[19] Application Working Group of the Scalable I/O Initiative, Preliminary Survey of I/O Intensive Applications, Scalable I/O Initiative Working Paper No. 1.

[20] M. Stonebraker, J. Frew, K. Gardels and J. Meredith. The SEQUOIA 2000 storage benchmark, ACM SIGMOD International Conerence on Management of Data, 1993.





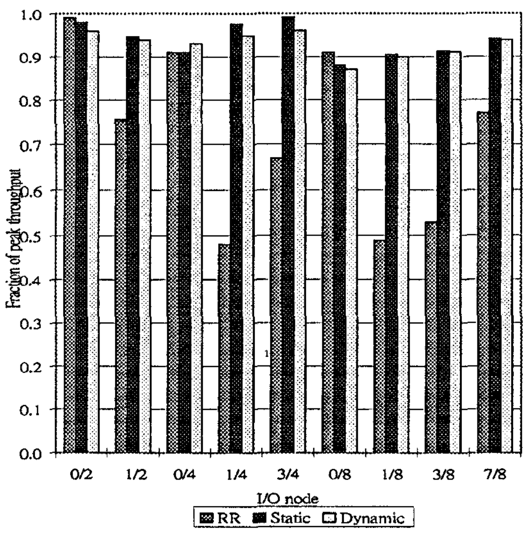


Figure 4. Throughputs for writing out a 512 MB array with 8 compute nodes (2x2x2 mesh) and different ratios of slow to total number of I/O nodes, where n/m indicates that n out of m I/O nodes are slow nodes, using a (BLOCK, BLOCK, BLOCK) distribution in memory and natural chunking on disk. The upper graph shows the aggregate throughput and the lower graph shows the fraction of peak throughput. The fraction of peak numbers are computed by dividing the aggregate throughput by the best aggregate throughput of the I/O subsystem, computed by (*total number of I/O nodes* $-$ *number of slow I/O nodes*) $\times$ 6.4 MB/sec $+$ *number of slow I/O nodes* $\times$ 3.2 MB/sec.

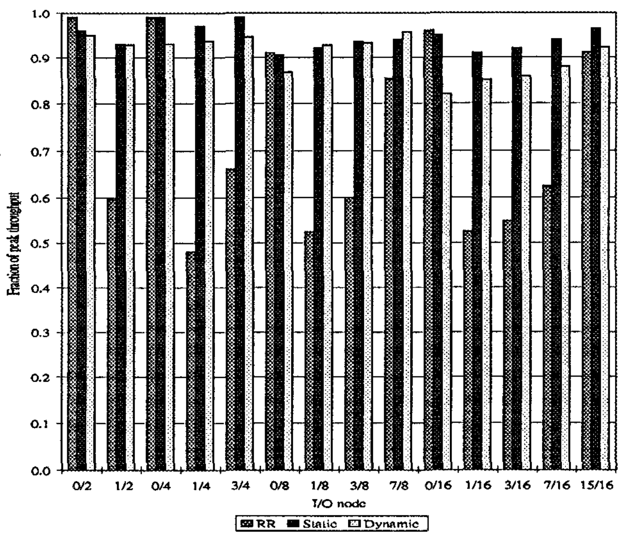
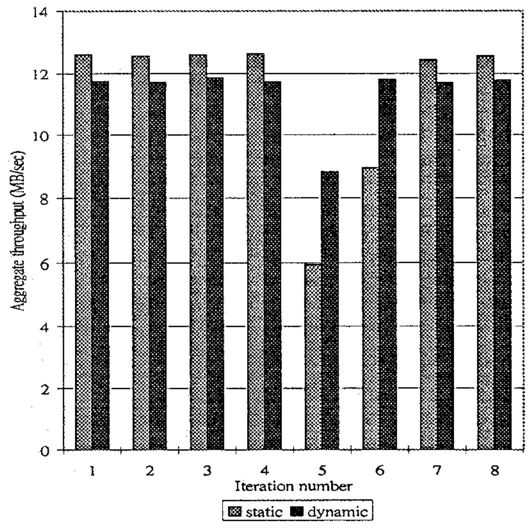


Figure 5. Throughputs for writing out a 512 MB array with 32 compute nodes (4x4x2 mesh) and different ratios of slow to total number of I/O nodes, using a (BLOCK, BLOCK, BLOCK) in-memory distribution and a (BLOCK, *, *) on-disk distribution, traditional row-major order. The upper graph shows the aggregate throughput and the lower graph shows the fraction of peak throughput.



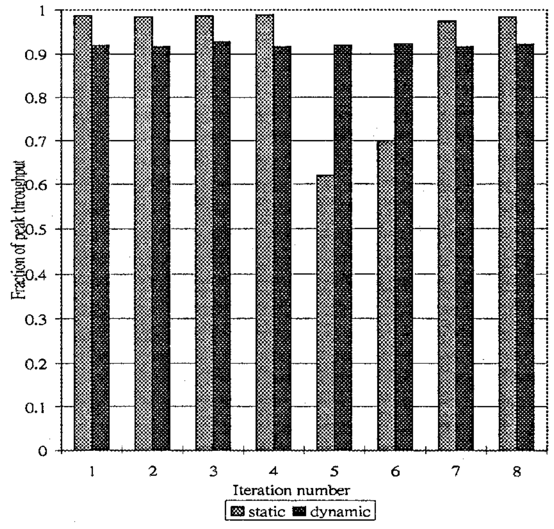
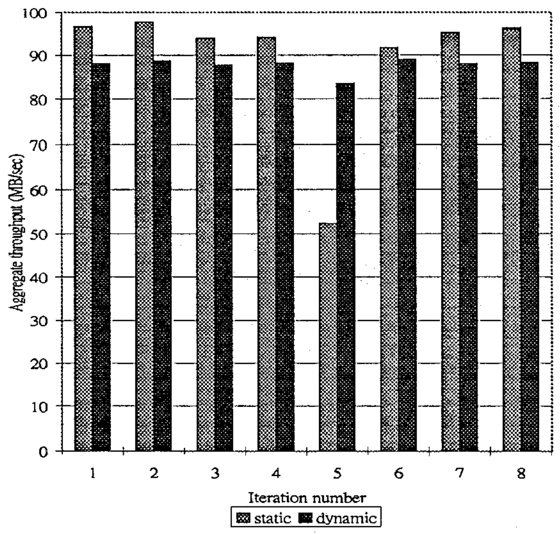


Figure 6. Throughputs for writing out a 512 MB array with 16 (4×2×2) compute nodes and 2 variable-performance I/O nodes, using a (BLOCK, BLOCK, BLOCK) memory distribution and a (BLOCK, *, *) disk distribution, traditional row-major order. The upper graph shows the aggregate throughput and the lower graph shows the fraction of peak throughput.



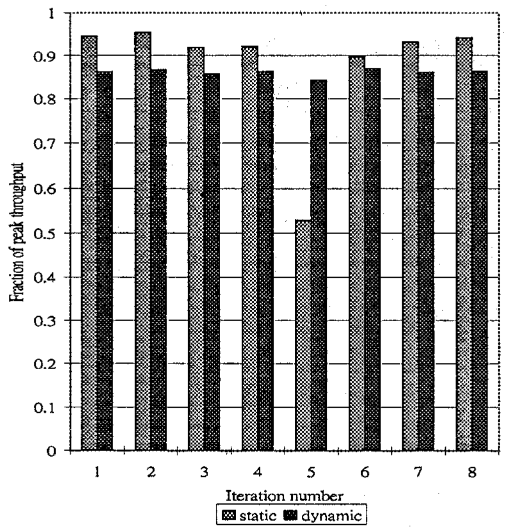


Figure 7. The same configuration as in Figure 6 except that 16 instead of 2 variable-performance I/O nodes are used.