# Scalable Message Passing in Panda

Y. Chen, M. Winslett, K. E. Seamons, S. Kuo, Y. Cho, M. Subramaniam
Computer Science Department, University of Illinois
{ying,winslett}@cs.uiuc.edu
(217) 244-6381 (voice) (217) 244-6500 (fax)

### Abstract

To provide high performance for applications with a wide variety of i/o requirements and to support many different parallel platforms, the design of a parallel i/o system must provide for efficient utilization of available bandwidth both for disk traffic and for message passing. In this paper we discuss the message-passing scalability of the server-directed i/o architecture of Panda, a library for synchronized i/o of multidimensional arrays on parallel platforms. We show how to improve i/o performance in situations where message-passing is a bottleneck, by combining the server-directed i/o strategy for highly efficient use of available disk bandwidth with new mechanisms to minimize internal communication and computation overhead in Panda. We present experimental results that show that with these improvements, Panda will provide high i/o performance for a wider range of applications, such as applications running with slow interconnects, applications performing i/o operations on large numbers of arrays, or applications that require drastic data rearrangements as data are moved between memory and disk (e.g., array transposition). We also argue that in the future, the improved approach to message-passing will allow Panda to support applications that are not closely synchronized or that run in heterogeneous environments.

## 1 Introduction

A parallel i/o software system for distributed-memory multiprocessor platforms generally contains two important components, the i/o subsystem that is responsible for actually reading and writing data and the message passing system that connects the individual processors and moves the data around the system. However, many newly proposed or developed i/o techniques focus only on the i/o subsystem of the entire parallel i/o system. The common assumption made is that the message passing system will always deliver data at the desired performance level, and hence, the i/o subsystem will always be the primary bottleneck of the system. However, the system bottleneck could quickly shift from the i/o subsystem to the message passing system if data cannot be shipped across the network as fast as they are transferred to or from the disk, as in the following scenarios:

- When the interconnect is disproportionately slow, compared to disk speed (e.g., on a local or wide area network);
- When the interconnect is heavily utilized by others, so that the message passing bandwidth available for i/o is disproportionately low compared to the available disk bandwidth;
- When the parallel i/o system introduces a large number of message transfers that do not utilize the available bandwidth well, such as when transferring data that resides in non-contiguous memory locations;[1]
- When the parallel i/o system carries out the i/o request using very small messages (small messages are often latency bound, and their transfer rate can be but a fraction of the peak bandwidth).

In section 2, we discuss the behavior of the Panda parallel i/o library as message passing becomes a bottleneck. We then explore a strategy that combines additional intelligence for Panda clients and the use of nonblocking message passing facilities in section 3. We also discuss how these modifications should improve Panda's scalability with respect to message passing by reducing Panda's internal communication and computation cost and increasing Panda's internal

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. Computing facilities were provided by NASA Ames Numerical Aeronautic Simulation (NAS) Division, Cornell Theory Center, and the National Center for Supercomputing Applications (NCSA) at the University of Illinois.

[1] Transfer of non-contiguous data can be useful when handling 'strided' i/o requests. Unfortunately, the MPI bandwidth for transfer of non-contiguous data ('derived data types') can be much lower than that for transferring the same amount of data stored contiguously in both sender's and receiver's memory, because the data stored in noncontiguous memory locations have to be copied into the network buffer piece by piece before they are sent over to the receiver side, and similarly, the data received in a temporary buffer have to be copied to noncontiguous memory locations piece by piece on the receiver side.

level of parallelism. As an important side effect, these modifications open up the possibility of supporting applications that are not closely synchronized or that run in a heterogeneous environment. In section 4 we show the performance gains that result for Panda on a set of benchmark runs on the NASA Ames NAS IBM SP2. Section 5 discusses the related work in this field and section 6 concludes the paper.

## 2 Message passing in Panda

Panda (URL http://bunny.cs.uiuc.edu/CADR/panda.html), a parallel array i/o library developed at the University of Illinois, is designed to provide high performance i/o for large, SPMD-style scientific applications running on both distributed memory parallel architectures and networks of workstations. Panda supports collective i/o operations where all the processors used by an application are closely synchronized and periodically cooperate in conceptually simple i/o operations. From our collaborations with scientists at NASA Ames Research Center and at NCSA at the University of Illinois who deal with multidimensional arrays, and the listed i/o requirements of several data-intensive applications collected by the Application Working Group of the Scalable I/O Initiative [Sio94], it is evident that synchronized collective i/o operations are indeed widely and intensively used.

Unfortunately, with traditional i/o techniques, the performance of such applications suffers from large numbers of random disk seeks, because each i/o node sees an apparently random sequence of i/o requests from processors within a single collective i/o request, even though the high-level view of the i/o request is not complicated at all. This results in poor i/o performance due to buffering errors and excessive disk seeks, the two flaws that Panda seeks to remedy.

As described in [Seamons95], Panda is distributed across both compute nodes and i/o nodes. Panda clients reside on the compute nodes, and Panda servers reside on the i/o nodes, as shown in Figure 1. Panda offers efficient support for synchronized collective i/o operations by combining the use of flexible array chunking storage formats, a high level interface, and a *server-directed i/o architecture*, which allows long sequential file reads and writes at each i/o node, making good use of native UNIX buffering facilities.

In Panda, arrays are distributed across processors and disks using HPF-style BLOCK and * distributions (schemas); applications can specify different schemas for arrays in memory and on disk. The pieces of an array that result from distribution are called 'chunks' in Panda. The unit of reads and writes is an array chunk on disk if the chunk size is 1 MB or less; otherwise, the array chunk is broken into roughly 1 MB subchunks, and the unit of reads and writes is a subchunk. The use of subchunks reduces memory buffer requirements at servers, and gives higher utilization of the underlying file system bandwidth for very large chunks. The current memory requirement of Panda on each server is no more than 1.7 MB—at most 1 MB for array chunk buffer space, about 600 KB for Panda server executable code itself, and less than 100 KB for Panda's internal data structures. In the future, we will explore techniques that make use of the available memory on Panda servers by sharing the i/o nodes with multiple applications. Similarly, the memory requirement on each client is only about 0.7 MB—600 KB for Panda client executable code, and less than 100 KB for the Panda client's internal data structures. Panda clients do not allocate new buffer space for storing the application's array

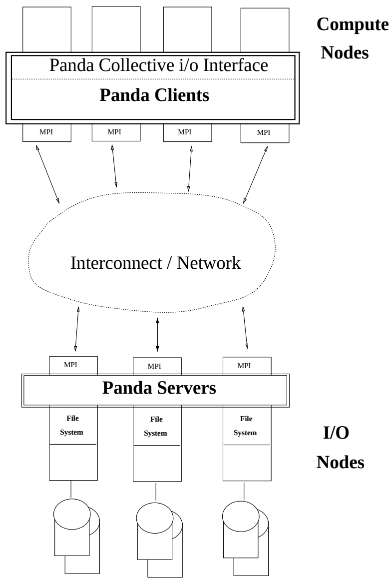


Figure 1: Panda server-directed i/o architecture

data, so there is no memory requirement for array data on Panda clients.

An i/o request in Panda begins when one compute node informs the servers of the desired layouts of array data in files on disk and also in memory. Then in order to realize sequential writes, Panda adopts a server-directed i/o approach. In the implementation of server-directed i/o in Panda 2.0 [Seamons95], Panda servers direct the flow of i/o requests by gathering the data for each array chunk in sequence. To gather a chunk, the server first sends data request messages to the clients who own pieces of the array chunk in parallel, and then waits for the clients to send the requested array data back. Once a chunk is gathered, the server writes it out to disk. The reverse strategy is employed for reads. Thus for each array data transfer between a client and a server, Panda 2.0 requires two messages, a data request message and an array data message. [Seamons95] presents the architecture of Panda 2.0 in more detail and shows the excellent performance levels Panda 2.0 achieves in reading and writing large arrays on the NASA Ames NAS IBM SP2.

The benchmark used to evaluate the performance of Panda 2.0 ('benchmark 2.0' from now on) characterizes the i/o requirements of applications that typically input/output *large* multidimensional arrays. The sizes of the array chunks are typically on the order of tens to hundreds of megabytes and the data message sizes are mostly on the order of several hundred kilobytes. Panda's high performance for benchmark 2.0 on the NAS SP2 can be traced to two factors. First, Panda servers are able to control the data flow in order to make sequential reads and writes; second, Panda's performance is limited by the peak throughput of the underlying file system, for the following reasons. Even though Panda supports only blocking collective i/o operations at the application level, Panda internally overlaps the file system i/o activities with its own communication and computation activities as much as possible. MPI performance approaches its peak bandwidth on the SP2 when message sizes are large, as they are in benchmark 2.0; thus the total communication and computation cost internal to Panda is extremely low compared to the file system cost. The result is that internal communication and computation costs are almost entirely overlapped with the file system cost, and Panda is limited by the throughput of the underlying file system.

Unfortunately, as pointed out in section 1, the NAS SP2 is not typical of all execution environments and benchmark 2.0 is not typical of all scientific applications. For example, we have found that the SP2 at Cornell Theory Center is sufficiently heterogeneous that Panda's performance can be improved significantly by taking heterogeneity into account in the implementation. Similarly, we have obtained an application from black hole researchers working under Professor E. Seidel at NCSA at the University of Illinois, for which message-passing throughput is a bottleneck for Panda on the NAS SP2.

The black hole researchers like to run on as many compute nodes as possible, with as large a data size as will fit in memory, and i/o costs are a large component of overall run time for them on most platforms. However, instead of very large arrays, the code uses a *large number of small arrays* distributed across all nodes, so that, for example, the typical i/o request asks for 200 arrays to be written out. Array sizes are typically 1 MB or less, with an array chunk on a processor ranging from 2 KB to 16 KB. The black hole researchers have recently ported the application to the IBM SP2 at Cornell Theory Center, and done runs with 256 and 512 nodes with the i/o portion of the code commented out. They want to use Panda for their production runs on the SP2, but the performance results of Panda 2.0 for the black hole benchmarks are fairly poor, as explained below.

Panda 2.0 uses at least as many data transfer messages during i/o as there are array chunks in memory, which in turn is equal to the number of compute nodes used times the number of arrays that need to be transferred. As an equal number of data request messages is required, the total number of messages can be quite large when there are many small arrays. Furthermore, computation costs are associated with each data request message: the server must describe the requested array chunk portion to the client in the message, and the client must parse and interpret the message, incurring a relatively high overhead per message. If we model the cost of each data transfer message by the following formula:

$$\begin{aligned} T_{message} &= T_{startup} + T_{transfer} \\ &= T_{startup} + \frac{S_{data}}{transfer\_rate} \end{aligned}$$

where $T_{startup}$ is a constant startup cost (latency) for each message, $T_{transfer}$ is the time to move the data across the network, $S_{data}$ is the message size, and $transfer\_rate$ is the transfer rate of the message passing system, then the total communication and computation overhead associated with each array data transfer in Panda can be modeled as follows:

$$\begin{aligned} T_{total\_message} &= T_{comp\_server} + T_{message}(request) \\ &\quad + T_{comp\_client} \\ &\quad + T_{message}(data) \\ &= (T_{startup} + T_{data\_request}) \\ &\quad + (T_{startup} + T_{data\_transfer}) \\ &\quad + (T_{comp\_server} + T_{comp\_client}) \\ &= 2 * T_{startup} + \frac{S_{request}}{transfer\_rate} \\ &\quad + \frac{S_{data}}{transfer\_rate} \\ &\quad + (T_{comp\_server} + T_{comp\_client}) \end{aligned}$$

where $T_{message}(request)$ is the communication cost of the data request, $T_{message}(data)$ is the communication cost of the array data transfer, $T_{comp\_server}$ is the computation cost for forming the data request on server before the request message can be sent, and $T_{comp\_client}$ is the computation cost for interpreting the request message. $T_{data\_request}$ is the time to transfer the data request, and $S_{request}$ is the size of the request message, which is typically less than 100 bytes for the experiments in benchmark 2.0. $S_{request}$ is also dependent on the array ranks that Panda is dealing with, since the request messages typically contain array names and the boundary points of the array portions that clients need to send to servers. Each boundary point contains $N$ numbers, where $N$ is the rank of the array.

Because Panda uses the MPI derived data type [mpi94] to transfer the portions of array data which may be stored in non-contiguous memory locations on the sender side, and received into non-contiguous memory locations on the receiver side, the transfer rate of such data can be quite low, and $T_{message}(data)$ can be quite high. Even when $T_{message}(request)$, $T_{comp\_server}$, and $T_{comp\_client}$ are small, the accumulated result for a large number of messages can be a significant value. Request message overhead can constitute half of the overall communication and computation cost of transferring one array data message. When Panda's communication and computation cost is higher than the file system i/o cost, the system bottleneck will move from the i/o system to the message passing system, as happens for the black hole application. In Figures 3 through 5, the message passing system becomes Panda's bottleneck when 8 i/o nodes are used.

The cost of array data messages in Panda can be reduced by optimizations not considered in this paper, such as combining multiple array data messages between the same client and server. The cost of array data request messages can be reduced if clients as well as servers know how the data will flow during the i/o request, as explained in the next section.

Panda 2.0 uses MPI's blocking message passing facility. A blocking send call cannot return until the message has been copied out of the user buffer space, and a blocking receive call cannot return until the message has been received in the user buffer space. This approach prevents Panda's computation from being overlapped with its communication on each node. Furthermore, receivers can only receive data in the order of receive calls, which may not be optimal; the receiver cannot choose to receive a data message, $x$, until all the receive calls in front of $x$ are finished even if $x$ arrives at the receiver side first. With MPI's nonblocking message passing facility, receivers can dynamically choose the order in which they accept messages, allowing them to determine an optimal order at runtime.

In the following example, we show how message transfers can be serialized when blocking message passing facilities, such as those of Panda 2.0, are used. In Figure 2, a 2D array is going to be output to disk. It is distributed across three compute nodes using a (BLOCK, *) in-memory schema and a $(3 \times 1)$ logical compute node mesh. The array shown at the top indicates how the array is distributed across 3 compute nodes. The arrows between the upper array chunks and the clients indicate the assignments of the array chunks to the compute nodes. The array is going to be stored on two i/o nodes using a (BLOCK, *) schema on disk and a

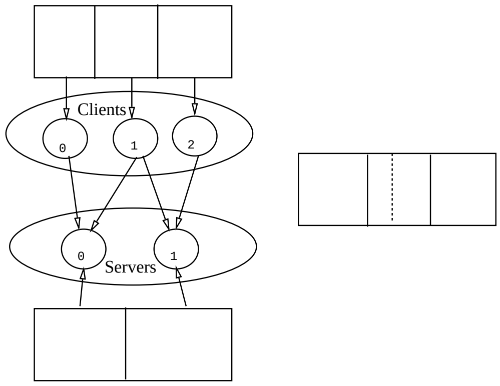


Figure 2: How message transfers are performed using blocking and nonblocking message passing facilities, for a write request.

$(2 \times 1)$ logical mesh. The bottom array shown in the figure illustrates how the array is distributed across two i/o nodes. The arrows between the lower array chunks and servers indicate the assignments of the array chunks to the i/o nodes. The array on the right-hand side of the graph shows how the array chunks on i/o nodes overlap with the array chunks on the compute nodes. The dotted line separates the array into two i/o node array chunks, and the solid lines separate the array into three compute node array chunks. In this case, each of the array chunks on the two i/o nodes overlaps with two array chunks on two compute nodes, and hence, in order to gather an array chunk, each i/o node needs to receive two array chunks from two compute nodes. The arrows between clients and servers illustrate this fact graphically. In Panda 2.0 with its blocking message passing facility, all three compute nodes can carry out the computation in parallel, then each of them can send the first data message to the relevant i/o nodes. Both compute node 0 and 1 could send messages to i/o node 0 simultaneously, and compute node 2 could send the message to i/o node 1. When a blocking message passing facility is used, a sender cannot return from an MPI send call until the data is copied out of the sender's send buffer, and the receiver actually starts to receive the message into its buffer space. Similarly, a receiver cannot return from the MPI receive until the data is received in the receive buffer. As a result, if we assume that i/o nodes do receive calls in increasing order of compute node numbers and compute nodes send in increasing order of i/o node numbers, then i/o node 0 cannot return from its first receive until compute node 0 issues an MPI send. After receiving the first data message, i/o node 0 will then ask to receive a message from compute node 1. Meanwhile, i/o node 1 expects data messages from compute node 1 and 2. But i/o node 1 cannot receive any data until compute node 1 finishes its send to i/o node 0, and i/o node 0 cannot start receiving a message from compute node 1 until it has received from compute node 0. In other words, i/o node 1 cannot receive any data until i/o node 0 has finished. The following section discusses the user of non-blocking message passing to overcome this problem of serialized message transfers.

## 3 Scalable message passing in Panda 2.1

As discussed in the previous section, in Panda 2.0 one request message is needed per array data transfer message because clients do not know which servers will communicate with them, and what data requests those servers will send to them. The new version of Panda discussed in this paper, Panda 2.1, eliminates the request messages by having each client compute which servers it should communicate with, and what array data those servers will need or provide. To carry out the computation, Panda clients must have certain information about the entire i/o request, such as the arrays' in-memory and on-disk schemas and the total number of servers in the system. Fortunately, most of this information is already supplied to the client when an application issues an i/o request through Panda's high level interface. When data request messages are eliminated, servers carry out the same computations as before to figure out from which clients they should receive or send pieces of array data that form an array chunk on disk.

The changes sketched above only affect the internal design of Panda, not the interface seen by users. As described in [Seamons95], the use of high level interfaces in Panda provides application portability, frees application developers from worrying about the low-level details of file system calls, and more importantly, it gives us great freedom to do performance optimizations and to experiment with new underlying implementations. The work described in this paper also serves as an concrete example of changing Panda's internal implementation without sacrificing application portability because of the use of Panda's high level interfaces. The rest of this section presents the detailed design of the improvements to message passing embodied in Panda 2.1.

In Panda 2.1, 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 desired schemas for arrays, in memory and on disk. After the initial request to the master server, the master server then informs all the other servers of the schema information. Each node (not just the servers, as in Panda 2.0) then begins to plan which nodes it should communicate with, and how it will send or receive its array chunks.

In the case of write operations in Panda 2.1, all the nodes 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. In Panda, each array is roughly equally distributed across clients, and hence, each client holds exactly one in-memory array chunk per array. For each array, the client calculates which servers will need pieces of the chunk currently in its memory, and what pieces they will need. Similarly, for each array chunk on disk, each server calculates which clients own which pieces of that on-disk chunk. Once a client finishes its calculation, it sends out the array data messages that belong to the entire in-memory chunk to all the servers who will need array data from it. On the other hand, once a server finishes its calculation about which clients it needs data from in order to assemble its first assigned chunk, it gathers all the data messages from those clients, and then performs the write. The reverse procedure holds for read operations.

Panda 2.1 eliminates the communication cost and much of the computation cost incurred by servers in Panda 2.0 for data request messages, while still allowing the sequential reads and writes used in Panda 2.0 to achieve high utilization of the available underlying file system throughput. Furthermore, the additional computation costs for Panda 2.1 clients can be entirely overlapped with the computation already carried out on Panda servers. Therefore, there is no extra cost introduced by adding intelligence to the clients when Panda handles an i/o operation. In general, as long as clients and servers agree on the strategies used to scatter/gather data and read/write data to the underlying file system (such as gathering/scattering one 1 MB subchunk at a time for large chunks), they both can carry out the computation correctly. Neither servers nor clients need to communicate with one another during the computation phase in Panda 2.0 and 2.1. In Panda 2.1, communication is required only for the purpose of moving the array data between clients and servers. Each data item is moved only once through the network, and there are no additional messages except for the schema message transfers at the beginning of the i/o request and the message transfers among the nodes at the end of the i/o request that signal the completion of the collective i/o request. In Panda 2.0 and 2.1, there is no communication between servers or between clients while servers are gathering array chunks for a write operation, or while servers are scattering array chunks to clients in a read operation. The only communication required between servers is at the beginning of the i/o operation when the master server node sends the schema string that describes the entire i/o operation to all other server nodes. And the only communication required among clients is when the master client informs all the clients about the completion of the i/o request.

There are several necessary conditions for Panda 2.1 to work efficiently, which we discuss in the remainder of the section. First, a nonblocking message passing facility, such as the MPI-F implementation used by Panda on the SP2, is needed for array data messages, to avoid serializing the data transfers. With nonblocking MPI communication [mpi94], a message transfer is separated into two phases. In the first phase, the message sends and receives are initiated but not completed. A sender can return from the initiation call before the message is copied out of the send buffer. Similarly, the receiver can return from the initiation call before the message is received in the receive buffer. In the second phase, a separate message completion call is needed to finish the message transfer. A send completion call will return if the message has been sent out from the send buffer, and a receive completion call will return if the message has been received in the receive buffer. In the case of write requests, the use of nonblocking message passing facilities allows a client to initiate all the message sends to all the relevant servers right away. As soon as a server is ready to receive from that client, the relevant data can be moved out of the send buffer. Similarly, servers use nonblocking receives to receive data. In this way, it is possible for servers to actually receive data from the relevant clients in an optimal order, such as the order of the data message arrivals, instead of some a priori order, such as the compute node number order, which may not be optimal. Nonblocking message passing also allows the overlapping of the communication and the computation activities. For instance, a server can receive one portion of array data from one client and compute what other portions it needs to gather at the same time. A client waits for the completion of all its message sends before it begins to output the next array in memory, and servers in Panda wait for the completions of message receives for the first on-disk array chunk before the chunk is written out to the underlying file system.

When a nonblocking message passing facility is used, all the nodes can initiate message transfers in parallel. When i/o nodes are ready to receive the data messages, the data can be copied out of the clients' send buffers right away. For instance, in Figure 2, when i/o node 0 is receiving the data message from compute node 1, i/o node 1 can choose to receive the data message from compute node 2 simultaneously. Then i/o node 0 can receive the data message from compute node 0, and i/o node 1 can receive the data message from compute node 1 simultaneously. The total communication time to transfer the array from compute nodes to i/o nodes could be halved in this example, compared to using the blocking message passing facility.

The ability to examine an incoming set of messages and choose which to receive opens up a new possibility for Panda: support for applications that are not globally synchronized. Currently, if Panda is used for such an application, then compute nodes that are ready to perform their i/o early must wait until slower nodes are also ready. This limits Panda's applicability for heterogeneous environments and for partially synchronized applications. For writes in Panda 2.1, servers could determine what data is currently available from clients and then make an optimal decision about what data to buffer locally and when to initiate file system writes. We plan to explore this possibility for Panda in the future.

The second necessary condition for Panda 2.1 to work efficiently is the use of some flow control mechanism to avoid receiver buffer overflow problems. For example, during writes, all the compute nodes can send data out in parallel, but i/o nodes only allocate space for one array chunk at a time. Thus if a compute node quickly finishes distributing its chunk of array $A$ and then moves on to its chunk from array $B$, the first server it tries to send data to may not yet be finished writing all of its assigned chunks from array $A$ and may not yet be ready to accept any data from array $B$. There are many possible ways to solve this buffer overflow problem; Panda 2.1 uses the synchronous mode of MPI nonblocking message sends, which is part of the MPI standard [mpi94]. Using the synchronous mode with nonblocking message sends, in the case of write requests, all clients can initiate message sends to the relevant servers, but the data messages will not be copied out of clients' send buffers until the corresponding servers initiate MPI receive calls. This ensures that servers will not be flooded by incoming data messages. A good implementation of the synchronous mode communication has another advantage: in the MPI-F message passing library on the NAS IBM SP2, the use of synchronous mode of MPI-F completely eliminates the memory-to-memory copying overhead on the receiver for a message transfer, which can be a substantial component of overall run time in Panda. Since the sender knows that the receiver has already allocated space for the incoming message, the message is copied right into the receiver's buffer in user space. Without synchronous mode (e.g., in Panda 2.0), the message goes to a temporary buffer at the MPI system level, then to user buffer space. The savings on memory-to-memory copying can be quite substantial, since data messages in Panda are often noncontiguous memory locations of a multidimensional array, for which copying can be very expensive. However, synchronous mode has the disadvantage that sending a message typically requires a longer startup time than when using standard mode. However, our experimental results show that the overall reduction in communication and computation cost that comes from eliminating data request messages is much higher than the increase in message startup cost.

The third necessary condition is that clients in Panda know the number of servers in order to figure out where data should go to or come from. We do not consider this to be a serious limitation since if the number of servers is static, informing the clients will require at most one round of messages during Panda initialization.

One potential disadvantage of eliminating data request messages is that clients and servers now have to agree beforehand on the strategies used to gather/scatter array chunks. This requirement could prevent servers from deciding dynamically at run time how to request and distribute data. For instance, if one server is currently overloaded with other work, the load cannot be rebalanced easily since clients did not know about the imbalance ahead of time. In the future, we would like to explore approaches that relax this restriction, while still avoiding unnecessary data request messages.

## 4 Performance comparison with Panda 2.0

We tested the performance of Panda 2.1 and Panda 2.0 on the IBM SP2 at NASA Ames Research Center. Table 1 gives the characteristics of the IBM SP2 at NAS; more detailed information can be found at http://lovelace.nas.nasa.gov/Parallel/SP2. Each processor in the SP2 has its own separate AIX file system, which Panda uses directly on each i/o node for storing array data. The partitions used to store array data were approximately 15-20% full at the time of the experiments.

We began experimentation by regression-testing Panda 2.1 against benchmark 2.0 (reading and writing very large arrays, both with real disks and simulated infinitely fast disks). As expected, we found that in all cases Panda 2.1 delivered the same or better performance than Panda 2.0 [Seamons95], as discussed briefly at the end of this section. We then moved on to experiments with large numbers of small arrays, discussed immediately below.

The new experiments characterize the i/o requirements of the black hole application as described in section 1. We measured Panda's performance to write 2048 arrays of size 64 KB or 128 KB each. The choice of array sizes is based on the characteristics of the black hole application, which typically has arrays of size 1 MB or less that are distributed across 256 or 512 nodes on the IBM SP2. Each in-memory array chunk is only 4 KB or less. Because of the difficulty in acquiring large numbers of nodes and the large blocks of machine time necessary for the large runs in our performance study, we scaled down the array size so that in-memory array chunks have sizes similar to what is used in the black hole application on 256 or 512 nodes of the SP2. Our runs vary the number of compute nodes and i/o nodes. All of our runs use 32 or fewer nodes, due to machine availability constraints. The number of compute nodes is 8 ($2 \times 2 \times 2$ mesh), 16 ($2 \times 2 \times 4$ mesh), or 32 ($2 \times 4 \times 4$ mesh) and the number of i/o nodes is 2, 4, or 8. In all cases, we used a (BLOCK, BLOCK, BLOCK) schema for in-memory arrays, and a (BLOCK, *, *) schema for arrays on disk. The logical mesh is ($n \times 1 \times 1$) on disk, where $n$ is the number of i/o nodes used. In this way, all the arrays will be stored in traditional row major order across several disks even though they are distributed very differently in memory. With these distributions, to gather an array chunk, a server will need array data from multiple clients.

| | |
|---|---|
| Total number of nodes | 160 nodes |
| Node information | |
| Node type | RS6000/590 workstation |
| Processor type | POWER2 RISC |
| Processor rate | 66.7 MHz |
| Node OS | AIX 3.2.5 |
| Total memory | 23.9 GB |
| Memory per node | 128 MB |
| Total disk space | 485 GB |
| Disk space per node | 2 GB |
| Total memory bandwidth | 342 GB/s |
| Peak performance | 42.8 Gflops |
| Switch bandwidth (hardware) | 40 MB/s |
| Peak disk transfer rate | 3.0 MB/s |
| I/O bus | SCSI |
| Peak I/O bus transfer rate | 10 MB/s |
| AIX file system block size | 4 KB |
| MPI-F latency | 43 microseconds |
| MPI-F bandwidth | 34 MB/s |
| Measured raw AIX file system performance | |
| Read | 2.85 MB/s |
| Write | 2.23 MB/s |

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

For performance data, we repeated each test four times and use the average elapsed time. The elapsed time is the maximum time spent by any compute node in 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 for reading or writing a file using AIX file system calls. 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. The experiments described below all measure write operations, as reads and writes in Panda show the same performance trends [Seamons95].

Under our experimental setup, we expect that when the number of i/o nodes is small, Panda's throughput on each i/o node can be very close to the peak AIX file system throughput. However, as the number of i/o nodes increases, Panda 2.0's throughput per i/o node will drop dramatically, for the following reasons. In Panda, the file system i/o cost on each i/o node is proportional to the amount of data written to disk, so as the number of i/o nodes increases and therefore the amount of data on each i/o node decreases, the file system cost also decreases. However, the communication cost does not decrease proportionally as the amount of data transferred to an i/o node decreases, and sometimes the communication cost may even increase because small data messages do not utilize available MPI bandwidth well. Once the total computation and communication cost dominates the overall cost of the i/o request, Panda's performance is limited by its message passing performance regardless of how fast the underlying file system can access data.

This phenomenon is illustrated by the experiments summarized in Figures 3 through 5. With two i/o nodes the throughputs of Panda 2.0 and 2.1 are limited by the performance of the underlying file system, and both obtain 90–98% of the peak AIX file system throughput at each i/o node. The throughputs for Panda 2.0 drop dramatically as the number of i/o nodes increases, so that with eight i/o nodes Panda 2.0's bottleneck is the message passing system. The graphs for four i/o nodes show the crossover point for Panda 2.0 as the bottleneck shifts from the file system (with 128 KB arrays) to message passing (with 64 KB arrays): the number of messages required to gather an array chunk is the same in both cases, so doubling the array size also doubles the array chunk size and the amount of data to be written out by each i/o node, and hence the file system cost. However, the communication cost is not doubled because MPI-F utilizes the available message passing bandwidth much more efficiently with the larger data messages. Thus, increasing the amount of data transferred does not stress the message passing system at all, and instead leads to better utilization of the available bandwidth than before as shown in Figures 6 through 8, which show the percentage utilization of peak MPI bandwidth in Panda. As we can see, the message passing bandwidth with 128 KB arrays is almost twice as high as that with 64 KB arrays, for a fixed number of compute and i/o nodes. Thus, the cost of communication in many cases increases only by a very small percentage, or even decreases as the size of arrays is doubled. Obviously, this does not apply to cases where Panda already comes close to the peak message passing bandwidth on the NAS IBM SP2. In those cases, the message passing system is not the system bottleneck anyway. In Figure 5 with 4 i/o nodes, 64 KB arrays, and a message-passing bottleneck for Panda 2.1, the message passing bandwidth used is only about 76% of the peak file system throughput (Figure 8 shows the message-passing throughput). When 128 KB arrays are used, the obtained message passing throughput is almost doubled, to around 150% of the peak file system throughput and Panda's performance bottleneck becomes the underlying file system. As long as Panda's internal communication and computation cost is kept below the file system i/o cost, Panda can achieve high performance, as shown in these figures.

Figures 3 through 5 also show that eliminating the data request messages of Panda 2.0 has greatly reduced the message passing cost of i/o requests in Panda 2.1. The throughputs of Panda 2.1 are about twice as high as for Panda 2.0 with 8 i/o nodes, and Panda 2.1 delivers at least 78% of the peak performance of the underlying file system on each i/o node, except for one case: 32 compute nodes with 8 i/o nodes and 64 KB arrays. In that case, the total communication and computation cost still dominates the file system i/o cost because of the large number of small messages used during the i/o request. Other configurations do not have as many messages because the number of compute nodes used is relatively small, and the number of in-memory chunks is small. Without techniques to further reduce the communication and computation cost, such as combining messages during the data transfer phase, Panda 2.1 still has a message passing bottleneck in this case.

We also conducted tests simulating an infinitely fast disk to observe how Panda utilizes the network bandwidth. Figures 6 through 8 show the normalized throughput (obtained by dividing Panda's throughput per i/o node by the peak MPI bandwidth, 34 MB/s) for these experiments. The cost for message passing is very high, as the throughput for writes ranges from 2-23% of peak MPI performance in Panda 2.0. When 8 i/o nodes are used, the message passing bandwidth utilization is only around 3% of the peak MPI performance, and 50% of the peak AIX file system throughput for write operations. It is clear that the system bottleneck is low message passing throughput in those cases. Panda 2.1 more than doubles the MPI throughputs achieved in the case of 8 i/o nodes. In order to approximate the excellent message-passing performance of Panda 2.0 and 2.1 for large arrays, however, additional techniques will be required to merge small array data messages into larger ones.

There are several other trends seen in the figures. First, when the file system is the limiting factor as shown in Figures 3 and 4 with Panda 2.1, as the number of i/o nodes increases, Panda's throughput per i/o node shows a gradual decrease. We explain this scenario using a simple performance model for Panda. The following formula models the total elapsed time for a collective i/o request at a compute node:

$$\begin{aligned} T_{total} &= T_{startup} + \max_{all\ i/o\ nodes}(T_{i/o}) \\ &\quad + T_{shutdown} \end{aligned} \tag{1}$$

Since Panda's performance is measured by taking the maximum elapsed time spent by any compute node in the collective i/o request, and Panda makes sure that the compute nodes will not return until all the i/o nodes have finished their work, $T_{total}$ is roughly the same for all the compute nodes in Panda, so we can use $T_{total}$ to approximate Panda's performance. Here, $T_{startup}$ is the startup cost that includes the time for the client master node to send the schema information to the server master node, for the server master node to send it to all other servers, and for them all to start to digest the information. $T_{shutdown}$ is the shutdown cost that includes the cost for the server master to inform the client master node about the completion of the collective i/o operation and the client master to send the termination message to all the other clients. $T_{i/o}$ represents the elapsed time for processing the collective i/o request at an i/o node. Since Panda makes sure that the compute nodes will not return until all the i/o nodes have finished their work, the elapsed time at the i/o nodes is bounded by the maximum of the elapsed time at each i/o node, as shown in formula 2. $T_{i/o}$ itself can be roughly formulated as the following:

$$T_{i/o} = \max(T_{file\_system\_i/o}, T_{comm}) \tag{2}$$

Here, $T_{file\_system\_i/o}$ is the total time required to read/write all the array chunks at that i/o node. $T_{comm}$ is the total communication and computation costs to gather and scatter all chunks at that i/o node. We have assumed that the file system is the limiting factor, and since each i/o node has about the same amount of data to read/write, $T_{i/o}$ is roughly the same for all the i/o nodes in Panda. $T_{i/o}$ and $T_{total}$ can be rewritten as follows:

$$\begin{aligned} T_{i/o} &= T_{file\_system\_i/o} && (3) \\ &= \frac{S_{data}}{file\_system\_throughput} \\ T_{total} &= T_{startup} + \frac{S_{data}}{file\_system\_throughput} && (4) \\ &\quad + T_{shutdown} \end{aligned}$$

where $S_{data}$ is the amount of data read/written by the underlying file system at an i/o node. Formula 4 holds because Panda is able to fully utilize AIX bandwidth at each i/o node. Panda's throughput $P$ is then modeled as follows:

$$\mathrm{P} = \frac{S_{data}}{T_{total}} \tag{5}$$

$$= \frac{S_{data}}{T_{startup} + \frac{S_{data}}{file\_system\_throughput} + T_{shutdown}}$$

$$= \frac{1}{\frac{(T_{startup}+T_{shutdown})}{S_{data}} + \frac{1}{file\_system\_throughput}}$$

As the number of i/o nodes increases, the startup and shutdown costs do not change much, but the amount of data $S_{data}$ per i/o node to be written out to disk decreases. Therefore, the denominator increases, and Panda's throughput per i/o node decreases. The second trend seen in the figures is that when the file system cost dominates as shown in Figures 3 and 4, as the array size increases for a fixed number of i/o and compute nodes, Panda's throughput increases. This can be explained by formula 5 as well. As the array size increases, $S_{data}$ increases, the denominator decreases, hence, Panda's throughput increases.

The experiments discussed so far compare Panda 2.1 with non-blocking message passing to Panda 2.0 with blocking message passing facilities. The performance improvements of Panda 2.1 come both from the elimination of the cost associated with the request messages and from the use of nonblocking message passing to increase Panda's internal level of parallelism. If nonblocking message passing were added to Panda 2.0 ("nonblocking Panda 2.0"), we expect a moderate improvement on the black hole benchmarks and benchmark 2.0, as explained below.

Using nonblocking messages, a message can be shipped over the network while both sender and receiver are computing, which is also equivalent to saying that $T_{transfer}$ in formula 1 can be overlapped with other computation activities on the sender and receiver. With very small messages, $T_{transfer}$ is also very small (tens of microseconds). Hence, the potential overlap of communication and computation costs is not very large. Furthermore, using nonblocking message passing introduces one more overhead in polling for the completion of the message sends and receives. As a result, the overall gain from overlapping communication and computation activities using nonblocking message passing is not very significant when messages are very small. Sometimes, the gain from overlapping communication and computation may be even smaller than the cost of extra polling. In those cases, the performance could be even worse using nonblocking message passing facilities than using blocking message passing facilities. For instance, in Figure 11, when 2 i/o nodes are used, the performance of Panda 2.0 using nonblocking message passing and simulating a fast disk is even worse than the performance of Panda 2.0 with the same configuration. However, as the sizes of the data messages increase, the amount of overlap increases, and the benefit of using nonblocking message passing is more obvious, as shown in Figures 9 and 10. The performance results for benchmark 2.0 experiments using nonblocking Panda 2.0 and Panda 2.1 also showed significant improvement over Panda 2.0, for simulated fast disk experiments; for real disk experiments, there was little difference in performance due to the overall heavy dominance of file i/o costs, which are the same for all three versions of Panda.

The gain from being able to receive data in an optimal order using nonblocking message passing is also not very obvious with nonblocking Panda 2.0. In the case of benchmark 2.0, the potential serialization penalty (roughly the time for each client to ship a 1 MB subchunk across the network) is dwarfed by file i/o costs. In the black hole benchmarks, the gain from an optimal receiving order would have been more obvious in a more heterogeneous computing environment where nodes are not naturally closely synchronized, e.g., if the hardware or software environment where Panda was run were heterogeneous, if the partition where we ran our experiments were shared with other application processes, or if the application were not closely synchronized across all compute nodes.

## 5 Related work

The common use of synchronized collective i/o operations in scientific applications has 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 cooperatively 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. Besides the Panda library, several parallel file systems and multidimensional array libraries have provided a collective i/o interface [Corbett95, del Rosario94, Karpovich94, Seamons94, Seamons95, Seligman94]; some of them have shown that collective i/o provides high performance given appropriate language, compiler and run-time system support.

[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. This strategy does not require extra request messages in order to transfer application data, but it does usually move the data items twice through the network; in Panda 2.1, each data item is moved once. In the two-phase strategy, the permutation phase and the i/o phase are not overlapped, while in Panda 2.1, the communication time is largely overlapped with the time to perform file system i/o operations, giving some extra time savings.

The design of Panda 2.0 is based on the 'disk-directed i/o' for collective i/o operations that is proposed in [Kotz94b]. While different in many respects, under both architectures nodes direct the data flow in an i/o request in order to make sequential file system requests and highly utilize the underlying file system throughput. Under disk-directed i/o, the file system on i/o nodes gathers data from contiguous regions of the clients' memory in a manner that will allow writes to occur in disk block order. Like the design in Panda 2.0, disk-directed i/o will have low message passing bandwidth utilization given a large number of small message transfers. We are not sure how to fix this potential performance problem in disk-directed i/o, as it would be too extreme to assume that clients know the exact disk layout of files on each i/o node.

## 6 Conclusions

In this paper we presented mechanisms to improve the scalability of message passing in the design of Panda 2.0. Our approach, embodied in Panda 2.1, reduces the communication and computation cost in Panda by eliminating data request messages from servers to clients and by increasing Panda's internal level of parallelism through use of MPI's nonblocking message passing facilities. Our experimental results show that Panda 2.1 can efficiently support a larger range of applications, by improving support for i/o of small data items. Panda 2.1 also shows promise for platforms with slower networks, as well as applications where compute nodes are not closely synchronized, and heterogeneous computing environments.

Panda 2.1 uses the same high level application interfaces of the Panda library as Panda 2.0, illustrating how Panda's high level interfaces allow us to provide different implementations for different platforms and to experiment with different optimization strategies without sacrificing the portability of user applications. In the near future, we will further extend Panda's range by implementing techniques to merge multiple data messages sent between a compute node and i/o node, to further reduce communication cost. We will test these techniques in a port of Panda to a different platform (a cluster of high-performance HP workstations) which has a relatively slow network.

## References


[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.

[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.

[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.

[mpi94] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard. Messge Passing Interface Forum, May 5, 1994.

[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.

[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.

[Seamons94] K. E. Seamons and M. Winslett, An Efficient Abstract Interface for Multidimensional Array I/O, Proceedings of Supercomputing '94, Washington D.C., November 1994.

[Seamons95] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, Server-Directed Collective I/O in Panda, Proceedings of Supercomputing '95, San Diego, CA, December 1995.

[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.

[Sio94] Application Working Group of the Scalable I/O Initiative. Preliminary Survey of I/O Intensive Applications. Scalable I/O Initiative Working Paper No. 1.

[Thakur95] Rajeev Thakur, Ewing Lusk, and William Gropp. I/O Characterization of a Portable Astrophysics Application on the IBM SP and Intel Paragon, Technical Report MCS-P534-0895, Argonne National Laboratory, August, 1995.

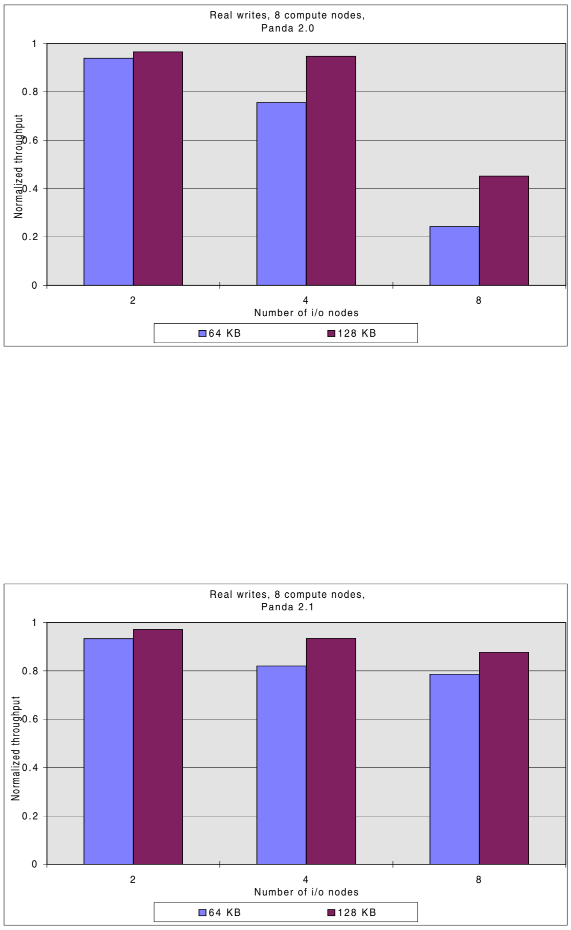


Figure 3: Normalized throughput (i.e., throughput obtained by Panda, expressed as a percentage of potential peak AIX throughput) at each i/o node for writing 2048 arrays of size 64 KB and 128 KB with 8 compute nodes, as a function of the number of i/o nodes. The schema on disk is row-major order (i.e., (BLOCK, *, *)), and the schema in memory is (BLOCK, BLOCK, BLOCK). The logical mesh is $(2 \times 2 \times 2)$ in memory and $(n \times 1 \times 1)$ on disk, where $n$ is the number of i/o nodes. The graph at the top shows the performance results of Panda 2.0 and the graph at the bottom shows the performance of Panda 2.1.

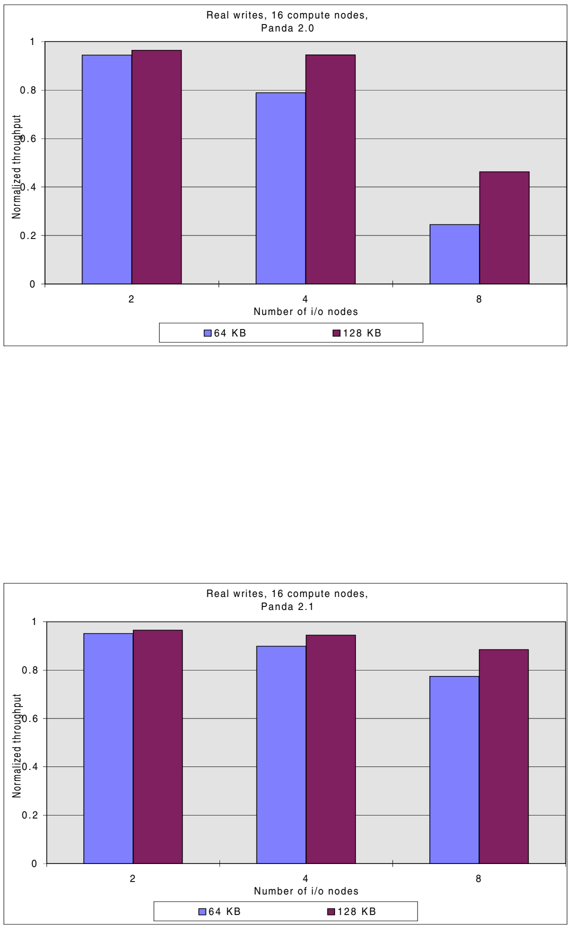


Figure 4: Normalized throughput with 16 compute nodes and a $(2 \times 2 \times 4)$ logical mesh in memory. Other parameter settings are the same as in Figure 3.

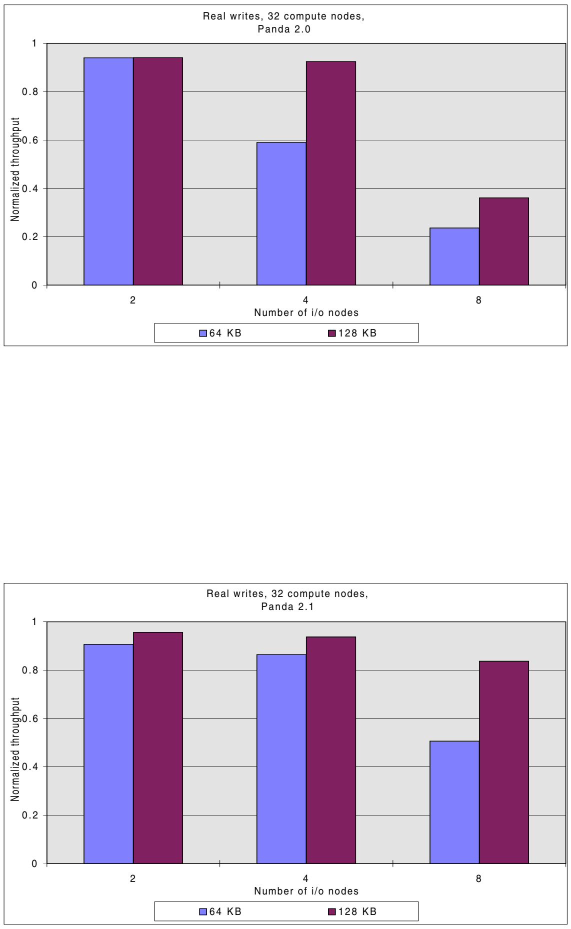


Figure 5: Normalized throughput with 32 compute nodes and a $(2 \times 4 \times 4)$ logical mesh in memory. Other parameter settings are the same as in Figure 3.

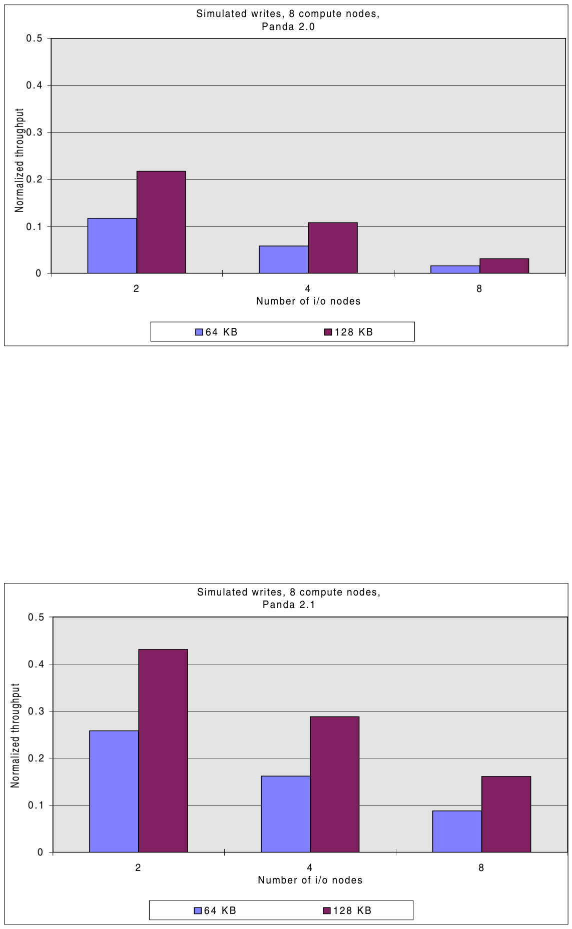


Figure 6: Normalized throughput with 8 compute nodes when simulating an infinitely fast disk. Other parameter settings are the same as in Figure 3.

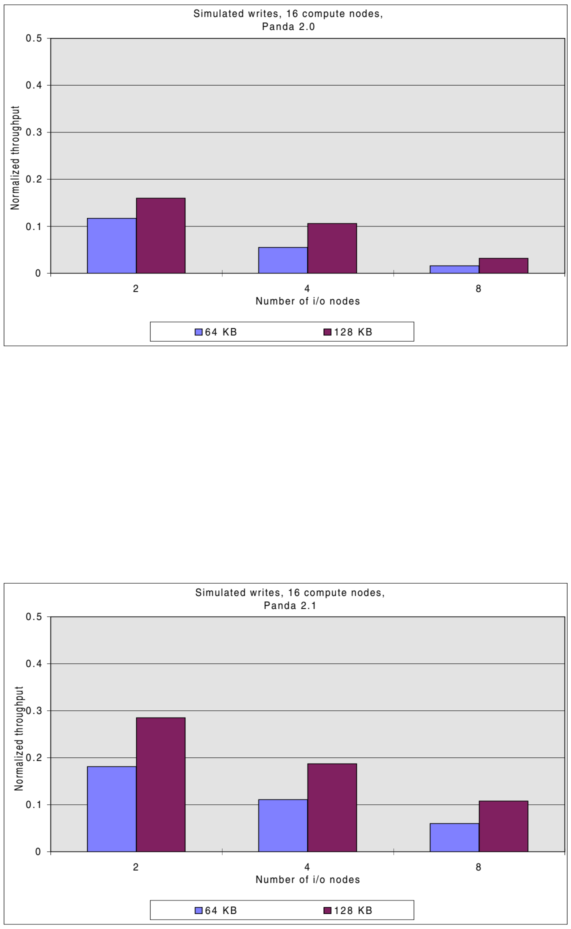


Figure 7: Normalized throughput with 16 compute nodes when simulating an infinitely fast disk. Other parameter settings are the same as in Figure 4.

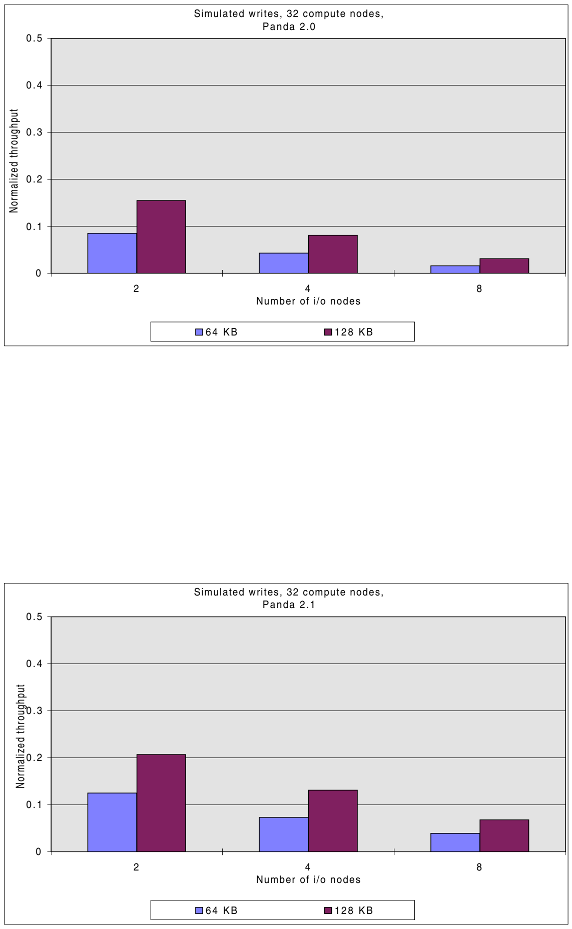


Figure 8: Normalized throughput with 32 compute nodes when simulating an infinitely fast disk. Other parameter settings are the same as in Figure 5.

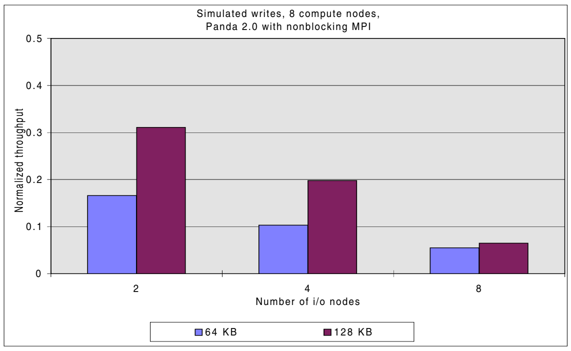


Figure 9: Normalized throughput for nonblocking Panda 2.0 with 8 compute nodes when simulating an infinitely fast disk. Other parameter settings are the same as in Figure 6.

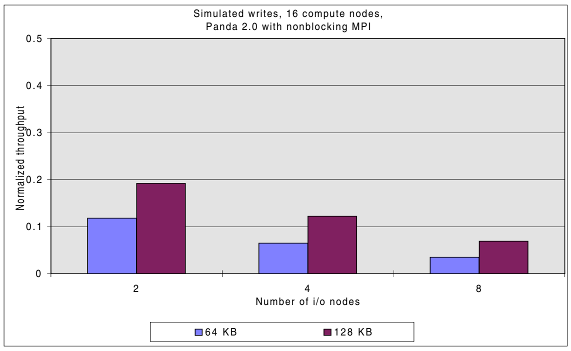


Figure 10: Normalized throughput for nonblocking Panda 2.0 with 16 compute nodes when simulating an infinitely fast disk. Other parameter settings are the same as in Figure 7.

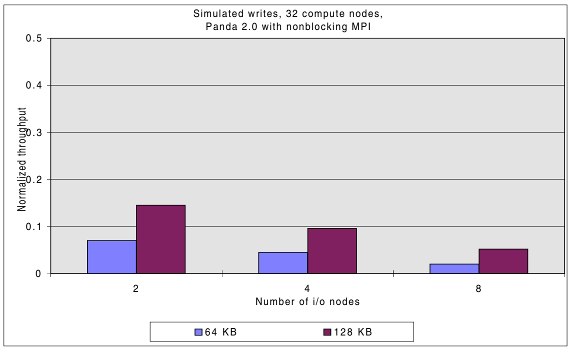


Figure 11: Normalized throughput for nonblocking Panda 2.0 with 32 compute nodes when simulating an infinitely fast disk. Other parameter settings are the same as in Figure 8.