Performance of an MPI-IO implementation using third-party transfer

Richard Hedges, Terry Jones, John May, R. Kim Yates[*]
Lawrence Livermore National Laboratory
P.O. Box 808, Livermore, CA 94551
rkyates@llnl.gov
tel +1-925-423-5535

Abstract

We present a unique new implementation of MPI-IO (as defined in the recent MPI-2 message passing standard) that is easy to use, fast, efficient, and complete. Our implementation is layered over the High-Performance Storage System, using HPSS's third-party transfers and parallel I/O descriptors.

1 Introduction

The MPI-2 standard [9] includes a chapter devoted to parallel I/O functions, often called "MPI-IO." Several partial or complete implementations have appeared (e.g., ROMIO [13], PMPIO [2], Sun MPI [11], and Fujitsu's MPI-2 [7]). This paper describes a new MPI-IO implementation that has several important and unique features:

- It provides large-scale scientific applications with an easy-to-use, high-performance, portable interface to petabyte archival storage systems.
- It shows good performance and very high utilization of the underlying storage system. Our tests have demonstrated peak MPI-IO bandwidth of up to 197 MB/s for collective read operations and up to 173 MB/s for write operations, out of a maximum available bandwidth of 207 MB/s on our test platform. Even though our test platform is small (see Sec. 3.1), this compares favorably with, for example, the 150 MB/sec maximum throughput reported for PDS/PIO on the Intel TFLOP [10], both in terms of absolute performance and efficiency.

[*]This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48.

- It uses a unique I/O mechanism known as *third-party transfer* in the High Performance Storage System (HPSS) archival storage system [1, 12, 14]. That is, the transfer of data from one processor to another may be arranged by a third processor, which does not participate in the actual movement of the data.

- It fully implements every MPI-IO function, including shared file pointers, error handlers, and automatic conversion between data representations.

- It is designed to work with any MPI-1 library, provided user applications are compiled with `mpio.h`. This header file defines all needed MPI-2 extensions for MPI-IO, including macros that enable MPI-IO to have access to the arguments given to MPI datatype constructors.

- It is thread-safe (as long as the underlying MPI library is also thread-safe).

Our MPI-IO implementation is a new user interface first provided with release 4.1 of HPSS. HPSS is an archival storage system that is designed to manage very large files on both disk and tape. HPSS is a joint project of IBM and several U.S. national laboratories, with a significant number of production installations.

The focus of our implementation has been to provide an efficient and scalable standard interface to the HPSS file system, providing the full functionality of the MPI-IO specification. We exploit HPSS I/O descriptors (Sec. 2.2), file striping, MPI-IO file hints, and third-party transfers to parallelize collective I/O. We shelter the user from HPSS details and constraints as much as possible. We utilize a threaded and distributed work model to minimize the latency of interactions with HPSS and to support the potential concurrency of nonblocking I/O. Our results affirm the promise of scalable performance with low overhead costs for layering MPI-IO over HPSS.

2 MPI-IO implementation

Our implementation of MPI-IO is specifically designed to run over HPSS and to take advantage of its third-party transfer capabilities. We have described this implementation elsewhere [5]; this section summarizes the design.

2.1 MPI-IO background

Two significant features of MPI-IO are *collective I/O* and access to discontiguous data chunks[1] using MPI *datatypes*. In a collective transfer, a group of processes in an application each perform a special MPI-IO read or write call, which can be implemented in such a way that information about each of the separate calls can be shared. Hence the library

[1]We use the term “chunk” to refer to a contiguous sequence of bytes in a file or memory.

can coordinate requests from multiple processes, and merge these requests to improve the locality of accesses within the file, potentially eliminating many needless disk accesses. MPI datatypes provide a mechanism for describing common static memory access patterns (e.g., slices of multidimensional arrays, records with arbitrary gaps, etc.) [3]. Using one datatype to address discontiguous regions of a file and another to address regions of a memory buffer, a single call can effect the complex data movement between them.

2.2 MPI-IO/HPSS implementation

The basic HPSS client API includes a mechanism for specifying third-party transfers using data structures called *IODs* (for "I/O descriptors"). An IOD specifies a transfer between a file and one or more client processes, and has two sides: one side describes a sequence of file chunks, and the other describes a sequence of client process memory chunks. Hence, file chunks accessed by multiple client processes can be collected into a single IOD that can be passed to HPSS through an `hpss_ReadList` or `hpss_WriteList` call.

IODs are a very flexible mechanism for describing parallel data transfers, but constructing an IOD requires quite a bit of detailed coding, and the interface is only usable for accessing HPSS files. In addition, use of the `hpss_ReadList/WriteList` interface requires an application to manage *client mover threads,* which connect application processes to remote storage devices via sockets to carry out data transfers. To offer applications a simpler and more portable programming interface, we have implemented MPI-IO on top of the HPSS IOD mechanism. The main tasks of our MPI-IO library are to manage the client mover threads and to convert MPI-IO requests into IODs.

To optimize collective I/O, it is necessary to submit I/O requests to HPSS from a single IOD. This means that collective MPI-IO calls must forward requests from many tasks to a single thread, which merges them to form an IOD. Our implementation does this using a set of *server threads* with one server thread running within each process of a parallel application. Each MPI-IO file is a single HPSS file; HPSS handles striping internally. One of our implementation's server threads is assigned to handle all collective operations on a given open file handle. However, different open file handles may be managed by different servers, so no single process bears the burden of managing all the open files. The server thread for each process is created when the application initializes MPI-IO, and a server thread only manages file handles for the parallel job that spawned it.

When an application makes a collective data access request, each process in the application forwards its part of the request to the server thread managing the request's file handle, which assembles an IOD and submits it to HPSS. HPSS carries out the data transfer to all participating processes (in parallel if possible) and then returns. Note that although *control* of the request is centralized at the server thread, the *data* itself does not flow through the server; with third-party transfer the data can move directly between storage devices and processes in parallel.

Our implementation attempts to translate collective MPI-IO requests into IODs that transfer data as efficiently as possible. Since HPSS sequentializes separate access requests for a given file handle, describing as much of a transfer as possible in a single IOD improves the opportunities for parallelism and thereby improves performance. In the ideal case requests can be merged into a transfer whose file descriptor list contains a single element (i.e., the merged transfer accesses a single contiguous chunk of the file) and whose client descriptor list uniformly stripes the transferred data across the client processes. Furthermore, optimal performance is achieved if the client side striping exactly matches the HPSS file striping specified through the HPSS class of service.

Our implementation recognizes when it can merge collective transfer requests from multiple processes into a request to access a single contiguous region of a file. If the requests cannot be merged into a single contiguous access, the IOD will require a separate file descriptor for each discontiguous chunk of the access. However, the file descriptor list of an HPSS IOD is limited to 64 descriptor elements at the time of the tests reported here. When a transfer requires an IOD with more than 64 file descriptors, our implementation automatically divides the request into multiple IODs with 64 or fewer file descriptors.

Moreover, the implementation recognizes when the collection of client memory chunks can be described compactly by a regularly "striped" pattern. Our implementation also recognizes when it can describe the client transfer mapping as uniformly distributed across the participating client processes. This simplifies the client source/sink descriptor given to HPSS by using a "striped" address. If the HPSS file is striped across the same number of devices as the number of participating processes, and if the chunk size of the client distribution matches the chunk size of the HPSS file stripe, HPSS will be able to achieve a one-to-one connection between its movers and our MPI-IO client movers for the transfer, allowing maximal concurrency.

If a transfer is not uniformly distributed across the clients, each contiguous chunk of the transfer per client requires a separate descriptor in the IOD. For these cases, we arrange the transfer description so that if there are n client processes, a maximum of n client descriptors will be used. For each client that accesses discontiguous regions of the file, however, this will result in multiple file descriptors. To summarize, irregular client distributions and/or discontiguous accesses to the file (e.g., holes in the file) will result in suboptimal performance.

3 Performance

This section reports the performance of our implementation on the platform described in Section 3.1. We begin by describing the test methods, and then we report the results of these tests. We conclude the section with a discussion of these results.

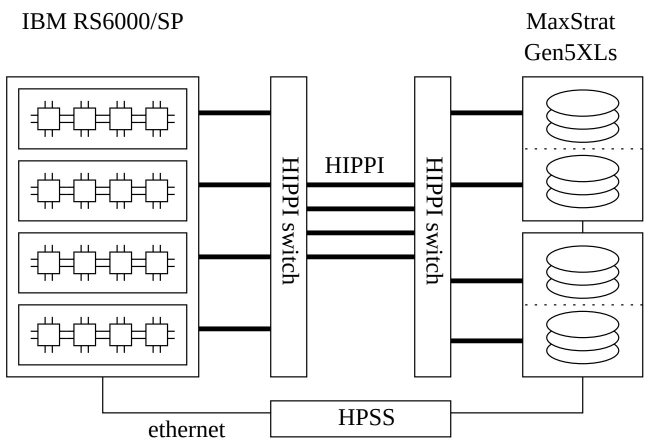

Figure 1: The test platform.

3.1 The test platform

An important feature of HPSS is its support of third-party data transfer. In this I/O model, a single process can issue a request to HPSS that will cause data to flow between the storage system and multiple processes on multiple compute nodes. (Recall that “third-party” refers to the fact that the process initiating a request need not be either the source or the destination of the data.) Using third-party transfer can simplify the management of parallel I/O transfers and reduce the number of times data is copied in a given operation. For example, a read request causes data to move between a single logical file and multiple destination processes. Third-party transfer allows this to happen with no need for intermediate buffering or shuffling of data between nodes. In this respect third-party transfer is similar to Kotz’s disk-directed I/O technique [6]. When a file is striped over multiple storage devices, and different nodes are accessing different stripes, a single I/O request can initiate parallel transfer of data. Of course, the degree of parallelism will depend on how well the striping matches the data distribution on the nodes.

The main components of our hardware test platform are a parallel computer, two RAID storage devices, an interconnection network, and an HPSS server (see Fig. 1). Application code runs on a 16-processor SMP cluster consisting of four IBM RS/6000 SP 604 High Nodes that each contain four 112-MHz PowerPC 604 processors. Each node has its own HIPPI adapter card. The four HIPPI cards are connected through a crossbar switch to two Maximum Strategy Gen5 XL RAID systems. Each of these systems is configured to operate as two independent RAID devices, each with its own HIPPI connection. The maximum theoretical HIPPI bandwidth between the two RAIDs and the SMP cluster is 400 MB/second. However, the adapter cards on the SMP nodes do not stream data at full HIPPI rates, and the maximum observed bandwidth from a given card depends on the number of simultaneous connections. IBM reports that a single connection can sustain 31.5 MB/s, and that four connections through the same adapter sustain 51.8 MB/s [4]. Furthermore, MaxStrat has suggested there is a maximum theoretical transfer rate through a single HIPPI adapter channel on each RAID system of 80 MB/s [8], with less than 70 MB/s in practice. We discuss the effect of these limitations further in Section 3.4.

3.2 Test methodology

We tested the MPI-IO code using a program that let us vary four parameters:

- Stripe factor: The number of devices over which a file is striped. For these tests, we used striping factors of 1, 2, 4, and 8. As noted above, our two RAID systems behave as four independent RAID devices. For n-way striping, the HPSS stripes are evenly distributed across the four RAID logical devices. We use a striping unit (number of contiguous bytes within each stripe segment of a file) of 8 MB.
- File size: The size of the file was varied from 1 KB to 64 MB.
- Chunk size: The size of contiguous chunks that are interleaved in the file. We tested chunk sizes ranging from 1 to 16 MB in power-of-two increments.
- Number of MPI processes: We used 1, 2, 4, 8, and 16 processes; we limited the number of processes to stay within the number of CPUs available on our test system.

In addition to this test program, we evaluated MPI-IO with three more tests. The first compares native and nonnative data representation, the second compares blocking and non-blocking I/O, and the third measures performance tuning.

We configured the HPSS storage classes and used appropriate hints to `hpss_Open` so that the files we were writing would require minimum allocation and metadata overhead. We configured our test environment to enable the HPSS IPI protocol (third-party transfers), which sends data over the HIPPI network connection. Through particular choice of HPSS class of service and MPI-IO "hints," we configured HPSS to optimize transfers of large files for large-chunk accesses, with large chunks), knowing that this would be inappropriate for small files and small chunks. A production HPSS system would provide configurations appropriate for a variety of file and access needs, beyond what we used in our testing.

In the performance plots in the following section, each data point represents an average of five (in a few cases, four) test measurements. For write operations the test program overwrites a file that has already been created and written. Therefore, write timings do not include the time needed to allocate file blocks. Equivalently, we could have preallocated the file before writing.

One limitation of the experiments presented here is that the file size never exceeds 256 MB. However, HPSS does not cache file data, so caching effects should be inconsequential. It is reasonable to expect that transfer of larger files will perform similarly to the largest files reported on here; future experiments on a new testbed will be conducted to verify this.

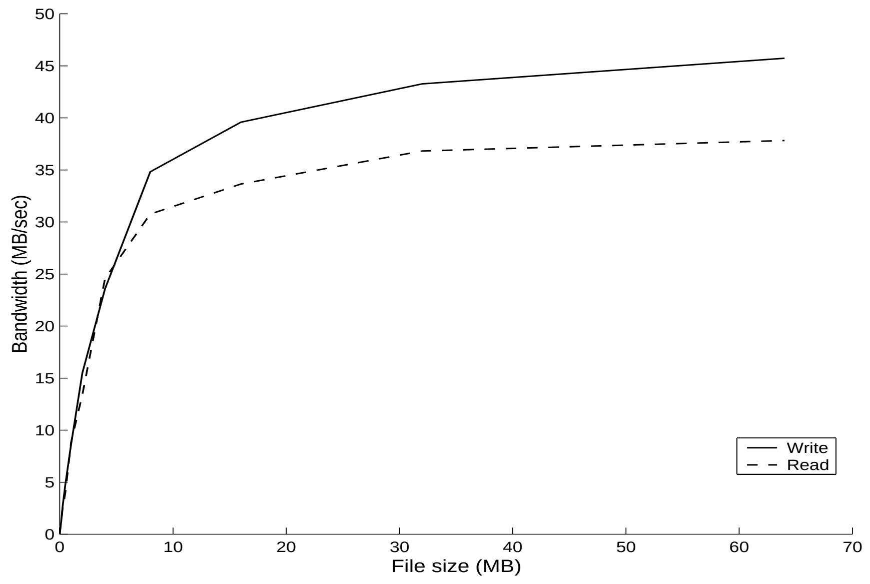

Figure 2: Baseline performance: 1 client process, stripe factor=1.

3.3 Performance results

We first show the read and write performance for independent operations with various file sizes. Figure 2 shows the baseline performance for MPI-IO: the throughput for a single process for a given file size with 1-way striping (i.e., no striping: only a single RAID is accessed). Thus, we can anticipate the maximum throughput of *n* processes using collective I/O with *n*-way striping to be *n* times this baseline performance.

For parallel or collective I/O, there are at least two possible ways to measure the I/O bandwidth. One is to take the average of the I/O times on the participating processes and divide this into the total amount of data moved in the operation. A second way is to take the I/O time as the interval between the earliest start time and the latest completion time of the processes. We chose the second method, which is more conservative. We inserted a barrier before each process began timing its portion of each collective I/O operation, so all the processes began each operation at about the same time.

Figure 3 shows the read and write performance for collective operations with various chunk sizes. These results are for parallel jobs with 16 MPI processes, where each process reads or writes a fixed amount of data. For these tests, we used only 8-way striping, but we vary the size of the contiguous chunk of data written by each process from 1 to 16 MB. The file size is 256 MB. At the largest reported chunk size of 16 MB there is just one chunk per process; all the other data points show transfers in which there are multiple chunks per process. These plots show the effectiveness of collective I/O, particularly as the chunk size is increased. Grouping the requests allows HPSS to handle accesses from multiple processes in parallel.

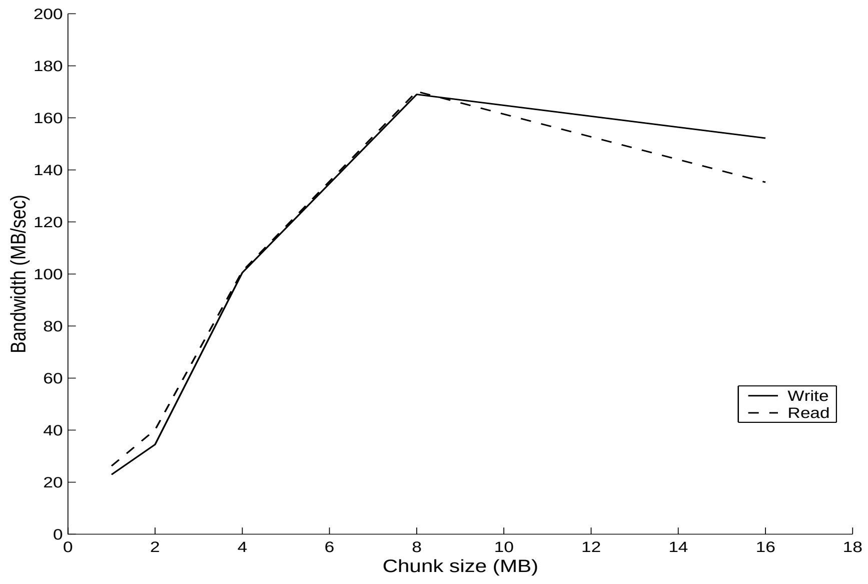

Figure 3: Collective write and read performance for varying chunk sizes
(stripe factor = 8, number of processes = 16, file size = 256 MB).

In Figures 4 and 5 we show how varying the number of tasks and the stripe factor affects performance. The purpose of these tests was to explore what parameters gave the best performance on our system. For these tests, we fixed the chunk size at 16 MB and we only present data for collective operations. We vary the number of tasks from 2 to 16, and we vary the stripe factor from 2 to 8. The results show that collective I/O performance continues to improve as the stripe factor and the number of processes increases.

In a separate test we measured the performance of converting numeric formats during reading and writing. Using `MPI_LONG_DOUBLE` and converting between native format and MPI-2's "external32" representations, we found that the combined effect of the extra buffering and the numeric conversion proper resulted in a slowdown from 12.0 MB/s to 0.8 MB/s for writes and from 14.3 MB/s to 1.3 MB/s for reads (the chunk size was 8 MB for 64-bit native, 16 MB for 128-bit external32).

We measured the impact of using nonnative instead of native data representation. We used a chunk size of 8 or 16 MB (depending on the size of `MPI_LONG_DOUBLE`: in native representation, it is 64 bits; in external32 representation, it is 128 bits), one MPI process, and a stripe factor of 1. We used external32 as the nonnative representation to test. For our platform, only `MPI_LONG_DOUBLE` types require a data conversion from native to external32. However, conversion requires that the data be buffered, which in itself impacts the performance. We isolated the buffering costs by transferring `MPI_BYTE` data, which cause the data to be buffered but not converted. The difference in performance resulted in a slowdown from 12.0 MB/s to 1.7 MB/s for writes and from 14.3 MB/s to 3.0 MB/s for reads. We found the additional effect for conversion by transferring `MPI_LONG_DOUBLE` data. This resulted in a further slowdown to 0.8 MB/s for writes and to 1.3 MB/s for reads.

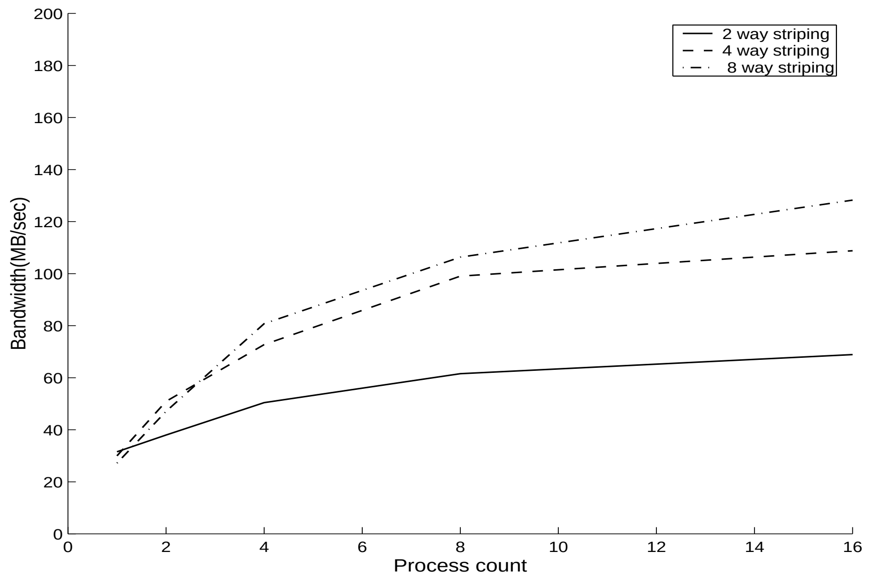

Figure 4: Collective read for varying number of processes
(stripe factor = 2/4/8, chunk size = 16 MB; file size = 16 MB / process).

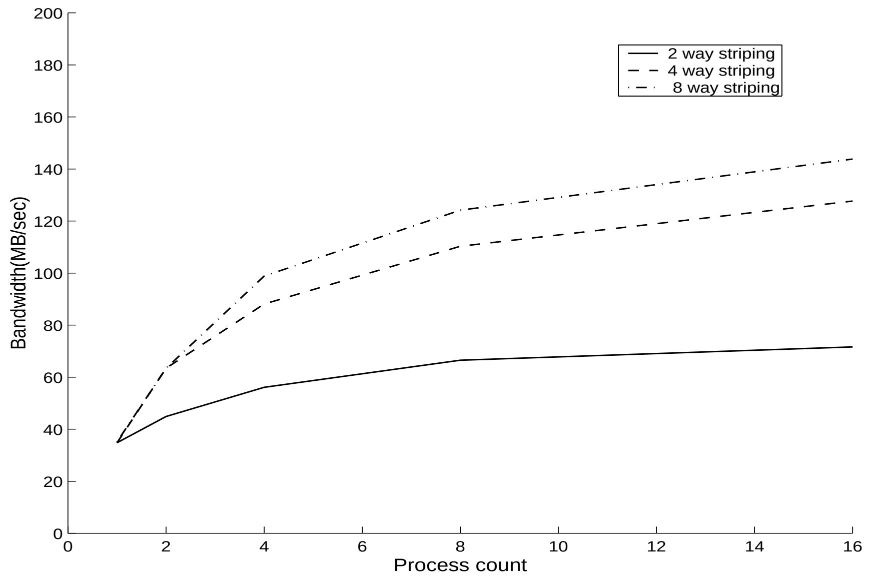

Figure 5: Collective write for varying number of processes
(stripe factor = 2/4/8, chunk size = 16 MB; file size = 16 MB / process).

We also examined the performance of blocking versus nonblocking I/O and the potential speedup of overlapping computation with I/O. We used a test that performed a floating point calculatation sequence for 16 million iterations, and then wrote the results of these calculations to a file. We constructed the calculation sequence so that the I/O and calculation times were approximately equal. We used 4 processes, a stripe factor of 4, and a chunk size of 16 MB, so the file size was 64 MB. We compared the time elapsed using sequential compute-then-write with the time elapsed when we overlap the compute and write phases using nonblocking I/O. We observed that the overlapped compute/write time (3.1 secs) was approximately 62% of the compute-then-write time (5.0 secs). We experimented with more MPI processes and stripe factors, but found that the best overlap was achieved with the four-process case. We believe this is due to contention among the threads that belong to each process when there are insufficient CPUs to assign to each active thread. That is, when there are more than 4 processes, each with multiple active threads, the processes are competing for the 4 CPUs per SMP node. When there is a single multithreaded process per node, there are 4 CPUs available for scheduling these threads.

Lastly, we constructed a test to utilize what we had learned about how to tune performance. We achieved maximum aggregate throughput for this test platform using collective I/O with 32 processes, 8-way striping and 8 MB chunks: we were able to read a 256 MB file at 197 MB/s and write it at 173 MB/s.

3.4 Analysis

The tests that vary file and chunk size show better performance for larger file sizes and chunks. This is expected since HPSS is designed for very large files and transfer sizes. It is worth examining what HPSS parameters contribute to performance variations, although we've alluded to some of these earlier.

Recall that we configured our test environment to use the IPI protocol for HPSS transfers. However, when the chunks are smaller than the HPSS stripe size, HPSS does not use the IPI protocol over HIPPI; instead, it uses TCP/IP over HIPPI. This is because HPSS defines its own blocks that are separate from, and typically larger than, the disk blocks that the storage devices uses. Transfers that do not fill a complete HPSS stripe block require special handling; to improve performance in this case we would need to collect data chunks into larger blocks. Performance peaked when the size of each chunk was 8 MB, our HPSS configuration's stripe size. For example, Fig. 3 shows that the performance of both reads and writes falls somewhat when the chunks grow from 8 to 16 MB.

Another source of performance variation is the contiguity of the data being accessed. For these tests, we deliberately avoided the use of MPI datatypes that contained holes (inaccessible regions) for the file types. That is, when the requests for all tasks are merged in a collective operation, they always form a single contiguous block of file data. Therefore, the limit on IOD length is not exceeded. Another discontiguity penalty is the metadata that is kept by the HPSS bitfile server and its limit on file fragmentation. For example, with even a single hole per filetype, the bitfile server is forced to keep two metadata records per filetype-sized chunk written, one describing the data and one describing the hole. Furthermore, the bitfile server has an upper bound (2K) on the number of fragments (metadata records) it can maintain per file. The creation and maintenance of the metadata hits the write performance severely and the read performance significantly. For the example of a single hole per filetype, writing and reading data with consecutive filetypes which are now discontigous, we see as little as only 10% of the contiguous performance.

Other HPSS configuration details that impact performance are how well the HPSS striping matches the distibution or striping of data over the client processes. This includes matching the size of a striping unit to the size of data being transferred by each client, as well as just matching number of clients and striping units. We see optimal performance when there is a 1-1 correspondence, as in the performance of 8 processes with 8-way striping which is better than the performance of 16 processes with 8-way striping.

Outside of HPSS, other limiting factors are the number of devices and connections available. Although there is a connection to each node, this is multiplexed to 4 CPUs on each node. Similarly, although there is a connection to each logical device, the IPI configuration of an 8-way stripe requires multiplexing of accesses to two stripe units per device.

On our testbed peak performance is limited by the HIPPI connections. Our measured read throughput of 197 MB/s is near the maximum aggregate performance of the HIPPI adapters on the compute nodes, which is 51.8 MB/s $\times$ 4 = 207.2 MB/s.

Although our results demonstrate scalability over the stripe factors and CPUs available for our tests, past performance is no guarantee of future returns. As the number of processors available increases, it is unlikely that all of those processors will have HIPPI interfaces available, and that would require additional data management, and possibly transferring data among nodes to maintain the collective I/O model we have implemented. HPSS is addressing this issue as well.

In summary, users will pay penalties for the flexibility of MPI-IO nonnative data representations and discontiguous accesses. The payoffs of collective I/O and concurrency of computation and I/O may ameliorate some of those penalties.

4 Future work

We are currently working on changes to improve the performance, and will be carrying out further experiments on larger testbeds and production systems.

5 Conclusions

We have examined the performance of a new MPI-IO implementation using third-party transfer and collective parallel I/O capabilities in the High Performance Storage System. Our implementation uses these capabilities to optimize file accesses from multiple processes in a parallel job. We have found the performance to be quite good, at least when reasonably large chunks are used.

Our implementation could be improved by optimizing it further to handle very finely interleaved data accesses. Other MPI-IO implementations use collective buffering to achieve this goal. Currently, our implementation does no data caching or buffering.

A further benefit of this implementation is that it adds to the list of platforms on which MPI-IO is supported efficiently, giving parallel programmers access to petabyte archives via a standard portable interface.

Acknowledgements

We thank John Allen, Rich Fischer, Dave Fox, Mark Gary, Robin Goldstone, Richard Mark, Dave Moffat, Ken Pollet, Joe Slavec, Stan Solbeck, and Steven Solbeck for helping us assemble and debug our test platform. HPSS developers Danny Cook (LANL) and Kurt Everson (IBM), along with Chris Wood of Maximum Strategy and Tom Torrico of IBM, offered several suggestions to help us maximize the performance of IPI transfers for our test platform. Jeanne Martin and Elsie Pierce contributed to earlier versions of this software.

References

[1] R. Coyne, H. Hulen, and R. Watson. The High Performance Storage System. In *Proceedings of Supercomputing '93*, November 1993.

[2] S. Fineberg, P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO—A portable implementation of MPI-IO. In *Frontiers '96, the Sixth Symposium on the Frontiers of Massively Parallel Computation*, October 1996.

[3] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, Mass., 1994.

[4] IBM Corporation. IBM's MCA High Performance Parallel Interface (HIPPI). http://www.austin.ibm.com/hardware/adapters/hippi.html, February 1998.

[5] T. Jones, R. Mark, J. Martin, J. May, E. Pierce, and L. Stanberry. An MPI-IO interface to HPSS. In *Proceedings of the Fifth NASA/Goddard Conference on Mass Storage Systems*, September 1996.

[6] D. Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[7] W. Krotz-Vogel. private communication.

[8] MAXSTRAT Corporation. Gen5 XLE Product Overview. http://www.maxstrat.com/product_1.html, 1998.

[9] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org, July 1997.

[10] J. Sturtevant, M. Christon, P. Heerman, and P.-C. Chen. PDS/PIO: Lightweight libraries for collective parallel I/O. In *Proceedings of Supercomputing '98*, November 1998.

[11] Sun Microsystems Computer Company, Palo Alto, CA. *Sun MPI 3.0 Guide*, November 1997.

[12] D. Teaff, R. W. Watson, and R. A. Coyne. The architecture of the High Performance Storage System (HPSS). In *Proceedings of the Goddard Conference on Mass Storage and Technologies*, March 1995.

[13] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

[14] R. Watson and R. Coyne. The parallel I/O architecture of the High Performance Storage System (HPSS). In *IEEE Symposium on Mass Storage Systems*, September 1995.