Exploiting Shared Memory to Improve Parallel I/O Performance

Andrew B. Hastings[1] and Alok Choudhary[2]

[1] Sun Microsystems, Inc.
andrew.hastings@sun.com
[2] Northwestern University
choudhar@ece.northwestern.edu

Abstract. We explore several methods utilizing system-wide shared memory to improve the performance of MPI-IO, particularly for non-contiguous file access. We introduce an abstraction called the *datatype iterator* that permits efficient, dynamic generation of (offset, length) pairs for a given MPI derived datatype. Combining datatype iterators with overlapped I/O and computation, we demonstrate how a shared memory MPI implementation can utilize more than 90% of the available disk bandwidth (in some cases representing a 5× performance improvement over existing methods) even for extreme cases of non-contiguous datatypes. We generalize our results to suggest possible parallel I/O performance improvements on systems without global shared memory.

Keywords: Parallel I/O, shared memory, datatype iterator, non-contiguous access, MPI-IO.

1 Introduction

The rich MPI derived datatype facility can describe arbitrary regions of in-memory and in-file data. Via this facility, an application using MPI-IO may issue I/O operations that are non-contiguous in memory and/or in a file[1]. Previous work has explored optimizing these operations via data sieving[2], the two-phase collective optimization[2], list I/O[3], and datatype I/O[4], primarily in the context of commodity clusters. The first two optimizations are broadly available through the open source ROMIO implementation distributed as part of MPICH2[5].

In a data sieving read, each process repeats this cycle: read the next large contiguous chunk of file data into a working buffer and extract the pieces needed by the MPI read operation. Each cycle typically transfers a subset of the data covered by the associated datatypes. Writes are similar, except locks serialize access to each chunk, and a read-modify-write may be necessary for each chunk. (ROMIO uses data sieving for non-interleaved collective I/O operations.)

In the two-phase collective optimization, a subset of processes are designated *aggregators*. All processes construct lists of (offset, length) pairs (*flattening* their memory and file datatypes), and send flattened file datatypes to the aggregators.

For each chunk all processes use flattened memory datatypes to transfer an appropriate subset of their data to or from the aggregators via MPI messages, and the aggregrators use the flattened file datatypes to transfer the data to or from the working buffer. The messages serve not only to transfer the data, but also to synchronize the processes and ensure that I/O is complete before accessing the working buffer.

In list I/O, the MPI-IO implementation flattens the memory and file datatypes to lists of (offset, length) pairs. These lists are then communicated (concurrently with the data on a write) to a new list I/O interface in the filesystem, where it may apply techniques such as data sieving to optimize the operation.

Since the memory and file lists can be quite long for non-contiguous datatypes, datatype I/O replaces the lists with two compact datatype representations extracted from the MPI derived datatypes specified by the MPI-IO call. As in list I/O, these compact representations are communicated (concurrently with the data on a write) to a new datatype I/O filesystem interface.

In this work we investigate algorithms for optimizing MPI-IO in the context of a shared memory computer. As participants in the DARPA High Productivity Computer Systems initiative[6], researchers at Sun Microsystems, Inc., have been exploring the use of shared memory in petascale computer systems[7]. Exploiting shared memory in the MPI-IO implementation offers an opportunity to improve I/O performance without altering how applications express I/O operations.

2 Exploiting Shared Memory

In a global shared memory system the filesystem typically has direct access to both user memory and I/O devices, and it is efficient to transfer data independently from control information. Further, the data transfer step in the two-phase collective optimization can be performed via shared memory, bypassing the packing, copying, and unpacking operations usually required to implement MPI messages. We extended ROMIO with new I/O methods that exploit shared memory.

mmap. In a shared memory system it is relatively efficient to use the POSIX *mmap* operation to map a file directly into the address space of several processes. In contrast, mmap on a cluster might require additional bookkeeping and data transfer overhead to provide distributed shared memory. In the mmap I/O method, processes must synchronize initially to compute and set the new file length, but then can proceed independently using a data sieving-like algorithm: map a file chunk, then use the flattened datatypes to copy data into or out of the mapped chunk. The advantage over data sieving is that pages of the file are shared in memory; thus I/O transfers happen only once. The disadvantage is that memory management hardware limitations require existing file contents to be read before each write operation, even when overwriting an entire page.

Collective shared data. In the ROMIO two-phase collective, processes exchange data with aggregators via MPI messages. In the *collective shared data* method, we arrange for aggregators to have direct access to every process's address space.[1] Each aggregator copies data between its working buffer and appropriate application memory locations in other processes without costly MPI messages. The locations in application memory and the working buffer are identified via the flattened datatypes. In contrast to the message-based collective, the processes need only synchronize at the very start and very end of the MPI-IO operation, no matter how many cycles through the working buffer are needed to complete the operation.

Collective shared buffer with flattened datatypes. In the *collective shared buffer* methods, we arrange for each process to have direct access to every aggregator's working buffer. Each process uses the flattened datatypes to copy its own application data to or from working buffers in the appropriate aggregators. With a single working buffer per aggregator it is necessary for all processes to synchronize before beginning their copy operations (to wait for the read or write of the previous chunk to complete) and after their copy completes (to notify the aggregators of copy completion and that it is safe to initiate the read or write of the next chunk). We eliminated one synchronization step by splitting each working buffer into multiple sub-buffers and performing I/O asynchronously. On each cycle of a write operation, for example, the aggregators: (1) wait for I/O to complete on the next sub-buffer, (2) synchronize with all processes, and (3) initiate I/O on the just-completed sub-buffer. We measured a 40-90% performance improvement for our collective shared buffer algorithms on the FLASH I/O benchmark (Section 3.3) by enabling sub-buffering.

Collective shared buffer with dynamic offset/length generation. This method replaces the flattened datatypes with dynamic (offset, length) generation. A problem with flattening is that the entire list must be generated before any actual I/O can begin. Also, the flattened list may be large and thus compete for space in the processor cache with the application data being transferred.

We introduced a new abstract data type called a *datatype iterator*, representing a cursor into a specific MPI datatype. The function `dtc_next` advances the cursor to the next contiguous block in the associated datatype and returns the (offset, length) for that block. `dtc_extent_tell` and `dtc_size_tell` return the extent or size within the datatype corresponding to the current cursor position. `dtc_extent_seek` and `dtc_size_seek` position the cursor to a specific extent or size within the datatype.

The datatype iterator concept is similar in some ways to the *segments* used to transfer a datatype subset (*partial processing*) in MPICH2's dataloops, although segments seem not to have been applied to MPI-IO[8]. Datatype iterators are also similar to the flattening-on-the-fly technique (like dataloops but with added optimizations useful for vector processors) of listless I/O, which specializes the MPI pack/unpack interfaces to perform partial processing[9]. However, our datatype iterator interface appears to be unique: it allows a data transfer where

[1] Our experimental implementation uses *shmget*/*shmat* to attach a single shared memory region to every process. Each process (or aggregator, for collective shared buffer) allocates its application data (or working buffer) in a contiguous subset of the region.

both source and destination buffers are non-contiguous, and factors out separate *seek* and *tell* operations while still supporting partial processing.

The key data structure in the datatype iterator implementation is a stack with depth equal to the maximum nesting level of the derived datatype. Each stack element tracks the current position within the corresponding nested derived datatype. The basic algorithm for advancing the cursor descends the derived datatype tree to look ahead to the next contiguous block of bytes. If the lookahead is contiguous to the current accumulated block, add it and continue; otherwise, remember the lookahead for the next call and return the accumulated block.

The main processing loop for the collective shared buffer with dynamic offset/length generation algorithm is similar to that with flattened datatypes (including use of asynchronous I/O). However, no flattening is necessary before starting the main loop; instead, each process constructs datatype iterators for its own file and memory datatypes. The core of the "copy data" step for a write operation (the pseudocode below) demonstrates the power of datatype iterators. The code copies data directly from the (possibly non-contiguous) application buffer to the (possibly non-contiguous) destination locations in the shared working buffer without the need to pack and/or unpack data in an intermediate buffer (in contrast to the *direct_pack_ff* technique of [10]). Further, a contiguous datatype is not a special case: the code works efficiently for both contiguous and non-contiguous datatypes.

```
while (file_off + file_len <= end_off) {
                 // Entire file block still fits in current chunk
  while (file_len >= mem_len) {   // Mem block fits in file block
    src = app_buf + mem_off;
    memcpy(dest, src, mem_len);       // Copy remaining mem block
    file_off += mem_len;
    file_len -= mem_len;
    dest += mem_len;
    (mem_off, mem_len) = dtc_next(mem_dtc);     // Next mem block
  }
  while (mem_len >= file_len) {   // File block fits in mem block
    dest = temp_buf + file_off - start_off;
    memcpy(dest, src, file_len);     // Copy remaining file block
    mem_off += file_len;
    mem_len -= file_len;
    src += file_len;
    (file_off, file_len) = dtc_next(file_dtc); // Next file block
    if (file_off + file_len > end_off)
      break;
  }
}          // Elided: post-loop handling of tail end of file block
```

Another illustrative paradigm is the pseudocode to position the memory and file datatype cursors to match the start of the current file chunk:

```
  dtc_extent_seek(file_dtc, start_off);
  file_size = dtc_size_tell(file_dtc);
  dtc_size_seek(mem_dtc, file_size);
```

The use of datatype iterators considerably simplifies the implementation of the two-phase collective I/O method. Using lines of code as a proxy for complexity, we compared our two collective shared-buffer implementations. The list-based routine required 952 lines (with 1210 lines of supporting functions), while the datatype iterator-based routine required 358 lines (with 617 lines of supporting functions, including the datatype iterator implementation), an overall savings of 62% for datatype iterators. It seems reasonable to expect similar savings for a non-shared-memory-based two-phase collective algorithm.

3 Performance Evaluation

To evaluate the performance of our new shared memory-based I/O methods against other existing methods, we ran three MPI-IO benchmarks. Our benchmark hardware is a Sun Fire^{TM} 6800 server with 24 processors at 1.2 GHz and 96 GBytes of RAM. Four Sun StorEdge^{TM} T3 disk arrays are connected via four dedicated 1 Gbit Fibrechannel host adapters. We used the Sun StorageTek^{TM} QFS 4.5 filesystem[11] and the Solaris^{TM} 9 operating system. The filesystem is configured with metadata on one disk array and data striped across the remaining three disk arrays using a 512 MByte disk allocation unit per array. Aggregate peak read or write bandwidth to the three data arrays does not exceed 300 MBytes/second. Some of the benchmark problem sizes are small enough to fit within the RAM cache of the disk arrays, so we explicitly disabled this cache for a fairer comparison to larger problems that do not fit in cache.

QFS offers both *buffered* I/O (caches file blocks in system memory, then copies or maps them to/from user memory) and *direct* I/O (host adapter copies file blocks directly between user space and disk array). Direct I/O usually delivers higher performance than buffered for writes and for reads of data not already present in buffer cache (lower bookkeeping overhead and one fewer copy operation) but requires user code to align file offsets. List I/O, datatype I/O, and mmap are by their nature restricted to buffered I/O. As a baseline, we measured both buffered and direct I/O results for data sieving and the ROMIO two-phase collective I/O methods, but only direct I/O for the remaining, higher-performing methods. QFS does not support datatype I/O; results for other buffered I/O methods suggest datatype I/O would have similar performance to those methods. In some cases we omit list I/O or data sieving results because their performance was so poor that the corresponding runs took too long to complete.

We required an MPI implementation that included ROMIO's implementation of MPI-IO and supported a shared memory transport on our test platform. LAM 7.1.1[12] seemed to be the only available implementation meeting both requirements at the time of our experiments. We implemented datatype iterators directly inside LAM, avoiding the public MPI interfaces for inspecting datatypes. We upgraded ROMIO to version 1.2.4 with additional flattening code from version 2005-06-09[5]. All code was compiled for a 64-bit execution model.

For each method, we picked one set of tuning parameters (primarily working buffer size, number of sub-buffers, and number of aggregators), chosen to obtain the best results across the selected range of problem types and sizes. The collective shared buffer with dynamic generation method required the least aggregate working buffer space of the collective methods and seemed least sensitive to parameter changes.

Each reported result is the average over three runs. Time constraints prevented us from flushing the filesystem buffer cache between runs, and one benchmark pre-reads file contents into the buffer cache before beginning measurements. Therefore, the results for I/O methods utilizing buffered reads include time to access the buffer cache but not time to transfer data from disk.

3.1 ROMIO 3D Block Test

The ROMIO 3D block test (coll_perf.c), included in the ROMIO test suite, measures bandwidth to a 600×600×600 array of integers stored in an 824 MByte file. Each process uses a contiguous memory datatype, but the portion of the array file accessed by each process is determined by a block distribution (MPI_DISTRIBUTE_BLOCK).

Figure 1 shows our results. When the number of processes is not an integer's cube, data is distributed unevenly among the processes, accounting for several zig-zags in the graphs. As expected, buffered methods outperform direct on reads, but suffer from cache management overhead and extra copying on writes. Among direct methods, the collective shared buffer with dynamic generation method achieves the best read and write performance with little sensitivity to the uneven data distribution. Data sieving has the poorest direct I/O performance; repeated access to the same disk block causes extra disk seeks and must be serialized for writes.

3.2 Tile Reader Benchmark

The tile reader benchmark[13] implements tiled access to a two-dimensional dense dataset. A *tile* represents an individual display unit; displays are arranged in an array to collectively present a large image to a human viewer. Each tile is 1024×768 pixels with 24 bits per pixel; the tiles overlap by 128 pixels vertically and 270 pixels horizontally to improve edge merging. Each process reads its corresponding tile from the file to a contiguous memory buffer.

Figure 2 presents results for array sizes from 2×2 to 6×4 with corresponding file sizes 7 to 37 MBytes. (The number of processes is the product of the two dimensions.) Buffered methods again benefit from a warm filesystem buffer cache to outperform direct methods. Among direct methods, the collective shared buffer with dynamic generation method consistently outperforms the other direct methods (even on small problems), and on larger problems achieves over

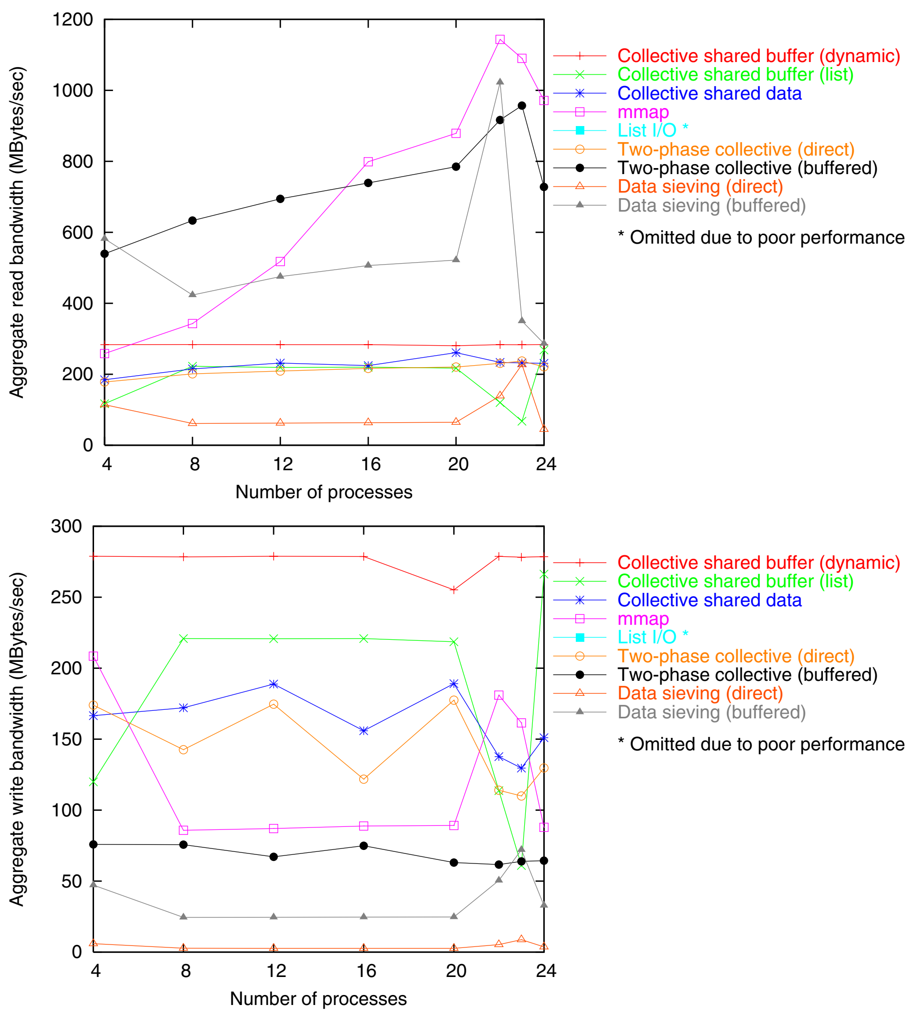


Fig. 1. ROMIO 3D block test performance results

100% of the available bandwidth.[2] Data sieving lags in performance due to repeated reads of the same disk block.

3.3 FLASH I/O Benchmark

The Argonne/Northwestern FLASH I/O benchmark (derived from the FLASH adaptive mesh refinement application[14]) substitutes synthetic data for the original computation, but makes the identical sequence of MPI-IO calls. Each

[2] How is this possible? The collective I/O methods read the overlapping data regions from the file only once, yet the benchmark counts the overlapping regions multiple times: $aggregate\_bytes\_read = number\_of\_processes \times data\_per\_process$.

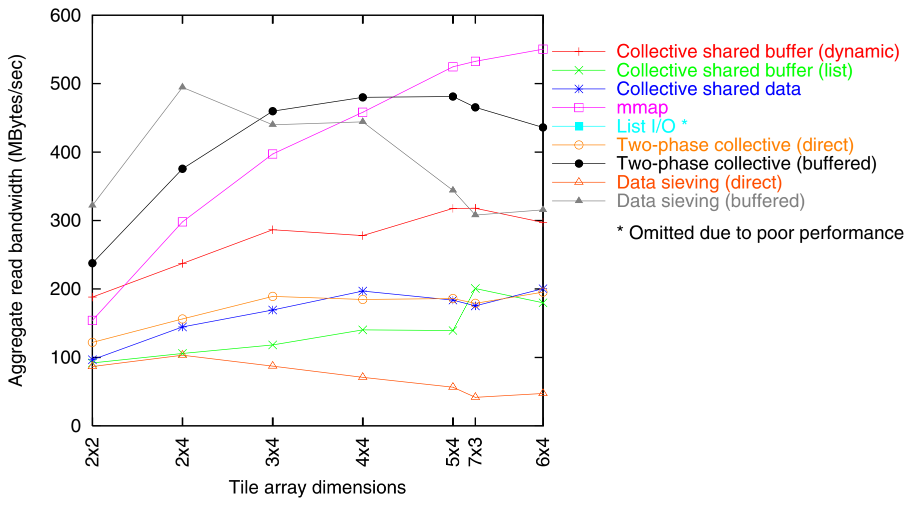


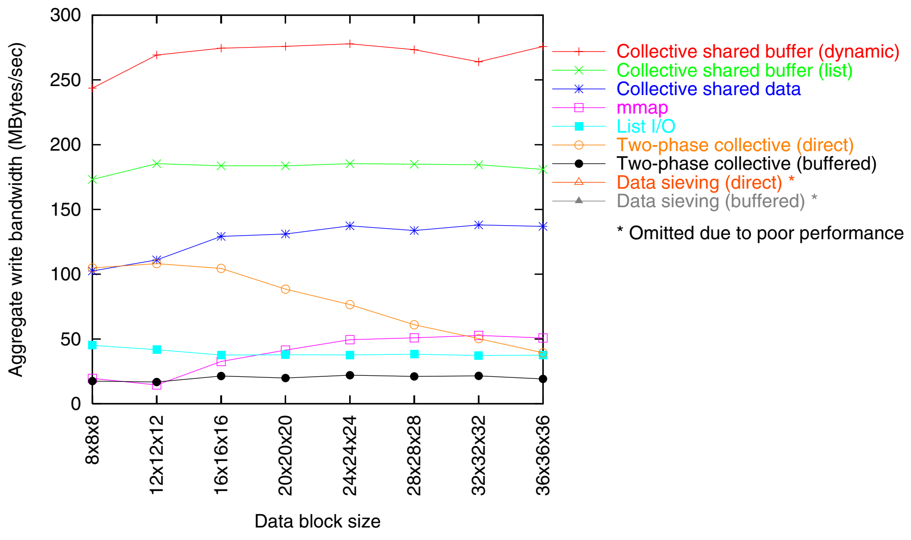
Fig. 2. Tile reader performance results

process contains 80 blocks. Each block is a three-dimensional array of data elements with each surface extended by four ghost cells. Each data element contains 24 variables. Data elements (but not ghost cells) are checkpointed to a file. In the file, data is rearranged so all values of variable 0 are stored first, then variable 1, and so on. Both file and memory datatypes are non-contiguous; each value is 8 bytes and is not contiguous in memory with other values of the same variable. (For our largest problem size the list-based I/O methods use an aggregate $O(10^9)$ list entries requiring twice the memory of the data they describe.)

We explored scalability along two dimensions. The top graph in Figure 3 reports results for a fixed number of processes (22) but a varying block size; file sizes range from 165 MBytes to 15 GBytes. The bottom graph reports results for a fixed block size ($20\times20\times20$) but a varying number of processes; file sizes range from 469 MBytes to 2.8 GBytes. The graphs show the collective shared buffer with dynamic generation method scaling well on both dimensions and providing the best performance. For the larger process counts it achieves over 90% of the available disk bandwidth, reflecting a $5\times$ improvement over the best existing method (two-phase collective). The buffered methods are limited by cache management overhead and extra copying and have the poorest performance.

4 Conclusion

We explored several new methods to improve MPI-IO in a shared memory computer system. A method that utilizes a shared working buffer, a single aggregator, overlaps I/O and computation via a generalized double-buffering scheme, and reduces startup cost to generate (offset, length) pairs dynamically offered the best aggregate performance for several application I/O patterns.



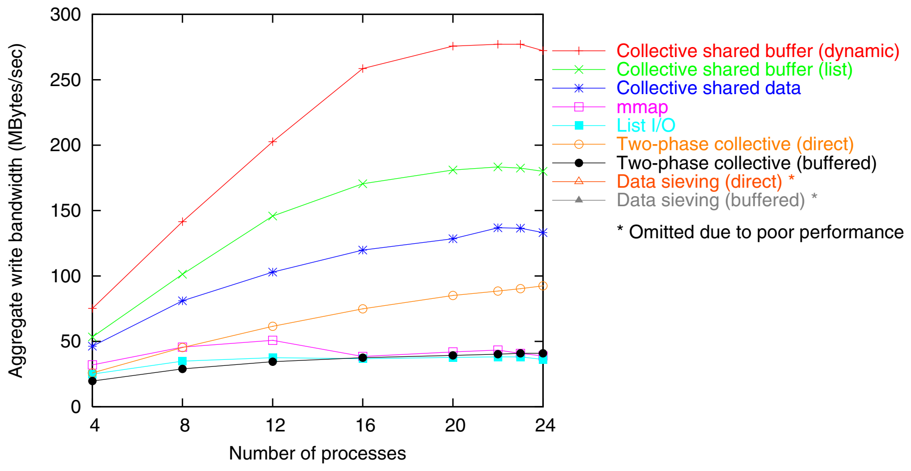


Fig. 3. FLASH I/O performance results: 22 processes (top), 20×20×20 block (bottom)

More generally, we rediscovered two important principles for obtaining good streaming I/O performance: (1) Reduce startup overhead and begin I/O early. (2) Overlap I/O and computation whenever possible. Our new abstraction, the datatype iterator, follows (1): initialization is cheap in contrast to the potentially high cost to generate (offset, length) lists. The sub-buffering mechanism we used in our collective shared buffer I/O methods follows (2).

We utilized the datatype iterator only in a shared memory system. Since the ROMIO two-phase collective uses lists extensively, and our research shows that datatype iterators in conjunction with overlapped I/O and computation can produce better performance with fewer lines of code than lists, an interesting area for future work would be the use of datatype iterators in traditional clusters.

Acknowledgements

Harriet Coverston and Anton Rang also contributed to this work. This material is based on work supported by the US Defense Advanced Research Projects Agency under Contract No. NBCH3039002. Sun, StorageTek, Sun Fire, Sun StorEdge, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc.

References

1. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI – The Complete Reference*, vol. 2. MIT Press, 1998.
2. 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*, pp. 182–189, IEEE Computer Society Press, February 1999.
3. R. Thakur, W. Gropp, and E. Lusk, “On implementing MPI-IO portably and with high performance,” in *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pp. 23–32, May 1999.
4. A. Ching, A. Choudhary, W.-K. Liao, R. Ross, and W. Gropp, “Efficient structured data access in parallel file systems,” in *Proceedings of the IEEE International Conference on Cluster Computing*, pp. 326–335, IEEE Computer Society Press, December 2003.
5. “MPICH2 home page,” August 2005. `http://www.mcs.anl.gov/mpi/mpich2/`.
6. “HPCS – High Productivity Computer Systems,” April 2006. `http://www.highproductivity.org/`.
7. M. Vildibill, “Sun’s Hero program: Changing the productivity game,” April 2006. `http://www.hpcwire.com/hpc/614805.html`.
8. R. Ross, N. Miller, and W. Gropp, “Implementing fast and reusable datatype processing,” in *Proceedings of the 10th European PVM/MPI Users Group Meeting*, pp. 404–413, Springer-Verlag, October 2003.
9. J. Worringen, J. L. Träff, and H. Ritzdorf, “Fast parallel non-contiguous file access,” in *Proceedings of SC2003: High Performance Networking and Computing*, IEEE Computer Society Press, November 2003.
10. J. Worringen, A. Gäer, and F. Reker, “Exploiting transparent remote memory access for non-contiguous- and one-sided-communication,” in *Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 163–172, IEEE Computer Society Press, April 2002.
11. “Sun StorageTek QFS software,” April 2006. `http://www.sun.com/storagetek/management_software/data_management/qfs/`.
12. “LAM/MPI parallel computing,” April 2005. `http://www.lam-mpi.org/`.
13. R. Ross, “Parallel I/O benchmarking consortium,” August 2005. `http://www.mcs.anl.gov/~rross/pio-benchmark/`.
14. “ASC center for astrophysical thermonuclear flashes,” April 2006. `http://flash.uchicago.edu/`.