Implementing Byte-Range Locks Using MPI One-Sided Communication

Rajeev Thakur, Robert Ross, and Robert Latham

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{thakur, rross, robl}@mcs.anl.gov

Abstract. We present an algorithm for implementing byte-range locks using MPI passive-target one-sided communication. This algorithm is useful in any scenario in which multiple processes of a parallel program need to acquire exclusive access to a range of bytes. One application of this algorithm is for implementing MPI-IO's atomic-access mode in the absence of atomicity guarantees from the underlying file system. Another application is for implementing data sieving, a technique for optimizing noncontiguous writes by doing an atomic read-modify-write of a large, contiguous block of data. This byte-range locking algorithm can be used instead of POSIX `fcntl` file locks on file systems that do not support `fcntl` locks, on file systems where `fcntl` locks are unreliable, and on file systems where `fcntl` locks perform poorly. Our performance results demonstrate that the algorithm has low overhead and significantly outperforms `fcntl` locks on NFS file systems on a Linux cluster and on a Sun SMP.

1 Introduction

Often, processes must acquire exclusive access to a range of bytes. One application of byte-range locks is to implement the atomic mode of access defined in MPI-IO, the I/O interface that is part of MPI-2 [7]. MPI-IO, by default, supports weak consistency semantics in which the outcome of concurrent overlapping writes from multiple processes to a common file is undefined. The user, however, can optionally select stronger consistency semantics on a per file basis by calling the function `MPI_File_set_atomicity` with `flag=true`. In this mode, called the atomic mode, if two processes associated with the same open file write concurrently to overlapping regions of the file, the result is the data written by either one process or the other, and nothing in between.

In order to implement the atomic mode, either the underlying file system must provide functions that guarantee atomicity, or the MPI-IO implementation must ensure that a process has exclusive access to the portion of the file it needs to access [13]. Many POSIX-compatible file systems support atomicity for contiguous reads and writes, such as those issued by a single `read` or `write` function call, but some high-performance parallel file systems, such as PVFS [1] and PVFS2 [9], do not. MPI-IO's atomic mode supports atomicity even for noncontiguous file accesses that are made with a single MPI function call by using noncontiguous file views. No file system supports atomicity for noncontiguous reads and writes. For such accesses, the MPI-IO implementation must explicitly acquire exclusive access to the byte range being read or written by a process.

Another use of byte-range locks is to implement data sieving [14]. Data sieving is a technique for optimizing noncontiguous accesses. For reading, it involves reading a large chunk of data and extracting the necessary pieces from it. For writing, the process must read the large chunk of data from the file into a temporary buffer, copy the necessary pieces into the buffer, and then write it back. This read-modify-write must be done atomically to prevent other processes from writing to the same region of the file while the buffer is being modified in memory. Therefore, the process must acquire exclusive access to the range of bytes before doing the read-modify-write.

POSIX defines a function `fcntl` by which processes can acquire byte-range locks on an open file [5]. However, many file systems, such as PVFS [1], PVFS2 [9], some installations of NFS, and various research file systems [2, 4, 8], do not support `fcntl` locks. On some file systems, for example, some installations of NFS, `fcntl` locks are not reliable. In addition, on some file systems, the performance of `fcntl` locks is poor. Therefore, one cannot rely solely on `fcntl` for file locking.

In this paper, we present an algorithm for implementing byte-range locks that can be used instead of `fcntl`. This algorithm extends an algorithm we described in an earlier work [12] for acquiring exclusive access to an entire file (not a range of bytes). Both algorithms have some similarities with the MCS lock [6], an algorithm devised for efficient mutex locks in shared-memory systems, but differ from it in that they use MPI one-sided communication (which does not have atomic read-modify-write operations) and can be used on both distributed- and shared-memory systems. Byte-range locks add significant complications to the algorithm for exclusive access to an entire file [12], which is essentially just a mutex. Byte-range locks are an important improvement because they enable multiple processes to perform I/O concurrently to nonoverlapping regions of the file, a feature whole-file locks preclude.

The rest of this paper is organized as follows. In Section 2, we give a brief overview of MPI one-sided communication, particularly those aspects used in our algorithm. In Section 3, we describe the byte-range locking algorithm. In Section 4, we present performance results. In Section 5, we conclude with a brief discussion of future work.

2 One-Sided Communication in MPI

To enable one-sided communication in MPI, a process must first specify a contiguous memory region, called a *window*, that it wishes to expose to other processes for direct one-sided access. Each process in the communicator must call the function `MPI_Win_create` with the starting address of the local memory window, which could be `NULL` if the process has no memory to expose to one-sided

Process 0	*Process 1*	*Process 2*
`MPI_Win_create(&win)`	`MPI_Win_create(&win)`	`MPI_Win_create(&win)`
`MPI_Win_lock(shared,1)`		`MPI_Win_lock(shared,1)`
`MPI_Put(1)`		`MPI_Put(1)`
`MPI_Get(1)`		`MPI_Get(1)`
`MPI_Win_unlock(1)`		`MPI_Win_unlock(1)`
`MPI_Win_free(&win)`	`MPI_Win_free(&win)`	`MPI_Win_free(&win)`

Fig. 1. An example of MPI one-sided communication with passive-target synchronization. Processes 0 and 2 perform one-sided communication on the window memory of process 1 by requesting shared access to the window. The numerical arguments indicate the target rank.

communication. `MPI_Win_create` returns an opaque object, called a *window object*, which is used in subsequent one-sided communication functions.

Three one-sided data-transfer functions are provided: `MPI_Put` (remote write), `MPI_Get` (remote read), and `MPI_Accumulate` (remote update). In addition, some mechanism is needed for a process to indicate when its window is ready to be accessed by other processes and to specify when one-sided communication has completed. For this purpose, MPI defines three synchronization mechanisms. The first two synchronization mechanisms require both the origin and target processes to call synchronization functions and are therefore called *active-target synchronization*. The third mechanism requires no participation from the target and is therefore called *passive-target synchronization*. We use this method in our byte-range locking algorithm because a process must be able to acquire a lock independent of any other process.

2.1 Passive-Target Synchronization

In passive-target synchronization, the origin process begins a synchronization epoch by calling `MPI_Win_lock` with the rank of the target process and indicating whether it wants shared or exclusive access to the window on the target. After issuing the one-sided operations, it calls `MPI_Win_unlock`, which ends the synchronization epoch. The target does not make any synchronization call. When `MPI_Win_unlock` returns, the one-sided operations are guaranteed to be completed at the origin and the target. Figure 1 shows an example of one-sided communication with passive-target synchronization.

An implementation is allowed to restrict the use of this synchronization method to window memory allocated with `MPI_Alloc_mem`. `MPI_Win_lock` is *not* required to block until the lock is acquired, except when the origin and target are one and the same process. In other words, `MPI_Win_lock` does not establish a critical section of code; it ensures only that the one-sided operations issued between the lock and unlock will be executed on the target window in a shared or exclusive manner (as requested) with respect to the one-sided operations from other processes.

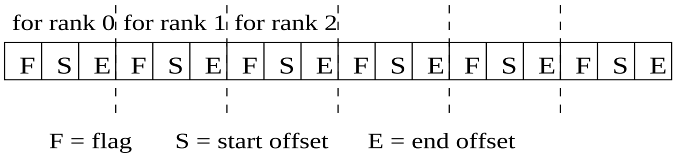

Fig. 2. Window layout for the byte-range locking algorithm

2.2 Completion and Ordering

MPI puts, gets, and accumulates are nonblocking operations, and an implementation is allowed to reorder them within a synchronization epoch. They are guaranteed to be completed, both locally and remotely, only when the synchronization epoch has ended. In other words, a get operation is not guaranteed to see the data that was written by a put issued before it in the same synchronization epoch. Consequently, it is difficult to implement an atomic read-modify-write operation by using MPI one-sided communication [3]. One cannot simply do a lock-get-modify-put-unlock because the data from the get is not available until after the unlock. In fact, the MPI Standard defines such an operation to be erroneous (doing a put and a get to the same location in the window in the same synchronization epoch). One also cannot do a lock-get-unlock, modify the data, and then do a lock-put-unlock because the read-modify-write is no longer atomic. This feature of MPI complicates the design of a byte-range locking algorithm.

3 Byte-Range Locking Algorithm

In this section, we describe the design of the byte-range locking algorithm together with snippets of the code for acquiring and releasing a lock.

3.1 Window Layout

The window memory for the byte-range locking algorithm is allocated on any one process—in our prototype implementation, on rank 0. Other processes pass `NULL` to `MPI_Win_create`. All processes needing to acquire locks access this window by using passive-target one-sided communication. The window comprises three values for each process, ordered by process rank, as shown in Figure 2. The three values are a flag, the start offset for the byte-range lock, and the end offset. In our implementation, for simplicity, all three values are represented as integers. The window size, therefore, is `3 * sizeof(int) * nprocs`. In practice, the flag could be a single byte, and the start and end offsets may each need to be eight bytes to support large file sizes.

3.2 Acquiring the Lock

The algorithm for acquiring a lock is as follows. The pseudocode is shown in Figure 3. The process wanting to acquire a lock calls `MPI_Win_lock` with the

```
Lock_acquire(int start, int end)
{
    val[0] = 1; /* flag */  val[1] = start; val[2] = end;

    while (1) {
        /* add self to locklist */
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
        MPI_Put(&val, 3, MPI_INT, homerank, 3*(myrank), 3, MPI_INT, lockwin);
        MPI_Get(locklistcopy, 3*(nprocs-1), MPI_INT, homerank, 0, 1, locktype1,
                lockwin);
        MPI_Win_unlock(homerank, lockwin);

        /* check to see if lock is already held */
        conflict = 0;
        for (i=0; i < (nprocs - 1); i++) {
            if ((flag == 1) && (byte ranges conflict with lock request)) {
                conflict = 1; break;
            }
        }

        if (conflict == 1) {
            /* reset flag to 0, wait for notification, and then retry the lock */
            MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
            val[0] = 0;
            MPI_Put(val, 1, MPI_INT, homerank, 3*(myrank), 1, MPI_INT, lockwin);
            MPI_Win_unlock(homerank, lockwin);

            /* wait for notification from some other process */
            MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, WAKEUP, comm,
                     MPI_STATUS_IGNORE);
            /* retry the lock */
        }
        else {
            /* lock is acquired */
            break;
        }
    }
}
```

Fig. 3. Pseudocode for obtaining a byte-range lock. The derived datatype `locktype1` is created at lock-creation time and cached in the implementation.

`lock_type` as `MPI_LOCK_EXCLUSIVE`, followed by an `MPI_Put`, an `MPI_Get`, and then `MPI_Win_unlock`. With the `MPI_Put`, the process sets its own three values in the window: It sets the flag to 1 and the start and end offsets to those needed for the lock. With the `MPI_Get`, it gets the three values for all other processes (excluding its own values) by using a suitably constructed derived datatype, for example, an indexed type with two blocks. After `MPI_Win_unlock` returns, the process goes through the list of values returned by `MPI_Get`. For all other processes, it first checks whether the flag is 1 and, if so, checks whether there is a conflict between that process's byte-range lock and the lock it wants to acquire. If there is no such conflict with any other process, it considers the lock acquired. If a conflict (flag and byte range) exists with any process, it considers the lock as not acquired.

If the lock is not acquired, the process resets its flag in the window to 0 by doing an `MPI_Win_lock`–`MPI_Put`–`MPI_Win_unlock` and leaves its start and end offsets in the window unchanged. It then calls a zero-byte `MPI_Recv` with

```
Lock_release(int start, int end)
{
    val[0] = 0; val[1] = -1; val[2] = -1;

    /* set start and end offsets to -1, flag to 0, and get everyone else's status */
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
    MPI_Put(val, 3, MPI_INT, homerank, 3*(myrank), 3, MPI_INT, lockwin);
    MPI_Get(locklistcopy, 3*(nprocs-1), MPI_INT, homerank, 0, 1, locktype2,
            lockwin);
    MPI_Win_unlock(homerank, lockwin);

    /* check if anyone is waiting for a conflicting lock. If so, send them a
       0-byte message, in response to which they will retry the lock. For
       fairness, we start with the rank after ours and look in order. */

    i = myrank;  /* ranks are off by 1 because of the derived datatype */
    while (i < (nprocs - 1)) {
        /* the flag doesn't matter here. check only the byte ranges */
        if (byte ranges conflict) MPI_Send(NULL, 0, MPI_BYTE, i+1, WAKEUP, comm);
        i++;
    }
    i = 0;
    while (i < myrank) {
        if (byte ranges conflict) MPI_Send(NULL, 0, MPI_BYTE, i, WAKEUP, comm);
        i++;
    }
}
```

Fig. 4. Pseudocode for releasing a byte-range lock. The derived datatype `locktype2` is created at lock-creation time and cached in the implementation.

`MPI_ANY_SOURCE` as the source and blocks until it receives such a message from any other process (that currently has a lock; see the lock-release algorithm below). After receiving the message, it tries again to acquire the lock by using the above algorithm (further explained below).

3.3 Releasing the Lock

The algorithm for releasing a lock is as follows. The pseudocode is shown in Figure 4. The process wanting to release a lock calls `MPI_Win_lock` with the `lock_type` as `MPI_LOCK_EXCLUSIVE`, followed by an `MPI_Put`, an `MPI_Get`, and then `MPI_Win_unlock`. With the `MPI_Put`, the process resets its own three values in the window: It resets its flag to 0 and the start and end offsets to -1. With the `MPI_Get`, it gets the start and end offsets for all other processes (excluding its own values) by using a derived datatype. This derived datatype could be different from the one used for acquiring the lock because the flags are not needed. After `MPI_Win_unlock` returns, the process goes through the list of values returned by `MPI_Get`. For all other processes, it checks whether there is a conflict between the byte range set for that process and the lock it is releasing. The flag is ignored in this comparison. For fairness, it starts with the next higher rank after its own, wrapping back to rank 0 as necessary. If there is a conflict with the byte range set by another process—meaning that process is waiting to acquire a conflicting lock—it sends a 0-byte message to that process, in response to which that process will retry the lock. After it has gone through the entire list of values and sent 0-byte messages to all other processes waiting for a lock that conflicts with its own, the process returns.

3.4 Discussion

In order to acquire a lock, a process opportunistically sets its flag to 1, before knowing whether it has got the lock. If it determines that it does not have the lock, it resets its flag to 0 with a separate synchronization epoch. Had we chosen the opposite approach, that is, set the flag to 0 initially and then set it to 1 after determining that the lock has been acquired, there could have been a race condition because another process could attempt the same operation between the two distinct synchronization epochs. The lack of an atomic read-modify-write operation in MPI necessitates the approach we use.

When a process releases a byte-range lock, multiple processes waiting on a conflicting lock may now be able to acquire their lock, depending on the byte range being released and the byte ranges those processes are waiting for. In the lock-release algorithm, we use the conservative method of making the processes waiting on a conflicting lock retry their lock instead of having the releasing process hand the lock to the appropriate processes directly. The latter approach can get fairly complicated for byte-range locks, and in Section 5 we describe some optimizations that we plan to explore.

If processes are multithreaded, the current design of the algorithm requires that either the user must ensure that only one thread calls the lock acquisition and release functions (similar to `MPI_THREAD_SERIALIZED`), or the lock acquisition and release functions themselves must acquire and release a thread mutex lock. We plan to extend the algorithm to allow multiple threads of a process to acquire and release nonconflicting locks concurrently.

The performance of this algorithm depends on the quality of the implementation of passive-target one-sided communication in the MPI implementation. In particular, it depends on the ability of the implementation to make progress on passive-target one-sided communication without requiring the target process to call MPI functions for progress. On distributed-memory environments, it is also useful if the implementation can cache derived datatypes at the target, so that the derived datatypes need not be communicated to the target each time.

4 Performance Evaluation

To measure the performance of our algorithm, we wrote two test programs: one in which all processes try to acquire a conflicting lock (same byte range) and another in which all processes try to acquire nonconflicting locks (different byte ranges). In both tests, each process acquires and releases the lock in a loop several times. We measured the time taken by all processes to complete acquiring and releasing all their locks and divided this time by the number of processes times the number of iterations. This measurement gave the average time taken by a single process for acquiring and releasing a single lock. We compared the performance of our algorithm with that using `fcntl` locks. We ran the tests on a Myrinet-connected Linux cluster at Argonne and on a 24-CPU Sun SMP at the University of Aachen in Germany.

On the Linux cluster, we used a beta version of MPICH2 1.0.2 with the GASNET channel running over GM. To measure the performance of `fcntl` locks, we used an NFS file system that mounted a GFS [10] backend. This is the only way to use `fcntl` locks on this cluster; the parallel file system on the cluster, PVFS, does not support `fcntl` locks. On the Sun SMP, we could not use Sun MPI because of a bug in the implementation that caused one of our tests to hang when run with more than four processes. We instead used a beta version of MPICH2 1.0.2 with the sshm (scalable shared-memory) channel. For `fcntl` locks, we used an NFS file system. When we ran our test for conflicting locks with more than 16 processes, `fcntl` returned an error with `errno` set to "no record locks available." This is an example of the unreliability of `fcntl` locks with NFS, mentioned in Section 1.

On the Linux cluster, this version of MPICH2 has two limitations that can affect performance of the byte-range locking algorithm. One limitation is that MPICH2 requires the target process to call MPI functions in order to make progress on passive-target one-sided communication. This restriction did not affect our test programs because the target (rank 0) also tried to acquire byte-range locks and therefore made MPI function calls. Furthermore, all processes did an `MPI_Barrier` at the end, which also guaranteed progress at the target. The other limitation is that MPICH2 does not cache derived datatypes at the target process, so they need to be communicated each time. Both these limitations will be fixed in a future release of MPICH2. These limitations do not exist on the Sun SMP because when the window is allocated with `MPI_Alloc_mem`, the sshm channel in MPICH2 allocates the window in shared memory and implements puts and gets by directly copying data to/from the shared-memory window.

Figure 5 shows the average time taken by a single process to acquire and release a single lock on the Linux cluster and the Sun SMP. On the Linux cluster, for nonconflicting locks, our algorithm is on average about twice as fast as `fcntl`, and the time taken does not increase with the number of processes. For conflicting locks, the time taken by our algorithm increases with the number of processes because of the overhead induced by lock contention. In Section 5, we describe some optimizations we plan to incorporate that will reduce communication traffic in the case of conflicting locks and therefore improve performance and scalability. The graph for conflicting locks with `fcntl` on NFS on the Linux cluster is not shown because the time taken was on the order of seconds—about three orders of magnitude higher than any of the other results!

On the Sun SMP, for nonconflicting locks, our algorithm is about 10 times faster than `fcntl`, and the time taken does not increase with the number of processes. For conflicting locks, our algorithm is 5–9 times faster than `fcntl`. As mentioned above, `fcntl` on NFS for conflicting locks failed when run on more than 16 processes on the Sun SMP.

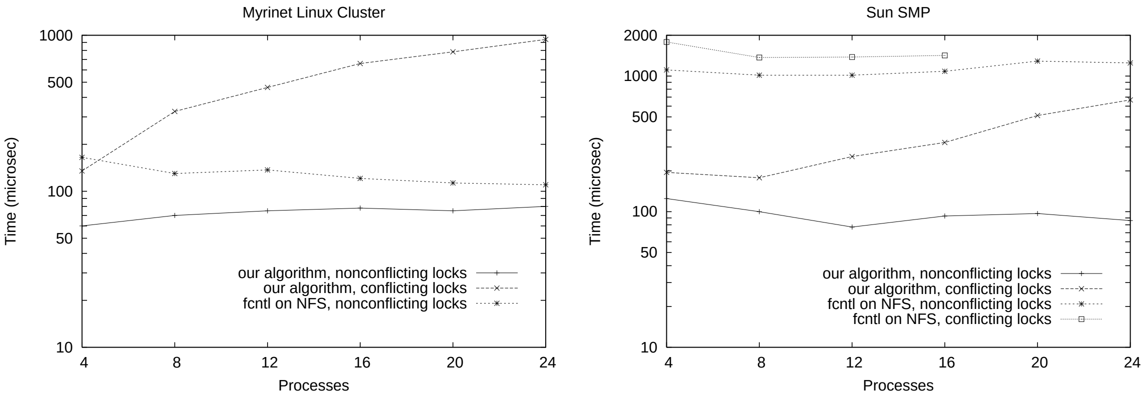

Fig. 5. Average time for acquiring and releasing a single lock on a Myrinet-connected Linux cluster (left) and a Sun SMP (right). The graph for conflicting locks with `fcntl` on the Linux cluster is not shown because the time was three orders of magnitude higher than the other results. On the Sun SMP, when run on more than 16 processes, `fcntl` on NFS for conflicting locks failed.

5 Conclusions and Future Work

We have presented an efficient algorithm for implementing byte-range locks using MPI one-sided communication. We have shown that our algorithm has low overhead and outperforms NFS `fcntl` file locks on the two environments we tested. We plan to use this algorithm for byte-range locking in our implementation of MPI-IO, called ROMIO [11].

This algorithm requires that the MPI implementation handle passive-target one-sided communication efficiently, which is not the case in many MPI implementations today. For example, with IBM MPI on the IBM SP at the San Diego Supercomputer Center, we observed wide fluctuations in the performance of our algorithm. We hope that such algorithms that demonstrate the usefulness of passive-target one-sided communication will spur MPI implementers to optimize their implementations.

While we have focused on making the algorithm correct and efficient, it can be improved in several ways. For example, in our current implementation of the lock-release algorithm, we make the processes waiting for conflicting locks retry their lock, instead of having the releasing process grant the locks directly. We chose this approach because, in general, the analysis required to determine which processes can be granted their locks is tricky. For special cases, however, such as processes waiting for a byte range that is a subset of the byte range being held by the releasing process, it is possible for the releasing process to grant the lock directly to the other process. With such an approach, the releasing process could grant the lock to processes for which it can easily determine that the lock can be granted, deny it to processes for which it can determine that the lock cannot be granted, and have others retry their lock. Preventing too many processes from retrying conflicting locks will improve the performance and scalability of the algorithm significantly. Another optimization for scalability is to replace the linear list of values with a tree-based structure, so that a process does not have to fetch and check the values of all other processes. We plan to explore such optimizations to the algorithm.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38. We thank Chris Bischof for giving us access to the Sun SMP machines at the University of Aachen.

References

1. Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta*, pages 317–327, October 2000.
2. Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.
3. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.
4. Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.
5. IEEE/ANSI Std. 1003.1. Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) [C Language], 1996 edition.
6. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 1991.
7. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.
8. Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.
9. PVFS2: Parallel virtual file system. `http://www.pvfs.org/pvfs2/`.
10. Red Hat Global File System. `http://www.redhat.com/software/rha/gfs`.
11. ROMIO: A high-performance, portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`.
12. Robert Ross, Robert Latham, William Gropp, Rajeev Thakur, and Brian Toonen. Implementing MPI-IO atomic mode without file system support. In *Proceedings of CCGrid 2005*, May 2005.
13. Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.
14. Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, January 2002.