Implementing MPI-IO Atomic Mode Without File System Support

Robert Ross Robert Latham William Gropp Rajeev Thakur Brian Toonen
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{rross,robl,gropp,thakur,toonen}@mcs.anl.gov

Abstract

The ROMIO implementation of the MPI-IO standard provides a portable infrastructure for use on top of any number of different underlying storage targets. These different targets vary widely in their capabilities, and in some cases, additional effort is needed within ROMIO to support the complete MPI-IO semantics. One aspect of the interface that can be problematic to implement is the MPI-IO atomic mode*. This mode requires enforcing strict consistency semantics. For some file systems, native locks may be used to enforce these semantics, but not all file systems have lock support. In this work, we describe two algorithms for implementing efficient mutex locks using MPI-1 and MPI-2 capabilities. We then show how these algorithms may be used to implement a portable MPI-IO atomic mode for ROMIO. We evaluate the performance of these algorithms and show that they impose little additional overhead on the system. Because of the low-overhead nature of these algorithms, they are likely useful in a variety of situations where distributed locks are needed in the MPI-2 environment.*

1 Introduction

MPI-IO [8] provides a standard interface for MPI programs to access storage in a coordinated manner. Implementations of MPI-IO, such as the portable ROMIO implementation [12] and the implementation for AIX GPFS [9] have aided in the widespread availability of MPI-IO. These implementations in particular include a collection of optimizations [11, 9, 6] that leverage MPI-IO features to obtain higher performance than would be possible with the less capable POSIX interface [5].

One component of the MPI-IO interface that has been difficult to implement portably is the *atomic mode*. This mode provides a more strict consistency semantic than the default MPI-IO mode or even POSIX I/O. Atomic mode is a very useful capability for applications and higher-level I/O components that need to share data through a file. One good example where atomic mode may be helpful is in HDF5, where internal data stored in the file is used by all processes to place application data in a consistent manner. In ROMIO the atomic mode is implemented through the use of file system locks where available. Unfortunately for file systems without locking systems, such as Lustre and PVFS2, atomic mode is not supported.

With the recent full implementation of MPI-2 one-sided operations in MPICH2 and other MPI packages, a new opportunity has arisen. By building up mutex locks from one-sided and point-to-point operations, we can implement atomic mode semantics without file system support. If this mutex lock can be operated on efficiently, it may be useful in other situations as well.

1.1 MPI-IO Atomic Mode

The MPI-IO atomic mode guarantees sequential consistency of writes to the same file by a group of processes who have previously collectively opened the file. It also guarantees that these writes will be immediately visible by other processes in this group. This semantic is primarily used for two purposes: simplifying communication through a shared file, and guaranteeing atomicity of writes to overlapping regions. The MPI-IO standard encourages applications to use the more relaxed default MPI-IO consistency semantics when peak performance is desired, as the MPI-IO implementation can more easily optimize the requests. Even though atomic mode might not be the fastest way to access the underlying file system, some programs need this capability, so it is important that we support the standard in its entirety where possible.

The ROMIO implementation builds MPI-IO on top of the I/O API supported by the underlying file system. For many file systems, this interface is POSIX. While the POSIX I/O `read`, `write`, `readv`, and `writev` calls also guarantee sequential consistency, they cannot describe all possible I/O operations through the MPI-IO interface,

particularly ones with noncontiguous data in file. The `lio_listio` function available as part of the POSIX real-time extensions is also inadequate because the list of operations are considered independent – there is no guarantee of atomicity with respect to the entire collection. Because of these characteristics, it is necessary to impose atomicity through additional means. For these file systems ROMIO uses `fcntl` locks, locking contiguous regions encompassing all the bytes that the process will access.

File systems such as PVFS [3] and PVFS2 do not guarantee atomicity of operations at all, instead relying on the MPI-IO layer to provide these guarantees. Other types of storage back-ends, such as GridFTP [1] and Logistical Networks [2] do not have locking capabilities either. In the existing ROMIO implementation atomic mode is simply not supported for these types of storage.

In order to implement atomic mode without file system support, we need to build a mechanism for coordinating access to a file, or regions of a file. Our approach is to provide a mutex lock for the entire file coupled with an efficient system for notifying subsequent processes on lock release. We will describe how we implement these capabilities in the following section.

2 Efficient, Scalable Mutex Locks with MPI

The MPI one-sided operations include both *active target* and *passive target* options[8]. Active target operations require that the process that owns the memory participate in the operation. These operations are not particularly useful in this context because our processes are performing independent operations; they do not know when other processes are acquiring or releasing locks. Passive target operations, on the other hand, do not require that the owner of the memory (the target) participate. These operations are ideal for our purposes.

Before MPI one-sided calls may be used, a collection of processes must first define a *window object*. This object contains a collection of memory *windows*, each associated with the rank of the process on which the memory resides. After defining the window object, MPI processes can then perform put, get, and accumulate operations into the memory windows of the other processes.

MPI passive target operations are organized into *access epochs* that are bracketed by `MPI_Win_lock` and `MPI_Win_unlock` calls. Clever MPI implementations [10] will combine all the data movement operations (puts, gets, and accumulates) into one network transaction that occurs at the unlock.

Implementing locks with MPI one-sided operations poses an interesting challenge: the standard does not define the traditional test-and-set and fetch-and-increment operations. In fact, no mechanism exists for both reading and writing a single memory region in an atomic manner in the MPI scheme. Two approaches are outlined in [4]. These approaches have some disadvantages, particularly in that they require many remote one-sided operations and poll on remote memory regions.

At a high level, our algorithm is simple. A process that wants to acquire the lock first adds itself to a list of processes waiting for the lock. If the process is the only one in the list, then it has acquired the lock. If not, it will wait for notification that the lock has been passed on to it. Processes releasing the lock are responsible for notifying the next waiting process (if any) at lock release time.

Both algorithms presented here were influenced by the MCS lock [7], an algorithm devised for efficient mutex locks in shared memory systems. The MCS lock has two characteristics that we mimic: spinning only on local memory regions, and $O(1)$ network transactions per lock acquisition. However, we are not able to meet their achievement of constant memory size per lock, mainly due to the constraint in MPI of not reading and writing to the same memory location in a single access epoch. Our use of MPI communication, and the approach we use for organizing memory windows, are unique to our algorithms.

2.1 Mutex Locks with One-Sided Operations

Our first algorithm uses only one access epoch to attempt to obtain the lock. In the presence of contention that access epoch is followed only by local polling. When unlocking, a single access epoch is used to release the lock, and if another process is waiting, a second access epoch notifies that process that it now holds the lock. We will call this approach the *one-sided algorithm*, because only one-sided operations are used in the algorithm.

The one-sided algorithm uses a pair of MPI window objects. The first, which we will call `waitlistwin`, consists of a single window of N bytes on a single process, one byte per process participating in the lock. The second window object, `pollbytewin`, consists of a single byte window on each of the processes. Figure 2 shows these windows, and Figure 1 shows MPI pseudocode for creating these windows.

The algorithm works as follows. Processes trying to obtain the lock use a single access epoch to read (get) all of the waitlistwin memory region except for the byte that corresponds to their rank, and they write (put) a non-zero value into the byte corresponding to their rank. This effectively places them in the list of processes that would like to obtain the lock. Following that access epoch the process examines the contents of the memory region that it read during the epoch. If all values are zero, then no one else had the lock, and they are now the owner.

Local spinning occurs when a process attempts to obtain

```
if (myrank == homerank) {
    MPI_Win_create(waitlistaddr, nprocs, 1,
                   MPI_INFO_NULL, comm, &waitlistwin);
}
else {
    MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL,
                   comm, &waitlistwin);
}

MPI_Win_create(pollbyteaddr, 1, 1, MPI_INFO_NULL,
               comm, &pollbytewin);
```

Figure 1. MPI pseudocode for creating windows in one-sided algorithm.

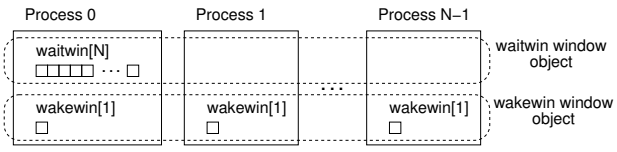


Figure 2. Two MPI windows are created in the one-sided algorithm.

the lock and sees that some other process already owns the lock. When this happens, the process continually reads the local byte in the pollbytewin, waiting for it to be set to a non-zero value. Another process, on releasing the lock, will notify the spinning process by writing a non-zero value into this memory region. MPI pseudocode for obtaining the lock is shown in Figure 3.

When the process is ready to release the lock, it performs a second access epoch, again reading all bytes except the one corresponding to their rank and writing a zero value into their rank. Following that access epoch the process examines the contents of the memory region that it read during the epoch. If all values are zero, then no one was waiting for the lock, and the process has finished releasing the lock. If there is a non-zero value, then one or more processes were waiting for the lock. For fairness purposes the process selects the next highest waiting rank after its own, wrapping back to rank zero as necessary. It then uses one additional access epoch to set the byte in that process's pollbytewin window to a non-zero value, notifying the process that it now owns the lock. MPI pseudocode for releasing the lock is shown in Figure 4.

2.2 Eliminating Polling with Point-to-Point

While the previous algorithm minimizes remote memory access, we would expect that spinning on local variables would waste many CPU cycles. This can be particularly important in systems where the memory bus is shared with other processors or processors are oversubscribed (i.e. more MPI processes than physical processors). One solution would be to use a back-off algorithm to mitigate CPU utilization, but that would incur additional latency in our lock acquisition.

Fundamentally we are using the pollbytewin for notification. We want one process to tell another one that it now owns the lock. Because we are in an MPI environment, we have a very effective mechanism for implementing notification: point-to-point operations. We will call our second algorithm, which uses both MPI-1 point-to-point and MPI-2 one-sided operations, the *hybrid algorithm*.

The hybrid algorithm eliminates the `pollbytewin` window object entirely. The process of acquiring and releasing the lock is similar to the one-sided algorithm, except that notification is handled by a single, simple `MPI_Send` on the process releasing the lock and by a `MPI_Recv` on the process waiting for notification, as shown in Figures 5 and 6. Because the waiting process does not know who will notify it that it now owns the lock, `MPI_ANY_SOURCE` is used to allow the receive operation to match any sender. A zero-byte message is used because all we are really interested in is synchronization; the arrival of the message is all that is needed.

3 Performance Evaluation

We can trivially enforce atomic mode semantics by using our mutex lock to implement a whole-file lock. Because we are primarily interested in correctness, this is a viable option for a portable implementation.

Our tests were run on a subset of Jazz, a 350 node Linux cluster at Argonne National Laboratory. Jazz has both Myrinet and Fast Ethernet networks We used both of these networks for testing, providing us with results on both low and high latency networks. We expect that we will see higher overhead in the Fast Ethernet tests because of the lower performance characteristics of the network. A CVS version of MPICH2 (version 1.0 plus minor enhancements) was used as our MPI implementation.

To isolate the performance characteristics of our algorithm from other artifacts in the system, we implemented synthetic benchmarks using the two algorithms presented.

Our first benchmark measures the time spent locking and unlocking without contention. This benchmark uses two

```
blklens[0] = mutex->myrank;
disps[0]   = 0;
blklens[1] = mutex->nprocs - mutex->myrank - 1;
disps[1]   = mutex->myrank + 1;
MPI_Type_indexed(2, blklens, disps, MPI_BYTE, &waitlisttype);

val = 1;
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, waitlistwin);
MPI_Get(waitlistcopy, nprocs-1, MPI_BYTE, homerank, 0, 1, waitlisttype, waitlistwin);
MPI_Put(&val, 1, MPI_BYTE, homerank, myrank, 1, MPI_BYTE, waitlistwin);
MPI_Win_unlock(homerank, waitlistwin);

/* check to see if lock is already held */
for (i=0; i < (nprocs - 1) && waitlistcopy[i] == 0; i++);
if (i < nprocs - 1) {
    unsigned char pollbytecopy = 0;

    /* spin on local variable until set by previous lock holder */
    while (!pollbytecopy) {
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, myrank, 0, pollbytewin);
        pollbytecopy = *pollbyteaddr;
        MPI_Win_unlock(myrank, pollbytewin);
    }

    /* set pollbyte back to zero */
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, mutex->myrank, 0, pollbytewin);
    *pollbyteaddr = 0;
    MPI_Win_unlock(myrank, pollbytewin);
}
```

Figure 3. MPI pseudocode for obtaining lock in one-sided algorithm. Note: `waitlisttype` **is actually created at lock creation time and cached in the actual implementation.**

```
val = 0;
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, waitlistwin);
MPI_Get(waitlistcopy, nprocs-1, MPI_BYTE, homerank, 0, 1, waitlisttype, waitlistwin);
MPI_Put(&val, 1, MPI_BYTE, homerank, myrank, 1, MPI_BYTE, waitlistwin);
MPI_Win_unlock(homerank, waitlistwin);

/* check to see if lock is already held */
for (i=0; i < (nprocs - 1) && waitlistcopy[i] == 0; i++);
if (i < nprocs - 1) {
    int nextrank;
    unsigned char pollbytecopy = 1;

    /* find the next rank waiting for the lock.  we start with the
     * rank after ours and look in order to ensure fairness.
     */
    nextrank = myrank;
    while (nextrank < (nprocs - 1) && waitlistcopy[nextrank] == 0) nextrank++;
    if (nextrank < nprocs - 1) nextrank++; /* nextrank is off by one */
    else {
        nextrank = 0;
        while (nextrank < myrank && waitlistcopy[nextrank] == 0) nextrank++;
    }

    /* set pollbyte on next rank (who is spinning) */
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, nextrank, 0, pollbytewin);
    MPI_Put(&pollbytecopy, 1, MPI_BYTE, nextrank, 0, 1, MPI_BYTE, pollbytewin);
    MPI_Win_unlock(nextrank, pollbytewin);
}
```

Figure 4. MPI pseudocode for releasing lock in one-sided algorithm.

```
/* add self to waitlist */
val = 1;
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0,
             waitlistwin);
MPI_Get(waitlistcopy, nprocs-1, MPI_BYTE,
        homerank, 0, 1, waitlisttype, waitlistwin);
MPI_Put(&val, 1, MPI_BYTE, homerank, myrank,
        1, MPI_BYTE, waitlistwin);
MPI_Win_unlock(homerank, waitlistwin);

/* check to see if lock is already held */
for (i=0; i < nprocs-1 && waitlistcopy[i] == 0; i++);
if (i < nprocs - 1) {
    /* wait for notification */
    MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE,
             WAKEUPTAG, comm, MPI_STATUS_IGNORE);
}
```

Figure 5. MPI pseudocode for obtaining lock in hybrid algorithm.

```
/* remove self from waitlist */
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0,
             waitlistwin);
MPI_Get(waitlistcopy, nprocs-1, MPI_BYTE,
        homerank, 0, 1, waitlisttype, waitlistwin);
MPI_Put(&val, 1, MPI_BYTE, homerank, myrank,
        1, MPI_BYTE, waitlistwin);
MPI_Win_unlock(homerank, waitlistwin);

for (i=0; i < nprocs-1 && waitlistcopy[i] == 0; i++);
if (i < nprocs - 1) {
    int nextrank = myrank;

    /* find the next rank waiting for the lock */
    while (nextrank < nprocs-1 &&
           waitlistcopy[nextrank] == 0) nextrank++;
    if (nextrank < nprocs - 1) {
        nextrank++; /* nextrank is off by one */
    }
    else {
        nextrank = 0;
        while (nextrank < myrank &&
               waitlistcopy[nextrank] == 0) nextrank++;
    }

    /* notify next rank with zero-byte message */
    MPI_Send(NULL, 0, MPI_BYTE, nextrank, WAKEUP, comm);
}
```

Figure 6. MPI pseudocode for releasing lock in hybrid algorithm.

MPI processes, where the first process owns the waitwin and the second process performs a sequence of locks and unlocks (across the network). This second process spins in a tight loop locking and unlocking, and we average the results of the iterations. Using this benchmark we found that the operation of locking and unlocking together take an average of 0.9 ms on Fast Ethernet and 0.1 ms on Myrinet in the absence of contention.

Our second synthetic benchmark simulates a collection of processes that all independently compete for the lock in order to perform a sequence of atomic operations. The process of performing these atomic operations is simulated by sleeping for an amount of time specified as a parameter to the test. We call this time simply the "work time". Each process performs 100 iterations of this lock/work/unlock cycle in a tight loop as part of a single test. We found that we were unable to accurately simulate a smaller amount of work than 10 ms because the `nanosleep` function call consistently delayed for no less than 10 ms regardless of how small a time value we passed to it.

In a real system this work time would be the time spent performing the I/O operations. Typical I/O operations take on the order of milliseconds. A high-end storage system capable of delivering 100 MBytes/sec can read or write about 1 MBytes in 10 ms – a fairly typical I/O size in the world of high-performance I/O. We vary our work time from 10 to 100 milliseconds in our experiments, simulating I/O operations of 1 Mbytes to 10 Mbytes in our hypothetical storage environment.

We know that there will be no I/O overlap using our algorithm, because we are serializing access to implement the atomic mode. This means that the total time to execute our synthetic benchmark is approximately n times our selected work time plus any overhead incurred by our locking algorithm, where n is the number of processes. We are interested in measuring this overhead and examining it as a percentage of the sum of the work times for all processes. To calculate the "overhead time", we subtract the total amount of time that processes actually spend working (not waiting for the lock) from the total elapsed time of the simulation. We then calculate the percentage of time lost due to our algorithm's overhead, which we term the "percent overhead".

To estimate the percent overhead for smaller work times, we ran our benchmark without any delay between locking and unlocking (effectively zero work time). This gave us an upper bound on the amount of time spent in the locking and unlocking operations. We used these values to estimate percent overhead at a 1 ms work time value.

In Figure 7(a) and Figure 7(b) we compare the percent overhead of using the one-sided algorithm to the percent overhead of using the hybrid algorithm as a percentage of total elapsed time. Work time is fixed at 100 msec per oper-

	1	4	8	16	32	64
Fast Ethernet	0.008	2.0	5.4	11.2	24.7	56.4
Myrinet	0.004	0.28	0.6	1.3	2.8	5.9

Table 1. Total overhead time from simulation with zero work time (ms)

ation. We see that the hybrid algorithm performs at least as well as the one-sided algorithm in all cases. If the CPUs are oversubscribed, hybrid performance does not degrade while the percent overhead of the one-sided method increases dramatically. Because we see performance improvements in the oversubscribed case and no penalty with one CPU per process, we will focus on the hybrid algorithm for the rest of this discussion.

In Figure 8(a) we focus on the overhead of the hybrid algorithm, varying both number of processes (X axis) and the time spent holding the lock ("working") on each iteration, using the Fast Ethernet network. We see that for simulated work times in the range of 32 ms and higher, overhead is negligible (less than 2%). Even at a 10 ms work time the overhead is sufficiently small for 64 processes (approximately 4.25%), and the rate at which the overhead increases with increasing numbers of processes indicates that the algorithm would scale to many more processes for this work granularity. The reason that we see only 4.25% overhead at 10 ms of work time when earlier we calculated that for Fast Ethernet the locking and unlocking sequence takes 0.9 ms is that processes attempting to acquire the lock can do so while the lock is held by another process, overlapping some lock communication with another process's work time.

Figure 8(b) shows a repeat of the experiment, this time making use of the Myrinet network. Percent overhead is an order of magnitude smaller, which is what we would expect given that Myrinet latency is roughly an order of magnitude lower than Ethernet.

Table 1 presents the results of running our simulation with no delay between locking and unlocking. These values give us an indication of the amount of overhead that would occur for very small work times. Numbers for the single process case are particularly low because all communication occurs locally. We can estimate an expected percent overhead for a given work time with the equation $100 * O_n/(O_n + n * Tw)$, where n is the number of processes concurrently operating, O_n is the total overhead for n processes on the given network, and Tw is the work time. This estimate is actually somewhat high because it does not account for overlap of locking attempts by some processes during the work time of another.

Using this equation we can estimate that we would spend approximately 47% of our time locking and unlocking in the case of a 1 ms work time on 64 processes on the Fast Ethernet network, or 8.5% of our time on the Myrinet network. We note that percentage overhead will always become a factor as the amount of work performed becomes smaller; this is an unavoidable result of the network operations necessary to acquire the distributed lock.

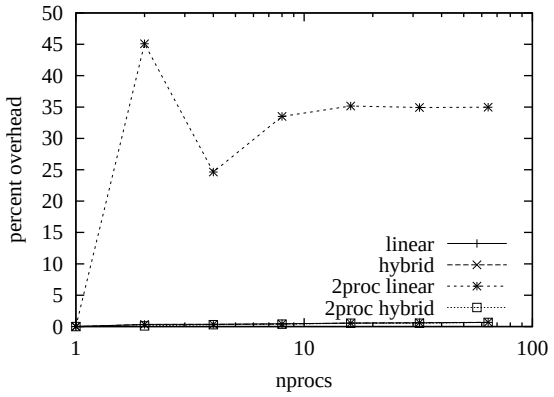
The experiments above show that for modest work times (as low as 10 ms), our algorithm incurs very little overhead on either network. For smaller work sizes the algorithm is efficient only on a high-performance network such as Myrinet.

4 Conclusions and Future Work

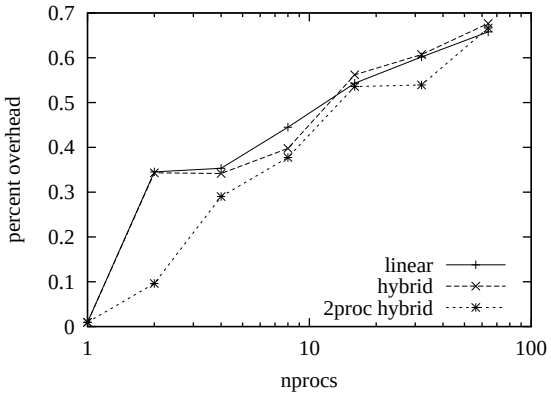
We have presented two new algorithms for implementing mutual exclusion with notification using MPI primitives. Our algorithms are capable of performing locking and unlocking in two access epochs in the absence of contention, and an additional remote access epoch or a single point-to-point is used for notification in the event of contention. The algorithms are also designed to avoid starvation by cycling through ranks. We have shown that the better of these two algorithms operates with very low overhead even in oversubscribed systems. We found that for operations that require 10 ms or more to complete, our algorithm was efficient even on a low-performance Fast Ethernet network. For operations that require less time, a higher-performance network would be necessary to maintain efficiency. On systems with these networks, the low overhead nature of our algorithm makes it a useful building block for applications in general.

We intend to use this algorithm for implementing MPI-IO atomic mode in a portable manner in ROMIO. This will provide atomic mode semantics for file systems whose locking subsystems are not yet complete (e.g. Lustre) and for file systems that lack locking subsystems entirely (e.g. PVFS2). Further investigation will be necessary to determine if this approach is more scalable than the locking implementations in some parallel file systems (e.g. GPFS). If so, we will modify ROMIO to use our scalable algorithm rather than the file system locks.

While this work has focused specifically on providing a correct and efficient implementation that is portable across file systems, there are a number of ways in which this work could be extended if we determined that higher performance atomic mode access was necessary. One manner in which this system could be improved is through the detection of non-overlapping file views. File views are the mechanism MPI-IO uses for specifying a subset of a file that a process will access. When the file view for a process does not overlap with the file views of other processes, locking is unnecessary – conflicts will not occur. Because of the complexity of the MPI datatypes used to describe file views, this is an

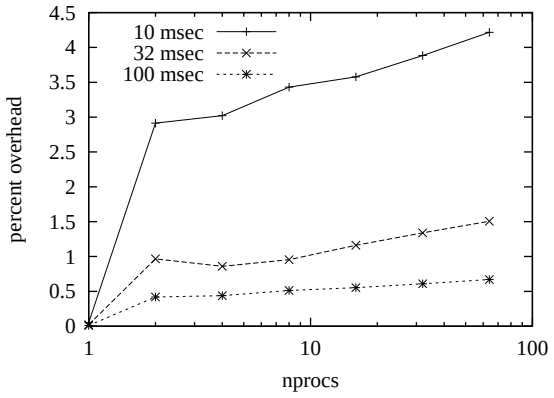


(a) The one-sided and hybrid techniques. '2proc' indicates two MPI processes per CPU

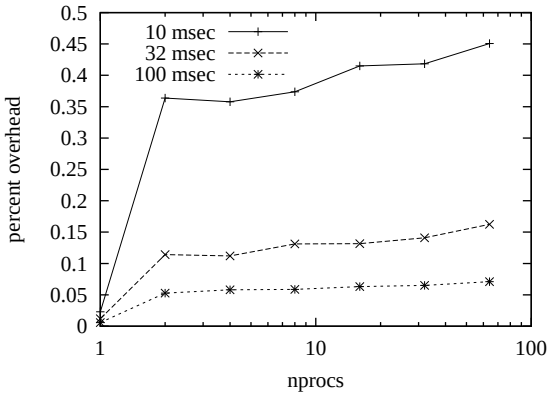


(b) Highlighting the three low-overhead cases in (a)

Figure 7. Percent overhead of locking strategies for 100 ms work time



(a) Percent overhead vs. work size for Fast Ethernet



(b) Percent overhead vs. work size for Myrinet. Observe that the scale of the Y-axis is one tenth that of Ethernet.

Figure 8. Interconnect effect on overhead

open research topic.

Another manner in which this work could be enhanced is through the use of multiple locks to partition a file into independent regions. Processes could then acquire only the locks needed to access regions that they were changing, allowing for concurrent access to separate regions. Ideally a range-based locking approach would be used. While maintaining the shared data structures necessary to store a list of ranges will undoubtedly require additional overhead, this approach might lead to an MPI-IO atomic mode that provides a level of concurrency and efficiency that beats that of the best file system locking implementations, eliminating the need for file locks in ROMIO entirely.

We have demonstrated that this work introduces little overhead for up to 64 processes. To handle even more processes (on the order of thousands), a tree algorithm might be more appropriate, where leaf nodes first acquire an intermediate lock before acquiring the lock itself. This level of indirection would limit contention on the byte array. Further testing at scale is necessary to determine if this extra degree of complexity in the algorithm is warranted.

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.

References

[1] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke. Data management and transfer in high performance computational grid environments. In *Parallel Computing Journal, Vol. 28 (5)*, pages 749–771, May 2002.

[2] Scott Atchley, Micah Beck, Jeremy Millar, Terry Moore, James S. Plank, and Stephen Soltesz. The logistical networking testbed. Technical Report Technical Report UT-CS-02-496, University of Tennessee Department of Computer Science, December 2002.

[3] 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*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[4] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[5] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[6] Robert Latham, Robert Ross, and Rajeev Thakur. The impact of file systems on MPI-IO scalability. In *Proceedings of EuroPVM/MPI 2004*, September 2004.

[7] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 1991.

[8] MPI-2: Extensions to the message-passing interface. The MPI Forum, July 1997.

[9] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an optimized implementation of mpi-io on top of gpfs. In *Proceedings of SC2001*, November 2001.

[10] Rajeev Thakur, William Gropp, , and Brian Toonen. Minimizing synchronization overhead in the implementation of MPI one-sided communication. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2004)*, pages 57–67, September 2004.

[11] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of SC98: High Performance Networking and Computing*. ACM Press, November 1998.

[12] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.