Implementing MPI-IO Atomic Mode and Shared File Pointers Using MPI One-Sided Communication

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

February 5, 2007

Abstract

The ROMIO implementation of the MPI-IO standard provides a portable infrastructure for use on top of a variety of underlying storage targets. These targets vary widely in their capabilities, and in some cases additional effort is needed within ROMIO to support all MPI-IO semantics. Two aspects of the interface that can be problematic to implement are MPI-IO *atomic mode* and the *shared file pointer* access routines. Atomic mode requires enforcing strict consistency semantics, and shared file pointer routines require communication and coordination in order to atomically update a shared resource. For some file systems, native locks may be used to implement these features, but not all file systems have lock support. In this work, we describe algorithms for implementing efficient mutex locks using MPI-1 and the one-sided capabilities from MPI-2. We then show how these algorithms may be used to implement both MPI-IO atomic mode and shared file pointer methods for ROMIO without requiring any features from the underlying file system. We show that these algorithms can outperform traditional file system lock approaches. Because of the portable nature of these algorithms, they are likely useful in a variety of situations where distributed locking or coordination is needed in the MPI-2 environment.

1 Introduction

MPI-IO (The MPI Forum 1997) provides a standard interface for MPI programs to access storage in a coordinated manner. Implementations of MPI-IO, such as the portable ROMIO implementation (Thakur, Gropp, and Lusk 1999) and the implementation for AIX GPFS (Prost, Treumann, Hedges, Jia, and Koniges

2001), have aided in the widespread availability of MPI-IO. These implementations in particular include a collection of optimizations (Thakur, Gropp, and Lusk 1998; Prost, Treumann, Hedges, Jia, and Koniges 2001; Latham, Ross, and Thakur 2004) that leverage MPI-IO features to obtain higher performance than would be possible with the less capable POSIX interface (IEEE 1996).

One component of the MPI-IO interface that has been difficult to implement in a portable manner 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 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, ROMIO cannot support atomic mode for file systems without locking systems.

Another feature that the MPI-IO interface provides is *shared file pointers*. A shared file pointer is an offset that is updated by any process accessing the file in this mode. This feature organizes accesses to a file on behalf of the application in such a way that subsequent accesses do not overwrite previous ones. This is particularly useful for logging purposes: it eliminates the need for the application to coordinate access to a log file.

Obviously, coordination must still occur; it just happens implicitly within the I/O software rather than explicitly in the application. Only a few historical file systems have implemented shared file pointers natively (Vesta (Corbett and Feitelson 1994), PFS (Intel Supercomputing Division 1993), CFS (Pierce 1989), SPIFFI (Freedman, Burger, and Dewitt 1996)), and they are not supported by parallel file systems being deployed today. Thus, today shared file pointer access must be provided by the MPI-IO implementation.

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 and the shared file pointer routines without file system support.

This paper discusses a novel method for supporting shared file pointer access within an MPI-IO implementation. This method relies only on MPI-1 and MPI-2 communication functionality and not on any storage system features, making it portable across any underlying storage. Sections 2 and 3 discuss the MPI-IO interface standard, the portions of this related to atomic mode and shared file pointers, and the way atomic mode and shared file pointer operations are supported in the ROMIO MPI-IO implementation. Section 4 describes our new approach to supporting atomic mode and shared file pointer operations within an MPI-IO implementation. Three algorithms are used: one for atomic-mode synchronization, one for independent shared-mode operations, and one for collective ordered calls. Section 5 evaluates the performance of our approaches on MPI-IO benchmarks. Section 6 concludes with a discussion of future work in this area.

2 MPI-IO Atomic Mode and ROMIO

The MPI-IO atomic mode guarantees sequential consistency of writes to the same file by a group of processes that have collectively opened the file. It also guarantees that these writes will be immediately visible by other processes in this group. This semantic is used primarily 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; hence, 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, atomicity must be imposed 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 v1 (Carns, Ligon III, Ross, and Thakur 2000) and PVFS v2 do not guarantee atomicity of operations, instead relying on the MPI-IO layer to provide these guarantees. Other types of storage back-ends, such as GridFTP (Allcock, Bester, Bresnahan, Chervenak, Foster, Kesselman, Meder, Nefedova, Quesnal, and Tuecke 2002) and logistical networks (Atchley, Beck, Millar, Moore, Plank, and Soltesz 2002), do not have locking capabilities either. The NFS file system provides advisory lock routines but makes no guarantees that locks will be honored across processes. In the existing ROMIO implementation atomic mode is simply not supported for these types of storage.

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. In Section 4 we describe how we implement these capabilities.

3 MPI-IO Shared File Pointers and ROMIO

The MPI-IO interface standard provides three options for referencing the location in the file at which I/O is to be performed: explicit offsets, individual file pointers, and shared file pointers. In the explicit offset calls, the process provides an offset that is to be used for that call only. In the individual file pointer calls, each process uses its own internally stored value to denote where I/O should start; this value is referred to as a file pointer. In the shared file pointer calls, each process in the group that opened the file performs I/O starting at a single, shared file pointer.

Each of these three ways of referencing locations has both independent (noncollective) and collective versions of read and write calls. In the shared file pointer case the independent calls have the `_shared` suffix (e.g., `MPI_File_read_shared`), while the collective calls have the `_ordered` suffix (e.g., `MPI_File_read_ordered`). The collective calls additionally guarantee that accesses will be ordered by rank of the processes. We will refer to the independent calls as the *shared-mode* accesses and the collective calls as the *ordered-mode* accesses.

The fundamental problem in supporting shared file pointers at the MPI-IO layer is that the implementation never knows when some process is going to perform a shared-mode access. This information is important because the implementation must keep a single shared file pointer value somewhere, and it must access and update that value whenever a shared-mode access is made by any process.

When ROMIO was first developed in 1997, most MPI implementations provided only MPI-1 functionality (point-to-point and collective communication), and these implementations were not thread safe. Thread safety makes it easier to implement algorithms that rely on nondeterministic communication, such as shared-mode accesses, because a separate thread can be used to wait for communication related to shared file pointer accesses. Without this capability, a process desiring to update a shared file pointer stored on a remote process could stall indefinitely waiting for the remote process to respond. The reason is that the implementation could check for shared-mode communication only when an MPI-IO operation was called. These constraints led the ROMIO developers to look for other methods of communicating shared file pointer changes.

Processes in ROMIO use a second hidden file containing the current value for the shared file pointer offset. A process reads from or writes into this file the value of the shared file pointer file before carrying out I/O routines. The hidden file acts as a communication channel among all the processes. File system locks serialize access and prevent simultaneous updates to the hidden file. This approach works well as long as the file system meets two conditions:

1. The file system must support file locks.
2. The file system locks must prevent access from other processes, and not just from other file accesses in the same program.

As discussed earlier, several common file systems do not provide file system locks. On such file systems ROMIO cannot correctly implement shared file pointers using the hidden file approach and currently must disable support for this feature. For this reason ROMIO needs a portable mechanism for synchronizing access to a shared file pointer that does not rely on any underlying storage characteristics.

```
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);
}
```

Figure 1: MPI pseudocode for creating windows in a one-sided algorithm. The "homerank" process hosts the actual memory, but all processes can access it.

4 Synchronization and Coordination with One-Sided Operations

The MPI-2 specification adds a new set of communication primitives, called the one-sided or remote memory access (RMA) functions. We are particularly interested in "passive target" RMA communication, which allows one process to modify the contents of remote memory without the remote process intervening. These passive target operations provide the basis on which to build a portable synchronization method within an MPI-IO implementation. This synchronization primitive can then be used to coordinate accesses when implementing atomic mode and shared file pointers.

MPI-2 one-sided operations do not provide a way to atomically read and modify a remote memory region. We can, however, construct an algorithm based on existing MPI-2 one-sided operations that lets a process perform an atomic modification. For atomic mode, we want to coordinate access to the entire file. In the shared file pointer case, we want to serialize access to the shared file pointer value.

Before performing one-sided transfers, 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. Figure 1 gives pseudocode for how this might be done.

MPI passive target operations are organized into *access epochs* that are bracketed by `MPI_Win_lock` and `MPI_Win_unlock` calls. Clever MPI implementations (Thakur, Gropp, and Toonen 2004) will combine all the data movement operations (puts, gets, and accumulates) into one network transaction that occurs at the unlock. The MPI-2 standard allows implementations to optimize RMA communication by carrying out operations in any order at the end of an epoch. Implementations take advantage of this fact to achieve much higher performance (Thakur, Gropp, and Toonen 2004). Thus, within one epoch a process cannot read a byte, modify that value, and write it back because the standard makes no guarantee about the order of the read-modify-write steps. This aspect of the standard complicates, but does not prevent, the use of one-sided routines

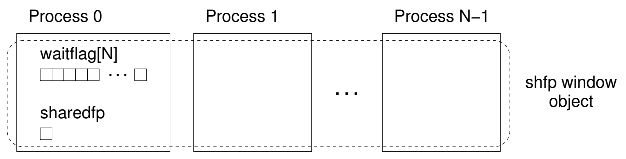


Figure 2: Depiction of MPI windows.

to build our data structures for coordination among MPI processes.

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 (Gropp, Lusk, and Thakur 1999). These approaches have some disadvantages, particularly in that they require many remote one-sided operations and poll on remote memory regions. Our approach requires only a few one-sided operations and makes no use of polling.

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.

The algorithms presented here were influenced by the MCS lock (Mellor-Crummey and Scott 1991), an algorithm devised for efficient mutex locks in shared-memory systems. Like the MCS lock, we carry out $O(1)$ network transactions per lock acquisition. However, we are not able to meet their achievement of constant memory size per lock, mainly because of 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. This general approach has been used in a simulation of a portable atomic mode algorithm (Ross, Latham, Gropp, Thakur, and Toonen 2005) and a simulated shared file pointer access (Latham, Ross, and Thakur 2005). Here we elaborate on both approaches, implement them in an MPI-IO implementation, and analyze behavior with actual MPI-IO programs.

4.1 The Hybrid Point-to-Point and One-Sided Approach

Were we to use only one-sided operations, we would end up polling on a particular byte to know when another process released the lock. While doing so minimizes remote memory access, we would expect that spinning on local variables would waste many CPU cycles. Avoiding this situation 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 proces-

```
/* 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 3: MPI pseudocode for obtaining lock in hybrid algorithm. The MPI datatype `waitlisttype` is an indexed type that accesss all the bytes except `waitflag[myrank]`.

sors). 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 want one process to inform another that it now owns the lock. Because we are in an MPI environment, we have an effective mechanism for implementing notification: point-to-point operations. We will call this algorithm, which uses both MPI-1 point-to-point and MPI-2 one-sided operations, the *hybrid algorithm*.

Our algorithms make use of the following data structure. We define a window object with an N-byte `waitflag` array and an `MPI_Offset`-sized `sharedfp`, both residing on a single process (Figure 2). In our discussion we will assume that this data structure is stored on process 0, but these structures could be distributed among different processes to balance the memory requirements if many files were being accessed. We also define on each process an MPI datatype designed to access all the bytes in `waitflag` except for the byte corresponding to the process's rank. As we will see, even though atomic mode, shared file pointers, and ordered mode accesses will access this data structure in different ways, this data structure contains all the information we need for the three modes.

Each byte in the `waitflag` array corresponds to a process. A process will request a lock (Figure 3) by putting a 1 in the byte corresponding to its rank in the communicator used to open the file. Doing so effectively adds it to the list of processes that want to access the file. In the same access epoch, the process will make use of `waitlisttype` to get only the remaining N-1 bytes of `waitflag`. Thus, at the end of the epoch, a process knows (because it has either read or written each of them) the value of all N bytes of `waitflag` array.

Should a process find that the `waitflag` array contains other 1 values, then some other process has the lock. The process will then call `MPI_Recv` and block until awakened by the lock-holding process. Upon receiving this message, the

```
/* 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 4: MPI pseudocode for releasing lock in hybrid algorithm.

process can safely assume it has been granted the lock. It does not need to re-examine the `waitflag` array.

If the `waitflag` array contains no other 1 values, then no other process is waiting for the lock. The process can proceed to access the file. When it releases the lock (Figure 4), it will initiate a second access epoch, placing a 0 in the corresponding byte of the `waitflag` array. During that second access epoch, the process will also get the remaining bytes of `waitflag`. If it finds a process waiting for the lock, it will call `MPI_Send` and wake it up.

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. 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.

4.2 Shared-Mode Synchronization

The MPI-2 standard makes no promises as to the order of concurrent shared-mode accesses. Additionally, the implementation does not need to serialize access to the file system, only the value of the shared file pointer. After a process updates the value of the file pointer, it can carry out I/O while the remaining processes attempt to gain access to the shared file pointer. By observing these

```
val = 1; /* add self to waitlist */
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, waitlistwin);
MPI_Get(waitlistcopy, nprocs-1, MPI_BYTE, homerank, FP_SIZE, 1,
        waitlisttype, waitlistwin);
MPI_Put(&val, 1, MPI_BYTE, homerank, FP_SIZE + myrank, 1, MPI_BYTE,
        waitlistwin);
MPI_Get(fpcopy, 1, fptype, homerank, 0, 0, fptype, 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(&fpcopy, 1, fptype, MPI_ANY_SOURCE, WAKEUPTAG, comm,
        MPI_STATUS_IGNORE);
}
```

Figure 5: MPI pseudocode for acquiring access to the shared file pointer.

restrictions, we can devise an approach that minimizes the time during which any one process has exclusive access to the shared file pointer.

In our shared-mode approach, we use the `waitflag` array to synchronize access to the shared file pointer. Figure 5 gives pseudocode for acquiring the shared file pointer, and Figure 6 demonstrates how we update the shared file pointer value.

Just like the atomic mode case, a process requests a lock by putting a 1 in the corresponding byte in the `waitlistwin`. In the same access epoch the process gets the remaining N-1 bytes of `waitflag` and the `sharedfp` value. (For atomic mode, we can omit the `sharedfp` value, since our atomic mode serializes access to the entire file.) This combination effectively implements a test and set. If a search of `waitflag` finds no other 1 values, then the process has permission to access the shared file pointer, and it already knows what that value is without another access epoch, having optimistically gotten that value at the same time as it got the N-1 bytes of the `waitflag` array.

In this case the process saves the current shared file pointer value locally for subsequent use in I/O. It then immediately performs a second access epoch (Figure 6). In this epoch the process updates `sharedfp`, puts a zero in its corresponding `waitflag` location, and gets the remainder of the `waitflag` array. Following the access epoch the process searches the remainder of `waitflag`. If all the values are zero, then no processes are waiting for access. If there is a 1 in the array, then some other process is waiting. For fairness the first rank after the current process's rank is selected to be awakened, and a point-to-point send (`MPI_Send`) is used to notify the process that it may now access the shared file pointer. The contents of the send is the updated shared file pointer value; this optimization eliminates the need for the new process to reread `sharedfp`. (For atomic mode, the send is carried out with a zero-byte payload.) Once the process has released the shared file pointer in this way, it performs I/O using the original, locally stored shared file pointer value. Again, by moving I/O after the shared file pointer update, we minimize the length of time the shared file

```
val=0; /* remove self from waitlist */
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, waitlistwin);
MPI_Get(waitlistcopy, nprocs-1, MPI_BYTE, homerank, FP_SIZE, 1,
        waitlisttype, waitlistwin);
MPI_Put(&val, 1, MPI_BYTE, homerank, FP_SIZE + myrank, 1,
        MPI_BYTE, waitlistwin);
MPI_Put(&fpcopy, 1, fptype, homerank, 0, 1, fptype, 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(&fpcopy, 1, fptype, nextrank, WAKEUPTAG, comm);
}
```

Figure 6: MPI pseudocode for updating the shared file pointer and (if needed) waking up the next process.

pointer is held by any one process.

If during the first access epoch a process finds a 1 in any other byte, some other process has already acquired access to the shared file pointer. The requesting process then calls MPI_Recv with MPI_ANY_SOURCE to block until the process holding the shared file pointer notifies it that it now has permission to update the pointer and passes along the current value. It is preferable to use point-to-point operations for this notification step, because they allow the underlying implementation to best manage making progress. We know, in the case of the sender, that the process we are sending to has posted, or will very soon post, a corresponding receive. Likewise, the process calling receive knows that very soon some other process will release the shared file pointer and pass it to another process. The alternative, polling using one-sided operations, has been shown less effective (Ross, Latham, Gropp, Thakur, and Toonen 2005).

4.3 Ordered-Mode Synchronization

Ordered-mode accesses are collective; in other words, all processes participate in them. The MPI-IO specification guarantees that accesses in ordered mode will be ordered by rank for these calls: the I/O from a process with rank N will appear in the file after the I/O from all processes with a lower rank (in the write case). However, the actual I/O need not be carried out sequentially. The implementation can instead compute *a priori* where each process will access the file and then carry out the I/O for all processes in parallel.

Process 0	Process 1 through $(N-2)$	Process $(N-1)$
Lock		
`MPI_Get`		
Unlock		
`MPI_Scan`	`MPI_Scan`	`MPI_Scan`
		Lock
		`MPI_Put`
		Unlock
MPI_Bcast	MPI_Bcast	MPI_Bcast
perform collective I/O	*perform collective I/O*	*perform collective I/O*

Figure 7: Synchronizing in the ordered mode case. Process 0 acquires the current value for the shared file pointer. After the call to `MPI_Scan`, process $(N-1)$ knows the final value for the shared file pointer after the I/O completes and can `MPI_Put` the new value into the window. Collective I/O can then be carried out in parallel with all processes knowing their appropriate offset into the file. An MPI_Bcast with process $(N-1)$ as the root ensures that the shared file pointer value is updated before any process exits the call.

Section 9.4.4.2 of the MPI-2.0 standard places several restrictions on collective I/O with shared file pointers. The most important one is that the application must ensure all outstanding independent I/O (e.g., shared-mode) routines have completed before initiating collective I/O (e.g., ordered-mode) ones. This restriction simplifies the implementation of the ordered-mode routines. However, the standard also states that

> In order to prevent subsequent shared offset accesses by the same processes from interfering with this collective access, the call might return only after all the processes within the group have initiated their accesses. When the call returns, the shared file pointer points to the next etype accessible.

This statement indicates that the implementation should guarantee that changes to the shared file pointer have completed before allowing the MPI-IO routine to return.

Figure 7 outlines our algorithm for ordered mode. Process 0 uses a single access epoch to get the value of the shared file pointer. Since the value is stored locally, the operation should complete with particularly low latency. While we use the same data structure for both shared-mode and ordered-mode accesses, note that ordered mode does not need to access `waitlist` at all, because the MPI specification requires the application not be performing shared-mode accesses at the same time. All processes can determine, based on their local `datatype` and `count` parameters, how much I/O they will carry out. In the call to `MPI_Scan`, each process adds this amount of work to the ones before it. After this call completes, each process knows its effective offset for subsequent I/O.

The $(N-1)$th process can compute the new value for the shared file pointer by adding the size of its access to the offset it obtained during the `MPI_Scan`. It performs a one-sided access epoch to put this new value into `sharedfp`, again ignoring the `waitlist`.

The MPI standard requires us to ensure no process races ahead of the others and starts carrying out other I/O operations before the last process can update the shared file pointer. We enforce this rule by having the $(N-1)$th process perform an `MPI_Bcast` of one byte after updating the shared file pointer. We use an `MPI_Bcast` here because we do not require all the syncronization steps of the more expensive `MPI_Barrier`. The single-byte payload ensures that a clever MPI implementation does not optimize away the call. All other processes wait for this `MPI_Bcast`, after which they know the last rank has completed its update of the shared file ponter. All processes may then safely carry out collective I/O and exit the call.

5 Performance Evaluation

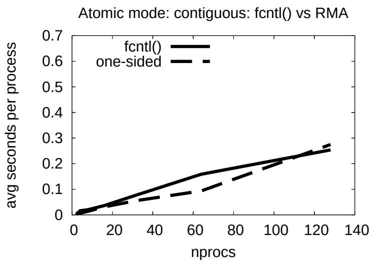
We integrated the above approaches in an experimental version of the ROMIO MPI-IO implementation as found in MPICH2. We carried out experiments on the NCSA Mercury machine. Mercury is an IA64 Linux cluster with Myrinet interconnect and a GPFS file system for parallel I/O. MPICH2 was configured to use the new "nemesis" channel over GM (Buntinas, Mercier, and Gropp 2006). Our experiments compare performance of several MPI-IO routines under ROMIO's older `fcntl()`-based locking and synchronization approach to our new RMA-based coordination methods.

5.1 Atomic Mode Performance

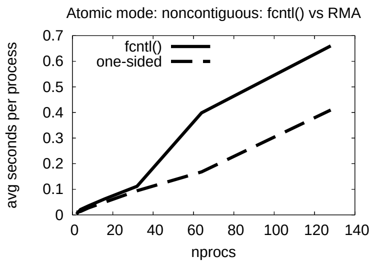
For our atomic mode comparison we instrumented ROMIO's `atomicity` test. This test initializes a file with zeros, then enables atomic mode. One process writes contiguous data to a file, while all other processes read data from the file. The sequence is then repeated with a noncontiguous access pattern. If the file system and MPI-IO implementation implement MPI-IO atomic mode correctly, processes should either see all old data or all new data, but never both. Note that under the default MPI-IO semantics, the results for such an operation would be undefined.

We plot results for a contiguous access pattern in Figure 8(a), where we see similar performance for the two approaches. We would expect contiguous accesses to be the best case for an `fcntl`-based approach because the amount of lock traffic in the contiguous case is significantly smaller than in the noncontiguous case. Both approaches see decreased performance (increased run time) as the number of processes — and the amount of time processes are waiting for their turn to take the lock — increases.

When we look at Figure 8(b) for the noncontiguous case, however, we see significant benefits to carrying out our synchronization via one-sided operations.



(a) Contiguous



(b) Noncontiguous

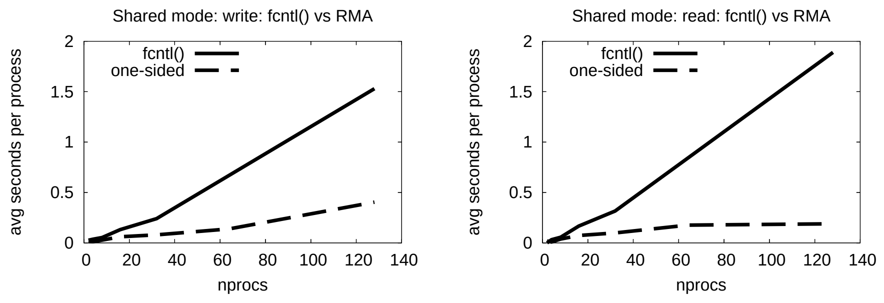
Figure 8: MPI-IO atomic mode. Average time per process to write or read from file.

The `fcntl` approach can potentially do well with a noncontiguous workload if the underlying file system implements byte-range locks. Even so, the RMA approach enjoys two advantages. First, the operations make use of MPI-2 access epochs to carry out the large number of requests more efficiently. The `fcntl` system call does not have enough contextual information to be able to make many optimizations when faced with a large number of requests. Second, the one-sided messaging traffic is carried out over a high-performance interconnect (in this case Myrinet) instead of trying to use the file system as a messaging layer. While MPI-IO atomic mode was never intended to be a high-performance mode of operation, we can handle a wider array of access patterns with higher performance by using one-sided operations.

5.2 Shared File Pointer Performance

For our shared file pointer analysis we instrumented the `shared_fp` test from ROMIO. In this test, all processes perform a shared write operation and then a shared read. While a typical application using shared file pointers would use variable-length blocks, this experiment uses fixed-sized records to eliminate any effects file system block alignment might have on a comparison. The one aspect we want to examine is the relative performance of the `fcntl` approach compared to the one-sided algorithm.

In Figures 9(a) and 9(b) we can see that the one-sided approach both performs and scales better than the older `fcntl` approach. As discussed earlier in this paper, the `fcntl` approach will make use of a hidden file, and each process will update that file with the appropriate value for the shared file pointer. The latency for a small file I/O operation can often be high. Here again, one-sided operations benefit from being able to make use of the high-performance network on this cluster instead of trying to communicate through the file system interface.



(a) Write

(b) Read

Figure 9: MPI-IO shared file pointers. Average time per process to write or read from file.

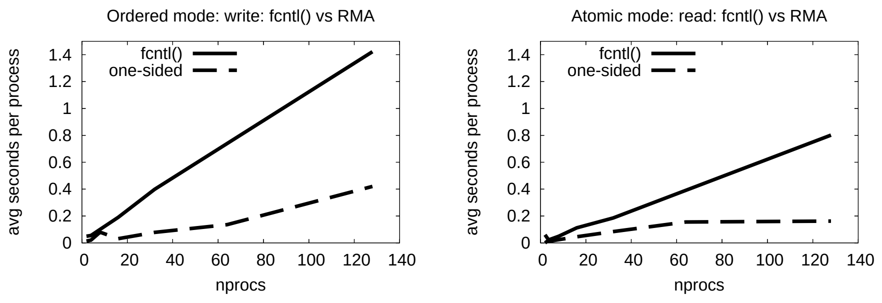
5.3 Ordered Mode Performance

To compare ordered-mode accesses, we modified the `shared_fp` test to make use of the collective ordered-mode routines. Otherwise, the workload is similar: all processes write a fixed amount of data to a file, then read it back. Under a correct implementation, all records will show up in the file, but will also show up in rank order (i.e., rank 0's data will show up before rank 1 and so on).

As is evident in Figures 10(a) and 10(b), the one-sided ordered-mode algorithm excels in this situation, delivering much better performance and remarkable scalability. We can identify three contributing factors. First, under the `fcntl` approach each process reads and updates the shared file pointer from the file system. Our one-sided algorithm stores the file pointer information in a memory window, not on disk, and so can avoid the overhead of acquiring a file system lock. Second, and most important, the one-sided algorithm takes advantage of the collective nature of the MPI-IO ordered routines. The memory window containing the shared file pointer information needs only to be read by one process and written by one other. The appropriate shared file pointer offset is disseminated with a collective call. Our ordered-mode algorithm can again make effective use of the high-performance interconnect on the Mercury cluster, but we also expect that even older or lower-performance interconnects would see performance gains.

6 Conclusions and Future Work

We have presented new algorithms for implementing mutual exclusion with notification using MPI primitives. Our algorithms perform locking and unlocking in two access epochs in the absence of contention and require only a single point-



(a) Write (b) Read

Figure 10: MPI-IO ordered mode. Average time per-process to write or read from file.

to-point message for notification in the event of contention. The algorithms also avoid starvation by cycling through ranks.

We implemented these algorithms in an experimental version of ROMIO as an option for implementing MPI-IO atomic mode and shared file pointers in ROMIO, providing atomic mode semantics for file systems whose locking subsystems are not yet complete and for file systems that lack locking subsystems entirely (e.g., PVFS). While our results compare favorably with the `fcntl` approach, further investigation will be necessary to determine whether the one-sided approach is in general 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 we have focused specifically on providing a correct and efficient implementation that is portable across file systems, this work could be extended in a number of ways if we determined that higher performance was necessary. One way in which the system could be improved is through the detection of nonoverlapping 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 open research topic.

Another way in which this work could be enhanced for atomic mode 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 implementation 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. Initial work toward this end has been done in (Thakur, Ross, and Latham 2005) but will require further analysis.

We have demonstrated that this work can outperform `fcntl` approaches for up to 128 processes. In order 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 whether this extra degree of complexity in the algorithm is warranted.

Our algorithms rely solely on MPI communication, using one-sided, point-to-point, and collective routines as appropriate. This removes any dependency on file system features and makes shared file pointer operations an option for all file systems. Performance in the shared-mode case scales as well as can be expected; performance in the ordered mode case scales very well.

Our synchronization routines have been used for MPI-IO atomic mode as well as MPI-IO shared file pointers. In future efforts we will look at using these routines to implement extent-based locking and other more sophisticated synchronization methods.

7 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. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

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

Atchley, S., M. Beck, J. Millar, T. Moore, J. S. Plank, and S. Soltesz (2002, December). The logistical networking testbed. Technical Report UT-CS-02-496, University of Tennessee Department of Computer Science.

Buntinas, D., G. Mercier, and W. Gropp (2006, September). Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem. In *Proceedings of EuroPVM/MPI.*

Carns, P. H., W. B. Ligon III, R. B. Ross, and R. Thakur (2000, October). PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, pp. 317–327. USENIX Association.

Corbett, P. F. and D. G. Feitelson (1994). Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pp. 63–70.

Freedman, C. S., J. Burger, and D. J. Dewitt (1996, November). SPIFFI — a scalable parallel file system for the Intel Paragon. *IEEE Transactions on Parallel and Distributed Systems 7*(11), 1185–1200.

Gropp, W., E. Lusk, and R. Thakur (1999). *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA: MIT Press.

IEEE (1996). *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. New York, NY USA: IEEE.

Intel Supercomputing Division (1993). *Paragon System User's Guide*. Intel Supercomputing Division.

Latham, R., R. Ross, and R. Thakur (2004, September). The impact of file systems on MPI-IO scalability. In *Proceedings of EuroPVM/MPI 2004*.

Latham, R., R. Ross, and R. Thakur (2005). Implementing MPI-IO shared file pointers without file system support. In *Proceedings of EuroPVM/MPI*.

Mellor-Crummey, J. M. and M. L. Scott (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*(1), 21–65.

Pierce, P. (1989, March). A concurrent file system for a highly parallel mass storage system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, pp. 155–160. Golden Gate Enterprises, Los Altos, CA.

Prost, J.-P., R. Treumann, R. Hedges, B. Jia, and A. Koniges (2001, November). MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Proceedings of SC2001*.

Ross, R., R. Latham, W. Gropp, R. Thakur, and B. Toonen (2005, May). Implementing MPI-IO atomic mode without file system support. In *Proceedings of CCGrid 2005*.

Thakur, R., W. Gropp, and E. Lusk (1998, November). A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of SC98: High Performance Networking and Computing*. ACM Press.

Thakur, R., W. Gropp, and E. Lusk (1999, May). 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.

Thakur, R., W. Gropp, and B. Toonen (2004, September). 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)*, pp. 57–67.

Thakur, R., R. Ross, and R. Latham (2005, September). Implementing byte-range locks using MPI one-sided communication. In *Proceedings of the 12th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2005), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, LNCS 3666, Springer*, pp. 119–128.

The MPI Forum (1997, July). MPI-2: Extensions to the Message-Passing Interface.