Implementing MPI-IO Shared File Pointers without File System Support

Robert Latham, Robert Ross, Rajeev Thakur, Brian Toonen

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{robl,rross,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 targets vary widely in their capabilities, and in some cases additional effort is needed within ROMIO to support all MPI-IO semantics. The MPI-2 standard defines a class of file access routines that use a *shared file pointer*. These routines require communication internal to the MPI-IO implementation in order to allow processes to atomically update this shared value. We discuss a technique that leverages MPI-2 one-sided operations and can be used to implement this concept without requiring any features from the underlying file system. We then demonstrate through a simulation that our algorithm adds reasonable overhead for independent accesses and very small overhead for collective accesses.

1 Introduction

MPI-IO [1] provides a standard interface for MPI programs to access storage in a coordinated manner. Implementations of MPI-IO, such as the portable ROMIO implementation [2] and the implementation for AIX GPFS [3], have aided in the widespread availability of MPI-IO. These implementations include a collection of optimizations [4, 3, 5] that leverage MPI-IO features to obtain higher performance than would be possible with the less capable POSIX interface [6].

One 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 [7], PFS [8], CFS [9], SPIFFI [10]) 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.

This paper discusses a novel method for supporting shared file pointer access within a 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. Section 2 discusses the MPI-IO interface standard, the portions of this related to shared file pointers, and the way shared file pointer operations are supported in the ROMIO MPI-IO implementation. Section 3 describes our new approach to supporting shared file pointer operations within an MPI-IO implementation. Two algorithms are used, one for independent operations and another for collective calls. Section 4 evaluates the performance of these two algorithms on synthetic benchmarks. Section 5 concludes and points to future work in this area.

2 Background

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 have both independent (non-collective) 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 also 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.

2.1 Synchronization of Shared File Pointers in ROMIO

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 or writes the value of the shared file pointer into this 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.

Unfortunately, several common file systems do not provide file system locks (e.g., PVFS, PVFS2, GridFTP) and the NFS file system provides advisory lock routines but makes no guarantees that locks will be honored across processes. On file systems such as these, ROMIO cannot correctly implement shared file pointers using the hidden file approach and hence disables support for shared file pointers. For this reason a portable mechanism for synchronizing access to a shared file pointer is needed that does not rely on any underlying storage characteristics.

3 Synchronization 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, that allow 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 general approach has been used in a portable atomic mode algorithm [11]. Here we extend that approach to manage a shared file pointer and additionally to address efficient ordered mode support.

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. In this 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.

MPI passive target operations are organized into *access epochs* that are bracketed by `MPI_Win_lock` and `MPI_Win_unlock` calls. Clever MPI implementations [12] 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 [12]. 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 to build our shared file pointer support.

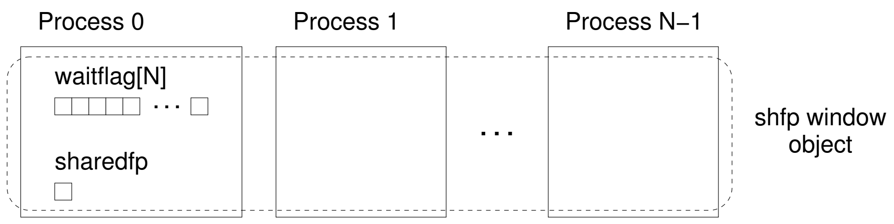


Fig. 1. Creation of MPI windows

Our algorithms operate by using 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 1). In our discussion we will assume that this data structure is stored on process 0, but for multiple files being accessed in shared file pointer mode, these structures could be distributed among different processes. This data structure is used differently for shared mode than for ordered mode access. We will discuss each in turn.

3.1 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. This approach 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 2 gives pseudocode for acquiring the shared file pointer, and Figure 3 demonstrates how we update the shared file pointer value.

Each byte in the `waitflag` array corresponds to a process. A process will request a lock 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 shared file pointer. In the same access epoch the process gets the remaining N-1 bytes of `waitflag` and the `sharedfp` value. This combination effectively implements a test and set. If a search of `waitflag` finds no

```
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);
}
```

Fig. 2. MPI pseudocode for acquiring access to the shared file pointer.

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.

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 3). 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`. 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 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 [11].

```
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 new value of shared file pointer */
    MPI_Send(&fpcopy, 1, fptype, nextrank, WAKEUPTAG, comm);
}
```

Fig. 3. MPI pseudocode for updating shared file pointer and (if needed) waking up the next process.

3.2 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.

MPI places several restrictions on collective I/O. The most important one, with regard to ordered mode, is that the application 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 4 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. It does not need to access `waitlist` at all, because the MPI specification leaves it to the application not to 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`.

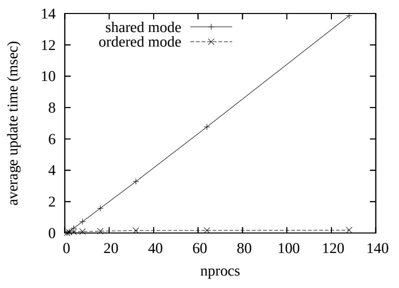
To ensure that a process doesn't race ahead of the others and start doing I/O before the shared file pointer has been updated, the $(N-1)$th process performs a `MPI_Bcast` of one byte after updating the shared file pointer. All other processes wait for this `MPI_Bcast`, after which they may all safely carry out collective I/O and then exit the call. If we used an `MPI_Barrier` instead of an `MPI_Bcast`, the $(N-1)$th process would block longer than is strictly necessary.

Process 0	Process 1 through (N minus 2)	Process (N minus 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*

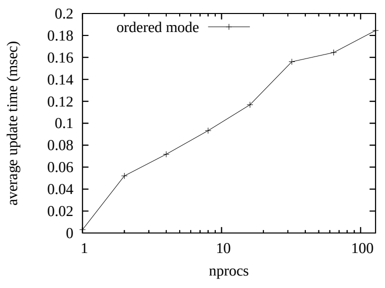
Fig. 4. Synchronizing in the ordered mode case. Process 0 acquires the current value for the shared file pointer. After the call to `MPI_Scan`, process $(N-1)$ knows the final value for the shared file pointer after the I/O completes and can `MPI_Put` the new value into the window. Collective I/O can then be carried out in parallel with all processes knowing their appropriate offset into the file. An MPI_Bcast ensures that the shared file pointer value is updated before any process exits the call, and imposes slightly less overhead than an MPI_Barrier.

4 Performance Evaluation

We simulated both algorithms (independent and collective) with a test program that implemented just the atomic update of the shared file pointer value. We ran tests on a subset of Jazz, a 350-node Linux cluster at Argonne National Laboratory, using the cluster's Myrinet interconnect.



(a) Average update time



(b) Focusing on ordered mode (log scale)

Fig. 5. Average time to perform one update of the shared file pointer.

Earlier in the paper we laid out the requirements for the hidden file approach to shared file pointers. On Jazz, none of the available clusterwide file systems meet those requirements. In fact, the NFS volume on Jazz has locking routines that not only fail to enforce sequential access to the shared file pointer but fail silently. Thus, we were unable to compare our approach with the hidden file technique.

This silent failure demonstrates another benefit of the RMA approach: if an MPI-I/O implementation tests for the existence of RMA routines, it can assume they work (otherwise the MPI-2 implementation is buggy). Tests for file locks, especially testing how well they prevent concurrent access, are more difficult, because such tests would have to ensure not just that the locking routines exist, but that they perform as advertised across multiple nodes.

4.1 Shared Mode Synchronization

In our simulation, processes repeatedly perform an atomic update of the shared file pointer in a loop. We measured how long processes spent updating the shared file pointer and then computed the average time for one process to carry out one shared file pointer update. Figure 5(a) shows how the number of processes impacts the average update time and gives an indication of the scalability of our algorithm.

All the independent I/O routines leave little room for the implementation to coordinate processes, so when N processes attempt to update the shared file pointer, we have to carry out $O(N)$ one-sided operations. Only one process can lock the RMA window at a time. This serialization point will have more of an impact as the number of processes — and the number of processes blocking — increases. The shared mode graph in Figure 5(a) confirms linear growth in time to update the shared file pointer value.

When considering performance in the independent shared file pointer case, one must bear in mind the nature of independent access. As with all independent MPI routines, the implementation does not have enough information to optimize accesses from multiple processes. Also, the simulation provides a worst-case scenario, with multiple processes repeatedly updating the shared file pointer as quickly as possible. In a real application, processes would perform some I/O before attempting to update the shared file pointer again.

4.2 Ordered Mode Synchronization

Implementations have more options for optimizing collective I/O, especially when performing collective I/O with a shared file pointer. As outlined in Section 3.2, N processes need only perform two access epochs — one for reading the current shared file pointer value, one for writing the updated value — from two processes. Our algorithm does use two collective routines (`MPI_Scan` and `MPI_Bcast`), so we would expect to see roughly $O(\log(N))$ increase in time to update the shared file pointer using a quality MPI implementation. Figure 5(a) compares update time for the ordered mode algorithm with that of the shared mode algorithm. The ordered mode algorithm scales quite well and ends up being hard to see on the graph. Figure 5(b) shows the average update time for just the ordered algorithm. The graph has a log scale X axis, emphasizing that the ordered mode algorithm is $O(\log(N))$.

5 Conclusions and Future Work

We have outlined two algorithms based on MPI-2 one-sided operations that an MPI-IO implementation could use to implement the shared mode and ordered mode routines. 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, while performance in the ordered mode case scales very well.

We designed the algorithms in this paper with an eye toward integration into ROMIO. At this time, one-sided operations make progress only when the target process hosting the RMA window is also performing MPI communication. We will have to add a progress thread to ROMIO before we can implement the shared file pointer algorithms. In addition to their use in ROMIO, the primitives used could be made into a library implementing test-and-set or fetch-and-increment for other applications and libraries.

The simulations in this paper focused on relatively small numbers of processors (128 or less). As the number of processors increases to thousands, we might need to adjust this algorithm to make use of a tree. Leaf nodes would synchronize with their parents before acquiring the shared file pointer. Such an approach reduces contention on the process holding the memory windows, but it also introduces additional complexity.

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.

6 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. The MPI Forum: MPI-2: Extensions to the Message-Passing Interface (1997)
2. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems. (1999) 23–32
3. Prost, J.P., Treumann, R., Hedges, R., Jia, B., Koniges, A.: MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In: Proceedings of SC2001. (2001)
4. Thakur, R., Gropp, W., Lusk, E.: A case for using MPI's derived datatypes to improve I/O performance. In: Proceedings of SC98: High Performance Networking and Computing, ACM Press (1998)
5. Latham, R., Ross, R., Thakur, R.: The impact of file systems on MPI-IO scalability. In: Proceedings of EuroPVM/MPI 2004. (2004)
6. IEEE/ANSI Std. 1003.1: Portable operating system interface (POSIX)–Part 1: System application program interface (API) [C language] (1996 edition)
7. Corbett, P.F., Feitelson, D.G.: Design and implementation of the Vesta parallel file system. In: Proceedings of the Scalable High-Performance Computing Conference. (1994) 63–70
8. Intel Supercomputing Division: Paragon System User's Guide. (1993)
9. Pierce, P.: 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, Golden Gate Enterprises, Los Altos, CA (1989) 155–160
10. Freedman, C.S., Burger, J., Dewitt, D.J.: SPIFFI — a scalable parallel file system for the Intel Paragon. IEEE Transactions on Parallel and Distributed Systems **7** (1996) 1185–1200
11. Ross, R., Latham, R., Gropp, W., Thakur, R., Toonen, B.: Implementing MPI-IO atomic mode without file system support. In: Proceedings of CCGrid 2005. (2005)
12. Thakur, R., Gropp, W., Toonen, B.: 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). (2004) 57–67