Evaluating Algorithms for Shared File Pointer Operations in MPI I/O

Ketan Kulkarni and Edgar Gabriel

Parallel Software Technologies Laboratory,
Department of Computer Science, University of Houston,
{knkulkarni,gabriel}@cs.uh.edu

Abstract. MPI-I/O is a part of the MPI-2 specification defining file I/O operations for parallel MPI applications. Compared to regular POSIX style I/O functions, MPI I/O offers features like the distinction between individual file pointers on a per-process basis and a shared file pointer across a group of processes. The objective of this study is the evaluation of various algorithms of shared file pointer operations for MPI-I/O. We present three algorithms to provide shared file pointer operations on file systems that do not support file locking. The evaluation of the algorithms is carried out utilizing a parallel PVFS2 file system on an InfiniBand cluster and a local ext3 file system using a 8-core SMP.

1 Introduction

The MPI standards provide many features for developing parallel applications on distributed memory systems, such as point-to-point communication, collective operations, and derived data types. One of the important features introduced in MPI-2 is MPI-I/O [1]. The MPI I/O routines provide a portable interface for accessing data within a parallel application. A feature defined in MPI-I/O is the notion of a *shared file pointers*, which is a file pointer jointly maintained by a group of processes.

There are two common usage scenarios for shared file pointers in parallel applications. The first one involves generating event logs of a parallel application, e.g., to document the progress of each application process or to generate error logs. For these scenarios, users typically want to know the chronological order of events. Without shared file pointers, the parallel applications would be required to coordinate events with the other processes in a sequential manner (in the order in which these events occurred) [2]. Using a shared file pointer, the MPI library will automatically order the events, and ease the generation of parallel log files significantly. The second usage scenario uses a shared file pointer for assigning chunks of work to each processes in a parallel application. If a process has finished computing the work currently assigned to it, it could read the next chunk from the shared input file using a shared file pointer. Thus, no additional entity is required to manage the distribution of work across the processes.

Although there are/were some file systems that have support for shared file pointers such as VESTA [3], the majority of file systems available today do not

provide a native support for these operations. Thus, the MPI library has to emulate shared file pointer operations internally. Today, the most wide-spread implementation of MPI I/O is ROMIO [4]. ROMIO relies on a hidden file which contains the shared file pointer for each MPI file. In order to modify the shared file pointer, a process has to lock the hidden file and thus avoid simultaneous access by multiple processes. However, many file systems have only limited or no support for file locking. As a result, shared file pointer operations often do not work with ROMIO [5]. Furthermore, since file locking is considered to be expensive, the performance of shared file pointer operations in ROMIO is often sub-optimal.

An alternative approach for implementing shared file pointer operations has been described in [6]. The solution proposed is to implement the shared file pointers using passive target one-sided communication operations, such as `MPI_Win_lock` and `MPI_Win_unlock`. A shared file pointer offset is stored in an MPI window. If a process wants to execute a read or write operation using the shared file pointer, it has to first *test* if any other process has acquired the shared file pointer. If this is not the case, it will access the shared file pointer from the root process. Else, it will wait for the signal from the process that currently has the shared file pointer. However, this approach is not available in a public release of ROMIO as of today, since it requires an implementation of one-sided operations, which can make progress outside of MPI routines, e.g. by using a progress thread [6]. In [7] Yu et.al. present an approach similar to one algorithm outlined in this paper by using the file-joining feature of Lustre to optimize the performance of collective write operations. In contrary to our algorithms however, there approach is bound to a specific file system and is not used to optimize shared file pointer operations.

In this paper we present three alternative algorithms for implementing shared file pointer operations on top of non-collaborative file systems. The first approach utilizes an additional process for maintaining the status of the shared file pointer. The second algorithm maintains a separate file for each process when writing using a shared file pointer. The third approach utilizes also an individual file on a per process basis, combines however the data of multiple MPI files into a single individual file. The remainder of the paper is organized as follows. In section 2 we describe the algorithms that have been developed, and document expected advantages, disadvantages and restrictions. Section 3 describes the test environment and the results of our evaluation over a parallel PVFS2 file system and a local ext3 file system. For the latter we also compare the performance of our algorithms to the ROMIO implementation of shared file pointers. Finally, section 4 summarizes the results of the paper and presents the ongoing work in this area.

2 Shared File Pointer Algorithms

The aim of designing new algorithms for shared file pointer operations is to avoid dependencies of the file systems on *file locks*, and to improve the performance of

shared file pointer operations. In the following, we describe three algorithms to achieve these goals, namely using an additional process to maintain the shared file pointer, using an individual file per process and MPI file, and using a single file per process across all MPI files. The algorithms described in this section have been implemented as a standalone library using the profiling interface of MPI, and use individual MPI file pointer operations to implement the algorithms. In order to do not mix up the individual file pointers to an MPI file and the shared file pointer, each MPI File is opened twice providing two independent MPI handles.

2.1 Using an Additional Process

The main idea behind the first algorithm is to have a separate process that maintains the shared file pointer. This mechanism replaces the hidden file used by ROMIO maintaining the shared file pointer, thereby preventing the file locking problems mentioned previously. In the current implementation, the additional process is created by dynamically spawning a new process upon opening a file. However, in a long term we envision to utilize an already existing process, such as *mpirun* for this purpose, since the workload of the additional process is relatively low in realistic scenarios.

Upon opening a new MPI file, all processes of a process group defined by an intra-communicator collectively spawned the management process using `MPI_Comm_spawn`. `MPI_Comm_spawn` returns a new inter-communicator that consists of a local group containing the already existing processes and a new remote group containing the newly spawned process. This inter-communicator will be used for communication between the local and remote groups. The management process initializes and stores the shared file pointer. Whenever a participating application process wants to perform an I/O operation using the shared file pointer, it first requests the current position of the shared file pointer respectively the file offset from the management process. The management process puts all requests in a request-queue and sends the current value of the shared file pointer offset back to the requesting process, and increases the value by the number of bytes indicated in the request.

The collective version of the write operation (`MPI_File_write_ordered`) for shared file pointers is implemented in multiple steps. First, a temporary root process, typically the process with the rank zero in that communicator, collects from each process in the group the number of bytes that they would write to the main file using a gather operation. The temporary root process sends the request to the management process with the total number of bytes to be written as a part of this collective write operation, and receives the offset of the shared file pointer. It than calculates the individual offset for each process based on the shared file pointer offset and the number of bytes requested by each process, and sends it to each process individually. Note that an alternative implementation using `MPI_Scan` could be used to determine the local offset for each process. Each process then writes to the main file using the explicit offset collective blocking write file routine `MPI_File_write_at_all`. Thus, the collective notion

of the operation is maintained. The collective read algorithm works in a similar fashion. This algorithm is able to implement the full set of functions defined in the MPI standard, and does not have any major restrictions form the usage perspective.

2.2 Using an Individual File per Process and MPI File

This algorithm prevents file locking on the main file during collective and non-collective write operations using the shared file pointer by having each process write its data into an individual data file. Henceforth, the file that is opened and closed in the application would be referred to as the main file. At any collective file I/O operation the entire data from the individual files are merged into the main file based on the *metadata* information stored along with the application data. *Metadata* is the information about the data, which is being written into the data file, and contains the information shown in Table 1.

Table 1. Metadata Record

Record Id	ID indicating the write operation of this record
Timestamp	Time at which data is being written to the data file
Local position	Offset in the individual data file
Record length	Number of bytes written to the data file

In the following, we detail the required steps in this algorithm. Each process opens an individual *data file* and a *metadata file* during the `MPI_File_open` operation. During any individual write operation, each process writes the actual data into its individual data file instead of the main file. The corresponding Metadata is stored in the main memory of each process using a linked list. Once a limit to the number of nodes in the linked list is reached, for example, as indicated by the parameter *MAX_METADATA_RECORDS*, the linked list is written into the metadata file corresponding to that process. During the merging step, we first check for metadata records in the metadata file before reading the records from the linked list. The most important and time-consuming operation in this algorithm is that of merging independent data written by each of the process. Note that merging needs to involve all processes in the group. Hence, it can be done only during collective operations such as `MPI_File_write_ordered`, `MPI_File_write_ordered_begin`, and `MPI_File_close`.

The merging of data requires the following five steps:

1. All processes compute the total number of *metadata nodes* by combining the metadata nodes written by each of the processes using an `MPI_Allgather` operation.
2. Each process collects the timestamps and record lengths corresponding to all metadata nodes stored on each process using an `MPI_Allgatherv` operation.

3. Once each process receives all the metadata nodes containing the timestamps and record lengths, it sorts them based on the timestamps in an ascending order.
4. Each process assigns a global offset to the sorted list of the metadata nodes. The *global offset* corresponds to the position at which the data needs to be written into the main file.
5. Once each process has global offsets assigned to each metadata record, it reads the data from the local data file using the local file offset from the corresponding metadata node, and writes the data into the main file based on the global offset, which was computed in the previous step.

For the merging step we also envision the possibility to create the final output file in a separate post processing step. This would give applications the possibility to improve the performance of their write operations using a shared file pointer, and provide an automatic mechanism to merge the separate output files of the different processes after the execution of the application.

This implementation has three major restrictions:

- The individual file algorithm does not support read operations using the shared file pointers. The assumption within this prototype implementation is that the user indicates using an `MPI_Info` object that shared file pointers would not be used for read operations, respectively the MPI library should not choose this algorithm if the required guarantee is not given, but fall back to a potentially slower but safer approach.
- In order to be able to compare the timestamps of different processes, this algorithm requires a globally synchronized clock across the processes. While there are some machines providing a synchronized clock, this is clearly not the case on most clusters. However, there are algorithms known in the literature on how to synchronize clocks of different processes [8], which would allow to achieve a relatively good clock synchronization at run time. Nevertheless, this algorithm has the potential to introduce minor distortions in the order of entries between processes.
- This implementation can not support applications that use both individual and shared file pointers simultaneously. In our experience this is however not a real restriction, since we could not find any application using both type of file pointers at the same time.

An additional hint passed to the library using an Info object could indicate whether the merge operation shall be executed at runtime or in a post-processing step.

2.3 Using a Single Individual File per Process

This algorithm offers a slight variation from the individual file per process algorithm. It maintains a single data and meta data file per process across all the files that have been opened by that process. Data pertaining to multiple files are written into single data and metadata files. The main reason behind

this approach is to ease the burden on the metadata servers of the file systems, whose main limitation is the number of simultaneous I/O requests that can be handled.

The main difference between this algorithm and the one described in the previous subsection is in the handling of collective write operations. When a metadata node is written to the metadata file for an individual operation, each process can individually determine the according timestamp of this operation. However, for collective operations, the data needs to be written in the order of the ranks of the processes. To ensure this behavior, the timestamp corresponding to the process with rank zero is broadcasted to all processes in the communicator, and used for the corresponding collective entry. To identify the collective operations at file close, each process maintains a linked list with the timestamps corresponding to the collective operations.

This algorithm has the same restrictions as the 'individual file per process and MPI file algorithm' described above.

3 Performance Evaluation

In the following, we present the performance of the algorithms described in the previous section. The three algorithms have been evaluated on two machines, namely, the *shark* cluster and *marvin*, an 8 core SMP server. The shark cluster consists of 24 compute nodes and one front end node. Each node consists of a dual-core 2.2GHz AMD Opteron processor with 2 GB of main memory. Nodes are connected by a 4xInfiniBand network. It has a parallel file system (PVFS2) mounted as '/pvfs2', which utilizes 22 hard drives, each hard drive is located on a separate compute node. The PVFS2 file system internally uses the Gigabit Ethernet network to communicate between the pvfs2-servers. Since two of the three algorithms presented in this paper only support write operations, our results section also focuses due to space constraints on the results achieved for various write tests.

Marvin is an eight processor shared memory system, each processor being a 2GHz single-core AMD Opteron processor. The machine has 8 GB of main memory and a RAID storage as its main file system using the Ext3 file system.

The results presented in this section are determined by using a simple benchmark program which writes data to a single file using either of the four shared file pointer write operations defined in MPI-2.

3.1 Results Using a PVFS2 File System

In the first test we constantly increase the total amount of data written by fixed number of processes. Tests have been performed for all four write operations using shared file pointers, namely, `MPI_File_write_shared`, `MPI_File_write_ordered`, `MPI_File_iwrite_shared`, `MPI_File_write_ordered_begin`. Due to space limitations, we show the results however only the blocking versions of these operations. Note that as explained in section 1, ROMIO does not support shared file

pointer operations over PVFS2 [5]. In the following subsections, results obtained after comparing all three algorithms are listed. Every test has been repeated at least three times, and the minimum execution time has been used for each scenario.

On PVFS2, the benchmark is run for each of the four write operations using shared file pointer. In each test case, data has been written from 10 MB to 10 GB among eight processes.

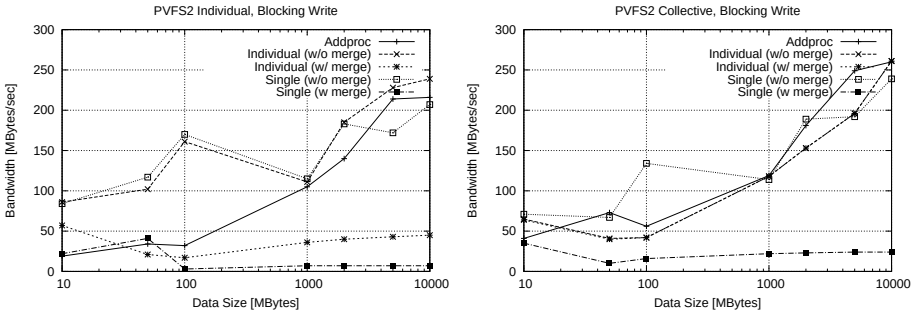

Fig. 1. Performance Comparison for Individual, Blocking write operation (left) and Collective Blockgin write-operations (right).

Figure 1 (left) gives the performance of individual and blocking write operation over PVFS2 file system. For the individual file and single file implementations, we show two separate lines for the execution time before merging (i.e., before calling `MPI_File_close`) and after merging. The reason for showing two lines is that these two algorithms are especially designed to minimize the time spent within the write operation itself, since the merging operation can potentially be performed in a post-processing step.

The individual file and single file implementations before merging of the data perform at par with the additional process implementation. All three implementations, with individual and single file performance compared before merging of data, show a relatively high bandwidth in the range of 140 MB/s to 240 MB/s, while writing 2-10 GB of data. The individual file algorithm achieves an overall bandwidth of around 50 MB/s after merging, while for the single file version bandwidth drops to around 10 MB/s including the merging step. The latter is the result of the large number of reduction and communication operations required in the merging step for that algorithm.

Figure 1 (right) gives the performance of the collective and blocking write operation over the PVFS2 file system. Note that the individual file implementation of the collective operations does not require merging, since the data is written directly into the main file for collective operations. Hence, only the single file implementation performance before the merging is compared with that of the other

two implementations. All the three implementations perform similarly, only the performance of the single file method after the merging deviates significantly. The performance improves as the amount of data being written increases.

Performance of the three algorithms has also been evaluated using by writing a fixed amount of data, and varying the number of processes writing the data. Accordingly, on PVFS2 2GB of data is being written by varying the number of processes from 2 to 32. The left part of Fig. 2 shows that the additional process algorithm mostly outperforms the other two implementations. Although it performs reasonably well up to 16 processes, the performance drops sharply for 32 processes. This performance drop can be due to either of the two reasons: the management process could be overwhelmed by the increasing number of (small) requests with increasing number of processes, or the file system or the hard drive could get congested with increasing number of requests.

Which of these two contributes stronger to the performance degradation observed for 32 processes can be answered by looking at the numbers of the collective write operations in the right part of Fig. 2. This graph shows the same behavior for the additional process implementation. However, as collective operations in this model only require one request to the management process per collective function call, but generate the same number of I/O requests to the hard drive and the file system, we are confident that the performance drop observed in the previous measurements is not due to a congestion at the additional process managing the shared file pointer.

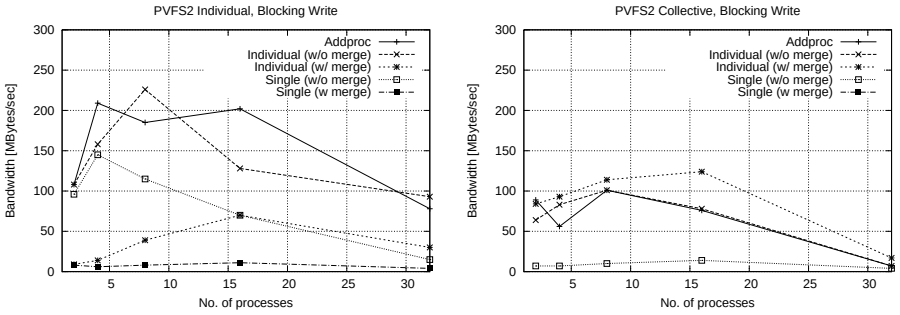

Fig. 2. Performance Comparison for Individual, Blocking write operation (left) and Collective Blockgin write-operations (right) for varying numbers of processors.

3.2 Results Using an EXT3 File System

In this subsection, the benchmark is run using all four write operations of the MPI specification for shared file pointer operations using all the three algorithms described earlier on a local file system using Ext3. The test machine utilized

consists of eight single core AMD Opteron processors. Since the Ext3 file system supports file locks, the performance of ROMIO over Ext3 could be compared with our three implementations. In each case, data from 1 GB to 10 GB is written among eight processes. The left part of Fig. 3 gives the performance of the individual and blocking write routine over Ext3 file system, while the right part shows the performance of the collective write routine. ROMIO performs better than the three new algorithms for smaller amounts of data. However, as the amount of data written increases, the performance of ROMIO decreases. The additional process implementation performance in these scenarios is better than ROMIO. There could be various reasons behind the additional process algorithm not being able to completely outperform ROMIO, such as

- All the evaluation has been done using eight processes. Marvin is a shared memory system which has eight processors. In the case of the additional process algorithm, when an additional process is spawned (the ninth in this case), the performance might decrease as nine processes run on eight processors.
- Secondly, when the dynamic process management feature is used, Open MPI restricts the communication among the original process group and the newly spawned processes from using shared memory communication. Instead, it uses the next available communication protocol, which is TCP/IP in case of Marvin. Hence, slow communication between the local and the remote groups of processes could be a factor affecting the performance of the additional process algorithm.
- Thirdly, in case of ROMIO, since the hidden file is small, it might be cached by the file system or the RAID controller. Hence, processes accessing the hidden file might not touch the hard drive, but access the hidden file from the cache to read and update the shared file pointer offset value.

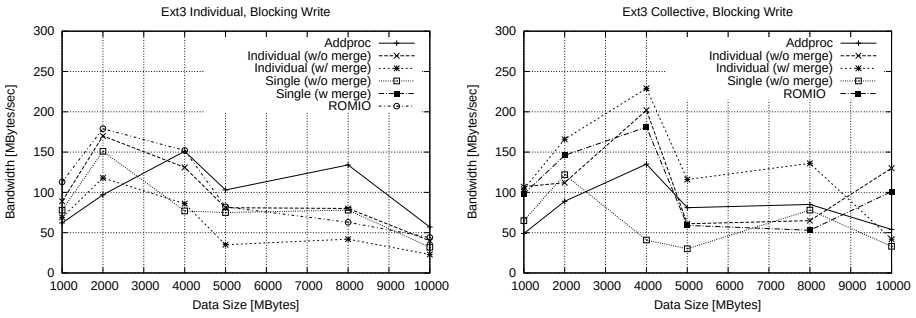

Fig. 3. Performance Comparison for Individual, Blocking write operation (left) and Collective Blockgin write-operations (right) on the ext3 file system.

4 Summary

This study aims at designing portable and optimized algorithms to implement shared file pointer operations of the MPI I/O specification. We have developed, implemented and evaluated three new algorithms for I/O operations using shared file pointers. Our evaluation shows, that the algorithms often outperform the current version in ROMIO, although there are notable exceptions. However, the algorithms do represent alternative approaches which could be used in case the file system does not support file locking. The 'additional process' algorithm is most general of the three approaches, and we plan to further extend it by overcoming the restrictions discussed in section 3, most notably having the management process being executed in an already existing process, such as `mpirun`. Furthermore, as of today, we have not yet evaluated the difference between the single file and the individual file approaches in case of multiple MPI files, and the impact on the Metadata server of the file system. The ultimate goal is to provide a collection of algorithms and give the end-user the possibility to select between those algorithms using appropriate hints.

References

1. Message Passing Interface Forum: MPI-2: Extensions to the Message Passing Interface, http://www.mpi-forum.org, (1997).
2. May, J.: Parallel I/O for High Performance Computing. Morgan Kaufmann Publishers (2003).
3. Corbett, P.F., Feitelson, D.G.: Design and implementation of the Vesta parallel file system. In: Proceedings of the Scalable High-Performance Computing Conference. 63 –70, (1994).
4. Thakur, R., Gropp, W., Lusk, E.: Data Sieving and Collective I/O in ROMIO. In: FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation, Washington, DC, USA, IEEE Computer Society 182 (1999).
5. Thakur, R., Ross, R., Lusk, E., Gropp, W.: Users Guide for ROMIO: A High Performance, Portable MPI-IO Implementation. Argonne National Laboratory (2004).
6. Latham, R., Ross, R., Thakur, R.: Implementing MPI-IO Atomic Mode and Shared File Pointers Using MPI One-Sided Communication. Int. J. High Perform. Comput. Appl. **21**(2) 132–143, (2007).
7. Yu, W., Vetter, J., Canon, R.S., Jiang, S.: Exploiting lustre file joining for effective collective io. In: CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, Washington, DC, USA, IEEE Computer Society 267–274, (2007).
8. Becker, D., Rabenseifner, R., Wolf, F.: Timestamp Synchronization for Event Traces of Large-Scale Message-Passing Applications. In: Proceedings of EuroPVM/MPI. Lecture Notes in Computer Science, Springer-Verlag 315–325, (2007).