The Impact of File Systems on MPI-IO Scalability*

Rob Latham, Rob Ross, and Rajeev Thakur

Argonne National Laboratory, Argonne, IL 60439, USA
{robl,rross,thakur}@mcs.anl.gov

Abstract. As the number of nodes in cluster systems continues to grow, leveraging scalable algorithms in all aspects of such systems becomes key to maintaining performance. While scalable algorithms have been applied successfully in some areas of parallel I/O, many operations are still performed in an uncoordinated manner. In this work we consider, in three file system scenarios, the possibilities for applying scalable algorithms to the many operations that make up the MPI-IO interface. From this evaluation we extract a set of file system characteristics that aid in developing scalable MPI-IO implementations.

1 Introduction

The MPI-IO interface [10] provides many opportunities for optimizing access to underlying storage. Most of these opportunities arise from the interface's ability to express noncontiguous accesses, the collective nature of many operations, and the precise but somewhat relaxed consistency model. Significant research has used these features to improve the scalability of MPI-IO data operations. Implementations use two-phase [13], data sieving [14], and data shipping [11], among others, to efficiently handle I/O needs when many nodes are involved.

On the other hand, little attention has been paid to the remaining operations, which we will call the *management operations*. MPI-IO semantics provide opportunities for scalable versions of open, close, resize, and other such operations. Unfortunately, the underlying file system API can limit the implementation's ability to exploit these opportunities just as it does in the case of the I/O operations.

We first discuss the opportunities provided by MPI-IO and the potential contributions that the parallel file system can make toward an efficient, scalable MPI-IO implementation. We then focus specifically on the issue of providing scalable management operations in MPI-IO, using the PVFS2 parallel file system as an example of appropriate support. We also examine the scalability of common MPI-IO management operations in practice on a collection of underlying file systems.

* 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.

1.1 MPI-IO Opportunities

Implementations can take advantage of three aspects of the MPI-IO specification to maximize scalability: semantics, noncontiguous I/O, and collective functions.

MPI-IO provides more relaxed consistency semantics than the traditional POSIX [6] interface provides. These semantics are relaxed on two fronts: in terms of the scope of the consistency (just the processes in the communicator) and the points in time at which views from different processes are synchronized. Under the default MPI-IO semantics, simultaneous writes to the same region yield an undefined result. Further, writes from one process are not immediately visible to another. Active buffering with threads [9], for example, takes advantage of MPI-IO consistency semantics to hide latency of write operations.

Additionally, MPI datatypes may be used to describe noncontiguous regions both in file and in memory, providing an important building block for efficient access for scientific applications. Several groups have implemented support for efficient noncontiguous I/O, including listless I/O [15], data sieving [14], list I/O [3], and datatype I/O [2].

MPI-IO also affords many opportunities for scalable implementations through collective operations. These collective functions enable the implementation to use scalable communication routines and to reorganize how operations are presented to the file system. The focus of optimizations of collective MPI-IO routines to this point has been on read and write operations. Optimizations such as two-phase have had a significant impact on the performance of collective I/O, particularly at large scale.

1.2 MPI-IO with POSIX and NFS

POSIX is not the ideal underlying interface for MPI-IO for three reasons. First, the `readv`, `writev`, and `lio_listio` calls are not efficient building blocks for noncontiguous I/O. The `readv` and `writev` calls only allow describing noncontiguous regions in memory, while these often occur in the file as well. The `lio_listio` API does allow for multiple file regions, but the language for describing them through the API is verbose, leading to descriptions larger than the data itself. Data sieving [14] is not so much an optimization as a workaround for these shortcomings; it is often more efficient to read an entire region containing the data of interest, discarding much of that data, than to construct and use a noncontiguous request with the POSIX noncontiguous functions.

The POSIX stateful model is also problematic. The `open`, `read`, `write`, `close` model of access requires that all processes desiring to access files directly perform many system calls. The file descriptor returned by `open` has meaning to just one client. Hence, file descriptors cannot be shared among all processors. Each client must make the `open` system call. In a large parallel program, opening a file on a parallel file system can put a large strain on the servers as thousands of clients simultaneously call `open`.

NFS provides an interesting contrast to POSIX. Clients access NFS file systems using the same functions as POSIX and must deal with same issues with

file descriptors. Although the API is the same, however, the consistency semantics are quite different and impose an additional set of problems. Because clients cache data aggressively and without synchronization among clients, it is difficult to predict when writes from one client will be visible to another. Metadata caching further complicates parallel I/O: when one process modifies the file size or file attributes, it is difficult to know when those modifications will be visible to the other processes. The NFS consistency semantics work well in the serial environment for which they were designed, but they are a poor fit for parallel I/O.

MPI-IO implementations can function on top of a wide variety of file systems, but the underlying file system can greatly help the implementation achieve real scalability, particularly if it addresses the problems outlined above.

1.3 Parallel File System Building Blocks

A parallel file system can provide three fundamentals to aid in a scalable MPI-IO implementation:

- Efficient noncontiguous I/O support
- Consistency semantics closely matching the MPI-IO model
- Client-independent references to files

Efficient noncontiguous I/O support in the file system has been a focus of a great deal of recent research [15]. The datatype I/O concept [3] in particular provides an efficient infrastructure for noncontiguous I/O in MPI-IO, with similar concepts seen in the View I/O [7] work.

One of the most significant influences on performance of a file system, both in data and metadata operations, is the consistency semantics implemented by the file system. For example, the POSIX consistency semantics require essentially sequential consistency of operations. Enforcing these semantics can result in high overhead and reduced parallelism. The NFS consistency semantics, on the other hand, require no additional overhead because there is no guarantee of consistency between clients. Thus, the consistency semantics drive caching policies, dictating how clients can cache data and when they must synchronize.

The "nonconflicting write" semantics of PVFS and PVFS2 are an ideal building block from the MPI-IO perspective. A write operation is nonconflicting with another write operation if no part of the two operations overlap (interleaved requests can still be nonconflicting). If two processes perform nonconflicting write operations, then all other processes will see the data from the writers after their respective writes have completed. If two clients write data to the same region of a file (i.e., a conflicting write), the result is undefined. The file system counts on the MPI-IO implementation handling any additional consistency requirements.

While relaxed data consistency can improve scalability, *metadata* consistency semantics have an impact on scalable optimizations as well. Some mechanism has to ensure that all clients have a consistent view of metadata from the MPI-IO perspective. For file systems such as NFS, all clients end up performing the same

operations because the MPI-IO implementation cannot control caching, limiting our ability to implement scalable operations. Thus, it is important to have not just relaxed consistency semantics but *appropriate* consistency semantics, and the right hooks to control the data and metadata caches.

A third mechanism that parallel file systems can use to achieve high performance is client-independent references to files. As opposed to the POSIX file descriptor, which has meaning only to one client, these references can be used by any client to refer to a file. By sharing these file references among all clients, programs place fewer demands on the file system. As we will see, this feature can significantly improve the performance of MPI-IO management operations.

Table 1. MPI-IO Management Operations (all clients call)

Function	Collective?	No. of FS Operations		
		NFS	POSIX	PVFS2
`MPI_File_get_size`	no	O(n)	O(n)	O(n)
`MPI_File_seek`	no	O(n)	O(n)	—
`MPI_File_delete`	no	O(1)	O(1)	O(1)
`MPI_File_open`	yes	O(n)	O(n)	O(1)
`MPI_File_close`	yes	O(n)	O(n)	O(1)
`MPI_File_sync`	yes	O(n)	O(n)	O(1)
`MPI_File_set_size`	yes	O(n)	O(1)	O(1)
`MPI_File_preallocate`	yes	O(1)	O(1)	O(1)
`MPI_File_set_info`	yes	O(n)	O(n)	O(1)
`MPI_File_set_view`	yes	O(n)	O(n)	O(1)
`MPI_File_get_position_shared`	no	—	—	—
`MPI_File_seek_shared`	yes	—	—	—

2 MPI-IO Management Operations

We can roughly split the MPI-IO operations into two groups: operations that read or write data, and operations that do not. We call this second group MPI-IO management operations. Table 1 lists the management operations that interact with the file system (calls such as `MPI_File_get_position` generally require no corresponding file system operations, using cached data instead). The point of the table is to help shed light on the options for creating scalable implementations of the MPI-IO functions.

Some functions, such as `MPI_File_get_size`, are not collective and thus cannot be optimized – every process wanting the file size would need to make the

call, or the application programmer would need to synchronize, make a single MPI-IO call, and then broadcast the result. Little can be done in this case. Functions such as `MPI_File_delete` fall into the same category, except that it is generally assumed that the application programmer will perform synchronization and make only one call, since calling this function many times would likely result in success on one process and failure on all others.

Next is a core set of collective management functions that are often used in MPI-IO applications, including `MPI_File_open` and `MPI_File_close` among others. The stateful nature of the POSIX and NFS APIs requires that open and close file system operations be performed on all processes. Likewise, the `fsync` operation that is the interface for synchronizing data in the POSIX and NFS APIs flushes changes only on the local node, requiring a file system operation per node (generally implemented by calling `fsync` on each process for simplicity). The NFS metadata caching makes relying on NFS file sizes problematic, so ROMIO chooses to call `ftruncate` on all processes. This situation could perhaps be avoided by maintaining file size data within ROMIO rather than relying on the file system, but implementing such a feature would touch many other calls.

For example, the file truncation function `MPI_File_set_size` is a collective operation that may be made scalable even under the POSIX API. A single POSIX `ftruncate` operation may be performed and the result broadcast to remaining processes (see Figure 1). We can use MPI-IO semantics here to further improve scalability: `MPI_File_set_size` is treated like a write operation, so the caller must synchronize client calls if that is desired. We note, however, that this cannot be done in the NFS environment, where metadata is not kept consistent between nodes; in that case we must perform the truncate on all nodes.

```
if (rank == 0) {
        /* perform the truncate on one node */
        ret = ftruncate(fd, size);
        MPI_Bcast(&ret, 1, MPI_INT, 0, comm);
} else {
        /* the result is broadcast to the other processors */
        MPI_Bcast(&ret, 1, MPI_INT, 0, comm);
}
/* at this point, all processors know the status of the ftruncate
 * call, even though only one processor actually sent the request */
```

Fig. 1. Scalable `MPI_File_set_size` (pseudocode)

The `MPI_File_sync` function offers a slightly different example. It, too, is a collective operation but has the added property that no outstanding write operations should be in progress on any process. Figure 2 demonstrates one possible scalable implementation. By using `MPI_Reduce` (or `MPI_Gather`), we can ensure that all processes have performed their write operations before one process initiates the flush. This scheme assumes there will be no client-side caching of data. In the POSIX and NFS environments, where local caching occurs and the

`fsync` flushes only local buffers, calls must be performed by all clients to ensure that changes on all nodes make it out to disk.

```
MPI_Reduce(&dummy1, &dummy1, 1, MPI_INT, MPI_SUM, 0, comm);

if (rank == 0) {
        ret = fsync(fd);
        MPI_Bcast(&ret, 1, MPI_INT, 0, comm);
} else {
        MPI_Bcast(&ret, 1, MPI_INT, 0, comm);
}
/* at this point, all processors know the status of the fsync call,
 * even though only one processor actually sent the request */
```

Fig. 2. Scalable `MPI_File_sync` (pseudocode). We do not want to sync until we know all other processors have finished writing. The call to `MPI_Reduce` ensures that all processes have completed any outstanding write operations. In this example, rank 0 will not call fsync until all other processors have sent rank 0 an integer.

The `MPI_File_set_info` and related calls are interesting because they may or may not require file system operations, depending on the hints passed in and supported by the file system. For example, setting the `MPIO_DIRECT_READ` option on file systems that support it (e.g., XFS) would require a file system call from each process.

Moreover, because no commonly used file systems support shared file pointers, the implementation in ROMIO uses a shared file approach to store the pointer. There may be more scalable options for implementing this support; this is an open research area.

3 File System Support: PVFS2

The new PVFS2 parallel file system [12] is a good example of providing efficient building blocks for scalable MPI-IO. It is no accident that PVFS2 is well suited for MPI-IO: it was expressly designed with such a goal in mind.

We took advantage of several PVFS2 features to optimize our MPI-IO implementation. Naturally, these features resemble the points laid out in Section 1.3:

- support for arbitrary noncontiguous I/O patterns
- consistency semantics well-suited for MPI-IO
- client-independent handles
- no client-side cache

Support for noncontiguous access in PVFS2, similar to the datatype I/O prototype in PVFS1 ([2],[4]), provides the necessary API for efficient independent I/O. Nonconflicting write consistency semantics, which leave the results of byte-overlapped concurrent writes undefined, provide sufficient consistency for building the nonatomic MPI-IO semantics.

Opaque, client-independent file references allow open operations to be performed scalably, and the stateless nature of the file system means that close operations are also trivial (a single synchronize operation is performed to flush all changes to disk).

For example, only one process in a parallel program has to perform actual PVFS2 function calls to create files (see Figure 3). One process performs a lookup, creating the file if it does not exist. The PVFS2 server responds with a reference to the file system object. The client then broadcasts the result to the other clients. `MPI_File_set_size` is another win: one client resizes the file and then broadcasts the result to the other clients. Servers experience less load because only one request comes in. The same approach is used for `MPI_File_sync` (Figure 2). Only one process has to ask the file system to flush data. The result is the same: all I/O servers write out their caches, but they have to handle only one request each to do so.

```
/* PVFS2 is stateless: clients perform a 'lookup' operation
 * to convert a path into a handle.  This handle can then
 * be passed around to all clients.  */

if (rank == 0) {
       ret = PVFS_sys_lookup(fs_id, path_name,
                credentials, &response, PVFS2_LOOKUP_LINK_FOLLOW);
       if (ret == ENOENT) {
                ret = PVFS_sys_create(name, parent, attribs,
                        credentials, NULL, &response);
       }
}
MPI_Bcast(&ret, 1, MPI_INT, 0, comm);
MPI_Bcast(&response, 1, MPI_INT, 0, comm);

/* now all processors know if the lookup succeeded, and
 * if it did, the handle for the entity   */
```

Fig. 3. Scalable open for PVFS2 (heavily simplified). In a real implementation, one could create an MPI datatype to describe the handle and error code and perform just one `MPI_Bcast`. We perform two for simplicity.

4 Results

To evaluate MPI-IO implementations, we performed experiments on the Jazz cluster at Argonne National Laboratory [8], the ALC cluster at Lawrence Livermore National Laboratory [1], and the DataStar cluster at NPACI/SDSC [5]. Jazz users have access to two clusterwide file systems: NFS-exported GFS volumes and PVFS (version 1). Additionally, we temporarily deployed PVFS2 across a subset of available compute nodes for testing purposes. The ALC cluster has a Lustre file system and the DataStar cluster has GPFS: we have included them for reference, even though we do not discuss their design elsewhere in this paper. In these tests, PVFS2 ran with 8 I/O servers, one of which also acted

as a metadata server. For fairness in these tests, the PVFS2 servers used TCP over Fast Ethernet. We used a CVS version of MPICH2 from mid-April 2004, including the ROMIO MPI-IO implementation (also from mid-April).

In the first experiment, we created 1,000 files in an empty directory with `MPI_File_open` and computed the average time per create. Table 2 summarizes the results. PVFS2 – the only stateless file system in the table – achieves consistent open times as the number of clients increased. Additionally, the average time to create a file on PVFS2 is an order of magnitude faster than the time it takes to do so on any of the other file systems. From a scalability standpoint, the NFS+GFS file system performs remarkably well. File creation may be relatively expensive, but as the number of clients increases, the cost to open a file remains virtually constant. PVFS1 demonstrates poor scalability with the number of clients, because PVFS1 each client must open the file to get a file descriptor (PVFS1 is stateful in this regard). With more clients, the metadata server has to handle increasingly large numbers of requests. Thus, the metadata server becomes a serialization point for these clients, and the average time per request goes up. While Lustre and GPFS both outperform PVFS1, they too must perform an open on each client. The time per operation increases as the number of clients and the demand placed on the file system increases significantly.

Table 2. Results: A Comparison of several cluster file systems(milliseconds)

	Create					
No. of Clients	NFS+GFS	Lustre	GPFS	PVFS1	S-PVFS1	PVFS2
1	3.368	8.585	16.38	41.78	-	18.82
4	178.1	51.62	29.77	221.8	-	18.73
8	191.6	56.68	45.80	292.4	-	22.02
16	176.6	67.03	280.2	241.1	-	20.66
25	183.0	146.8	312.0	2157	-	19.05
50	204.1	141.5	400.6	2447	-	24.73
75	212.7	231.2	475.3	3612	-	24.82
100	206.8	322.2	563.9	1560	-	28.14
128	204.0	463.3	665.8	1585	-	32.94
	Resize					
1	0.252	1.70	7.0	1.26	1.37	0.818
4	3.59	05.39	14.4	2.23	1.54	0.823
8	1.75	13.49	36.0	3.25	1.44	0.946
16	14.88	29.7	36.6	2.75	1.86	0.944
25	36.5	66.0	35.7	25.0	4.07	0.953
50	1960	113	39.5	16.2	2.02	1.11
75	2310	179	43.5	15.0	2.62	1.26
100	4710	233	40.5	19.1	3.10	1.46
128	2820	254	42.4	18.6	3.38	1.07

In the second experiment, we opened a file and then timed how long it took to perform 100 calls to `MPI_File_set_size` on one file with a random `size` parameter (ranging between 0 and `RAND_MAX`. We then computed the average time for one resize operation. Table 2 summarizes our results. The file-based locking of GFS clearly hurts resize performance. The GFS lock manager becomes the serialization point for the resize requests from the clients, and performance degrades drastically as the number of clients increases. The NFS client-side caches mean we must resize the file on each client to ensure consistency, so we cannot use the scalable techniques outlined earlier. Again, as with the create test, we see Lustre's performance getting worse as the number of clients increases. GPFS performance appears virtually independent of the number of clients; it would be interesting to know what approach GPFS takes to achieving scalable operations for this case. The PVFS column shows performance without the scalable algorithm from Figure 1. We modified the ROMIO PVFS1 resize routine to use the more scalable approach(the S-PVFS1 column) . Using the algorithm shown in Figure 1, both PVFS1 and PVFS2 both show consistent performance. The small increase in time as the number of clients increases can be attributed to the increased synchronization time at the reduce and the time taken to broadcast the results to a larger number of processors.

5 Conclusions and Future Directions

Many opportunities exist for optimizing MPI-IO operations, even in general purpose file systems. Some of these opportunities have been heavily leveraged, in particular those for collective I/O. Others require additional support from the file system. In this work we have described the collection of MPI-IO operations and categorized these based on the ability of the MPI-IO implementor to optimize them given specific underlying interfaces. We have pointed out some characteristics of file system APIs and semantics that more effectively serve as the basis for MPI-IO implementations: support for noncontiguous I/O, better consistency semantics, client-independent file references, a stateless I/O model, and a caching model that allows a single file system operation to sync to storage (in the case of PVFS2, no client-side caching at all). By building parallel file systems with these characteristics in mind, MPI-IO implementations can leverage MPI collective communication and achieve good performance from the parallel file system even as the number of clients increases.

In future work, we will examine scalable support for the shared file pointer and atomic access modes of the MPI-IO interface. While inherently less scalable than private file pointers and the more relaxed default semantics, these are important components in need of optimization. It is not clear whether file system support for shared file pointers and the more strict atomic data mode is warranted or if this support should be provided at the MPI-IO layer. Additionally, we continue to examine options for more aggressively exploiting the I/O semantics through client-side caching at the MPI-IO layer.

6 Acknowledgments

We gratefully acknowledge use of "Jazz," a 350-node computing cluster operated by the Mathematics and Computer Science Division at Argonne National Laboratory as part of its Laboratory Computing Resource Center. Jianwei Li at Northwestern University and Tyce McLarty at Lawrence Livermore National Laboratory contributed benchmark data. We thank them for their efforts.

References

1. ALC, the ASCI Linux Cluster. `http://www.llnl.gov/linux/alc/`.
2. A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp. Efficient structured data access in parallel file systems. In *Proceedings of Cluster 2003*, Hong Kong, November 2003.
3. Avery Ching, Alok Choudhary, Kenin Coloma, Wei keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O accesses through MPI-IO. In *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 104–111, Tokyo, Japan, May 2003. IEEE Computer Society Press.
4. Avery Ching, Alok Choudhary, Wei keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, September 2002.
5. IBM DataStar Cluster. `http://www.npaci.edu/DataStar/`.
6. IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.
7. Florin Isaila and Walter F. Tichy. View I/O: Improving the performance of noncontiguous I/O. In *Proceedings of IEEE Cluster Computing Conference, Hong Kong*, December 2003.
8. LCRC, the Argonne National Laboratory Computing Project. `http://www.lcrc.anl.gov`.
9. Xiasong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Improving MPI IO output performance with active buffering plus threads. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, April 2003.
10. MPI-2: Extensions to the message-passing interface. The MPI Forum, July 1997.
11. Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Proceedings of Supercomputing 2001*, November 2001.
12. The Parallel Virtual File System, version 2. `http://www.pvfs.org/pvfs2`.
13. Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.
14. Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of SC98: High Performance Networking and Computing*. ACM Press, November 1998.
15. Joachim Worringen, Jesper Larson Traff, and Hubert Ritzdorf. Fast parallel noncontiguous file access. In *Proceedings of SC2003: High Performance Networking and Computing*, Phoenix, AZ, November 2003. IEEE Computer Society Press.