Self-Consistent MPI-IO Performance Requirements and Expectations

William D. Gropp[1], Dries Kimpe[2], Robert Ross[3], Rajeev Thakur[3], and Jesper Larsson Träff[4]

[1] Computer Science Department, University of Illinois at Urbana-Champaign Urbana, IL 61801, USA
wgropp@uiuc.edu
[2] Scientific Computing Research Group, K. U. Leuven Celestijnenlaan 200A, B-3001 Leuven, Belgium
dries.kimpe@cs.kuleuven.be
[3] Mathematics and Computer Science Division Argonne National Laboratory, Argonne, IL 60439, USA
{rross, thakur}@mcs.anl.gov
[4] NEC Laboratories Europe, NEC Europe Ltd. Rathausallee 10, D-53757 Sankt Augustin, Germany
traff@it.neclab.eu

Abstract. We recently introduced the idea of self-consistent performance requirements for MPI communication. Such requirements provide a means to ensure consistent behavior of an MPI library, thereby ensuring a degree of performance portability by making it unnecessary for a user to perform implementation-dependent optimizations by hand. For the collective operations in particular, a large number of such rules could sensibly be formulated, without making hidden assumptions about the underlying communication system or otherwise constraining the MPI implementation. In this paper, we extend this idea to the realm of parallel I/O (MPI-IO), where the issues are far more subtle. In particular, it is not always possible to specify performance *requirements* without making assumptions about the implementation or without *a priori* knowledge of the I/O access pattern. For such cases, we introduce the notion of performance *expectations*, which specify the desired behavior for good implementations of MPI-IO. I/O performance requirements as well as expectations could be automatically checked by an appropriate benchmarking tool.

1 Introduction

In [7], we introduced the notion of self-consistent performance requirements for MPI implementations. Such requirements relate various aspects of the semantically strongly interrelated MPI standard to each other. The requirements are based on meta-rules, stating for instance that no MPI function should perform worse than a combination of other MPI functions that implement the same functionality, that no specialized function should perform worse than a more general

function that can implement the same functionality, and that no function with weak semantic guarantees should perform worse than a similar function with stronger semantics. In other words, the *library-internal* implementation of any arbitrary MPI function in a given MPI library should not perform any worse than an *external (user)* implementation of the same functionality in terms of (a set of) other MPI functions. Otherwise, the performance of the library-internal MPI implementation could trivially be improved by replacing it with an implementation based on the external user implementation. Such requirements, when fulfilled, would ensure consistent performance of interrelated parts of MPI and liberate the user from having to perform awkward and non-portable optimizations to cope with deficiencies of a particular implementation. For the MPI implementer, self-consistent performance requirements serve as a sanity check.

In this paper, we extend this idea to the MPI-IO part of MPI [1, Chapter 7]. The I/O model of MPI is considerably more complex than the communication model, with performance being dependent to a much larger extent on external factors beyond the control of both the application and the MPI library. Also, the I/O access patterns of different processes are not known beforehand. As a result, in some instances, we can only formulate performance *expectations* instead of performance *requirements*. Performance expectations are properties that are expected to hold most of the time and would be desirable for an MPI implementation to fulfill (from the perspective of the user and for performance portability).

We use the following notation in the rest of this paper. The performance relationship that MPI function $\mathsf{MPI\_}A(n)$ should perform no worse than function(s) $\mathsf{MPI\_}B(n)$ for total I/O volume n is expressed semi-formally by $\mathsf{MPI\_}A(n) \preceq \mathsf{MPI\_}B(n)$. To distinguish between requirements and expectations, we use the notation $\mathsf{MPI\_}A(n) \subseteq \mathsf{MPI\_}B(n)$ to indicate the performance relationship that MPI function $\mathsf{MPI\_}A(n)$ is expected to perform no worse than function $\mathsf{MPI\_}B(n)$ for total I/O volume n.

One value in defining expectations more formally is that they can suggest the need for additional hints to help an MPI implementation achieve the user's expectation of performance. In Section 5, we illustrate this point by describing some situations where achieving performance expectations may require additional information and suggest new standard hints that could be adopted in revisions of the MPI standard. Furthermore, this approach to defining standard hints is arguably a better approach than attempting to standardize common practice, as was done for the I/O hints in MPI 2.0. The formal definitions also help guide the development of tests to ensure that implementations meet user expectations.

As with the set of self-consistent performance requirements for MPI communication, the MPI-IO requirements and expectations can, in principle, be automatically checked with an appropriate, configurable benchmark and experiment-management and mining tool. We have not developed such a tool so far.

2 MPI-IO

MPI-IO [1] is an interface for parallel file I/O defined in the spirit of MPI and building on the same key concepts. Access patterns and data layouts in files are described by (derived) datatypes, and data is then sent (written) from memory into a region or regions in file, or received (read) from file into memory.

The MPI-IO model can be analyzed along different dimensions [1, Chapter 7, page 204]. File I/O operations can be classified as

1. independent vs. collective,
2. blocking vs. nonblocking, and
3. blocking collective vs. split collective

Positioning within a file can be done via

1. explicit offsets,
2. individual file pointers, or
3. shared file pointers.

These different classes of file I/O operations and positioning mechanisms have different semantics and performance characteristics. In addition to these I/O modes, we must also consider the side effects of other MPI-IO calls, such as those that change the bytes of a file that a process may access (by defining *file views*) or supply (implementation- and system-dependent) *hints* that may impact underlying behavior. For example, a call to MPI_File_set_info that changes the cb_nodes hint for a file could have a dramatic impact on subsequent collective I/O on that same file by limiting the number of processes that actually perform I/O. This example also helps explain the difficulty in defining performance requirements for MPI-IO.

Further details about MPI-IO can be found in [1, 2].

3 Requirements versus Expectations

We define both requirements and expectations for MPI-IO performance. Performance *requirements* are conditions that a good MPI-IO implementation should be able to fulfill. In some cases, however, it is not possible to specify requirements for a variety of reasons discussed below. For such cases, we define performance *expectations*, which would be desirable for an implementation to fulfill.

For example, it is tempting to specify that collective I/O should perform no worse than independent I/O. However, the I/O access pattern specified by the collective I/O function is not known unless the implementation analyzes the request, which may require communication among processes. If, after analysis, the implementation determines that collective I/O optimization is not beneficial for this request, and uses independent I/O instead, the cost of the analysis is still incurred, and the collective I/O function is slower than if the user had directly called the equivalent independent I/O function. Also, I/O performance is influenced by characteristics of the (parallel) file system, only some of which

can be controlled by (or are even visible to) the MPI implementation. This fact also makes it difficult to specify requirements in some cases.

The situation is further complicated by hints. In the MPI-IO model, the interaction with the file system can be influenced through hints that are supplied to MPI_File_open, MPI_File_set_info, or MPI_File_set_view calls. The MPI standard defines a number of such hints that can deeply affect performance. It is the user's responsibility to use these hints sensibly, which often requires knowledge of the underlying file system. It is easy to supply hints that have a negative effect on performance. For example, Figure 1 shows the effect of an unfortunate choice for the standard-defined cb_buffer_size hint (buffer size for collective buffering) on the performance of the noncontig benchmark [8]. This benchmark generates a regular, strided, nonoverlapping access pattern perfectly suited for collective buffering. However, the optimal value for cb_buffer_size is hard to determine, and depends on many factors such as the number of processes involved and the characteristics of the file system and communication network. In this graph we see that there is a small range of values for which optimal read performance is achieved. In fact, the default value of 4 MB used by the MPI-IO implementation, ROMIO [3, 5], does not happen to fall into that range on this system. Selection of an appropriate cb_buffer_size can be further complicated by the potential for interaction between alignment of these buffers during collective I/O and the granularity of locks in the (parallel) file system, if the file system uses locks.

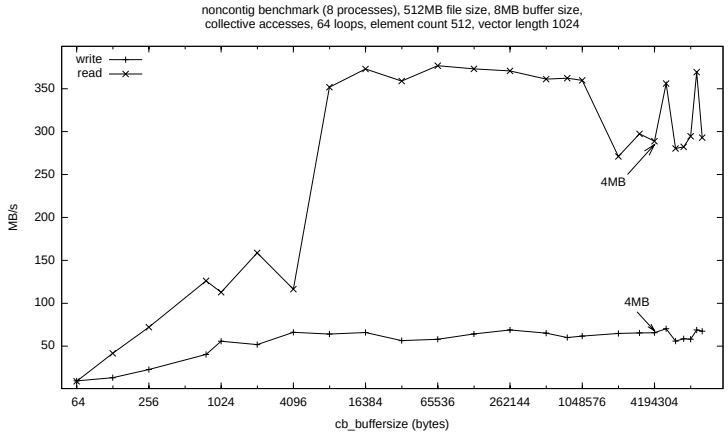


Fig. 1. Effect of the hint cb_buffer_size on the noncontig benchmark

Two features further complicate the matter. First, MPI implementations are allowed to support their own hints in addition to the ones described in the standard. We, of course, cannot say anything about the impact of such hints here. Second, a conforming MPI implementation is allowed to ignore all hints (including standard-defined hints), and their effect may therefore be void. In all,

hints are a nonportable feature (performance wise). We therefore cannot formulate strict performance requirements, but by making assumptions about how an implementation could (or should) sensibly use hint information, we can nevertheless formulate reasonable performance *expectations*. Similar to the performance requirements, such expectations can be stated as rules that can be checked by a suitable tool.

4 Performance Requirements

Where an MPI-IO operation can, in an obvious way, be implemented by other, possibly more general MPI-IO operations, a self-consistent performance requirement states that this alternative implementation should not be faster than the original operation. A comprehensive, but not exhaustive set of such requirements is presented in the following.

An I/O operation with an explicit offset should be no slower than implementing it with a call to MPI_File_seek followed by the corresponding individual file pointer operation.

$$\text{MPI\_File\_}\{\text{read}|\text{write}\}\text{\_at} \preceq \text{MPI\_File\_seek} + \text{MPI\_File\_}\{\text{read}|\text{write}\} \tag{1}$$

A collective I/O operation should be no slower than the equivalent split collective operation.

$$\text{MPI\_File\_}\{\text{read}|\text{write}\}\text{\_all} \preceq \tag{2}$$

$$\text{MPI\_File\_}\{\text{read}|\text{write}\}\text{\_all\_begin} + \text{MPI\_File\_}\{\text{read}|\text{write}\}\text{\_all\_end} \tag{3}$$

A blocking I/O operation should be no slower than the corresponding nonblocking operation followed by a wait.

$$\text{MPI\_File\_}\{\text{read}|\text{write}\} \preceq \text{MPI\_File\_}\{\text{iread}|\text{iwrite}\} + \text{MPI\_Wait} \tag{4}$$

By the assumption that an operation with weaker semantic guarantees should be no slower (presumably faster) than a similar operation with stronger guarantees, we can formulate the following requirement

$$\text{Write with default consistency semantics} \preceq \text{write with atomic mode} \tag{5}$$

Preallocating disk space for a file by using MPI_File_preallocate should be no slower than explicitly allocating space with MPI_File_write.

$$\text{MPI\_File\_preallocate} \preceq \text{MPI\_File\_write\_}* \tag{6}$$

I/O access using an individual file pointer should be no slower than I/O access using the shared file pointer (because accessing the shared file pointer may require synchronization).

$$\text{MPI\_File\_write} \preceq \text{MPI\_File\_write\_shared} \tag{7}$$

Similarly, collective I/O using an individual file pointer should be no slower than collective I/O using the shared file pointer.

$$\mathsf{MPI\_File\_write\_all} \preceq \mathsf{MPI\_File\_write\_ordered} \tag{8}$$

I/O operations in native file format should be no slower than I/O operations in `external32` format, since `external32` may require conversion, and it provides stronger semantics guarantees (on portability). An MPI implementation where `external32` is faster than native format could be "fixed" by using the same approach to native I/O as in the `external32` implementation, only without performing any data conversion.

$$\mathsf{I/O\ in\ native\ format} \preceq \mathsf{I/O\ in\ external32\ format} \tag{9}$$

5 Performance Expectations

We describe some examples of performance expectations, which for various reasons cannot be mandated as performance requirements.

5.1 Noncontiguous Accesses

MPI-IO allows the user to access noncontiguous data (in both memory and file) with a single I/O function call. The user can specify the noncontiguity by using derived datatypes. A reasonable performance expectation would be that a read or write with a noncontiguous datatype $T(n)$ that specifies t contiguous regions of size n in memory or file is no slower than if the user achieved the same effect by t individual, contiguous reads or writes. Ideally, an implementation should do better.

$$\mathsf{MPI\_File\_\{read|write\}}*(T(n)) \subseteq \underbrace{\mathsf{MPI\_File\_\{read|write\}\_at}(n)}_{t \text{ individual calls}} \tag{10}$$

We cannot state this as a requirement because the performance in such cases can be highly dependent on the access pattern, which must be determined from the memory and file layouts specified. This analysis can itself take some time. If after the analysis it is determined that no optimization is beneficial, and instead multiple contiguous reads/writes should be performed, the above relation will not hold. A good implementation should perform the analysis as efficiently as possible to minimize the overhead.

Figure 2 shows an example where this expectation is not met. Here, for small, sparse accesses (described in Figure 3[a]), multiple contiguous writes perform better than a single noncontiguous write. In this case, the MPI library mistakenly decided to apply the data-sieving optimization [6] to the file access, when in fact multiple contiguous accesses perform better because of the sparsity of the access pattern.

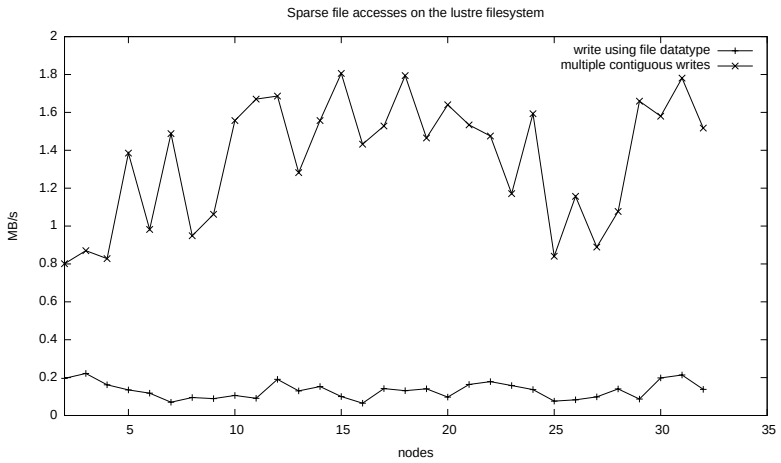
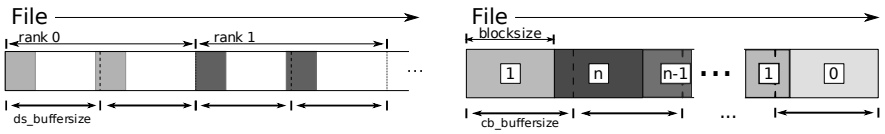


Fig. 2. Using file datatypes versus multiple contiguous accesses



(a) for Figure 2 (b) for Figure 4

Fig. 3. Access patterns (color indicates rank)

5.2 Implications for Hints

These considerations suggest several possible hints that would aid an MPI implementation in achieving the performance expectations. General hints that could be used with any MPI implementation (because they describe general features of the program and data) include:

`write_density` A measure of the ratio between the extent and size of datatypes used for writing to a file. Implementations may want to use independent I/O if the density is low and collective I/O if the density is high.

`read_density` Similar to `write_density`, but for reading.

Specific hints to control a particular MPI implementation can be used when the implementation is unable to meet a performance expectation without additional knowledge. These might include:

`use_only_contig_io` Only use contiguous I/O.

The advantage of such a hint is that the MPI program is more *performance portable*: Rather than replacing one set of MPI calls with a different set everywhere they occur, the hint can allow the MPI implementation to avoid the analysis and simply choose the appropriate method. This places most of the performance tuning at the point where the hint is provided, rather than at each location where file I/O is performed. Currently, ROMIO implements the hints romio_ds_read and romio_ds_write to allow users to selectively disable the use of noncontiguous I/O optimizations, overriding the default behavior. Use of these hints could obtain desired performance in the example shown in Figure 2 without reverting to the use of multiple I/O calls.

5.3 Collective I/O

Collective I/O faces a similar problem. MPI_File_write_all can choose to make use of the collective nature of the function call to merge the file accesses among all participating ranks in order to optimize file access. Most I/O systems perform relatively well on large contiguous requests (compared to small fragmented ones), and, as a result, merging accesses from different processes usually pays off. A user may therefore expect better performance with MPI_File_write_all than with MPI_File_write.

However, merging access patterns will not always be possible (for example, when accesses are not interleaved). When this happens, the implementation of MPI_File_write_all will usually fall back to calling MPI_File_write. Because of the additional synchronization and communication performed in determining the global access pattern, MPI_File_write_all will actually have performed worse compared with directly calling its independent counterpart. In ROMIO, users have the option of using the romio_cb_read and romio_cb_write hints to disable collective I/O optimizations when they know these optimizations aren't beneficial.

Of course, merging access patterns is just one possible optimization. Given the diverse hardware and software encountered in I/O systems, we do not want users to call the independent functions because they happen to perform better on a certain system. This would take away global optimization opportunities from the MPI library, possibly reducing performance on other systems or different MPI implementations. Instead, using suitable hints, the application should be able to provide the MPI-IO implementation with enough information to make sure the collective version does not perform worse than the corresponding independent call.

Therefore, we can only state the following as an expectation.

$$\mathsf{MPI\_File\_\{read\_all|write\_all\}} \subseteq \mathsf{MPI\_File\_\{read|write\}} \tag{11}$$

Figure 4 shows an example where this expectation does not hold. The corresponding access pattern is shown in Figure 3[b]. Here, independent reads and writes are faster than collective reads and writes. The access pattern was specially crafted to trigger bad performance with the two-phase algorithm [4] implemented in ROMIO. This algorithm, which is only enabled if interleaved access

is detected, first analyzes the collective access pattern to determine the first and last offset of the access region. Next, this region is divided among a configurable subset of processes, and each process is responsible for all I/O for its portion of the file. In this example, most of the data that each process accesses is destined for another process, resulting in most data passing over the network twice.

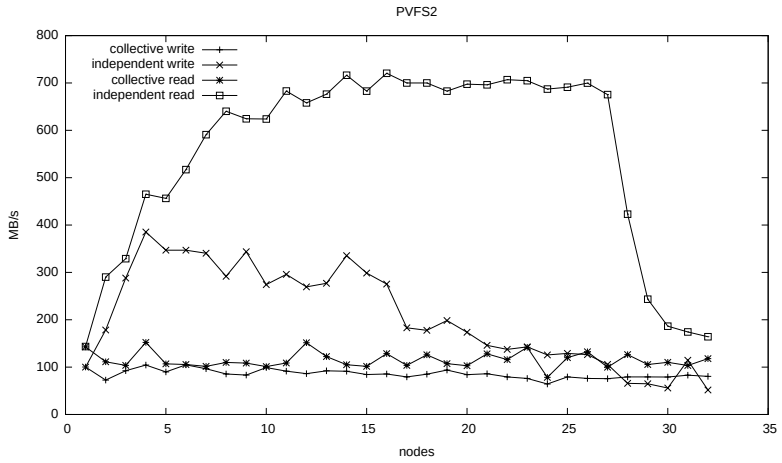


Fig. 4. Collective versus independent file access for a case that is not suitable for collective I/O

Another performance expectation is that an MPI-IO operation for which a "sensible" hint has previously been supplied should perform no worse than the operation without the hint.

6 Conclusions

Codifying performance requirements and expectations provides users and implementers with a common set of assumptions (and goals) for working with MPI-IO. From the user's perspective, these requirements and expectations encourage the use of collective I/O and file views to combine I/O operations whenever possible, to use default consistency semantics when appropriate, to avoid shared file pointers when independent file pointers are adequate, and to use hints for tuning rather than breaking from "best practice." Implementers must strive to meet these expectations wherever possible without the use of additional hints, and also support hints that enable users to correct the implementation's behavior when it goes astray. Implementers and users should strive to improve and standardize the hints used for this purpose across multiple MPI-IO implementations.

At present, some MPI-IO hints are inherently nonportable. The lack of portability is not because conforming implementations are allowed to ignore them, but

because setting them meaningfully requires intimate knowledge of the I/O layers below the MPI implementation. As demonstrated in Section 3, setting such hints carelessly can damage performance. Portable hints only give additional information about the application itself. As an example, consider a hint describing how much data is interleaved in collective accesses. Without this information, an implementation is forced to do additional calculation and communication, possibly making the collective access functions perform worse than their independent counterparts. At the same time, the availability of this information (assuming it is correct) should not degrade performance.

We have shown that some of the performance problems with the MPI-IO functions can be attributed to lack of information—the MPI implementation does not possess enough information to determine the optimal algorithm. However, many problems should be solvable by implementing smarter MPI-IO optimization algorithms. Much work still remains to be done in optimizing MPI-IO functionality!

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 DE-AC02-06CH11357.

References

1. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI – The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998.
2. W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.
3. ROMIO: A high-performance, portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`.
4. R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.
5. R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.
6. R. Thakur, W. Gropp, and E. Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, January 2002.
7. J. L. Träff, W. Gropp, and R. Thakur. Self-consistent MPI performance requirements. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 14th European PVM/MPI Users' Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 36–45. Springer, 2007.
8. J. Worringen, J. L. Träff, and H. Ritzdorf. Fast parallel non-contiguous file access. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 60, Washington, DC, USA, 2003. IEEE Computer Society.