Self-Consistent MPI Performance Requirements*

Jesper Larsson Träff[1], William Gropp[2], and Rajeev Thakur[2]

[1] NEC Laboratories Europe, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
traff@ccrl-nece.de
[2] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{gropp,thakur}@mcs.anl.gov

Abstract. The MPI Standard does not make any performance guarantees, but users expect (and like) MPI implementations to deliver good performance. A common-sense expectation of performance is that an MPI function should perform no worse than a combination of other MPI functions that can implement the same functionality. In this paper, we formulate some performance requirements and conditions that good MPI implementations can be expected to fulfill by relating aspects of the MPI standard to each other. Such a performance formulation could be used by benchmarks and tools, such as *SKaMPI* and *Perfbase*, to automatically verify whether a given MPI implementation fulfills basic performance requirements. We present examples where some of these requirements are not satisfied, demonstrating that there remains room for improvement in MPI implementations.

1 Introduction

For good reasons MPI (the *Message Passing Interface*) [4, 9] comes without a performance model and, apart from some "advice to implementers," without any requirements or recommendations as to what a good implementation should satisfy regarding performance. The main reasons are, of course, that the implementability of the MPI standard should not be restricted to systems with specific interconnect capabilities and that implementers should be given maximum freedom in how to realize the various MPI constructs. The widespread use of MPI over an extremely wide range of systems, as well as the many existing and quite different implementations of the standard, show that this was a wise decision.

On the other hand, for the analysis and performance prediction of applications, a performance model is needed. Abstract models such as LogP [3] and

* This work was supported in part 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.

BSP [13] tend to be too complex (for full applications), too limited, or too abstract to have predictive power. MPI provides much more flexible (but also much more complex) modes of communication than catered to in these models.

An alternative is to use MPI itself as a model and analyze applications in terms of certain basic MPI primitives. This may work well for restricted usages of MPI to, say, the MPI collectives, but full MPI is probably too large to be a tractable model for performance analysis and prediction.

A related consideration is a hard-to-quantify desire for *performance portability*—the desire that an application should, in some qualitative sense, behave the same when ported to a new system or linked with a different MPI library. Detailed, public benchmarks of MPI constructs can help in translating the performance of an application on one system and MPI library to another system with another MPI library [8]. Accurate performance models would also facilitate translation between systems and MPI libraries, but in their absence simple MPI-intrinsic requirements to MPI implementations might serve to guard against the most unpleasant surprises.

MPI has many ways of expressing the same communication (patterns), with varying degrees of generality and freedom for the application programmer. This kind of universality makes it possible to relate aspects of MPI to each other also in terms of the expected performance. This is utilized already in the MPI definition itself, where certain MPI functions are explained in a semi-formal way in terms of other MPI functions.

The purpose of this paper is to discuss whether it is possible, sensible, and desirable to formulate system-independent, but MPI-intrinsic performance requirements that a "good" MPI implementation should fulfill. Such requirements should not make any commitments to particular system capabilities but would enforce a high degree of performance consistency of an MPI implementation. For example, similar optimizations would have to be done for collective operations that are interlinked through such performance rules. Furthermore, such rules, even if relatively trivial, would provide a kind of "sanity check" of an MPI implementation, especially if they could be checked automatically. In this paper, we formulate a number of MPI-intrinsic performance requirements by semi-formally relating different aspects of the MPI standard to each other, which we refer to as *self-consistent performance requirements*. By their very nature the rules can be used only to ensure *consistency*—a trivial, bad MPI implementation could fulfill them as well as a carefully tuned library.

Related work includes quality of service for numerical library components [5, 6]. Because of the complexity of these components, it is not possible to provide the sort of definitive ordering that we propose for MPI communications.

2 General Rules and Notation

We first formulate and discuss a number of general self-consistent MPI performance requirements, presupposing reasonable familiarity with MPI. We consider the following relationships (metarules) between MPI routines:

1. Replacing all communication with the appropriate use of `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` should not reduce the communication time. In the context of MPI-2, this can be applied even to the MPI one-sided communication routines.
2. Subdividing messages into multiple messages should not reduce the communication time.
3. Replacing a routine with a similar routine that provides additional semantic guarantees should not reduce the communication time.
4. For collective routines, replacing a collective routine with several routines should not reduce the communication time. In particular, the specification of most of the MPI collective routines includes a description in terms of other MPI routines; each MPI collective should be at least as fast as that description.
5. For process topologies, communicating with a communicator that implements a process topology should not be slower than using a random communicator.

The first of these requirements provides a formal way to derive relationships between the MPI communication routines—write each routine in terms of an equivalent use of `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait`, and then compare the time taken. In the rest of this paper, we give more specific examples of each of these rules.

We use the notation that

$$\texttt{MPI\_}A(n) \preceq \texttt{MPI\_}B(n) \tag{1}$$

means that MPI functionality A is not slower than B when evoked with parameters resulting in the same amount of communication or computation n. Note that MPI buffers are not always specified in this way. We use p to denote the number of processes involved in a call, and $\texttt{MPI\_}A\{c\}$ for A called on communicator c. As an example

$$\texttt{MPI\_Send}(n) \preceq \texttt{MPI\_Isend}(n) + \texttt{MPI\_Wait} \tag{2}$$

states that an `MPI_Send` call is possibly faster, but at least not slower than a call to `MPI_Isend` with the same parameters followed by an `MPI_Wait` call. In this case, it would probably make sense to require more strongly that

$$\texttt{MPI\_Send}(n) \approx \texttt{MPI\_Isend}(n) + \texttt{MPI\_Wait} \tag{3}$$

which means that the alternatives perform similarly. Quantifying the meaning of "similarly" is naturally contentious. A strong definition would say that there is a small confidence interval such that for any data size n, the running time of the one construct is within the running time of the other with this confidence interval. A somewhat weaker definition could require that the running time of the two constructs is within a small constant factor of each other for any data size n.

3 General Communication

Rule 2 can be made more precise as follows: Splitting a communication buffer of kn units into k buffers of n units, and communicating separately, never pays off.

$$\texttt{MPI\_}A(kn) \preceq \underbrace{\texttt{MPI\_}A(n) + \cdots + \texttt{MPI\_}A(n)}_{k} \tag{4}$$

For an example where this rule is violated with $A = \texttt{Bcast}$, see [1, p. 68].

Similarly, splitting possibly structured data into its constituent blocks of fixed size k should also not be faster.

$$\texttt{MPI\_}A(kn) \preceq \underbrace{\texttt{MPI\_}A(k) + \cdots + \texttt{MPI\_}A(k)}_{n} \tag{5}$$

One might be able to elaborate this requirement into a formal requirement for the performance of user-defined MPI datatypes, but this issue would require much care.

Note that many MPI implementations will violate Rule 2 and (4) because of the use of eager and rendezvous message protocols. An example is shown in Figure 1. A user with a 1500-byte message will achieve better performance on this system by sending two 750-byte messages. This example shows one of the implementation features that competes with performance portability—in this case, the use of limited message buffers. To satisfy Rule 2, an MPI implementation would need a more sophisticated buffer management strategy, but in turn this could decrease the performance of all short messages.

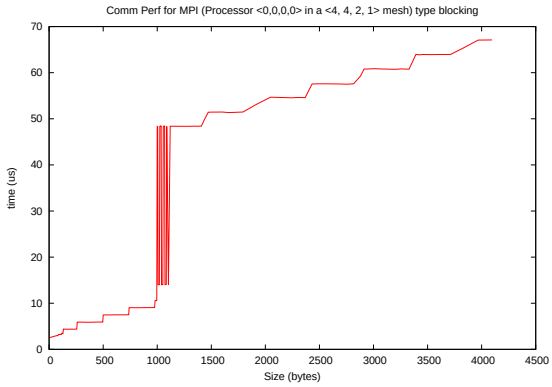

Fig. 1. Measured performance of short messages on IBM BG/L. Note the large jump around 1024 bytes; this is the transition from eager to rendezvous protocol in the MPI implementation.

As an example of Rule 3, we have the following

$$\texttt{MPI\_Send} \preceq \texttt{MPI\_Ssend} \tag{6}$$

Since the synchronous send routine has an additional semantic guarantee (the routine cannot return until the matching receive has started), it should not be faster than the regular send.

4 Collective Communication

The MPI collectives are strongly interrelated semantically, and often one collective can be implemented in terms of one or more other related collectives. A general requirement (metarule) is that a specialized collective should not be slower than a more general collective. Thus, with good conscience, users can be given the advice to always use the most specific collective (which is, of course, exactly the motivation for having so many collectives in MPI).

4.1 Regular Communication Collectives

The following three rules are instances of the metarule that specialized functions should not be slower than more general ones.

$$\texttt{MPI\_Gather}(n) \preceq \texttt{MPI\_Allgather}(n) \tag{7}$$

$$\texttt{MPI\_Scatter}(n) \preceq \texttt{MPI\_Allgather}(n) \tag{8}$$

$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Alltoall}(n) \tag{9}$$

The next rule implements a collective operation in terms of two others. Again the specialized function (`MPI_Allgather`) should not be slower.

$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Gather}(n) + \texttt{MPI\_Bcast}(n) \tag{10}$$

This is not as trivial as it may look. If, for instance, a linear ring algorithm is used for the `MPI_Allgather`, but tree-based algorithms for `MPI_Gather` and `MPI_Bcast`, the relationship will not hold (at least for small n).

A less obvious requirement relates `MPI_Scatter` to `MPI_Bcast`. The idea is to implement the `MPI_Scatter` function, which scatters individual data to each of the p processes, by a broadcast of the combined data of size pn; each process copies out its block from the larger buffer.

$$\texttt{MPI\_Scatter}(n) \preceq \texttt{MPI\_Bcast}(pn) \tag{11}$$

Again this is a nontrivial requirement for small n for MPI libraries with an efficient `MPI_Bcast` implementation and forces an equally efficient implementation of `MPI_Scatter`.

A currently popular implementation of broadcast for large messages is by a scatter followed by an allgather operation [2, 10]. Since this is an algorithm expressed purely in terms of collective operations, it makes sense to require that the native broadcast operation should behave at least as well.

$$\texttt{MPI\_Bcast}(n) \preceq \texttt{MPI\_Scatter}(n) + \texttt{MPI\_Allgather}(n) \tag{12}$$

4.2 Reduction Collectives

The second half of the next rule states that a good MPI implementation should have an `MPI_Allreduce` that is faster than the trivial implementation of reduction to root followed by a broadcast.

$$\begin{aligned} \texttt{MPI\_Reduce}(n) &\preceq \texttt{MPI\_Allreduce}(n) \\ &\preceq \texttt{MPI\_Reduce}(n) + \texttt{MPI\_Bcast}(n) \end{aligned} \tag{13}$$

A similar rule can be formulated for `MPI_Reduce_scatter`.

$$\begin{aligned} \texttt{MPI\_Reduce}(n) &\preceq \texttt{MPI\_Reduce\_scatter}(n) \\ &\preceq \texttt{MPI\_Reduce}(n) + \texttt{MPI\_Scatterv}(n) \end{aligned} \tag{14}$$

The next two rules implement `MPI_Reduce` and `MPI_Allreduce` in terms of `MPI_Reduce_scatter` and are similar to the broadcast implementation of requirement (12).

$$\texttt{MPI\_Reduce}(n) \preceq \texttt{MPI\_Reduce\_scatter}(n) + \texttt{MPI\_Gather}(n) \tag{15}$$

$$\texttt{MPI\_Allreduce} \preceq \texttt{MPI\_Reduce\_scatter}(n) + \texttt{MPI\_Allgather}(n) \tag{16}$$

For the reduction collectives, MPI provides a set of built-in binary operators, as well as the possibility for users to define their own operators. A natural requirement is that a user-defined implementation of a built-in operator should not be faster.

$$\texttt{MPI\_Reduce}(\texttt{MPI\_SUM}) \preceq \texttt{MPI\_Reduce}(\texttt{user\_sum}) \tag{17}$$

A curious example where this is violated is again given in [1, p. 65]. For a particular vendor MPI implementation, a user-defined sum operation was significantly faster than the built-in `MPI_SUM` operation!

4.3 Irregular Communication Collectives

The irregular collectives of MPI, in which the amount of data communicated between pairs of processes may differ, are obviously more general than their regular counterparts. It is desirable that performance be similar when an irregular collective is used to implement the functionality of the corresponding regular collective. Thus, we have requirements like the following.

$$\texttt{MPI\_Gatherv}(v) \approx \texttt{MPI\_Gather}(n) \tag{18}$$

This requires that the performance of `MPI_Gatherv` be in the same ballpark as the regular `MPI_Gather` for uniform p element vectors v with $v[i] = n/p$. Again this is not a trivial requirement. For instance, there are easy tree-based algorithms for `MPI_Gather` but not for `MPI_Gatherv` (at least not as easy because the irregular counts are not available on all processes), and thus performance characteristics of the two collectives may be quite different [11]. Thus, $\approx$ should be formulated carefully and leave room for some overhead.

4.4 Constraining Implementations

In addition to the rule above that relates collective operations to other collective operations, it would be tempting to require that a good MPI implementation fulfill some minimal requirements regarding the performance of its collectives. For example, the MPI standard already explains many of the collectives in terms of send and receive operations.

$$\texttt{MPI\_Gather}(n) \preceq \underbrace{\texttt{MPI\_Recv}(n/p) + \cdots + \texttt{MPI\_Recv}(n/p)}_{p} \tag{19}$$

Extending this, one could define a set of "minimal implementations," for example, an `MPI_Bcast` implementation by a simple binomial tree. Correspondingly one could require that the collectives of an MPI library perform at least as well. This requirement could prevent trivial implementations from fulfilling the rules, but how far this idea could and should be taken is not clear at present.

5 Communicators and Topologies

Let c be a communicator (set of processes) of size p representing an assignment of p processes to p processors. Let c' be a communicator representing a different (random) assignment to the same processors. A metarule like

$$\texttt{MPI\_}A\{c\} \approx \texttt{MPI\_}A\{c'\} \tag{20}$$

can be expected to hold for homogeneous systems for any MPI communication operation A. For non-homogeneous systems, such as SMP clusters with a hierarchical communication system, such a rule will not hold.

For some collectives, it is still reasonable to require communicator independence (irrespective of system), for example, the following.

$$\texttt{MPI\_Allgather}\{c\} \approx \texttt{MPI\_Allgather}\{c'\} \tag{21}$$

This is not a trivial requirement. A linear ring or logarithmic algorithm designed on the assumption of a homogeneous system may, when executed on a SMP system and depending on the distribution of the MPI processes over the SMP nodes, have communication rounds in which more than one MPI process per SMP node communicates with processes on other nodes. The effect of the resulting serialization of communication is shown in Figure 2.

It seems reasonable to require communicator independence for all symmetric (non-rooted communication) collectives, that is, requirement (20) for $A \in \{\texttt{Allgather}, \texttt{Alltoall}, \texttt{Barrier}\}$.

MPI contains routines for defining process topologies, and these should not decrease performance for their preferred communication patterns. As an example of Rule 5, using a Cartesian communicator c and then communicating to the Cartesian neighbors should be no slower than using an arbitrary communicator c'.

$$\texttt{MPI\_Send}\{c\} \preceq \texttt{MPI\_Send}\{c'\} \tag{22}$$

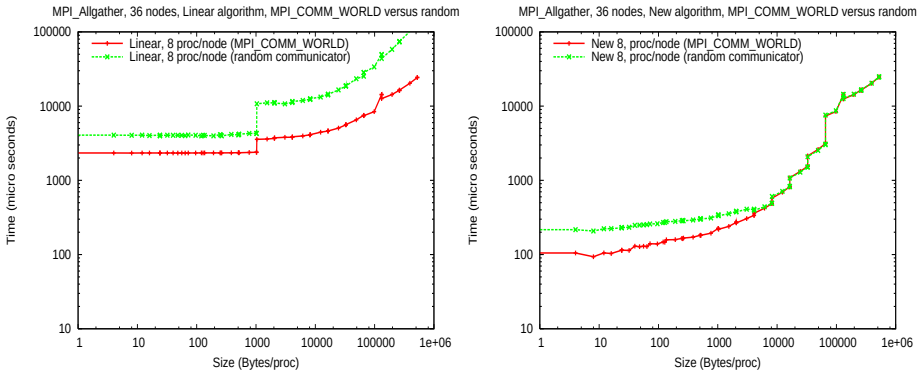

Fig. 2. Left: performance of a simple, non-SMP-aware linear ring algorithm for `MPI_Allgather` when executed on the ordered `MPI_COMM_WORLD` communicator and on a communicator where the processes have been randomly permuted. Right: performance of an SMP-aware algorithm for `MPI_Allgather` on ordered and random communicator. The degradation for small data for the random communicator is due to specifics of the target (vector) system; see [12].

6 One-Sided Communication

The one-sided communication model of MPI-2 is interrelated to both point-to-point and collective communication, and a number of performance requirements can be formulated. We give a single example. For the fence synchronization method, the performance of a fence-put-fence should be no worse than a barrier on the same communicator, followed by an `MPI_Send` of the datatype and address information, followed by another barrier:

$$\texttt{MPI\_Win\_fence} + \texttt{MPI\_Put}(n) + \texttt{MPI\_Win\_fence} \preceq \texttt{MPI\_Barrier} + \texttt{MPI\_Send}(d) + \texttt{MPI\_Send}(n) + \texttt{MPI\_Barrier} \tag{23}$$

where d represents information about the address and datatype on the target.

Figure 3 shows an example where this requirement is violated. On an IBM SMP system with IBM's MPI, the performance of a simple nearest-neighbor halo exchange is about three times worse with one-sided communication and fence synchronization compared with regular point-to-point communication, even when two processors are available for each MPI process.

7 Automating the Checks

With a precise definition of the $\preceq$ and $\approx$ relations, it would in principle be possible to automate the checking that a given MPI implementation (on a given

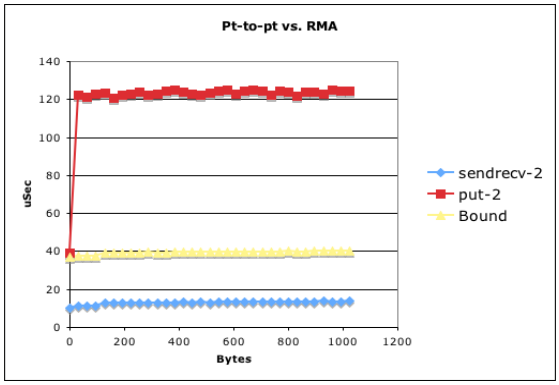

Fig. 3. Performance of `MPI_Put` with fence synchronization versus point-to-point communication on an IBM SMP with IBM's MPI for a simple nearest-neighbor halo exchange. The middle line shows the performance bound based on using barrier and send.

system) fulfills a set of self-consistent performance requirements. A customizable benchmark such as *SKaMPI* [1, 7, 8] already has some patterns that allow the comparison of alternative implementations of the same MPI functionality, similar to many of the rules formulated above. It would be easy to incorporate a wider set of rules into *SKaMPI*. By combining this with an experiments-management system such as *Perfbase* [14, 15], one could create a tool that automatically validates an MPI implementation as to its intrinsic performance. (We have not done so yet.)

8 Concluding Remarks

Users often complain about the poor performance of some of the MPI functions in MPI implementations and of the difficulty of writing code whose performance is portable. Solving this problem requires defining performance standards that MPI implementations are encouraged to follow. We have defined some basic, intrinsic performance rules for MPI implementations and provided examples where some of these rules are being violated. Further experiments might reveal more such violations. We note that just satisfying these rules does not mean that an implementation is good, because even a poor, low-quality implementation can trivially do so. They must be used in conjunction with other benchmarks and performance metrics for a comprehensive performance evaluation of MPI implementations.

References

1. W. Augustin and T. Worsch. Usefulness and usage of SKaMPI-bench. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th Euro-*

pean PVM/MPI Users' Group Meeting, volume 2840 of *Lecture Notes in Computer Science*, pages 63–70, 2003.

2. M. Barnett, S. Gupta, D. G. Payne, L. Schuler, R. van de Geijn, and J. Watts. Building a high-performance collective communication library. In *Supercomputing'94*, pages 107–116, 1994.
3. D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.
4. 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.
5. L. C. McInnes, J. Ray, R. Armstrong, T. L. Dahlgren, A. Malony, B. Norris, S. Shende, J. P. Kenny, and J. Steensland. Computational quality of service for scientific CCA applications: Composition, substitution, and reconfiguration. Technical Report ANL/MCS-P1326-0206, Argonne National Laboratory, Feb. 2006.
6. B. Norris, L. McInnes, and I. Veljkovic. Computational quality of service in parallel CFD. In *Proceedings of the 17th International Conference on Parallel Computational Fluid Dynamics, University of Maryland, College Park, MD, May 24–27*, 2006. To appear.
7. R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A detailed, accurate MPI benchmark. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 5th European PVM/MPI Users' Group Meeting*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59, 1998.
8. R. Reussner, P. Sanders, and J. L. Träff. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.
9. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.
10. R. Thakur, W. D. Gropp, and R. Rabenseifner. Improving the performance of collective operations in MPICH. *International Journal on High Performance Computing Applications*, 19:49–66, 2004.
11. J. L. Träff. Hierarchical gather/scatter algorithms with graceful degradation. In *International Parallel and Distributed Processing Symposium (IPDPS 2004)*, page 80, 2004.
12. J. L. Träff. Efficient allgather for regular SMP-clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 58–65. Springer, 2006.
13. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
14. J. Worringen. Experiment management and analysis with perfbase. In *IEEE International Conference on Cluster Computing*, 2005.
15. J. Worringen. Automated performance comparison. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 402–403. Springer, 2006.