Self-Consistent MPI Performance Guidelines

Jesper Larsson Träff, William D. Gropp, and Rajeev Thakur

Abstract

Message passing using the *Message Passing Interface* (MPI) is at present the most widely adopted framework for programming parallel applications for distributed-memory and clustered parallel systems. For reasons of (universal) implementability, the MPI standard does not state any specific performance guarantees, but users expect MPI implementations to deliver good and consistent performance in the sense of efficient utilization of the underlying parallel (communication) system. For performance portability reasons, users also naturally desire communication optimizations performed on one parallel platform with one MPI implementation to be preserved when switching to another MPI implementation on another platform.

We address the problem of ensuring performance consistency and portability by formulating performance guidelines and conditions that are desirable for good MPI implementations to fulfill. Instead of prescribing a specific performance model (which may be realistic on some systems, under some MPI protocol and algorithm assumptions, etc.), we formulate these guidelines by relating the performance of various aspects of the semantically strongly interrelated MPI standard to each other. Common-sense expectations, for instance, suggest 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, that no function with weak semantic guarantees should perform worse than a similar function with stronger semantics, and so on. Such guidelines may enable implementers to provide higher quality MPI implementations, minimize performance surprises, and eliminate the need for users to make special, non-portable optimizations by hand.

We introduce and semi-formalize the concept of *self-consistent performance guidelines* for MPI, and provide a (non-exhaustive) set of such guidelines in a form that could be automatically verified by benchmarks and experiment-management tools. We present experimental results that show cases where guidelines are not satisfied in common MPI implementations, thereby indicating room for improvement in today's MPI implementations.

J. L. Träff is with NEC Laboratories Europe. W. D. Gropp is with University of Illinois at Urbana-Champaign. R. Thakur is with Argonne National Laboratory

Index Terms

Parallel processing, message passing, Message-passing Interface, MPI, performance portability, performance prediction, performance model, public benchmarking, performance guidelines.

I. Introduction

In the past decade, MPI (Message Passing Interface) [11], [31], [22] has emerged as the *de facto* standard for parallel application programming in the message-passing paradigm. Despite upcoming new languages, notably of the *Partitioned Global Address Space* (PGAS) family like UPC [6], Titanium [41], X10 [30] and others, and frameworks like Google's MapReduce [5], MPI is likely to retain this position for its intended application domain (tightly coupled applications with non-trivial communication requirements and patterns on systems with substantial communication capabilities) for the foreseeable future.

MPI deliberately 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 that the implementability of the MPI standard should not be restricted to systems with specific interconnect and hardware capabilities and that implementers should be given maximum freedom in how they 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, application programmers expect that their MPI library delivers good *performance* on their chosen hardware, and hope that their applications are *portable* to other systems and MPI libraries in the sense that the communication parts of their code do not have to be rewritten or tweaked for performance reasons. Even within a fixed environment, good performance means not only that the communication capabilities of the underlying system are used efficiently, but also that it is not be necessary to replace MPI constructs (typically collective operations) by "hand-written" implementations in terms of other MPI functions in order to achieve the expected performance. That the latter is sometimes the case is unfortunate and known to MPI implementers and users, but not often documented in the literature. One case where an MPI_Allreduce collective had to be rewritten by hand is insightfully documented in [38]. Some examples of hand-improvements of MPI_Alltoall and point-to-point communication can be found

in [13], [24]. In fact, a large number of papers on improvements to various collective operations of MPI (sometimes for specific systems) are motivated by applications that did not perform well using the native implementation of some collective operation; see for instance [18] for a different example.

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, not only semantically but also in terms of the expected performance. This interrelatedness is already used in the MPI standard itself, where certain MPI functions are defined or explained in a semi-formal way in terms of other MPI functions. For example, the semantics of many collective operations is illustrated in terms of point-to-point operations [31, Chapter 4]. The MPI standard, however, does not take the (natural) step to relate the performance of such alternative definitions.

The purpose of this paper is to discuss whether it is possible, sensible, and desirable to formulate relative, system-independent, MPI intrinsic performance guidelines that MPI implementations would want to fulfill. Such guidelines should not make any commitments to particular system capabilities, but would encourage a high(er) degree of performance consistency in an MPI implementation. Such guidelines would also enhance performance portability by discouraging the user from system- and implementation-specific communication optimizations that might not be beneficial for other systems and MPI libraries. Finally, even if relatively trivial, such guidelines would provide a kind of “sanity check” on an MPI implementation, especially if they could be checked automatically.

We formulate a number of MPI intrinsic performance guidelines by semi-formally relating different aspects of the MPI standard to each other with regard to performance. We refer to such rules as *self-consistent MPI performance guidelines*. We believe that such concrete rules can guide both the MPI user and the MPI implementer, and, in the cases where they are fulfilled, aid both single-system performance and performance portability. By their very nature, the rules can be used only to ensure performance *consistency*—a trivial, poor, but consistent MPI implementation could fulfill them perhaps more easily than a carefully tuned library. In that sense such rules raise the bar for the very ambitious MPI implementations. Clearly, the rules should not be interpreted such as to exclude optimizations, but direct the attention of the MPI implementer towards possibly negative side-effects that partial optimizations may have. Or more positively

put: if one part of an MPI library is improved that is performance-wise related to another part, then the rules indicate an opportunity to also improve this other, related part. Otherwise, the user would again be tempted to perform optimizations by hand to compensate for the performance-inconsistency of his MPI library. The list of rules presented here is not exhaustive, but cover the main communication models of MPI (point-to-point, collective, and one-sided), explicate some non-trivial relationships, and in general indicate how and where different parts of MPI can be related with respect to performance. More rules along these lines can surely be established, and other aspects of the MPI standard covered by similar rules (although this becomes more subtle).

More generally, this paper makes the software-engineering suggestion that performance guidelines and benchmark procedures be part of the initial design of communication and other application-support libraries. Performance guidelines can be either self-consistent as discussed here, or model based and more absolute. This would oblige the library designer to think about performance and performance portability from the outset, and contribute towards the internal consistency of the concepts of a library design. MPI is an example of a library design that can be retrofitted with and benefit from performance guidelines. Related work in this direction includes quality of service for numerical library components [21], [23]. Because of the complexity of these components, it is not possible to provide the sort of definitive ordering that we propose for MPI communications. A recent, theoretical model for design of parallel algorithms called the *Parallel Resource Optimal* (PRO) model [7] incorporates a notion of quality by relating each parallel algorithm to a sequential reference algorithm. The requirement enforces performance and scalability for algorithms to be acceptable in the model.

We stress that the performance guidelines presented in this paper explicate common-sense performance expectations, which MPI implementations would mostly want and be able to fulfill without unnecessary burden. They are not intended to constrain or hamper implementations. Rather, explicit performance guidelines should be an aid to implementers to alert them of potential performance inconsistencies in their libraries, which they may otherwise be unaware of. In many cases, the fixes to revealed performance inconsistencies may be simple and implementers would want to do them. In a few cases, there may be special considerations or trade-offs that prevent easy fixes, and an implementer may therefore choose not to fix a particular problem. Nonetheless, that would be a deliberate choice rather than an unfortunate side-effect as is presently the case. Performance guidelines therefore benefit both implementers and users. They help implementers

deliver higher quality MPI implementations, help minimize performance surprises, and eliminate the need for users to perform special, non-portable optimizations by hand.

A. Outline

The remainder of this article is organized as follows. Section II discusses performance models and portability in more detail. Section III states performance meta-rules from which concrete, self-consistent performance guidelines will be derived, and presents the notation that will be used. Concrete performance guidelines for all MPI communication, in particular point-to-point communication, are derived in Section IV. Collective communication guidelines, which due to the strong interrelatedness of the MPI collectives form the bulk of the paper, are derived in Section V. Rules governing MPI virtual topologies and process reorderings are discussed in Section VI, and the more difficult to capture one-sided communication model of MPI is touched upon in Section VII. A brief discussion of approaches to automatic validation of MPI libraries with respect to conformance to self-consistent performance guidelines is given in Section VIII. Summary and outlook conclude the paper in Section IX.

II. Performance Models and Portability

The notion of *performance portability* is hard to quantify (see e.g. [25], [19]), but at least implies that some qualitative aspects of performance are preserved when going from one system to another. For MPI applications, in particular, communication optimizations performed on one system should not be counteracted by the MPI implementation on another. As argued we believe that a degree of performance portability is attainable *without* an explicit performance model by adhering instead to self-consistent performance guidelines. Similar notions of performance portability, as well as the unfortunate consequences in required application restructuring, were explored in [15] for shared virtual memory and hardware cache-coherent systems, and in [37]. In contrast to the *suggestive* approach to performance portability advocated here which suggests to delegate obtaining the best performance across systems of operations captured in a library to the implementations of that library, *descriptive* approaches provide aids towards understanding and translating performance across systems, but ultimately leaves the user with the responsibility of restructuring the application to best fit the system and library at hand. The two approaches are orthogonal, but more focus has so far been given to descriptive approaches.

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 [27], [25], and can help the user both in choosing the right MPI construct for a given system, and in indicating where rewrite may be necessary when switching to another system and/or MPI library. Unfortunately, such benchmarks are not widespread enough, and do not provide the detail necessary to facilitate such complex decisions. Most established MPI benchmarks (Intel MPI Benchmark, SpecMPI, NetPIPE, ...), after all focus on base performance of isolated constructs, and not on comparisons of different ways of achieving a desired user functionality.

Accurate, detailed MPI performance models would make a quantitative translation between systems and MPI libraries possible. Abstract models such as $LogP$ [4] and BSP [36], that are used in the design of applications and communication algorithms, tend to be too complex (for full applications), too limited (enforcing a particular programming style), or too abstract to have predictive power for actual MPI applications, even when characteristics of the hardware, e.g. network topology and capabilities, are sometimes accounted for. Furthermore, MPI provides much more flexible (but also much more complex) modes of communication than typically catered to in such models (blocking and nonblocking communication with possibilities for overlapping of communication and computation, optional synchronous semantics and buffering, rich set of collective operations). 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 surely too large to serve as a tractable model for performance analysis and prediction.

An interesting approach to performance optimization and portability was advocated in [9], [10]. In this approach, applications are designed at a high level using solely MPI collectives for communication. Performance is improved and adapted to MPI implementations with specific characteristics by the use of *transformation rules*. Some of these aim at combining collectives in a given context, whereas some decompose or rephrase collectives with presumably inefficient implementations into sequences or instances of other MPI collectives. These latter type transformations are intimately guided by knowledge of the target system and MPI implementation (performance model and concrete parameters). This is exactly the opposite of what we propose in this paper. Indeed, many of the transformations of [9] could never be beneficial for MPI libraries fulfilling the self-consistent performance guidelines derived in the following sections.

In this sense, the paper [9] is an “afterthought” addressing and alleviating problems that a good MPI library should in our opinion not have.

Even for relatively simple models of point-to-point communication, establishing concrete values for the model parameters for a given system is not simple, and a number of benchmarks have been developed explicitly for this purpose [17], [14]. Automatic generation of performance models has been addressed by the DIMEMAS project [8], [29], but models for collective communication operations are notoriously difficult to generate without apriori, (too) simplifying assumptions about the underlying communication algorithms and system. A further complication is that the concrete, physical topology under which an MPI application is running at a given time may not be known apriori (due to a scheduler allocating different partitions of a large machine). Methods and systems for predicting the performance of full applications without relying explicitly on performance models in the simple sense described above have been explored in [20], [16] and many other works.

III. META-RULES

We first introduce a set of general principles, *meta-rules*, which capture user expectations on how an MPI (or other communication) library should behave. We assume reasonable familiarity with the MPI standard [11], [31], [22], although the discussion should be intuitively understandable to the general reader. The rationale captured by the meta-rules is that 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 other MPI functions. If such were the case, the performance of the library-internal MPI implementation could, all other things being equal, be improved by replacing it with an implementation based on the user implementation. Here we focus exclusively on the (communication) time taken by MPI functions, and not on how this interacts with other computation time of the application. Thus, we do not address the possibility of overlapping communication and computation as made possible by the MPI standard and supported by some MPI implementations, and ignore also other context-sensitive factors that could favor one way of realizing MPI communication over another. The meta-rules are as follows.

(I) Subdividing messages into multiple smaller messages should not reduce the communication time.

(II) Replacing an MPI function with a similar function that provides additional semantic guarantees should not reduce the communication time.

(III) Replacing a specific (collective) MPI operation with a more general operation by which the same functionality can be expressed should not reduce communication time.

(IV) Replacing a (collective) operation by a sequence of other (collective) operations implementing the same functionality should not reduce communication time.

(V) Reranking the processors through a new MPI communicator should not reduce the communication time between processors.

(VI) A virtual process topology should not make communication between all pairs of topological neighbors slower than communication between the same pairs of processes in the communicator from which the virtual topology was built.

Rule I reflects the understanding that MPI is designed to be particularly efficient for communication of large messages. In situations where large messages have to be sent, all other circumstances being equal, it should therefore not be necessary for the user to manually subdivide messages. It is a meta-rule and covers all types of MPI communication, be it point-to-point, one-sided, or collective. Rule II expresses the expectation that semantic guarantees usually come with a cost. Were that not the case, instances of operations with weak semantic guarantees could be replaced by the corresponding operations with stronger guarantees, which would not compromise program correctness, but improve performance. An analogous meta-rule could be formulated for MPI constructs that require certain semantic *preconditions* to be fulfilled. Such operations would not be expected to be slower than corresponding operations not making such requirements. We apply to this analogous meta-rule in two examples in Section IV and V.

Rules III and IV were motivated above and relieve the user of the temptation to implement MPI functions in terms of other MPI functions in order to improve performance.

Rules V and VI are rather MPI specific and cater to the various capabilities of MPI to change the numbering (ranking) of the processors. Processes are bound to processors, and are identified by an associated rank in their communication domain, the *communicator* in MPI terms. MPI provides constructs for creating new communicators from existing ones, and thereby to change the ranking of the processors. Rule V states that the rank that a processor happens to have in a communicator should not determine its communication performance. Rather the location of each processor in the communication system is the determining factor. Indeed, if an MPI library had

a preferred communicator (say `MPI_COMM_WORLD`), in which communication between ranks i and j was faster than communication between the *same* processors with ranks i' and j' in some other communicator, the user would be confronted with the option of manually mapping communication between i' and j' in the new communicator back to communication between i and j in the preferred communicator. This is a very strong meta-rule that is discussed in more detail in Section VI.

MPI provides a means for designating processes as *neighbors* that are expected to communicate more intensively. In MPI terms such a specification is called a *virtual process topology*[31, Chapter 6]. An MPI library can use this specification to create a communicator in which neighboring ranks will be bound to processors that can indeed communicate faster. Rule VI states that communication between at least one pair of such neighbors should not be slower (read: can be expected to be faster) in the reordered communicator. Indeed, if all neighbor pairs communicate slower in the reordered communicator, the user is better off not creating the virtual topology at all, and this is not what is expected of a good MPI implementation. This is explored further in Section VI.

Similar meta-rules rules can most likely be formulated for other communication and application specific libraries. The application to MPI is particularly meaningful since the operations of the MPI standard are semantically strongly interrelated, and because MPI provides the additional support functionality to make implementation of complex functions possible in terms of other, simpler functions. In the following sections, we discuss the meta-rules in more detail and use them to derive a list of concrete, *self-consistent MPI performance guidelines.*

A. Notation

As shorthand for the concrete examples, and to provide a quantitative measure we use the semi-formal notation

$$\mathsf{MPI\_}A(n) \quad \preceq \quad \mathsf{MPI\_}B(n)$$

to mean that MPI function A is *not slower* (alternative reading: *possibly or usually faster*) than MPI function B implementing the same operation when evoked with parameters resulting in at most the same amount of communication or computation n, all other circumstances (communicator, datatypes, source and destination ranks, . . .) being the same. When necessary, we

use p to denote the number of processes involved in a call, and $\mathsf{MPI\_}A\{c\}$ for MPI functionality A invoked on communicator c. Note that the amount of communication n in actual MPI calls is not determined by a single argument but specified implicitly or explicitly by a combination of arguments (datatypes, repetition counts, count vectors, etc.).

We use

$$\mathsf{MPI\_}A \preceq \mathsf{MPI\_}B$$

to mean that functionality A is possibly faster than functionality B for (almost) all communication amounts n.

Finally, we use

$$\mathsf{MPI\_}A \approx \mathsf{MPI\_}B$$

to express that functionalities A and B perform similarly for almost all communication amounts n. Quantifying the meaning of "similarly" is naturally contentious. A strong definition would say that there is a small confidence interval independent of n such that for any data size n, the running time of one construct is within the running time of the other plus/minus some *small additive constant*. A 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. The relation $\preceq$ should be an order relation, and $\preceq$ and $\approx$ defined such that $\mathsf{MPI\_}A \preceq \mathsf{MPI\_}B$ and $\mathsf{MPI\_}B \preceq \mathsf{MPI\_}A$ implies $\mathsf{MPI\_}A \approx \mathsf{MPI\_}B$.

As we discuss in Section VIII, it is not necessary to actually fix the (constant factors in the) relations $\preceq$ and $\approx$ as described above in order to be able to check to what extent an MPI implementation fulfills a set of self-consistent performance guidelines.

IV. General and Point-to-point Communication

The following performance guidelines are concrete applications of meta-rule I. Splitting a communication buffer of kn units into k buffers of n units, and communicating the pieces separately, should not pay off in an MPI implementation.

$$\mathsf{MPI\_}A(kn) \preceq \underbrace{\mathsf{MPI\_}A(n) + \cdots + \mathsf{MPI\_}A(n)}_{k} \tag{1}$$

Likewise, splitting possibly structured data into its constituent blocks of fixed size k should also not be faster than communicating the data in one operation.

$$\mathsf{MPI\_}A(kn) \preceq \underbrace{\mathsf{MPI\_}A(k) + \cdots + \mathsf{MPI\_}A(k)}_{n} \tag{2}$$

Guideline (2) ensures that "blocking by hand", that is, manually splitting buffers into smaller parts of fixed size, will not pay off performance wise, all other circumstances being equal. Of course, in pipelined algorithms or in situations where there is a possibility for overlapping communication with computation, blocking by hand could be an option. The guideline makes no statement about such situations.

The guidelines (1) and (2) are nevertheless non-trivial, and many MPI libraries will violate them for point-to-point communication for some range of n because of the use of different (short, eager and rendezvous) message protocols. An example is given in Figure 1 for a particular system and MPI implementation. The performance of MPI_Send has been measured (with another process performing the matching MPI_Recv) for varying data size n. Because of the large, discrete jump in communication time around $n = 1K$, a user with a 1500-byte message will achieve better performance on this system by sending instead two 750-byte messages. This example illustrates an optimization that competes with performance portability—in this case, the use of small, preallocated message buffers and special protocols. To satisfy guideline (1), an MPI implementation would need a more sophisticated buffer-management strategy, but in turn this could decrease the performance of all short messages.

An example of an MPI library and system that violates guideline (1) with $A = \mathsf{Bcast}$ was given in [1, p. 68]. For this case the broadcast operation has a range of data sizes n where splitting into 2 or 4 blocks of size $n/2$ and $n/4$ respectively and performing instead 2 or 4 broadcast operations is faster than a single broadcast with data size n.

In both examples users are tempted to improve performance by splitting messages by hand in his code, and in both examples performance portability suffers because other systems and MPI libraries may either not have the problem, or the ranges of data size where the problem appears may be completely different.

MPI has a very general mechanism for communicating non-contiguous data by means of so-called user-defined (or derived) datatypes [31, Chapter 3]. Derived datatypes can be used universally in communication operations. Let $T(k)$ be an MPI derived datatype containing k

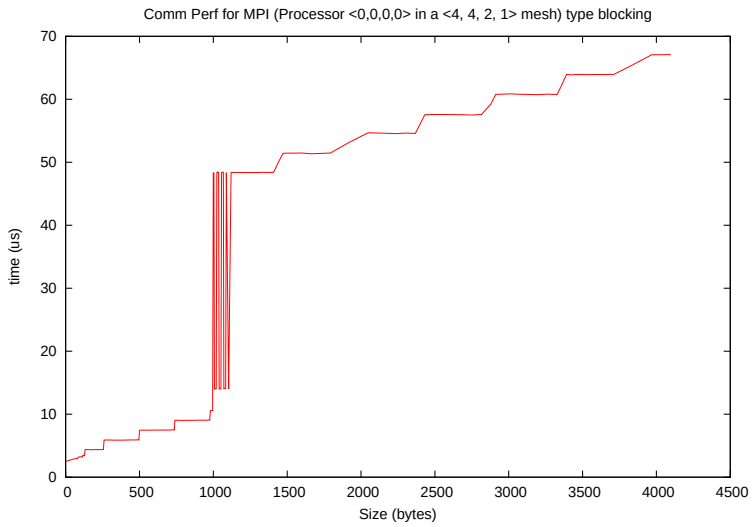

Fig. 1. Measured performance of short message point-to-point communication on IBM BG/L with MPI_Send and MPI_Recv. Because of the switch from eager to rendezvous protocol in the MPI implementation, there is a large jump in performance around 1024 bytes. A user could get better performance for a 1500-byte message by sending this as two 750-byte messages instead. This behavior is typical of many (most) current MPI implementations.

basic elements. We would expect the handling of datatypes in any MPI operation A to be at least as good as first packing the non-contiguous data into a contiguous block using the MPI packing functionality [31, Chapter 3, page 175], followed by operation A on the consecutive buffer. Semi-formally this can be expressed as follows.

$$\mathsf{MPI\_}A(n/k, T(k)) \quad \preceq \quad \mathsf{MPI\_Pack}(n/k, T(k)) + \mathsf{MPI\_}A(n) \tag{3}$$

where $\mathsf{MPI\_}A(n/k, T(k))$ means that operation A is called with n/k elements of type $T(k)$ (for a total of n units). Hence, $\mathsf{MPI\_Pack}(n/k, T(k))$ packs n/k elements of type $T(k)$ into a consecutive buffer of size n. There would be a similar guideline for MPI_Unpack.

The pack and unpack functionalities should be constrained such that packing by subtype does not pay off. This is captured by the following guideline.

$$\mathsf{MPI\_Pack}(n/k, T(k)) \quad \preceq \tag{4}$$

$$\mathsf{MPI\_Pack}(n_0/k_0, T_0(k_0)) + \ldots + \mathsf{MPI\_Pack}(n_{t-1}/k_{n-1}, T_{t-1}(k_{t-1}))$$

where $T_i(k_i)$ for $i = 0, \ldots, t-1$ are the subtypes of $T(k)$, each of k_i elements, and $k_0 + \ldots + k_{t-1} = k$ and $n_0 + \ldots + n_{t-1} = n$. Note that guideline (3) does not require MPI handling of non-consecutive data described by derived datatypes to be "faster" than what can be done by hand, but solely relates datatype handling implicit in the MPI communication operations to explicit handling by the user with the MPI pack and unpack functionality. Recursive application of guideline (4) does, however, limit the allowed overhead in derived-datatype handling, and thus reassures the user of a certain, relative base performance in the use of derived datatypes. The derived datatype functionality powerfully illustrates the gains in (performance) portability that would be offered by even relatively weak self-consistent guidelines like the above. The decision whether to use derived datatypes often has a major impact on application code-structure, and maintaining code versions with and without derived datatypes is often not feasible. In other words, the amount of work required to port a complex application implemented with MPI derived datatypes that performs well on a system with a good implementation of the datatype functionality to a system with poor datatype support can be considerable. Since many early MPI libraries had relatively poor implementations of the derived datatype functionality, this fact did and still does detract users from relying on a functionality that can often simplify the coding. Performance guidelines would assure that the performance of MPI communication with structured data would be at least as good as a certain type of hand-coding, and that this base performance would be portable across systems.

Application of meta-rule II to MPI_Send and MPI_Isend first gives

$$\mathsf{MPI\_Isend}(n) \quad \preceq \quad \mathsf{MPI\_Send}(n)$$

since MPI_Send provides the additional semantic guarantee that the send buffer is free for use after the call. Of course it rarely makes sense to replace an MPI_Send operation with a nonblocking MPI_Isend without at some point issuing a corresponding MPI_Wait call. Since an MPI_Wait has no effect and therefore should be practically for free for the MPI_Send operation, we infer the guideline

$$\mathsf{MPI\_Isend}(n) + \mathsf{MPI\_Wait} \quad \preceq \quad \mathsf{MPI\_Send}(n) \tag{5}$$

By meta-rule IV we also have that

$$\mathsf{MPI\_Send}(n) \quad \preceq \quad \mathsf{MPI\_Isend}(n) + \mathsf{MPI\_Wait} \tag{6}$$

and in combination with guideline (5) it can be deduced that an MPI_Send operation should be in the same ballpark as an MPI_Isend followed by an MPI_Wait:

$$\mathsf{MPI\_Send}(n) \approx \mathsf{MPI\_Isend}(n) + \mathsf{MPI\_Wait} \tag{7}$$

Another similar application of meta-rule II leads to the guideline that

$$\mathsf{MPI\_Send}(n) \preceq \mathsf{MPI\_Ssend}(n) \tag{8}$$

Since the synchronous send operation has the additional semantic guarantee that the function cannot return until the matching receive has started, it should not be faster than the regular send. Along the same lines it can be expected that

$$\mathsf{MPI\_Rsend}(n) \preceq \mathsf{MPI\_Send}(n) \tag{9}$$

If the semantic *precondition* that the receiver is already ready is fulfilled the special ready-send should not be slower than an ordinary send operation. If that would be the case, the library (and user) should simply use MPI_Send instead. We did not explicitly introduce a meta-rule on semantic preconditions.

Guidelines for other point-to-point communication operations can be similarly deduced. For MPI_Sendrecv it is for instance sensible to expect that

$$\mathsf{MPI\_Sendrecv} \preceq \mathsf{MPI\_Isend} + \mathsf{MPI\_Recv} + \mathsf{MPI\_Wait} \tag{10}$$

$$\mathsf{MPI\_Sendrecv} \preceq \mathsf{MPI\_Irecv} + \mathsf{MPI\_Send} + \mathsf{MPI\_Wait} \tag{11}$$

which follows from meta-rule IV.

V. Collective Communication

The MPI collectives [31, Chapter 4] are semantically strongly interrelated, and often one collective operation can be implemented in terms of one or more other, related collectives. A general guideline to an MPI implementation is that a specialized collective should not be slower than a more general collective, as stated by meta-rule III. If such guidelines are fulfilled, users can, with good conscience, be given the advice to always use the most specific collective applicable in the given situation. Naturally, this is one of the motivations for having so many collectives in MPI, and many current MPI implementations do use specialized, more efficient algorithms for specific collectives. The literature on this is extensive.

A general, very MPI specific set of guidelines, that follow from meta-rule II concern the use of the `MPI_IN_PLACE` option. For some collectives it can be used to specify that part of the input has already been placed at its correct position in output buffer, leading to a potential reduction in local memory copies or communication and thus a performance benefit. In such cases the `MPI_IN_PLACE` option is a semantic *precondition* (similar to the precondition in guideline (9)), leading to guidelines like

$$\mathsf{MPI\_}A(\texttt{MPI\_IN\_PLACE}, n) \quad \preceq \quad \mathsf{MPI\_}A(n) \tag{12}$$

for collectives A where the `MPI_IN_PLACE` option has the meaning described above. This is the case for MPI_Gather, MPI_Scatter, MPI_Allgather (and their irregular counterparts). In the reduction collectives like MPI_Allreduce the `MPI_IN_PLACE` option is partly a precondition, partly a semantic guarantee (that a replacement has been performed), and therefore performance guidelines are more difficult to argue.

A. Regular Communication Collectives

Most MPI collectives exist in regular and irregular (vector) variants. In the former the involved processes contribute the same amount of data. These are algorithmically easier and usually perform better than their corresponding irregular variants, as will be discussed further in Section V-C.

The following two guidelines are obvious instances of meta-rule III.

$$\mathsf{MPI\_Gather}(n) \quad \preceq \quad \mathsf{MPI\_Allgather}(n) \tag{13}$$

$$\mathsf{MPI\_Allgather}(n) \quad \preceq \quad \mathsf{MPI\_Alltoall}(n) \tag{14}$$

These expectations also follow from the fact that the more general collectives on the right hand side of the equations require more communication, and an MPI implementation that does not fulfill them as n grows would indeed be problematic. For instance, in MPI_Allgather each process eventually sends (only) n/p data and receives n data, whereas MPI_Alltoall both sends and receives n data. An implementation of MPI_Allgather in terms of MPI_Alltoall would furthermore require each process to make p copies of the n/p contributed data prior to calling MPI_Alltoall. This alone should eventually cause the performance expectation (14) to hold.

The next guideline follows by meta-rules III and IV from a straight-forward implementation of the collective MPI_Allgather operation in terms of two other.

$$\mathsf{MPI\_Allgather}(n) \preceq \mathsf{MPI\_Gather}(n) + \mathsf{MPI\_Bcast}(n) \tag{15}$$

The MPI library implemented MPI_Allgather should not be slower than the user implementation in terms of MPI_Gather and MPI_Bcast. This is not as trivial as it may look. If, for instance, a linear ring algorithm is used for the native MPI_Allgather implementation, and tree-based algorithms for MPI_Gather and MPI_Bcast, the relationship will not hold, at least not for small data sizes n. Such guidelines thus contribute towards consistent performance between different MPI collectives, and would render user optimizations based on inconsistently optimized collectives unnecessary.

A less obvious guideline relates MPI_Scatter to MPI_Bcast. By meta-rule III

$$\mathsf{MPI\_Scatter}(n) \preceq \mathsf{MPI\_Bcast}(n) \tag{16}$$

since the MPI_Scatter operation can be done by more generally broadcasting the n data and then letting each process filter out the subset of the data it needs. For MPI libraries with an efficient implementation of MPI_Bcast, this is a nontrivial guideline for small n, and enforces an equally efficient implementation of MPI_Scatter. An example where this guideline is violated was found with the IBM Blue Gene/P MPI library, which contains an optimized implementation of MPI_Bcast, but not of MPI_Scatter. As a result, MPI_Scatter is about four times slower than MPI_Bcast as shown in Figure 2. This is certainly not what a user would expect, and such behavior would encourage non-portable replacements of MPI_Scatter by MPI_Bcast.

A currently popular implementation of broadcast for large messages reduces broadcast to a scatter followed by an allgather operation [2], [3], [32]. Since this algorithm can be expressed purely in terms of collective operations, it makes sense to require that the native broadcast of an MPI library should behave at least as well:

$$\mathsf{MPI\_Bcast}(n) \preceq \mathsf{MPI\_Scatter}(n) + \mathsf{MPI\_Allgather}(n) \tag{17}$$

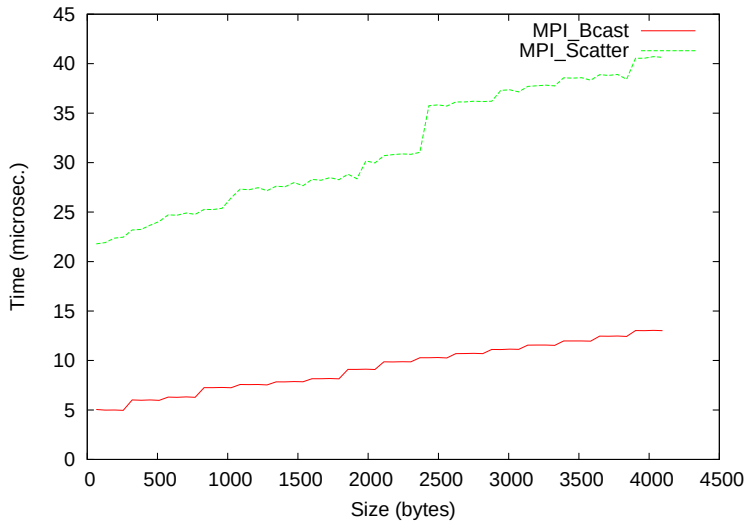

Fig. 2. Performance of MPI_Scatter versus MPI_Bcast on IBM Blue Gene/P. For both collectives the size is the total size of the buffer sent from the root. A natural expectation would require the scatter operation to be no slower than the broadcast. This is violated here.

B. Reduction Collectives

The reduction collectives perform an operation on data contributed from all involved processes, and come in several variants in MPI.

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. It follows from meta-rule IV, whereas the first part is a trivial instantiation of rule III.

$$\begin{aligned} \mathsf{MPI\_Reduce}(n) &\preceq \mathsf{MPI\_Allreduce}(n) \\ &\preceq \mathsf{MPI\_Reduce}(n) + \mathsf{MPI\_Bcast}(n) \end{aligned} \tag{18}$$

A similar rule can be formulated for MPI_Reduce_scatter. This is an irregular collective, since the result blocks eventually scattered over the processes may differ in size:

$$\mathsf{MPI\_Reduce\_scatter}(n) \preceq \mathsf{MPI\_Reduce}(n) + \mathsf{MPI\_Scatterv}(n) \tag{19}$$

The next two rules implement MPI_Reduce and MPI_Allreduce in terms of MPI_Reduce_scatter and are similar to the broadcast implementation of guideline (17).

$$\mathsf{MPI\_Reduce}(n) \preceq \mathsf{MPI\_Reduce\_scatter}(n) + \mathsf{MPI\_Gather}(n) \quad (20)$$

$$\mathsf{MPI\_Allreduce}(n) \preceq \mathsf{MPI\_Reduce\_scatter}(n) + \mathsf{MPI\_Allgather}(n) \quad (21)$$

Since MPI_Allreduce is a more general operation than MPI_Reduce_scatter it should, similarly to guideline (16), hold that

$$\mathsf{MPI\_Reduce\_scatter}(n) \preceq \mathsf{MPI\_Allreduce}(n) \quad (22)$$

MPI libraries with trivial implementations of MPI_Reduce_scatter but efficient implementations of MPI_Allreduce will fail this guideline. A further complication arises because MPI_Reduce_scatter is an irregular collective, allowing result blocks of different sizes to be scattered over the MPI processes. In extreme cases where the complete result is scattered to one process only, the guideline could be difficult to meet. The paper [34] shows that the guideline is nevertheless reasonable by giving an adaptive algorithm for this collective with comparable performance to similar, good algorithms for MPI_Allreduce.

For the reduction collectives, MPI provides a set of built-in binary operators, as well as the capability for users of defining their own operators. A natural expectation is that a user-defined implementation of the functionality of a built-in operator should not be faster. By meta-rule III this gives rise to guidelines like the following.

$$\mathsf{MPI\_Reduce}(\mathsf{n}, \texttt{MPI\_SUM}) \preceq \mathsf{MPI\_Reduce}(\texttt{user\_sum}) \quad (23)$$

where `user_sum` implements element-wise summation just as the built-in operator `MPI_SUM`. A curious example of a vendor MPI implementation where exactly this is violated is again given in [1, p. 65]. For this particular system it turned out that summation with a user-defined operation was a factor 2-3 faster than summation with the built-in operator for larger problem sizes.

The following observation relates gather operations to reductions for consecutive data. A consecutive block of data n_i can gathered from each process i by summing contributions of size n_i with all processes except i contributing blocks of n_i zeroes (neutral element for the operation).

This gives rise to two non-trivial performance guidelines, namely

$$\mathsf{MPI\_Gather}(n) \preceq \mathsf{MPI\_Reduce}(n) \quad (24)$$

$$\mathsf{MPI\_Allgather}(n) \preceq \mathsf{MPI\_Allreduce}(n) \quad (25)$$

which also hold for the irregular MPI_Gatherv and MPI_Allgatherv collectives. The guidelines could also be extended to non-consecutive data by using a user-defined reduction operator operating on non-consecutive data, and thus to cover the gather collectives in full generality. The IBM Blue Gene/P has very efficient hardware support for reduction and broadcast (collective network). The equations above suggest that similar performance be achieved for MPI_Gather(v) and MPI_Allgather(v). In the Blue Gene/P MPI library the observation above is used exactly for this purpose (Sameer Kumar, personal communication). More traditional mesh- or ring-algorithms for MPI_Gather or MPI_Allgather would have difficulties fulfilling such guidelines, and the example show that self-consistent performance guidelines, if formulated carefully, do not compromise the use of special hardware support to optimize an MPI library. But they oblige (and show how) to exploit such hardware support consistently.

C. 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 would be desirable for the performance to be similar when an irregular collective is used to implement the functionality of the corresponding regular collective. This would releive the user of irregular collectives of the temptation to detect regular patterns and call the regular operations in such cases. The $\preceq$ part of this optimistic expectation follow from meta-rule III:

$$\mathsf{MPI\_Gather}(n) \preceq \mathsf{MPI\_Gatherv}(v) \quad (26)$$

$$\mathsf{MPI\_Scatter}(n) \preceq \mathsf{MPI\_Scatterv}(v) \quad (27)$$

$$\mathsf{MPI\_Allgather}(n) \preceq \mathsf{MPI\_Allgatherv}(v) \quad (28)$$

$$\mathsf{MPI\_Alltoall}(n) \preceq \mathsf{MPI\_Alltoallv}(v) \quad (29)$$

for uniform p element vectors v with $v[i] = n/p, 0 \leq i < p$. Strengthening these to $\approx$ and requiring the performance of MPI_Gatherv to be in the same ballpark as the regular MPI_Gather

would be a highly non-trivial guideline. For instance, there are easy tree-based algorithms for MPI_Gather that do not easily generalize to MPI_Gatherv, because the MPI definition of MPI_Gatherv is such that the count vector v is not available on all processes. Thus, performance characteristics of these collectives may be quite different, at least for small n [33].

The MPI_Alltoallv and MPI_Alltoallw functions are universal collectives capable of expressing each of the other data-collecting collectives. Also, broadcast can be implemented by an irregular MPI_Allgatherv operation by a gather-vector v_b with $v_b[r] > 0$ for root r and $v_b[i] = 0$ for $i \neq r$. Again, by rule III, the following further guidelines can be trivially deduced.

$$\mathsf{MPI\_Bcast}(n) \preceq \mathsf{MPI\_Allgatherv}(v_b) \tag{30}$$

$$\mathsf{MPI\_Gatherv}(v_g) \preceq \mathsf{MPI\_Alltoallv}(v_g) \tag{31}$$

$$\mathsf{MPI\_Scatterv}(v_s) \preceq \mathsf{MPI\_Alltoallv}(v_s) \tag{32}$$

$$\mathsf{MPI\_Allgatherv}(v_a) \preceq \mathsf{MPI\_Alltoallv}(v_a) \tag{33}$$

where v_b, v_g, v_s, v_a are p element vectors expressing the broadcast, irregular gather, scatter and all-gather problems, respectively. Strengthening to $\approx$ does not follow from the meta-rules and is too strict a guideline which no current MPI libraries would satisfy.

A similar guideline to (16) for MPI_Scatterv would *not* hold:

$$\mathsf{MPI\_Scatterv}(n) \preceq \mathsf{MPI\_Bcast}(n)$$

is, due to the asymmetry of the rooted, irregular collectives explained above, too strong. Only the root has all data sizes and therefore the non-root processes cannot know the offset from which to filter out their blocks. Further communication is necessary, therefore the performance guideline is rather

$$\begin{aligned} \mathsf{MPI\_Scatterv}(n) &\preceq \mathsf{MPI\_Scatter}(p) + \mathsf{MPI\_Bcast}(n) \\ &\preceq \mathsf{MPI\_Bcast}(p) + \mathsf{MPI\_Bcast}(n) \end{aligned} \tag{34}$$

where latter part follows by application of guideline (16).

D. Constraining Implementations

The guidelines deduced in the previous sections, which relate the performance of collective operations to that of other collective operations, could be expanded to more absolute performance

guidelines by requiring collective performance to be bound by the performance of a set of predefined, standard algorithms (implemented in MPI). Such more elaborate performance bounds would actually follow by repeated application of meta-rule IV. The MPI standard already explains many of the collectives in terms of send and receive operations. This for instance leads to the following, basic performance guideline for MPI_Gather.

$$\mathsf{MPI\_Gather}(n) \preceq \underbrace{\mathsf{MPI\_Recv}(n/p) + \cdots + \mathsf{MPI\_Recv}(n/p)}_{p} \tag{35}$$

Although we do not include them here, such additional, implementation constraining guidelines based on point-to-point implementations of the MPI collectives would bound the collective performance from above. Such a set of upper bound guidelines would include for instance algorithms based on meshes, binomial or binary trees, linear pipelines and others for MPI_Bcast, MPI_Reduce and similar collectives. If a sufficiently large and broad set of constraining implementations were defined and incorporated into an automatic validation tool (see Section VIII), reflecting common networks and assumptions on communication systems, useful performance guarantees clearly showing where simple user-optimizations make no sense could be given.

VI. Communicators and Topologies

In this section we expand on the meta-rules V and VI.

Let c be a communicator (set of processes) of size p representing an assignment of p processes to p processors. Within c processes have consecutive ranks $0, \ldots, p-1$. Let c' be a communicator derived from c (by MPI_Comm_split or other communicator creating operation) representing a different (random) assignment to the same processors. Let i' be the rank in c' of the process with rank i in c. Rule V states that

$$\mathsf{MPI\_}A(i,n)\{c\} \preceq \mathsf{MPI\_}A(i',n)\{c'\} \tag{36}$$

where $\mathsf{MPI\_}A(i,n)\{c\}$ is an MPI operation A performed by rank i in communicator c. In other words, switching to the ranking provided by the new communicator c' should not be expected, all other things being equal, to lead to a performance improvement. Note that this guideline is not requiring that rank i performs similarly in c and c', which would only be possible in special cases

(e.g. systems with a homogeneous communication system). Indeed, since rank i can have been remapped to another processor in c', communication performance can be completely different.

Rule VI addresses this point. If ranks i and j have been specified as neighbors in a virtual topology c' derived from c, at least for one such neighbor pair it should hold that

$$\mathsf{MPI\_Sendrecv}(i,j,n)\{c'\} \preceq \mathsf{MPI\_Sendrecv}(i,j,n)\{c\} \tag{37}$$

This is possible because, as explained above, ranks i and j in c' may be "closer" to each other than they were in c by being mapped to different processors. In an MPI library not fulfilling such guidelines, there is little (performance) advantage in using the virtual topology mechanism. The user is better advised staying with the original communicator c.

Guidelines derived from rule V are far from trivial to meet by MPI implementations. Consider for instance the `MPI_Allgather` collective. A linear ring or logarithmic tree 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 have to communicate with processes on other nodes. This would lead to serialization and slow-down of such rounds and thus break guideline (36). The extent to which this can degrade performance is shown in Figure 3, where the running times of MPI_Allgather for two different communicators are plotted against each other. A non-SMP aware algorithm will be highly sensitive to the process numbering, whereas the SMP-aware algorithms is only to a small extent, which shows that guideline (36) can be fulfilled by a corresponding algorithm modification.

A further difficulty for collective operations to fulfill guideline (36) is that resulting data must be stored in rank (or, for the irregular collectives, in a user-defined) order in the output buffer. Thus, even though MPI_Allgather collects the same data (on all processes) whether it is called over communicator c or c', the order in which data are received may be different in the two cases depending on the algorithm used to implement the collective operation. Gathering in rank order may sometimes be easier than gathering in some random order, for instance because intermediate buffering may not be needed. Therefore, the running time on a communicator c' with an easy ordering could be smaller than on the original c', thus violating guideline (36). Such algorithm dependent factors need to be taken into account when making the performance guidelines quantitative. Figure 3 (right) illustrates such a case: for small data sizes the performance on the

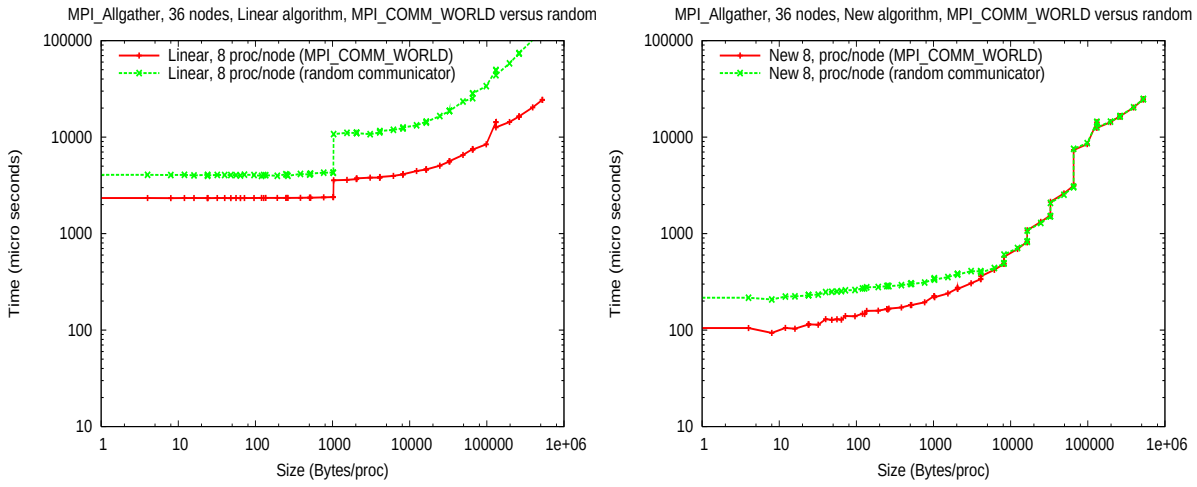

Fig. 3. Left: performance of a simple, non-SMP-aware linear ring algorithm for MPI_Allgather when executed on the ordered MPI_COMM_WORLD communicator and on a communicator where the processes have been randomly permuted. Right: performance of an SMP-aware algorithm for MPI_Allgather on ordered and random communicator. The non-SMP aware algorithm violates the performance expectations captured in the guidelines, whereas the SMP-aware algorithm arguably does not.

random communicator is up to a factor two worse than on the sequential MPI_COMM_WORLD communicator. The algorithm in this example uses an intermediate buffer for small data. If data are received in rank order in this buffer, copying into the user buffer can be done in one operation, which is significantly faster than the sequence of copy operations needed if the intermediate buffer stores data in some random order.

For reduction collectives with non-commutative operators meta-rule V will not hold. It is therefore formulated as a rule purely for communication operations.

As discussed for guideline (24), guideline (36) does not preclude optimizations based on hardware support. For instance the NEC SX-8 communication network provides a small number of hardware counters that can be used for efficient barrier synchronization. This is exploited in the NEC MPI/SX library [28] which allocates a barrier counter to each new communicator if one is available. Thus, it is possible for MPI_Barrier to be significantly slower on a new communicator c' than on the original communicator c—which is not in violation of (36), all other things being equal (concretely, that a hardware counter has not become available when c' is created from a c

without counter). Guideline (36) does suggest, however, that `MPI_COMM_WORLD`, from which all other communicators are derived, be assigned such special resources.

A. Hierarchical Implementations

For systems with a hierarchical communication structure, e.g. clusters of SMP nodes, some collective operations can conveniently and efficiently be implemented by hierarchical algorithms. It would therefore be sensible to require a good MPI implementation to do this, instead of tempting the user into writing his own, hierarchical algorithms. The guideline below for MPI_Allreduce formalizes this. Similar guidelines can easily be formulated for other collectives admitting of hierarchical implementations of this kind.

Let $c_0, \ldots c_{k-1}$ be a partition of the given communicator c into k parts, and let C be a communicator consisting of one process from each of the communicators c_i.

$$\mathsf{MPI\_Allreduce}\{c\} \preceq \|_{i=0}^{k-1} \mathsf{MPI\_Reduce}\{c_i\} + \mathsf{MPI\_Allreduce}\{c\} + \|_{i=0}^{k-1} \mathsf{MPI\_Bcast}\{c_i\} \tag{38}$$

The guideline states that the library implementation of MPI_Allreduce should not be slower than a set of local, concurrent MPI_Reduce operations on the c_i communicators followed by a global MPI_Allreduce and followed by a set of local, concurrent MPI_Bcast of the final result. This is supposed to hold for all splits of the communicator c.

In [38] a case where exactly this kind of tedious rewrite was necessary for the user to attain the expected performance is described. Other users of SMP-like systems often raise similar complaints. Encouraging MPI libraries to fulfill performance guidelines like (38) would save the user from that kind of trouble, and make application codes (more) performance portable across different libraries and systems.

VII. One-Sided Communication

The one-sided communication model of MPI-2 is related to both point-to-point and collective communication, and a number of performance guidelines can be formulated. However, because of the different semantics of the point-to-point, one-sided and collective communication models, much more care is required when formulating guidelines for this communication model. We only

point out the kind of guidelines that can be formulated, but do not explicitly state such because of the subtle semantic differences of especially the point-to-point and one-sided communication models.

Assuming a communication window spanning only two processes, and assuming that data are sent as a consecutive block (no derived datatype), it might be reasonable to expect a blocking send to perform comparably to an MPI_Put with the proper synchronization, at least for larger data sizes n.

$$\mathsf{MPI\_Send}(n) \approx \mathsf{MPI\_Win\_fence} + \mathsf{MPI\_Put}(n) + \mathsf{MPI\_Win\_fence}$$

Such a guideline ignores overhead for tag matching in point-to-point communication, may severely underestimate the synchronization overhead in the semantically strong MPI_Win_fence mechanism, and cannot cater for derived datatypes that have an inherent overhead in one-sided communication, and other factors. Great care is needed for an actual set of fair guidelines.

The issues may be less involved when relating one-sided to collective communication. The following guideline can be seen as a further example of a constraining implementation in the sense of Section V-D.

$$\mathsf{MPI\_Gather}(n) \preceq$$
$$\mathsf{MPI\_Win\_fence} + \underbrace{\mathsf{MPI\_Get}(n/p) + \cdots + \mathsf{MPI\_Get}(n/p)}_{p} + \mathsf{MPI\_Win\_fence}$$

VIII. Verification of Conformance

Although the number of performance guidelines derived in the previous sections is already large, the list is not (and cannot be) exhaustive. Furthermore, each guideline is intended to hold for all system sizes, all possible communicators, all datatypes, etc, except where noted otherwise. These factors have to be considered when attempting to assess or validate the performance consistency of a given MPI implementation. The form of the guidelines is to contrast two implementations of the same MPI functionality. One of these implementations can be quite elaborate, as shown in the example for hierarchical collectives in Section VI-A.

To aid in the validation, a flexible MPI benchmark is therefore needed that makes it possible to script implementations of some functionality at a high-level and contrast the performance of

various alternatives. One such benchmark is *SKaMPI* [1], [26], [27], which already contains *patterns* that correspond to some of the implementation alternatives of the performance guidelines. *SKaMPI* can easily be customized with more such alternatives. The benchmark also makes it possible to compare different measurements.

To assess whether performance guidelines are violated and to what extent, an experiment-management system is needed to go through the large amount of data produced, and to compare results between different experiments. An example of such a system is *Perfbase* [39], [40], which can be used to mine a performance database for cases where the performance of two alternative implementations differ by more than a preset threshold. The *Perfbase* system allows a flexible notion of threshold, thus it is not necessary for the validation to quantify exactly the $\preceq$ and $\approx$ relations.

Building a tool for extensive, semi-automatic verification of performance guidelines for MPI is an an extensive and challenging task, that is beyond the scope of this paper.

IX. Concluding Remarks

Users of MPI often complain about the poor performance (relatively speaking) of some of the MPI functions in their MPI libraries, and about obstacles to writing code whose performance is portable. Defining a performance model for something as complex as MPI is untenable and infeasible. We instead propose self-consistent MPI performance guidelines to help in ensuring performance consistency of MPI libraries and a degree of performance portability of application codes. Conformance to such guidelines could, in principle, be checked automatically. We believe that good MPI implementations are developed with some such guidelines in mind, at least implicitly, but we also showed examples of MPI libraries and systems where some of the rules are violated. Further experiments and benchmarks with other MPI implementations will undoubtedly reveal more such violations. This would be a positive contribution, revealing where more work in improving the quality of MPI implementations is needed. Of course, in some cases, system, resource, or other constraints may lead an implementation to choose not to fix a particular violation of the guidelines.

We think that a similar approach may be possible and beneficial for other software libraries. In this wider context, MPI is an example of a library with a high degree of internal consistency

between concepts and functionality, and this made it feasible to formulate a large set of self-consistent performance guidelines.

For MPI, similar performance guidelines can also be formulated for the parallel I/O functionality defined in MPI-2 [11, Chapter 7], but that topic is too specialized, subtle (as was already the case for one-sided communication in Section VII), and extensive to be considered here. I/O performance is crucially influenced by factors that can only partially be controlled by the MPI library, possibly only through hints that an MPI implementation is not obliged to take. Therefore, many desirable rules can only be formulated as *performance expectations* to the MPI library. Our initial thoughts are detailed in [12].

Another interesting direction for further work is to consider whether sensible, self-consistent guidelines to the *scalability properties* of an MPI implementation can be formulated, again without constraining the underlying hardware or prescribing particular algorithms and implementations.

The most important, imminent work, however, is to construct a tool to assist in the verification of conformance of an MPI library to a set of performance guidelines as formulated in this paper. As explained this would comprise a flexible, easily extensible MPI benchmark and a performance management system that would together make it possible to contrast the two sides of the equations making up the performance guidelines and discover points of violation.

Acknowledgments

The ideas expounded in this paper were first introduced in [35], although in a less finished form. We thank a number of colleagues for sometimes intense discussions on these. At various stages the paper has benefitted significantly from the comments of anonymous reviewers, whose insightfulness, time and effort we also hereby gratefully acknowledge. The 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.

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 European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 63–70. Springer, 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] E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. van de Geijn. On optimizing collective communication. In *IEEE International Conference on Cluster Computing (CLUSTER 2004)*, 2004.

[4] 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.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming.* Wiley, 2005.

[7] A. H. Gebremedhin, M. Essaïdi, I. G. Lassous, J. Gustedt, and J. A. Telle. PRO: A model for the design and analysis of efficient and scalable parallel algorithms. *Nordic Journal of Computing*, 13:215–239, 2006.

[8] S. Girona, J. Labarta, and R. M. Badia. Validation of Dimemas communication model for MPI collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, pages 39–46. Springer, 2000.

[9] S. Gorlatch. Toward formally-based design of message passing programs. *IEEE Transactions on Software Engineering*, 26(3):276–288, 2000.

[10] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems*, 26(1):47–56, 2004.

[11] 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.

[12] W. D. Gropp, D. Kimpe, R. Ross, R. Thakur, and J. L. Träff. Self-consistent MPI-IO performance requirements and expectations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 15th European PVM/MPI Users' Group Meeting*, volume 5205 of *Lecture Notes in Computer Science*, pages 167–176. Springer, 2008.

[13] J. Hein, S. Booth, and M. Bull. Exchanging multiple messages via MPI. Technical Report HPCxTR0308, EPCC, The University of Edinburgh, 2003.

[14] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. Netgauge: A network performance measurement framework. In *High Performance Computing and Communications (HPPC)*, pages 659–671, 2007.

[15] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 217–229, 1997.

[16] D. J. Kerbyson and A. Hoisie. S05 - a practical approach to performance analysis and modeling of large-scale systems. In *ACM/IEEE SC06 Conference on High Performance Networking and Computing*, page 206, 2006.

[17] T. Kielmann, H. E. Bal, and K. Verstoep. Fast measurement of LogP parameters for message passing platforms. In *IPDPS 2000 Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 1176–1183, 2000.

[18] M. Kühnemann, T. Rauber, and G. Rünger. Optimizing MPI collective communication by orthogonal structures. *Cluster Computing*, 9(3):257–279, 2006.

[19] C. Lin and L. Snyder. *Principles of Parallel Programming.* Pearson/Addison-Wesley, 2008.

[20] M. M. Mathis, D. J. Kerbyson, and A. Hoisie. A performance model of non-deterministic particle transport on large-scale systems. *Future Generation Computer Systems*, 22(3):324–335, 2006.

[21] 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.

[22] MPI Forum. MPI: A message-passing interface standard. version 2.1, September 4th 2008. www.mpi-forum.org.

[23] 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.

[24] M. Plummer and K. Refson. An LPAR-customized MPI_Alltoallv for the materials science code CASTEP. Technical Report HPCxTR0401, EPCC, The University of Edinburgh, 2004.

[25] R. Reussner. Using SKaMPI for developing high-performance MPI programs with performance portability. *Future Generation Computing Systems*, 19(5):749–759, 2003.

[26] 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. Springer, 1998.

[27] R. Reussner, P. Sanders, and J. L. Träff. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.

[28] H. Ritzdorf and J. L. Träff. Collective operations in NEC's high-performance MPI libraries. In *International Parallel and Distributed Processing Symposium (IPDPS 2006)*, page 100, 2006.

[29] G. Rodríguez, R. M. Badia, and J. Labarta. Generation of simple analytical models for message passing applications. In *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 183–188. Springer, 2004.

[30] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: Concurrent programming for modern architectures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 271, 2007.

[31] 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.

[32] 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.

[33] J. L. Träff. Hierarchical gather/scatter algorithms with graceful degradation. In *International Parallel and Distributed Processing Symposium (IPDPS 2004)*, page 80. IEEE Press, 2004.

[34] J. L. Träff. An improved algorithm for (non-commutative) reduce-scatter with an application. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 129–137. Springer, 2005.

[35] 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.

[36] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[37] P. Worley and J. Drake. Performance portability in the physical parameterizations of the community atmospheric model. *International Journal for High Performance Computing Applications*, 19(3):187–202, 2005.

[38] P. H. Worley. Comparison of Cray XT3 and XT4 scalability. In *49th Cray User's Group Meeting (CUG)*, 2007.

[39] J. Worringen. Experiment management and analysis with perfbase. In *IEEE International Conference on Cluster Computing*. IEEE Press, 2005.

[40] 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.

[41] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, P. C. K. Datta, and T. Wen. Parallel languages and compilers: Perspective from the Titanium experience. *International Journal of High Performance Computing Applications*, 21:266–290, 2007.