MPI at Exascale

Rajeev Thakur,[1] Pavan Balaji,[1] Darius Buntinas,[1] David Goodell,[1] William Gropp,[2] Torsten Hoefler,[2] Sameer Kumar,[3] Ewing Lusk,[1] Jesper Larsson Traff[4]

[1]Argonne National Laboratory, Argonne, IL 60439, USA
[2]University of Illinois, Urbana, IL 61801, USA
[3]IBM Research, Yorktown Heights, NY 10598, USA
[4]Dept. of Scientific Computing, Univ. of Vienna, Austria

Abstract. With petascale systems already available, researchers are devoting their attention to the issues needed to reach the next major level in performance, namely, exascale. Explicit message passing using the Message Passing Interface (MPI) is the most commonly used model for programming petascale systems today. In this paper, we investigate what is needed to enable MPI to scale to exascale, both in the MPI specification and in MPI implementations, focusing on issues such as memory consumption and performance. We also present results of experiments related to MPI memory consumption at scale on the IBM Blue Gene/P at Argonne National Laboratory.

1. Introduction

We have already reached an era where the largest parallel machine in the world has close to 300,000 cores (the IBM Blue Gene/P at J¨ulich Supercomputing Center). Table 1 shows the top five largest parallel machines in terms of number of cores in the latest (June 2010) edition of the Top500 list [31]. These machines range in size from 150,000 to 300,000 cores. The most commonly used model for programming large-scale parallel systems today is MPI, and in fact, all the systems in Table 1 use MPI implementations that are derived from MPICH2 [23].

Although MPI runs successfully today on up to 300,000 cores, future extreme-scale systems are expected to comprise millions of cores. These systems may have several hundred thousand "nodes" (Figure 1), and each node itself may have large numbers (hundreds) of cores. The cores may comprise a mix of regular CPUs and accelerators such as GPUs, as illustrated in Figure 2. Many researchers and users wonder whether MPI will scale to systems of this size, or what is required to make scale MPI to this level. Furthermore, at exascale, MPI is likely to be used as part of a "hybrid" programming model

Table 1. Top five machines with the largest number of cores in the June 2010 Top500 list [31].

System	Location	Number of Cores
IBM BG/P	Jülich Supercomputing Center	294,912
Cray XT5	Oak Ridge National Laboratory	224,162
IBM BG/L	Lawrence Livermore National Laboratory	212,992
IBM BG/P	Argonne National Laboratory	163,840
IBM BG/P	Lawrence Livermore National Laboratory	147,456

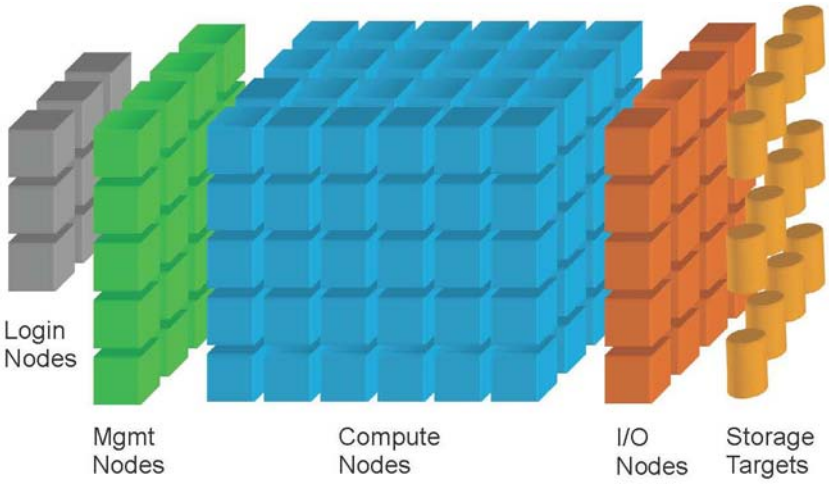

Figure 1. Future extreme-scale platforms with hundreds of thousands of nodes, each node with hundreds of cores.

(MPI+X), much more so than it is today. In such a model, MPI will be used to communicate between address spaces in conjunction with some other "shared memory" programming model (such as OpenMP, UPC, CUDA, or OpenCL) for programming within an address space. In other words, MPI and MPI implementations will need to support efficient hybrid programming. In this paper, we discuss the issues of scaling MPI to exascale, in terms of both what is needed in the MPI specification and what is needed from MPI implementations.

2. Scaling MPI to Exascale

Although the original designers of MPI were not thinking of exascale, MPI was always designed with scalability in mind. For example, a design goal was to enable implementations that maintain very little global state per process. Another design goal was to require very little memory management within MPI; all memory for communication can be in user space. MPI defines many operations as collective (called by a group of processes), which enables them to be implemented scalably and efficiently. Nonetheless, examination of the MPI standard reveals that some parts of the specification may need to be fixed for exascale. Section 3 describes these issues. Many of the issues are being addressed by the MPI Forum for MPI-3.

The main factors affecting MPI scalability are performance and memory consumption. A nonscalable MPI function is one whose time or memory consumption per process increase linearly (or worse) with the total number of processes. For example, if the memory consumption of `MPI_Comm_dup` increases linearly with the number of processes, it is not scalable. Similarly, if the time taken by `MPI_Comm_spawn` increases linearly or more with the number of processes being spawned, it indicates a nonscalable implementation of the function. Such examples need to be identified and fixed in the specification and in implementations. The goal should be to use constructs that require only constant space per process.

From an implementation perspective, the main requirement is that consumption of resources (memory, network connections, etc.) must not scale linearly with the number of processes. Since failures are expected to be common because of the large number of components, the implementation also needs to be resilient and tolerant to faults. Fault tolerance is needed from all levels of the stack—hardware, system software, and applications—and the MPI library must play its role. Since hybrid programming is likely to be common at exascale, the MPI implementation must also efficiently support concurrent communication from multiple threads of a process. In addition, the MPI

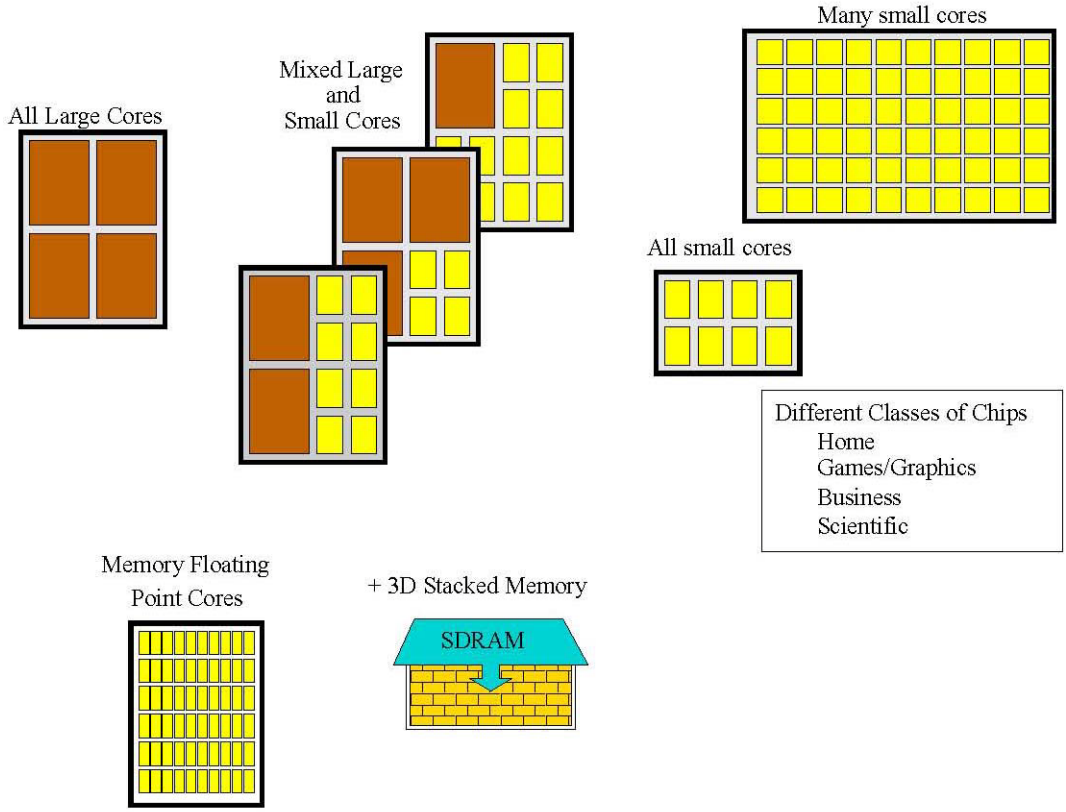

Figure 2. Many cores on a node.

implementation must provide excellent performance for the entire range of message sizes and for all MPI functions, not just simple latency and bandwidth benchmarks. In other words, there should be fewer performance surprises.

We discuss all these issues in further detail in the following sections.

3. Scalability Issues in the MPI Specification

Below we discuss aspects of the MPI specification that may have issues at extreme scale.

3.1. Sizes of Arguments to Some Functions

Some MPI functions take arguments that are arrays of size equal to the number of processes. An example is the irregular or "v" (vector) version of the collectives, such as `MPI_Gatherv` and `MPI_Scatterv`. These functions allow users to transfer different amounts of data among processes, and the amounts are specified by using an array of size equal to the number of processes. Using arrays that scale linearly with system size is nonscalable. An extreme example is the function `MPI_Alltoallw`, which takes six such arrays as arguments: counts, displacements, and datatypes for both send and receive buffers. On a million processes, each array will consume 4 MiB on each process (assuming 32-bit integers), requiring a total of 24 MiB just to call the function `MPI_Alltoallw` (i.e., just to pass the parameters to the function).

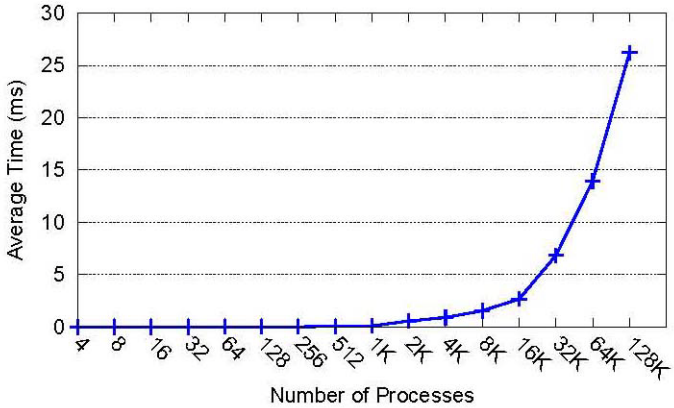

Figure 3. Time for doing a 0-byte alltoallv on IBM BG/P (no actual communication takes place).

Furthermore, an MPI implementation is forced to scan most of these arrays to determine what data needs to be communicated. On large numbers of processes, the time to read the entire array itself can be large, and it increases linearly with system size. This is particularly harmful in cases where a process needs to communicate with only a small number of other processes. For example, Figure 3 shows the time for a 0-byte `MPI_Alltoallv` on an IBM Blue Gene/P. No actual communication takes place since it is specified as 0 bytes. The time taken is just for each process to scan through the input array to determine that no communication is needed. As the number of processes increases, this time itself becomes significant. The MPI Forum is discussing ways to address this problem in MPI-3 [19], such as by defining sparse collective operations. A concrete proposal has been put forth in [13].

3.2. Graph Topology

MPI allows users to define virtual process topologies that express the communication pattern in the application. This feature in turn provides the implementation an opportunity to map the processes to the underlying system in a way that minimizes communication costs. Two types of virtual topologies are supported: a Cartesian topology and a general graph topology. The graph topology, which has existed since the first version of MPI, is one of the most nonscalable parts of the standard. It requires that each process specify the *entire* communication graph of *all* processes, not just its own communication information. The memory required for the purpose would clearly make it unusable on an exascale system. Other limitations of this interface are discussed in [32].

Thankfully, this problem has already been fixed in the latest version of the MPI Standard, MPI 2.2 [18]. Two new functions, `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent`, are defined that allow the user to specify the graph in a distributed, memory-efficient manner. However, the old interface is still available; and unless applications using the old interface make the effort to switch to the new interface, they will encounter scalability issues at large scale [12].

3.3. All-to-all Communication

MPI defines functions that allow users to perform all-to-all communication. All-to-all communication, however, is not a scalable communication pattern. Each process has a unique data item to send to every other process, which leads to limited opportunities for optimization compared with other collectives in MPI. This is not a problem with the MPI specification but is something that applications should be aware of and avoid as far as possible. Avoiding the use of all-to-all may require applications to use new algorithms.

3.4. Fault Tolerance

On exascale systems, the probability of failure or some other error in some part of the system is expected to be relatively high. As a result, greater resilience against failure is needed from all components of the software stack, from low-level system software to libraries and applications. The MPI specification already provides some support to enable users to write programs that are resilient to failure, given appropriate support from the implementation [11]. For example, when a process dies, instead of aborting the whole job, an implementation can return an error to any other process that tries to communicate with the failed process. The application then must decide what to do at that point.

However, more support from the MPI specification is needed for true fault tolerance. For example, the current set of error classes and codes needs to be extended to indicate process failure and other failure modes. Support is needed in areas such as detecting process failure, agreeing that a process has failed, rebuilding a communicator in the event of process failure or allowing it to continue to operate in a degraded state, and timing out for certain operations such as the MPI-2 dynamic process functions. A number of other researchers have studied the issue of fault tolerance in MPI in greater detail [4, 9, 14]. The MPI Forum is actively working on adding fault-tolerance capabilities to MPI-3 [20].

3.5. One-Sided Communication

Many applications have been shown to benefit from one-sided communication, where a process directly accesses the memory of another process instead of sending and receiving messages. For this reason, MPI-2 also defined an interface for one-sided communication that uses put, get, and accumulate calls and three different synchronization methods. This interface, however, has not been widely used for a number of reasons, the main being that its performance is often worse than regular point-to-point communication. The culprit is often the synchronization associated with one-sided communication. Another limitation is the lack of a convenient way to do atomic read-modify-write operations, which are useful in many parallel algorithms. Other issues with MPI one-sided communication are discussed in [3]. The MPI Forum is considering ways to fix these problems in MPI-3 [22].

3.6. Representation of Process Ranks

Another nonscalable aspect of MPI is the explicit use of lists of process ranks in some functions, such as the group creation routines `MPI_Group_incl` and `MPI_Group_excl`. While more concise representations of collections of processes are possible (for example, some group routines support ranges), the use of this sort of unstructured, nonscalable enumeration in some functions is problematic. Eliminating the explicit enumeration should be considered as an option for large scale.

4. MPI Implementation Scalability

MPI implementations must pay attention to two aspects as the number of processes is increased: memory consumption of any function and performance of *all* collective functions (not just the commonly optimized collective communication functions but also functions such as `MPI_Init` and `MPI_Comm_split`). We discuss some specific issues below.

4.1. Process Mappings

MPI communicators usually contain a mapping from MPI process ranks to processor ids. This mapping is often implemented as an array of size equal to the number of processes, which enables simple, constant-time lookup. Although convenient and fast, this solution is not scalable because it requires linear space per process per communicator and quadratic space over the system. To alleviate the problem, as a first step, communicators with the same process-to-processor mapping can share mappings. For example, if a communicator is duplicated with `MPI_Comm_dup`, the new communicator can share the mapping with the original communicator. (The MPICH2 implementation already does this.)

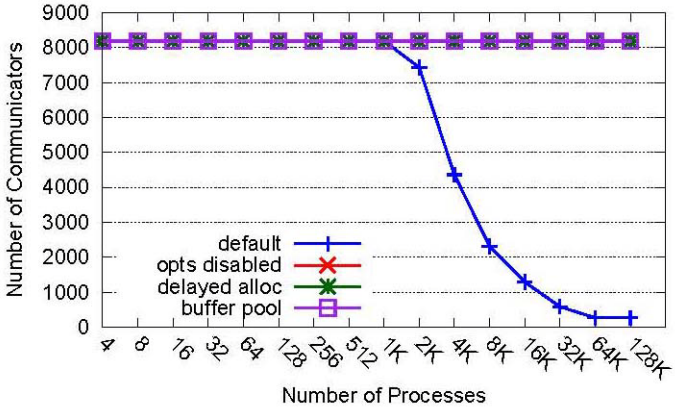

Figure 4. Maximum number of communicators that could originally be created with `MPI Comm dup` on IBM BG/P for varying numbers of processes.

In general, however, there is a need to explore more memory-efficient mappings, at least for common cases. Simple (and very restricted) solutions within the context of Open MPI were considered in [5]. A more general approach could be based on representations of mappings by simple linear functions, $ia + b \bmod p$. The identity mapping is often all that is needed for `MPI_COMM_WORLD`. Such linear representations, when possible, can be easily detected and cover many common cases, for example, subcommunicators that form consecutive segments from `MPI_COMM_WORLD`. A solution in this direction was explored in [34]. Other approaches, incorporated into the FG-MPI implementation, were described in [16]. However, this simple mapping covers only a small fraction of the $p!$ possible communicators, most of which cannot be represented by such simple means. For more general approaches to compact representations of mappings, see the citations in [34].

4.2. Creation of New Communicators

Creating duplicate communicators can consume a lot of memory at large scale if care is not taken. In fact, an application (Nek5000) running on the IBM Blue Gene/P at Argonne National Laboratory initially failed at only a few thousand processes because it ran out of memory after less than 60 calls to `MPI_Comm_dup`.

To study this issue, we ran a simple test that calls `MPI_Comm_dup` in a loop several times until it fails. We ran this test on the IBM BG/P and varied the number of processes. Figure 4 shows the results. Note that the maximum number of communicators supported by the implementation by default is 8,189 (independent of `MPI_Comm_dup`) because of a limit on the number of available context ids.

The number of new communicators that can be created drops sharply starting at about 2,048 processes. For 128K processes, the number drops to as low as 264. Although the MPI implementation on the BG/P does not duplicate the process-to-processor mapping in `MPI_Comm_dup`, it allocates some memory for optimizing collective communication. For example, it allocates memory to store "metadata" (such as counts and offsets) needed to optimize `MPI_Alltoall` and its variants. This memory usage is linear in `p`. Having such metadata per communicator is useful as it allows different threads to perform collective operations on different communicators in parallel. However, the per communicator memory usage increases with system size. Since the amount of memory per process is limited on the BG/P (512 MiB in virtual node mode), this optimization also limits the total number of communicators that can be created with `MPI_Comm_dup`.

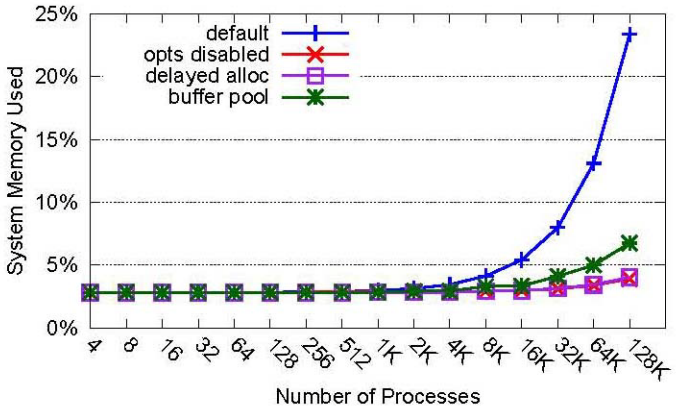

Figure 5. MPI memory usage on BG/P after 32 calls to `MPI Comm dup` of `MPI COMM WORLD` (in virtual node mode).

This scalability problem can be avoided in a number of ways. The simplest way is to use a BG/P environment variable to disable collective optimizations, which eliminates the extra memory allocation. However, it has the undesirable impact of decreasing the performance of all collectives. Another approach is to use an environment variable that delays the allocation of memory until the user actually calls `MPI_ Alltoall` on the communicator. This approach helps only those applications that do not perform `MPI_ Alltoall`.

A third approach, which we have implemented, is to use a buffer pool that is sized irrespective of the number of communicators created. Since the buffers exist solely to permit multiple threads to invoke `MPI_Alltoall` concurrently on different communicators, it is sufficient to have as many buffers as the maximum number of threads allowed per node, which on the BG/P is four. By using a fixed pool of buffers, the Nek5000 application scaled to the full system size without any problem.

Figure 5 shows the memory consumption in all these cases after 32 calls to `MPI_Comm_dup`. The fixed buffer pool enables all optimizations for all collectives and takes up only a small amount of memory.

4.3. Scalability of MPI Init

Since the performance of `MPI_Init` is not usually measured, implementations may neglect scalability issues in `MPI_Init`. On large numbers of processes, however, a nonscalable implementation of `MPI_Init` may result in `MPI_Init` itself taking several minutes. For example, on connection-oriented networks where a process needs to establish a connection with another process before communication, it is easiest for an MPI implementation to set up all-to-all connections in `MPI_Init` itself. This operation, however, involves $\Omega(p^2)$ amount of work and hence is nonscalable. A better approach is to establish no connections in `MPI_Init` and instead establish a connection when a process needs to communicate with another. This method does make the first communication more expensive, but only those connections that are really needed are set up. It also minimizes the number of connections, since applications written for scalability are not likely to have communication patterns where all processes directly communicate with all other processes.

Figure 6 shows the time taken by `MPI_Init` on a Linux cluster with TCP when all connections are set up eagerly in `MPI_Init` and when they are set up lazily. The eager method is clearly not scalable.

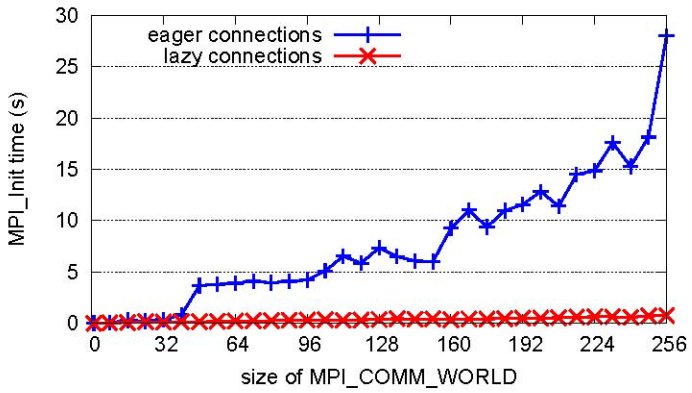

Figure 6. Time taken by MPI Init with eager versus lazy connections on an eight-core-per-node cluster using TCP.

4.4. Scalable Algorithms for Collective Communication

MPI implementations already use sophisticated algorithms for collective communication. Different algorithms are used depending on the message size: for short messages, they use an algorithm that minimizes latency; for long messages, they use algorithms that minimize bandwidth consumption. For example, broadcast is often implemented by using a binomial tree for short messages and a scatter-allgather for long messages [6, 30]. The scatter-allgather may become inefficient at large scale because the scatter may result in blocks that are too small. For example, for a 1 MiB broadcast on one million processes, the scatter phase will result in blocks of size 1 byte. Such problems can be countered by using hybrid algorithms that first do a logarithmic broadcast to a subset of nodes and then a scatter/allgather on many subsets at the same time [33]. Algorithms with similar properties for reduction operations are given in [26].

Global collective acceleration supported by many networks such as Quadrics, InfiniBand, and Blue Gene may be another solution for collectives on `MPI_COMM_WORLD`. On the Blue Gene/P, for example, the `MPI_Broadcast`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Scatter`, `MPI_Scatterv`, and `MPI_ Allgather` collectives take advantage of the combine and broadcast features of the tree network [2].

4.5. Enabling Hybrid Programming

Exascale machines are expected to have many more cores per node than today, but the memory per core is likely to remain the same. Applications, however, want to access more and more memory from each process. A solution to this problem is to use a combination of shared-memory and message-passing programming models (MPI+X): MPI for moving data between address spaces and some shared-memory model (X) for accessing data within an address space that spans multiple cores or even multiple nodes. Options for X include the following.

- OpenMP. This option has been well studied [15, 17, 25, 27]. One limitation is that the shared memory is usually restricted to the address space of a single physical node.
- PGAS languages such as UPC [8] or CoArray Fortran [24]. This option is not as well understood, although there has been some recent preliminary work with UPC [7]. The advantage of this approach is that the shared address space can span the memories of multiple nodes.
- CUDA/OpenCL. This option is needed for GPU-accelerated systems.

What makes hybrid programming possible is the carefully defined thread-safety semantics of MPI [10]. This specification does not require a particular threads implementation or library, such as Pthreads or OpenMP. Rather, it defines a mechanism by which the user can request the desired level of

thread safety and the implementation can indicate the level of thread safety that it provides. This mechanism allows the implementation to avoid incurring the cost of providing a higher level of thread safety than the user needs.

The MPI Forum is exploring further enhancements to MPI to support efficient hybrid programming [21]. An interesting proposal being discussed is to extend MPI to support multiple communication endpoints per process [28]. The basic idea is as follows. In MPI today, each process has one communication endpoint (rank in `MPI_COMM_WORLD`). Multiple threads communicate through that one endpoint, requiring the implementation to do use locks, which turn out to be expensive [29]. This proposal allows a process to define multiple endpoints. Threads within a process attach to different endpoints and communicate through those endpoints as if they are separate ranks. The MPI implementation can avoid using locks if each thread communicates on a separate endpoint.

4.6. Fewer Performance Surprises

Sometimes we hear the following from users:

- "I replaced `MPI_Allreduce` by `MPI_Reduce + MPI_Bcast` and got better results."
- "I replaced `MPI_Send(n)` by `MPI_Send(n/k) + MPI_Send(n/k) + ... + MPI_Send(n/k)` and got better results."
- "I replaced `MPI_Bcast(n)` by my own algorithm for broadcast using `MPI_Send(n)` and `MPI_ Recv(n)` and got better results."

None of these situations should happen. If there is an obvious way intended by the MPI standard of improving communication time, a sound MPI implementation should do so, and not the user! We refer to these as performance surprises.

Although MPI is portable, there is considerable performance variability among MPI implementations— lots of performance surprises. In [35], we defined a set of common-sense, self-consistent performance guidelines for MPI implementations, for instance, the following:

- Subdividing messages into multiple messages should not reduce the communication time. For example, an `MPI_Send` of 1500 bytes should not be slower than two calls to `MPI_Send` for 750 bytes each.
- Replacing an MPI function with a similar function that provides additional semantic guarantees should not reduce the communication time. For example, `MPI_Send` should not be slower than `MPI_ Ssend`.
- Replacing a specific MPI operation by a more general operation by which the same functionality can be expressed should not reduce communication time. For example, `MPI_Scatter` should not be slower than `MPI_Bcast` (Figure 7).

Nonetheless, we found instances where such simple requirements are violated. For example, on the IBM BG/P, scatter is about four times slower than broadcast, as Figure 8 shows. The reason is because broadcast is implemented by using the hardware support provided by the interconnection network, whereas scatter is implemented in software. However, it should be possible to implement scatter by doing a broadcast and discarding the unnecessary data and achieve four times better performance. It would, of course, need extra memory allocation in the implementation.

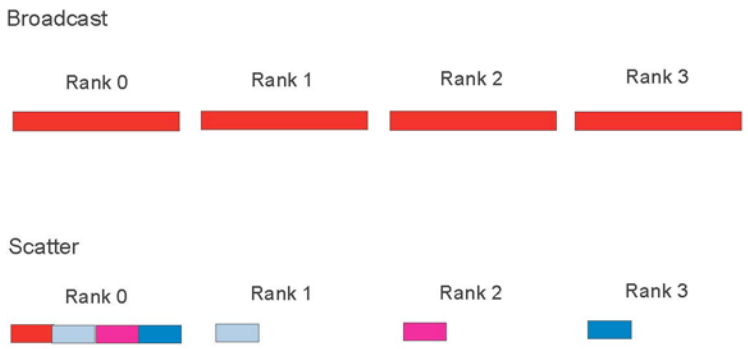

Figure 7. Scatter should be faster, or at least no slower, than a broadcast because it does less work.

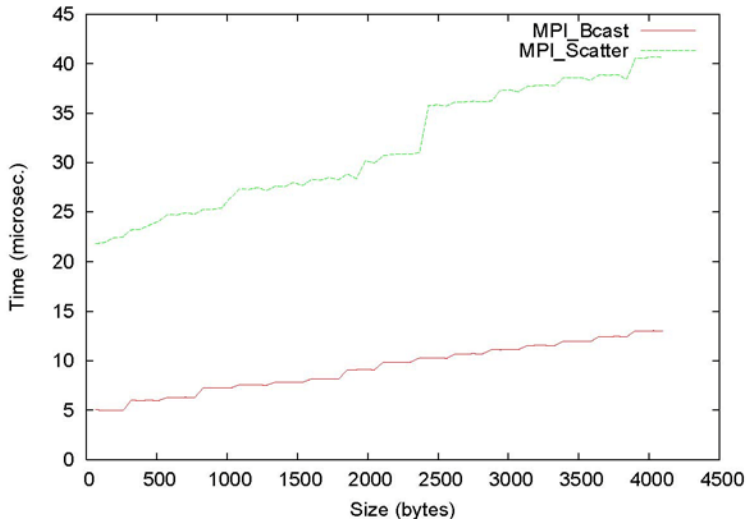

Figure 8. Performance of MPI Scatter versus MPI Bcast on IBM BG/P.

A similar performance surprise is encountered in most MPI implementations because of the switch from eager to rendezvous protocol at a particular message size. Short messages are sent eagerly to the destination assuming that there is enough memory available to store them. At some message size, the implementation switches to a rendezvous protocol, where it waits for an acknowledgment from the destination that the matching receive has been posted and hence memory is available to store in the incoming message. This often leads to a performance graph similar to the one in Figure 9, where there is a sharp jump in communication time when the rendezvous threshold is crossed. In this example, a user could get better performance for a 1500-byte message by sending it as two 750-byte messages instead.

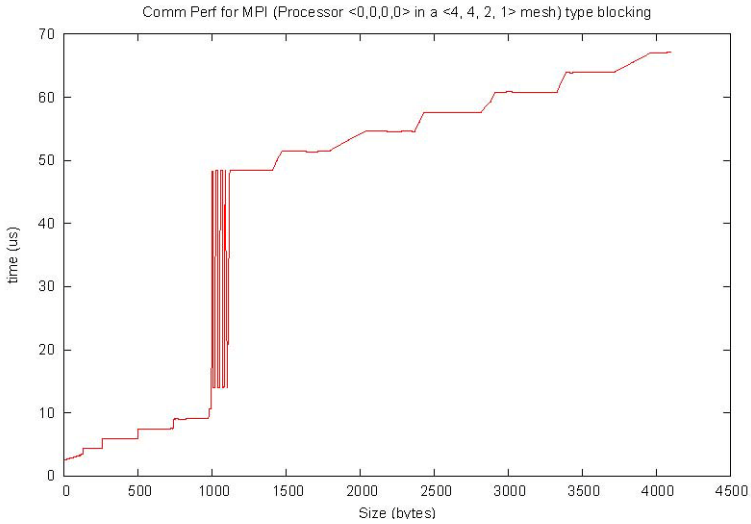

Figure 9. Latency on IBM BG/L. The large jump at 1024 bytes is due to change of protocol.

If care is not taken, such performance surprises will be even more common at exascale because of the large numbers of processes and unexpected interactions among communication patterns at large scale. Tools need to be written to check for these requirements, which would help implementers identify and fix problems.

5. Conclusions

Exascale systems are expected to be available in less than a decade. Although MPI runs successfully on today's petascale systems, for it to run efficiently on exascale systems with millions of cores, some issues need to be fixed both in the MPI specification and in MPI implementations. These issues are related primarily to memory consumption, performance, and fault tolerance. At small scale, memory consumption is often overlooked, but it becomes critical at large scale particularly because the amount of memory available per core is not expected to increase. The MPI Forum, which is currently meeting regularly to define the next version of MPI (MPI-3), is addressing several of the issues that must be fixed in the MPI specification. Correspondingly, MPI implementations are also addressing the issues that they must fix in order to scale to exascale. As a result of these efforts, we believe MPI will run successfully on exascale systems when they are available. Further details on issues related to scaling MPI to millions of cores can be found in [1].

Acknowledgments

We thank the members of the MPI Forum who participated in helpful discussions of the presented topics. We also thank the anonymous reviewers for comments that improved the manuscript. This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DEAC02-06CH11357 and award DE-FG02-08ER25835, and in part by the National Science Foundation award 0837719.

References

[1] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Traff. MPI on millions of cores. *Parallel Processing Letters*, 2011. To appear.

[2] Pavan Balaji, Anthony Chan, Rajeev Thakur, William Gropp, and Ewing Lusk. Toward message passing for a million processes: Characterizing MPI on a massive scale Blue Gene/P. *Computer Science – Research and Development*, 24(1–2):11–19, 2009.

[3] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In *2nd Workshop on Hardware/Software Support for High Performance Sci. and Eng. Computing*, 2003.

[4] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cedile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vencent Neri, and Anton Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Proc. of SC 2002*. IEEE, 2002.

[5] Mohamad Chaarawi and Edgar Gabriel. Evaluating sparse data storage techniques for MPI groups and communicators. In *Computational Science. 8th International Conference (ICCS)*, volume 5101 of *Lecture Notes in Computer Science*, pages 297–306. Springer-Verlag, 2008.

[6] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert A. van de Geijn. Collective communication: Theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[7] James Dinan, Pavan Balaji, Ewing Lusk, P. Sadayappan, and Rajeev Thakur. Hybrid parallel programming with MPI and Unified Parallel C. In *Proceedings of the ACM International Conference on Computing Frontiers*, May 2010.

[8] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, May 2005.

[9] Graham E. Fagg and Jack J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. 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 346–353. Springer-Verlag, 2000.

[10] William Gropp and Rajeev Thakur. Thread safety in an MPI implementation: Requirements and analysis. *Parallel Computing*, 33(9):595–604, 2007.

[11] William D. Gropp and Ewing Lusk. Fault tolerance in MPI programs. *Int'l Journal of High Performance Computer Applications*, 18(3):363–372, 2004.

[12] Torsten Hoefler, Rolf Rabenseifner, Hubert Ritzdorf, Bronis R. de Supinski, Rajeev Thakur, and Jesper Larsson Tr¨aff. The scalable process topology interface of MPI 2.2. *Concurrency and Computation: Practice and Experience*, 2010. To appear.

[13] Torsten Hoefler and Jesper Larsson Tr¨aff. Sparse collective operations for MPI. In *Proc. of 14th Int'l Workshop on High-level Parallel Programming Models and Supportive Environments at IPDPS*, 2009.

[14] Hideyuki Jitsumoto, Toshio Endo, and Satoshi Matsuoka. ABARIS: An adaptable fault detection/recovery component framework for MPIs. In *Proc. of 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS '07) in conjunction with IPDPS 2007*, 2007.

[15] Gabriele Jost, Haoquiang Jin, Dieter an Mey, and Ferhat F. Hatay. Comparing the OpenMP, MPI and hybrid programming paradigms on an SMP cluster. In *Proceedings of EWOMP'03*, 2003. http: //www.nas.nasa.gov/News/Techreports/2003/PDF/nas-030019.pdf.

[16] Humaira Kamal, Seyed M. Mirtaheri, and Alan Wagner. Scalability of communicators and groups in MPI. In *ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[17] Ewing Lusk and Anthony Chan. Early experiments with the OpenMP/MPI hybrid programming model. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2008. IWOMP, 2008.

[18] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009. available at: http://www.mpi-forum.org (July 2010).

[19] MPI Forum Collectives Working Group (Lead: Torsten Hoefler). http://meetings.mpi-forum.org/mpi3.0 collectives.php (July 2010).
[20] MPI Forum Fault Tolerance Working Group (Lead: Rich Graham). http://meetings.mpi-forum.org/mpi3.0 ft.php (July 2010).
[21] MPI Forum Hybrid Programming Working Group (Lead: Pavan Balaji). http://meetings.mpi-forum.org/mpi3.0 hybrid.php (July 2010).
[22] MPI Forum RMA Working Group (Leads: William Gropp and Rajeev Thakur). http://meetings.mpi-forum.org/mpi3.0 rma.php (July 2010).
[23] Mpich2. http://www.mcs.anl.gov/mpi/mpich2.
[24] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, August 1998.
[25] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proc. of 17th Euromicro Int'l Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009)*, pages 427–236, 2009.
[26] Rolf Rabenseifner and Jesper Larsson Tr¨aff. More efficient reduction algorithms for message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 36–46. Springer-Verlag, 2004.
[27] Ashay Rane and Dan Stanzione. Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems. In *Proc. of 10th LCI Int'l Conference on High-Performance Clustered Computing*, 2009.
[28] Marc Snir. MPI-3 hybrid programming proposal, version 7. http://meetings.mpi-forum.org/mpi3.0_hybrid.php.
[29] Rajeev Thakur and William Gropp. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Computing*, 35(12):608–617, December 2009.
[30] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *Int'l Journal of High-Performance Computing Applications*, 19(1):49–66, 2005.
[31] Top500 list. http://www.top500.org/lists/2010/06, June 2010.
[32] Jesper Larsson Tr¨aff. SMP-aware message passing programming. In *Proc. of 8th Int'l Workshop on High-level Parallel Programming Models and Supportive Environments at IPDPS*, pages 56–65, 2003.
[33] Jesper Larsson Tr¨aff. A simple work-optimal broadcast algorithm for message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 173–180. Springer-Verlag, 2004.
[34] Jesper Larsson Tr¨aff. Compact and efficient implementation of the MPI group operations. In *Recent Advances in Message Passing Interface. 17th European MPI Users' Group Meeting*, volume 6305 of *Lecture Notes in Computer Science*, pages 170–178. Springer-Verlag, 2010.
[35] Jesper Larsson Tr¨aff, William D. Gropp, and Rajeev Thakur. Self-consistent MPI performance guidelines. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):698–709, 2010.