Parallel Processing Letters
© World Scientific Publishing Company

MPI ON MILLIONS OF CORES*

PAVAN BALAJI,[1] DARIUS BUNTINAS,[1] DAVID GOODELL,[1]
WILLIAM GROPP,[2] TORSTEN HOEFLER,[2] SAMEER KUMAR,[3]
EWING LUSK,[1] RAJEEV THAKUR,[1] JESPER LARSSON TRÄFF[4†]

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

Received December 2009
Revised August 2010
Communicated by J. Dongarra

ABSTRACT

Petascale parallel computers with more than a million processing cores are expected to be available in a couple of years. Although MPI is the dominant programming interface today for large-scale systems that at the highest end already have close to 300,000 processors, a challenging question to both researchers and users is whether MPI will scale to processor and core counts in the millions. In this paper, we examine the issue of scalability of MPI to very large systems. We first examine the MPI specification itself and discuss areas with scalability concerns and how they can be overcome. We then investigate issues that an MPI implementation must address in order to be scalable. To illustrate the issues, we ran a number of simple experiments to measure MPI memory consumption at scale up to 131,072 processes, or 80%, of the IBM Blue Gene/P system at Argonne National Laboratory. Based on the results, we identified nonscalable aspects of the MPI implementation and found ways to tune it to reduce its memory footprint. We also briefly discuss issues in application scalability to large process counts and features of MPI that enable the use of other techniques to alleviate scalability limitations in applications.

Keywords: MPI, scalability

*This paper is an extended version of a paper titled "MPI on a Million Processors" that was presented at the 16th European PVM/MPI User's Group Meeting 2009 and printed in volume 5759 of *Lecture Notes in Computer Science*, Springer-Verlag, 2009.
†The work of this author was done at NEC Laboratories Europe, St. Augustin, Germany.

1. Introduction

We are fast approaching an era where the largest supercomputers in the world will have more than a million processing cores. For example, the Sequoia machine to be deployed at Lawrence Livermore National Laboratory in 2012 will have close to 1.6 million cores [1]. The *Message-Passing Interface* (MPI) [25] is currently the predominant model for programming large-scale parallel machines, the largest of which today has close to 300,000 cores (the IBM Blue Gene/P at the Jülich Supercomputing Center). As systems scale to millions of cores, many users and researchers are concerned whether MPI (and applications written in MPI) will scale to that level. There are multiple aspects to the scalability issue. First, is the MPI *specification* itself scalable, or are there aspects of the interface that may have issues at large scale? Related to this, is there missing functionality that would enable more scalable programming and/or enhance the utilization of large-scale systems? Second, are typical MPI *implementations* scalable, and what do implementations need to address to improve their scalability? Third, are the parallel algorithms that MPI applications use themselves scalable to millions of cores? We examine these issues in this paper.

Factors affecting scalability include time and space (memory) consumption. A *nonscalable implementation* of an MPI function is an implementation whose running time or memory consumption *per process* increases linearly (or worse) with the number of processes, all other things being equal. A *nonscalable specification* of an MPI function is one that forces *any* implementation of the construct to consume time or memory that grows linearly (or worse) with the number of processes. For example, if the time taken by MPI_Comm_spawn increases linearly with the number of processes being spawned, it indicates a nonscalable implementation of the function (or a scalability problem with the interface). Similarly, if the memory consumption of MPI_Comm_dup increases linearly with the number of processes, it is not scalable. An MPI function such as MPI_Alltoallw that takes several array arguments of size proportional to the total number of processes is a nonscalable MPI construct, since no implementation can circumvent this requirement. Such examples of nonscalability need to be identified and fixed, both in the MPI specification and in implementations. The goal should be to design and use constructs whose time and space requirements scale sublinearly with the number of processes.

2. Scalability Issues in the MPI Specification

Although the developers of MPI did not envision million-core systems when MPI was first designed, MPI was nonetheless designed with scalability in mind. For example, a design goal was to enable implementations of MPI that maintain very little global state per process and require very little memory management within MPI (all memory for communication can be in user space) [14]. MPI also 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

specification reveals that some parts of it may have issues at large scale, particularly with respect to memory consumption.

For the discussion below, we use p to denote the number of processes in a communicator.

2.1. *Irregular collectives*

Many collectives in MPI have an *irregular* (or *vector*, "v") version that allows users to transfer unequal amounts of data among processes. These collectives take one or more arguments that are arrays of size p, for example, the arrays of counts and displacements in MPI_Gatherv and MPI_Scatterv. An extreme case is MPI_Alltoallw, which takes *six* such arrays as arguments: counts, displacements, and datatypes for both send and receive buffers. Using such parameters is nonscalable: on a million processes, each array will consume 4 MiB on each process (assuming 32-bit integers). Furthermore, any MPI implementation is forced to scan most of these arrays to determine which data have to be communicated.

Irregular collectives are often used in applications because MPI lacks other ways to express communication within a sparse subset of processes in a communicator. For example, in applications that require nearest-neighbor communication in a Cartesian grid, each process may perform an MPI_Alltoallv on `MPI_COMM_WORLD` and specify 0 bytes for all processes other than its neighbors.[a] The PETSc library [35], for example, uses MPI_Alltoallv in this manner. While most MPI implementations optimize this pattern by communicating only with processes that have nonzero data, the MPI implementation must still scan through the entire array of data sizes to know which processes have nonzero data, and the user must allocate and initialize this array. On large numbers of processes, the time to read the entire array itself can be large and increases linearly with system size, even though the number of neighbors a process communicates with remains fixed. Figure 1 shows this effect on an IBM Blue Gene/P for calling MPI_Alltoallv with zero-byte messages (no actual communication).

To avoid this problem, some computational libraries, such as PETSc, disable MPI_Alltoallv-based communication by default and instead perform direct point-to-point communication among nearest neighbors, which may not be as efficient as a concisely represented collective operation could be. The MPI Forum is discussing ways to improve this situation in MPI-3 by means of sparse collective operations. A concrete proposal has been put forth in [19].

[a]This communication cannot be done easily by using subcommunicators because each process may belong to many subcommunicators and the collectives would have to be carefully ordered to avoid deadlocks. Such a scheme would also serialize much of the communication. Note that these concerns would be partly alleviated by collectives with nonblocking semantics.

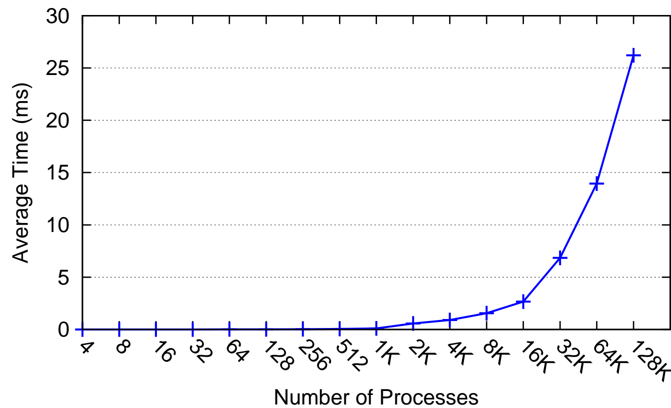

Fig. 1. Zero-byte Alltoallv time on IBM Blue Gene/P (no actual communication).

2.2. *Graph topology*

One of the most nonscalable constructs in MPI (memorywise) is the general graph topology. An MPI program can specify the communication pattern of the application as a directed graph, with edges of the graph representing the communication between processes. This allows the MPI implementation to optimize communication by appropriate reordering or placement of processes. The problem with the specification is that it requires the *entire* communication graph to be supplied on *each* process. It therefore requires $\Omega(p + e)$ or $\mathcal{O}(p^2)$ space per process, where e is the number of edges in the graph, and $\Omega(p^2 + pe)$ or $\mathcal{O}(p^3)$ in total (across all processes). Other limitations of this interface are discussed in [44].

The latest version of the MPI Standard, MPI-2.2 [25], introduces a new graph topology interface, MPI_Dist_graph_create, which enables a fully distributed specification of the communication topology with only $\mathcal{O}(p + e)$ total memory consumption, as further explained in [18]. A scalable implementation, where no single process needs to store the entire graph at any time, is important at large scale.

2.3. *MPI groups and communicators*

An essential, but potentially nonscalable, feature of MPI is the functionality for forming arbitrary subsets of process groups and building communicators. This rich and general set of operations leads implementations to use explicit enumerations for the mapping of processes to processors or cores. Such explicit enumerations, which may be impossible to avoid in the general case, may become an issue at large scale. Recently, researchers have explored compact representations for process groups [22, 46], but further work is needed.

2.4. *Fault tolerance*

On systems with millions of cores, the probability of failure or unrecoverable error in some part of the system becomes very 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 [16]. 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 timeouts 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 [3, 4, 10, 11, 12, 21]. The MPI Forum is actively working on adding fault-tolerance capabilities to MPI-3 [27].

Some of the interfaces for detecting and reporting errors in MPI are also not scalable. Consider the MPI dynamic process routines, MPI_Comm_spawn and MPI_Comm_spawn_multiple. When used with the `soft` info key, these routines provide a "best effort" attempt to spawn all of the processes, so that if some of the spawn operations fail, the implementation is allowed to return less than the requested number of processes. However, the error codes for the attempted spawn operations are returned in an array of size equal to the number of processes being spawned. A more scalable approach would be for the MPI implementation to allocate memory and return an array of size equal to the number of failures. This approach, however, would be a departure from the current convention in MPI in which the user allocates such memory. Another possible solution would be to extend the user-defined error handler mechanism to provide an iterator interface for accessing error codes, which may reduce the memory required to represent the error codes at the cost of requiring the application to iterate over each error code.

2.5. *Collective communication*

The collective communication operations in MPI have blocking semantics, and slight load imbalances or delays (operating system or network noise) can lead to significant process synchronization (waiting) times at large scale [34]. Synchronization issues are worse for collective communications that potentially synchronize entire process groups. Synchronization delays can often be mitigated by nonblocking communication operations [17], which relax the synchronization by splitting the operation into separate start and wait phases. The time window between start and wait can be used

as a *buffer* to mitigate the influence of load imbalance or delays. The MPI Forum originally believed (as of MPI 2.0) that threads could be used to provide efficient nonblocking collective operations. However, experience with systems that provide only one thread per core and with systems where the necessary thread safety also adds a significant performance overhead has caused the MPI Forum to consider the addition of nonblocking collective operations in MPI-3 [26].

An alternative or orthogonal implementation approach is to design algorithms for collective operations that are more resilient to nonsynchronized process arrival patterns. A start in this direction has been outlined in [33].

3. MPI Implementation Scalability

In terms of scalability, MPI implementations must pay attention to two aspects as the number of processes is increased: memory consumption of any function and the performance of *all* collective functions (including functions such as MPI_Init and MPI_Comm_split).

3.1. *Point-to-point communication*

In communication patterns where there are unexpected messages (the sends occur before the matching receives are posted), parts of the messages or control messages need to be buffered at the receiver side [9]. At large scale, the number of unexpected messages could become very large, requiring an inordinate amount of memory for buffering. This problem needs to be dealt with, for example, by using some kind of flow-control system. A protocol for the Blue Gene systems was designed and evaluated in [13]. A related problem is the number of connections (p^2) needed on systems that are connection oriented and the need to establish connections in a lazy (as-needed) manner (also see Section 3.5).

Another potential issue arises when MPI is implemented over RDMA networks. Implementations often choose to expose a memory region, called *mailbox*, to remote processes in order to facilitate fast RDMA puts without synchronization overheads. A naïve implementation would create one mailbox for each communication peer and thus occupy $\mathcal{O}(p)$ memory per process (depending on lazy connection establishment). A scalable implementation should limit the number of mailboxes and fall back to synchronizing communication mechanisms. This can be done by dynamically allocating a fixed pool of mailboxes to communication neighbors (i.e., a cache of mailboxes).

3.2. *Process mappings*

MPI communicators usually contain a mapping from MPI process ranks to processor id's. This mapping is typically implemented by an array of p entries for direct, constant-time lookup, possibly with optimizations for particular mappings (identity mapping, other very regular patterns). A number of other mappings are often

maintained, for instance, to enable fast navigation within and across the nodes of an SMP cluster. Although convenient and very fast, this solution, which requires linear space per process per communicator and quadratic space over the system, is clearly not scalable.

To alleviate the problem, communicators with the same process-to-processor mapping can share mappings. For example, if a communicator is dup'ed with MPI_Comm_dup, the new communicator can share the mapping with the original communicator.

A solution to this problem is needed. Simple (and very restricted) solutions within the context of Open MPI were considered in [6]. 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, e.g., subcommunicators that form consecutive segments from MPI_COMM_WORLD. A solution in this direction was explored in [46]. Other approaches, incorporated into the FG-MPI implementation, were described in [22]. However, this simple mapping covers only a very small fraction of the $p!$ possible communicators, most of which cannot be represented by such simple means. The regular structure is often lost when process topologies are remapped (`reorder=1` during creation), and simple schemes will fail to compress the storage. For more general approaches to compact representations of mappings, see the citations in [46].

3.3. *Memory overheads in communicator creation*

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 large scale because it ran out of memory after less than 60 calls to MPI_Comm_dup.

To study this issue, we ran experiments on the Argonne Blue Gene/P to measure the memory overhead involved in creating new communicators. Figure 2 shows the results of an experiment to determine, for different numbers of processes, how many communicators can be created by calling MPI_Comm_dup of MPI_COMM_WORLD in a loop until it fails. 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.

With the default settings, 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 Blue Gene/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

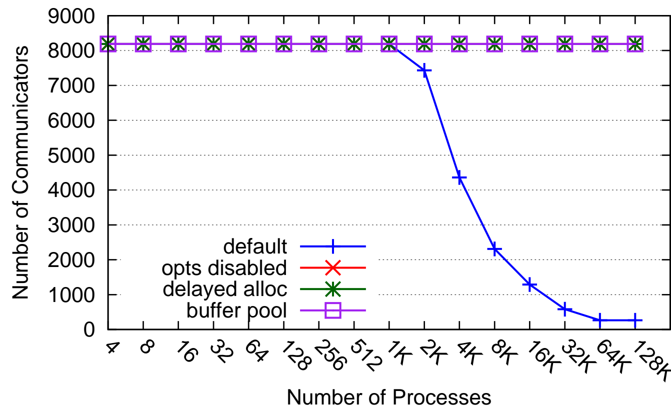

Fig. 2. Maximum number of communicators that can be created with MPI_Comm_dup of MPI_COMM_WORLD on IBM BG/P for different sizes of MPI_COMM_WORLD.

different communicators in parallel. However, the per communicator memory usage increases with system size. Since the amount of memory per process is very limited on the Blue Gene/P (512 MiB in virtual node mode), this optimization also limits the total number of communicators that can be created with MPI_Comm_dup.

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 that 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 Blue Gene/P is four. By using a fixed pool of buffers, the Nek5000 application scaled to the full system size without any problem.

Figure 3 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.

3.4. *Memory overheads in one-sided communication implementations*

One-sided operations readily lend themselves to be implemented on architectures that offer remote direct memory access (RDMA). In MPI one-sided communication,

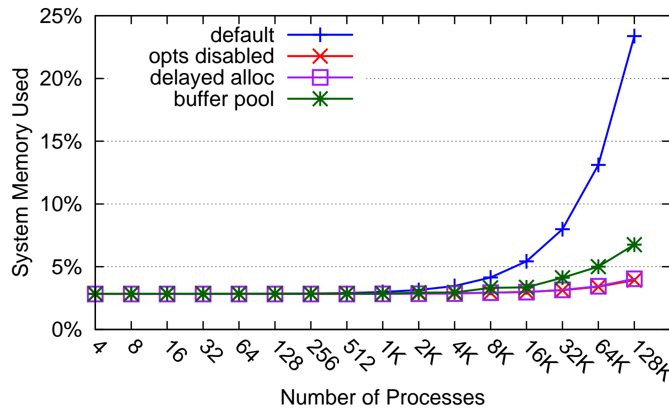

Fig. 3. MPI memory usage on BG/P after 32 calls to MPI_Comm_dup of `MPI_COMM_WORLD` (in virtual node mode).

since the base of the window might have a different address at each process, an RDMA put-based method seems to require a table of size $\Omega(p)$ at each process. (Another parameter than can be different on each process and may require a table of size $\Omega(p)$ is the "displacement unit" passed to MPI_Win_create.)

In order to retain the advantages of RDMA and also achieve constant memory overhead, an implementation could create a translation cache of fixed size at each process (similar to a TLB). When a one-sided request cannot be served from the cache, the library can fetch the required translation from the target process (e.g., with RDMA-Get) and add it to the local cache. A similar technique could be used for the displacement units; however, because the expected number of different displacement units (architectures) is low, other techniques could be used (e.g., compression schemes similar to the ones proposed in Section 3.2).

3.5. *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 tempting for an MPI implementation to set up all-to-all connections in MPI_Init itself. This operation involves $\Omega(p^2)$ amount of work and is inherently 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

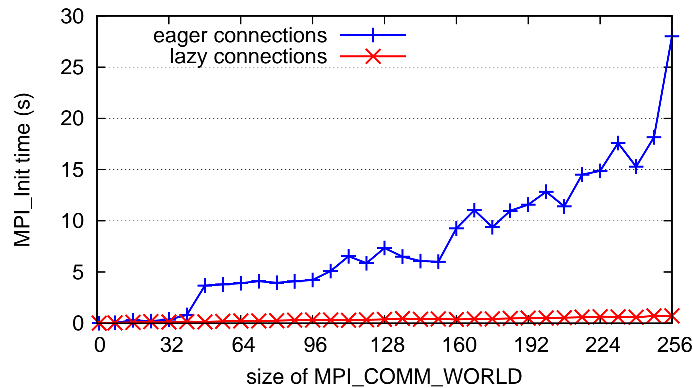

Fig. 4. Duration of MPI_Init with eager versus lazy connection establishment. Results are from an eight-core-per-node cluster using TCP/IP as the communication protocol.

written for scalability are not likely to have communication patterns where all processes directly communicate with all other processes.

Figure 4 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.

3.6. *Scalable algorithms for collective communication*

Good MPI implementations have collectives with low latency (proportional to the diameter of the communication network) for small messages and high bandwidth for large messages. They also carefully adapt to and exploit the capabilities of the underlying communication system (clustered, single- or many-ported, tree- or mesh-shaped; special hardware capabilities; etc.). We note that some algorithms that are attractive in principle may run into problems at large scale. For instance, a broadcast implemented as a scatter followed by an allgather [7, 42] may, if implemented naïvely, give rise to very small blocks in the allgather phase. For example, for a 1 MiB broadcast on one million processes, the allgather phase may involve one-byte messages. Such problems can be countered by less naïve implementations, which switch from scatter to a logarithmic broadcast when message blocks go under a certain threshold; the allgather phase will then consist of multiple simultaneous allgather operations on disjoint subsets of processes [45]. Algorithms with similar properties for reduction operations are given in [38].

Topology-specific optimizations are also essential at large scale. Most interconnects have smaller diameters than the size of the network ($\mathcal{O}(\log p)$ on switched networks and $\mathcal{O}(\sqrt[3]{p})$ on 3D torus networks). A pipelined algorithm that streams data on a spanning tree embedded in the network topology will provide more scalable performance because the throughput of the collective is determined by $\frac{\text{message_size}}{\text{diameter}}$.

For example, on the BG/P, the six-color torus algorithm can keep 95% of all the links busy during an 8 MiB broadcast operation [24].

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. Enabling Application Scalability

As emerging hardware architectures make greater degrees of parallelism available, and even necessary, existing applications are facing the problem of scaling up. The complexity of solving this problem depends entirely on the basic algorithms used by the application, and so no completely general approach will do. In this section, we describe some ways in which features of MPI, perhaps not being used in the current version of a particular application, can play an important role in enabling that application to run effectively on more processors. In many cases, it may be possible to retain most of the existing application code, which is of course desirable from the application's point of view.

4.1. *All-to-all communication*

All-to-all communication 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. This is not a problem with the MPI specification but is something applications should be aware of and avoid as far as possible. Avoiding the use of all-to-all may require new algorithms.

4.2. *Higher-dimensional decompositions with MPI*

One relatively straightforward case occurs when the application consists of calculations carried out on a rectangular two- or three-dimensional mesh with nearest-neighbor communication, but the application has parallelized the computation with a one-dimensional decomposition of the mesh. This approach results in contiguous buffers for the MPI sends and receives, which simplifies the application. Straightforward arithmetic shows that as the number of processors and mesh cells scales up, it becomes more efficient to use a two- or three-dimensional decomposition of the mesh. This results in noncontiguous communication buffers for sending and receiving edge or face data. MPI can help by providing the functions for assembling MPI datatypes that describe these noncontiguous areas of memory. Modern MPI implementations then use particularly efficient algorithms for communicating these areas [40, 47].

4.3. *Enabling topology mapping*

The MPI-2.2 standard allows a scalable specification of graph communication topologies, and the `reorder` argument allows for optimized mappings to the underlying topology. Especially on sparse topologies like 3D tori, an efficient process-to-node mapping can be crucial to large-scale application performance. MPI users should consider specifying their communication topologies and reordering the application data as indicated by the MPI library (according to the new ranks in the distributed graph communicator).

4.4. *Use of threads with MPI*

In the earlier parts of this paper, we treated "MPI on millions of cores" as if it meant that the application would have millions of separate MPI ranks. This is unlikely to be the case in practice. As the amount of memory per core decreases, applications will be increasingly motivated to use a shared-memory programming model on multicore nodes, while continuing to use MPI for communication among address spaces. MPI supports this transition by having clear semantics for interoperation with threads, based on four levels of thread safety that can be required by an application and provided by an MPI implementation. Although no particular thread system is mentioned in the MPI standard, the MPI specification of levels of thread safety meshes particularly well with the OpenMP standard. This feature has made OpenMP+MPI the currently most widely used hybrid programming method [8, 37, 39]. The MPI Forum is also discussing extensions to MPI in MPI-3 for more efficient support of hybrid programming [28].

4.5. *Use of MPI-based libraries to hide complexity*

We describe an example of how MPI enables the development of libraries that make it easier to write applications.

One of the most obviously nonscalable approaches to parallel programming is the "manager-worker" paradigm [15], which can achieve good load balancing at the expense of having a single manager process to coordinate the dispensing of work to the worker processes, collection of results, and perhaps addition of new work to the work queue. We recently worked with a Monte Carlo application in nuclear physics [36] that used a variation of this approach and was stuck at about 2,000 processors, with the ambition of going to tens of thousands. MPI helped solve this problem by enabling the construction of a general-purpose library called ADLB (Asynchronous Dynamic Load Balancing) [5] that eliminated the single manager as a bottleneck by providing a simple put/get interface to a distributed work queue. The application actually became simpler than before because the MPI communication disappeared; any application process simply puts new work to the queue or retrieves work from it. The ADLB implementation, however, is relatively complex and for scalability and efficiency requires a full range of MPI features, including thread

safety, multiple communicators, derived datatypes, asynchronous sends and receives, and the "ready" send operation, all of which are hidden from the application. In this way, MPI supports application scalability while actually simplifying application code.

4.6. *Performance and debugging tools*

An important feature of the MPI standard is a clever definition of an interface, called PMPI, that enables portable profiling of the MPI calls in an application without modifying (or even having access to) the application's source code. This feature has enabled the development of a number of performance and debugging tools, which immensely help application development, testing, and tuning. Examples of these tools include TAU [41], Intel Trace Collector and Trace Analyzer [20], Vampir [48], Paraver [32], mpiP [30], KOJAK [23], Paradyn [31], and TotalView [43]. More work is needed in this area, however, particularly tools that will scale to jobs running on millions of cores. The MPI Forum is also addressing this issue by investigating additional support for tools in MPI-3 [29].

5. Conclusions

We believe MPI is ready for scaling to millions of cores, although issues such as those discussed in this paper need to be fixed in both the MPI standard and in MPI implementations. Examples of nonscalable parts of the MPI standard include irregular collectives and some other functions that take array arguments of size proportional to the total number of processes. There is also a need for investigating systematic approaches to compact, adaptive representations of process groups. MPI implementations must pay careful attention to the memory requirements of functions and systematically root out data structures whose size grows linearly with the number of processes. To obtain scalable performance for collective communication, MPI implementations may need to become more topology aware or rely on global collective acceleration support. MPI also provides other features, such as support for building complex libraries and clear semantics for interoperation with threads, that enable applications to use other techniques to scale when limited by memory or data-size constraints.

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 DE-AC02-06CH11357 and award DE-FG02-08ER25835, and in part by the National Science Foundation award 0837719.

References

[1] 20 petaflop Sequoia supercomputer. `http://www-304.ibm.com/jct03004c/press/us/en/pressrelease/26599.wss` (July 2010).

[2] P. Balaji, A. Chan, R. Thakur, W. Gropp, and E. 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] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Proc. of SC 2002*. IEEE, 2002.

[4] G. Bosilca, A. Bouteiller, T. Herault, P. Lemarinier, and J. Dongarra. Dodging the cost of unavoidable memory copies in message logging protocols. In *Recent Advances in Message Passing Interface. 17th European MPI Users' Group Meeting*, Lecture Notes in Computer Science. Springer-Verlag, 2010. Accepted.

[5] R. Butler and E. Lusk. ADLB library. `http://www.cs.mtsu.edu/~rbutler/adlb/` (July 2010).

[6] M. Chaarawi and E. 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.

[7] E. Chan, M. Heimlich, A. Purkayastha, and R. A. van de Geijn. Collective communication: Theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[8] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[9] R. Cypher and S. Konstantinidou. Bounds on the efficiency of message-passing protocols for parallel computers. *SIAM Journal on Computing*, 25(5):1082–1104, 1996.

[10] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1495, 2001.

[11] G. E. Fagg and J. 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.

[12] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. Dongarra. Extending the MPI specification for process fault tolerance on high performance computing systems. In *Proc. of the International Supercomputer Conference (ICS) 2004*. Primeur, 2004.

[13] M. Farreras, T. Cortes, J. Labarta, and G. Almasi. Scaling MPI to short-memory MPPs such as BG/L. In *ACM International Conference on Supercomputing (ICS)*, pages 209–217, 2006.

[14] W. Gropp and E. Lusk. Goals guiding design: PVM and MPI. In *Proc. of the IEEE Int'l Conference on Cluster Computing (Cluster 2002)*, pages 257–265, 2002.

[15] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd edition*. MIT Press, Cambridge, MA, 1999.

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

[17] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proc. of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, 2007.

[18] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Träff. The scalable process topology interface of MPI 2.2. *Concurrency and Computation: Practice and Experience*, 2010. To appear.

[19] T. Hoefler and J. L. Träff. Sparse collective operations for MPI. In *Proc. of 14th Int'l Workshop on High-level Parallel Programming Models and Supportive Environments at IPDPS*, 2009.

[20] Intel Trace Analyzer and Collector. `http://www.intel.com/cd/software/products/asmo-na/eng/306321.htm` (July 2010).

[21] H. Jitsumoto, T. Endo, and S. 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.

[22] H. Kamal, S. M. Mirtaheri, and A. Wagner. Scalability of communicators and groups in MPI. In *ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[23] KOJAK. `http://www.fz-juelich.de/jsc/kojak/` and `http://icl.cs.utk.edu/kojak/` (July 2010).

[24] S. Kumar, G. Dozsa, J. Berg, B. Cernohous, D. Miller, J. Ratterman, B. Smith, and P. Heidelberger. Architecture of the Component Collective Messaging Interface. 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 23–32. Springer-Verlag, 2008.

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

[26] MPI Forum. MPI: A Message-Passing Interface Standard – Working-Draft for Nonblocking Collective Operations, April 2009.

[27] MPI Forum Fault Tolerance Working Group (Lead: Rich Graham). `http://meetings.mpi-forum.org/mpi3.0_ft.php` (July 2010).

[28] MPI Forum Hybrid Programming Working Group (Lead: Pavan Balaji). `http://meetings.mpi-forum.org/mpi3.0_hybrid.php` (July 2010).

[29] MPI Forum Tools Working Group (Lead: Martin Schulz). `https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/MPI3Tools` (July 2010).

[30] mpiP. `http://mpip.sourceforge.net/` (July 2010).

[31] Paradyn. `http://www.paradyn.org` (July 2010).

[32] Paraver. `http://www.cepba.upc.edu/paraver/` (July 2010).

[33] P. Patarasuk and X. Yuan. Efficient MPI_Bcast across different process arrival patterns. In *22nd International Parallel and Distributed Processing Symposium (IPDPS)*, page 32, 2008.

[34] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the ACM/IEEE Conf. on High Performance Networking and Computing*, page 55. IEEE/ACM, 2003.

[35] PETSc library. `http://www.mcs.anl.gov/petsc` (July 2010).

[36] S. C. Pieper and R. B. Wiringa. Quantum monte carlo calculations of light nuclei. *Annu. Rev. Nucl. Part. Sci.*, 51:53–90, 2001.

[37] R. Rabenseifner, G. Hager, and G. 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.

[38] R. Rabenseifner and J. L. Träff. 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.

[39] A. Rane and D. 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.

[40] R. Ross, N. Miller, and W. Gropp. Implementing fast and reusable datatype processing. 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 404–413. Springer-Verlag, 2003.

[41] TAU - Tuning and Analysis Utilities. `http://www.cs.uoregon.edu/research/tau` (July 2010).

[42] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *Int'l Journal of High-Performance Computing Applications*, 19(1):49–66, 2005.

[43] TotalView Parallel Debugger. `http://www.totalviewtech.com` (July 2010).

[44] J. L. Träff. 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.

[45] J. L. Träff. 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.

[46] J. L. Träff. 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.

[47] J. L. Träff, R. Hempel, H. Ritzdoff, and F. Zimmermann. Flattening on the fly: Efficient handling of MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting*, pages 109–116. Lecture Notes in Computer Science 1697, Springer-Verlag, 1999.

[48] Vampir - Performance Optimization. `http://www.vampir.eu` (July 2010).