Scalable Memory Use in MPI: A Case Study with MPICH2

David Goodell[1], William Gropp[2], Xin Zhao[2], and Rajeev Thakur[1]

[1] Argonne National Lab., Argonne, IL 60439, USA, {goodell,thakur}@mcs.anl.gov
[2] Univ. of Illinois, Urbana, IL 61801, USA, {wgropp,xinzhao3}@illinois.edu

Abstract. One of the factors that can limit the scalability of MPI to exascale is the amount of memory consumed by the MPI implementation. In fact, some researchers believe that existing MPI implementations, if used unchanged, will themselves consume a large fraction of the available system memory at exascale. To investigate and address this issue, we undertook a study of the memory consumed by the MPICH2 implementation of MPI, with a focus on identifying parts of the code where the memory consumed per process scales linearly with the total number of processes. We report on the findings of this study and discuss ways to avoid the linear growth in memory consumption. We also describe specific optimizations that we implemented in MPICH2 to avoid this linear growth and present experimental results demonstrating the memory savings achieved and the impact on performance.

1 Introduction

We have already reached an era where the largest parallel machines in the world have a few hundred thousand cores and are soon approaching an era of million-core systems. For example, an IBM Blue Gene/Q system (Sequoia) to be deployed at Lawrence Livermore National Laboratory in 2012 will have more than 1.5 million cores and a peak speed of 20 petaflops. Roadmaps for future systems indicate that we can expect systems with many millions of cores over the next 5–10 years. For example, a DOE technology and architecture roadmap for exascale envisions a 1 exaflop/s machine by 2018 with 1 *billion* cores [8]. Another significant trend is that although the number of cores is increasing rapidly, the amount of memory available per core is not increasing.

As systems grow to these sizes, many researchers and users wonder whether MPI will scale to such large systems. Scalability of performance is not the only concern; an often-cited concern is the potential memory consumption of MPI at scale. It is generally believed that as the system size grows, the memory consumed by MPI on each process also grows linearly. Given the limited amount of memory per core, it is believed that, unless steps are taken, MPI itself will consume a large fraction of available memory on exascale systems.

Anecdotal evidence exists of isolated examples indicating memory consumption issues in some functions in some MPI implementations, often reflecting a bug in the code or oversight on the part of the developers. At small scale, developers tend to make assumptions or take shortcuts that need to be fixed at scales several orders of magnitude higher. However, quantitative data on MPI memory consumption at scale is hard to find. Particularly lacking is information about what aspects are merely bugs that need to be fixed and what are more intrinsic problems that require a redesign or rethinking of conventional ways of implementing MPI, including changes that may incur a performance penalty at small scale but are necessary for the code to even run at large scale.

To investigate these aspects and address potential problems, we undertook a study of the memory consumed by MPICH2 [7], an MPI implementation that is widely used on many of the largest machines in the Top500 list. We focused particularly on parts of the code where the memory consumption increases linearly with system size. We report on the findings of this study and discuss ways in which such linear growth in memory can be avoided. We also designed and implemented specific optimizations in MPICH2 to avoid this linear memory growth. We describe these optimizations and present experimental results demonstrating the memory savings achieved and the negligible impact on performance.

Related Work. Balaji et al. [3] discuss issues related to scaling MPI to millions of cores, in terms of what is needed both in the MPI specification and in MPI implementations. The authors consider implementation issues in general, not specific to any particular MPI implementation. In this paper, on the other hand, we focus on identifying and fixing memory scalability issues in the MPICH2 implementation of MPI. Other researchers have also explored memory-space related optimizations for MPI implementations, such as the memory required for storing communicators and groups [5, 6, 9].

2 Apparent Nonscalable Memory Use in MPI

At first glance, MPI appears to have a number of areas where it must store $\mathcal{O}(p)$ data on each MPI process, where p is the number of processes in the MPI program. In this section, we discuss some of these areas and comment on what MPI really requires for them. For simplicity and to match the behavior of most MPI implementations on large systems, we assume that all processes are in `MPI_COMM_WORLD` (e.g., no dynamic processes).

Group Representation. An MPI *group* describes a collection of processes. The obvious implementation is an enumeration of processes by some identifier, such as rank in `MPI_COMM_WORLD` or an IP address and process ID. However, MPI only requires that this information be available, not the form in which it is stored. Lossless compression of the data is permitted; for example, for `MPI_COMM_WORLD`, the group can be represented as simply `0:p-1` (all ranks from 0 to $p-1$, requiring only a few words of storage).

Connections and Message Buffers. MPI allows a process to communicate directly with all other processes. It is sometimes alleged that this feature requires MPI to maintain $\mathcal{O}(p)$ data for such connections and to allocate significant buffer space to each possible connection. For example, providing only 16 KB for each connection for eager message delivery would require 16 GB on each process for a million-process MPI program (a total of 16 petabytes of memory). However, MPI does not specify when connections are established or how buffer memory is allocated and associated with connections; in fact, MPI does not even define "connections." For example, an MPI implementation may instantiate a connection only when needed and dynamically associate buffer memory to active connections. For a scalable application (which by definition cannot communicate with $\mathcal{O}(p)$ other processes), only a small number of such connections can be active.

RMA Windows. Each MPI RMA window is created with its own displacement value, start address, size, and info object for hints. Because RMA is for one-sided operations, it is natural to store information about the remote windows locally, where the information can be quickly accessed. However, locally storing the information for all ranks is not required by MPI. Other options include using a cache strategy for such data, acquiring it on first use, or even fully distributing the data and using one-sided operations to acquire the data.

Nonscalable Arguments. Some MPI routines have array parameters of size p. These are nonscalable routines and simply cannot be used in a scalable application. They do not reflect a problem in an MPI implementation.

In all of these cases, allocating memory for each of the $\mathcal{O}(p)$ items both simplifies the implementation and may be (slightly) faster. However, $\mathcal{O}(p)$ memory is not required, and we argue that the performance cost is often negligible.

3 Memory Usage in MPICH2

MPICH2 has been carefully designed and developed to be adaptable to environments with a paucity of memory resources. The current design is parsimonious with memory in certain areas, such as the usage and representation of MPI groups. In other areas of the code, decisions were consciously made to trade increased memory consumption to obtain decreased algorithmic running times. In a severely memory-constrained environment, some of these decisions could be revisited and potentially altered when such a change would be beneficial. Yet unsurprisingly, several memory inefficiencies remain in the current code. We discuss these strengths and weaknesses of the current stable version of MPICH2 in this section.

3.1 Link-Time Program Text Size Savings

MPICH2 was designed from the beginning to be highly modular. Less-used code is organized so that the code and the associated data structures are included (by the linker) only when actually used by the application. For example, the buffered send code is included only if the user references one of the buffered send routines. The code for each of the MPI collectives is another example. This reduces the size of the executable code, which is good for both very large systems and ones where use of dynamically loaded code from shared libraries may stress the I/O system, such as nodes without local or nearby disks.

3.2 One-Sided Communication

The current implementation of MPICH2 stores a copy of the window start address, size, and displacement unit of all processes locally on each process for easy lookup. This clearly requires $\mathcal{O}(p)$ space on each process, which is nonscalable. Possible approaches to fix this problem are outlined in Section 2, and we will consider them as part of our future work.

3.3 MPI Groups

Within every MPI process, the process assigns to each other process to which it is connected[3] a local process ID, or *LPID*, in the range $[0, p)$, where p is the number of connected processes. Note that LPIDs are not unique across processes; they exist as a purely local concept to simplify process-related bookkeeping operations.

An MPI group is a totally ordered set of processes in which each process is indexed by an integer rank in the range $[0, p_g)$, where p_g is the size of the group. This set is currently stored as a dense `int` array of LPIDs, where element i in the array stores the LPID of the process corresponding to rank i in the group. This information is sufficient, though nonoptimal, to be able to correctly implement all local `MPI_Group_` operations.

As a performance optimization, a list of indices sorted by increasing LPID order can also be constructed and stored in the group object, which significantly improves the performance of `MPI_Group_translate_ranks`, `MPI_Group-_compare`, and `MPI_Group_union`. In order to conserve memory (and list construction time) in codes that do not use these routines, this sorted LPID list is constructed lazily only when these routines are first invoked. Constructing this list requires an $\mathcal{O}(p_g \log p_g)$ time sorting step and roughly doubles the size of the group object for nontrivial values of p_g.

3.4 Virtual Connections

In most practical MPI implementations, each process must maintain at least a modicum of state for each other process with which it is communicating. In MPICH2, this state is kept in a *virtual connection* object (or *VC*) associated with the remote process. MPICH2's current implementation creates one of these objects for each other process in the system, on every process. That is, across an entire MPI application these VC objects consume $\mathcal{O}(p^2)$ memory.

This obvious scalability issue is addressed in Section 4. However, even the current design is more scalable than a naïve implementation. Many buffers that

[3] See MPI-2.2, § 10.5.4, for a formal definition of "connected" in this context.

are attached to the VC object are not created until communication actually occurs with the process corresponding to that VC. For connection-oriented communication substrates such as TCP, these connections are not created until communication actually occurs, thereby conserving operating system resources.

The VC implementation provides another example of a location where additional space is consumed in exchange for reduced access time. Each VC contains a "scratch pad" area that may be used by lower-level code to store per-VC information. In order to decouple lower layers from the upper layers of MPICH2, such storage space must exist. However, it would also be sufficient for this space to be just large enough to hold a pointer, such that the lower-level code could allocate a separate object and store a pointer to it in this minimal scratch pad region. This approach would require an additional pointer dereference for the lower-level code to access its own VC-specific data. Instead, by making the scratch pad region larger, this additional pointer dereference is saved for latency-sensitive data accesses that can be fit into the scratch pad. Of course, tuning the size of this scratch pad becomes critical for large p.

3.5 Communicator and Topology Information

Among many responsibilities, MPI communicators are responsible for storing enough data in order determine which underlying process corresponds to a given rank in that communicator. For example, when the user calls `MPI_Send(..., 5,...,comm)`, the implementation must be able to determine that rank 5 in `comm` will result in communication with a particular process on a particular network host. More concretely in the case of MPICH2, this means that given a communicator and a rank, the implementation must be able to produce a VC object. This translation is currently supported by a *virtual connection reference table* (or *VCRT*).

VCRTs consist of a dense array of *VC references* (or *VCRs*), indexed by communicator rank. The VCR is an opaque type, but because of practical details of the interface, it must typically be implemented as a pointer to the underlying VC. Each communicator stores a pointer to its VCRT and manipulates reference counts inside that VCRT. This reference counting permits shallow copies of the VCRT for the common case of `MPI_Comm_dup`, reducing memory consumption. However, besides sharing a VCRT between two communicators, VCRTs themselves have only $\mathcal{O}(p)$ per communicator space scalability in the general case.

MPICH2 stores additional information on a per process and per communicator basis in order to support hierarchical collective communication algorithms. For each connected process a *node ID* is stored, consuming $\mathcal{O}(p)$ memory on each process. This approach enables creating two internal subcommunicators for each user-created communicator: one that contains only "node leaders" and another that contains only processes on the same node. For a top-level communicator of size p that is spread evenly over k nodes, the node-leaders communicator will contain k members, while the node-local communicator will contain p/k members. Every process will be a member of a node-local communicator, but only the node leaders will be a member of the leader communicator. In MPICH2's current implementation these communicators will consume $\mathcal{O}\big(p^2/k + pk\big) \Rightarrow \mathcal{O}\big(p^2\big)$ memory across the whole system (assuming constant k).

4 Steps to Reduce MPICH2 Memory Consumption

The current deficiencies in MPICH2 memory usage mentioned above can be addressed in several ways. We detail here solutions we have implemented in an experimental version of MPICH2, and we outline several additional solutions we intend to implement in the near future.

4.1 Implemented Solutions

The most serious scalability problem discussed earlier is the $\mathcal{O}\big(p^2\big)$ memory consumption by VCs (across the whole system) even when communication takes place with zero or few partners. To rectify this issue, we have developed a prototype version of MPICH2 that substantially overhauls the way VCs are managed.

Under the new scheme, entire VC objects are created lazily only as needed instead of statically at `MPI_Init` time. This change required a fundamental shift in the way VCs are stored and accessed. The per communicator VCRTs discussed in Section 3.5 have been eliminated and replaced with a similar, yet more efficient concept: the *LPID mapping* (or *LPM*). These objects perform a similar role; but rather than mapping communicator ranks to VCs directly and always via a dense array mechanism, the LPM maps communicator ranks to LPIDs. This mapping decouples the upper-level code, for example MPI collective routines, from any notion of VCs that exist only at the lower level.

Unlike VCRTs, LPMs are truly opaque objects that are accessed only via function calls and macros. This design provides the opportunity to encode the communicator representation in the most succinct, memory-efficient manner possible. Examples include using compression techniques that take advantage of domain-specific knowledge [9] or more general compression methods [4]. Another example of domain-specific compression is supporting identity mappings, wherein the LPID is always equal to the communicator rank. Implementing this identity mapping is trivial, given the new interface, and reduces per process memory consumption from $\mathcal{O}(p)$ to $\mathcal{O}(1)$ for communicators for which this mapping holds (such as `MPI_COMM_WORLD`).

Conveniently, the LPM concept and interface also permitted us to unify the representation of groups and the representation of communicator VC contents. Future improvements to this common LPM facility will yield dividends in both the group and communicator subsystems of MPICH2.

At a lower level, VCs are obtained only via APIs that refer to them by their LPIDs. This design permits true lazy instantiation and storage of VC objects, such as in a hash table, since upper-level code no longer holds pointers to all VCs. This hash-based approach is exactly what we implemented, with a runtime environment variable to select between the hash table and a dense, fully populated array.

For convenience and robustness, we used the open source `uthash` package [10]. Additional constant factor time and memory savings may be possible with an alternative implementation.

4.2 Proposed Solutions

Though we implemented several space-saving techniques in our experimental version of MPICH2, there remain many that we did not have time to implement. For example, data on the same SMP node can be shared, such as information about `MPI_Win` objects. Data-caching strategies can be employed, particularly if efficient remote memory access is available. We leave these approaches to future work.

5 Results

In this section we provide experimental evidence that an MPI implementation can limit the use of memory for scalable applications without a significant performance impact. We first look at some simple benchmarks, including ping-pong performance microbenchmarks, and then evaluate several application-based benchmarks.

All results were gathered on the "Fusion" cluster at ANL. Each node consists of two Intel Xeon X5550 quad-core processors, and the nodes are connected by QDR Infiniband. MPICH2 was configured as `--with-device=ch3:nemesis:tcp` and `--enable-fast`.

5.1 Scalable Memory Use

To validate the expected memory consumption of the prototype, we crafted three microbenchmarks that isolate basic communication behavior from more sophisticated application MPI usage. These microbenchmarks respectively perform no communication, scalable communication (a single `MPI_Allreduce`), and non-scalable communication (pairwise communication between all processes). Furthermore, the MPI library was instrumented to permit memory consumption measurements to be taken. The results from running these simple programs respectively provide minimum, modest, and maximum memory consumption baselines that are harder to observe as clearly in applications with more sophisticated communication patterns.

Figure 1 shows the results of running these experiments with the lazy initialization prototype code enabled. As expected, the "no communication" and "allreduce" benchmarks consumed essentially no additional memory per process as the job size was increased, while the "all communication" benchmark showed per process memory consumption increasing linearly with job size. This increase indicates an $\mathcal{O}(p^2)$ systemwide memory consumption scalability problem, one that our technique has addressed for programs with a scalable communication pattern.

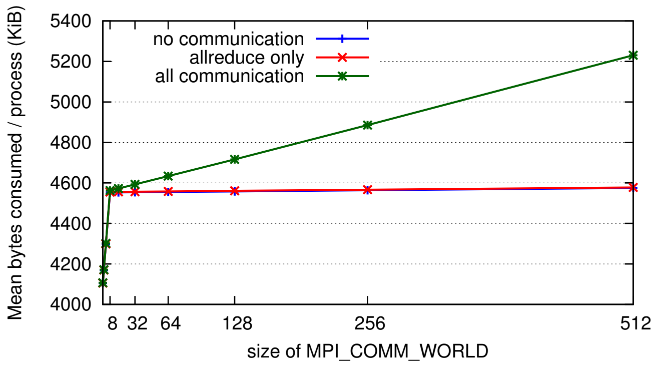

Fig. 1. Per process memory consumption in the prototype for three microbenchmarks.

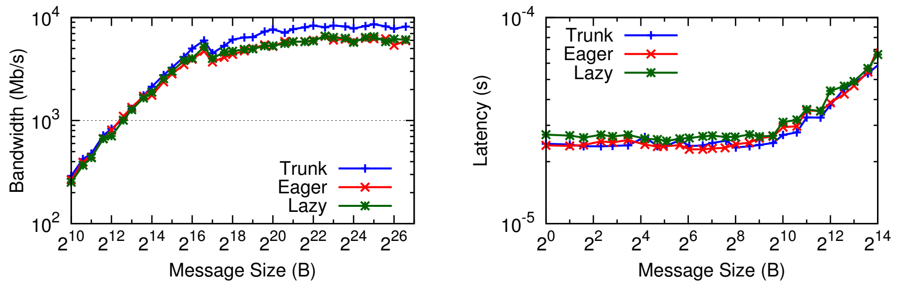

Fig. 2. Netpipe ping-pong performance results (log-log plot for relevant message sizes).

5.2 Performance Impact

The techniques discussed in Section 4 are expected to at least slightly impact performance. Figure 2 shows MPI-level bandwidth and one-way latency numbers for the stable ("Trunk") version of MPICH2 as a reliable baseline, as well as the prototype configured to use an eagerly constructed dense array ("Eager") or lazily constructed sparse hash table ("Lazy") for VC storage. Both a slight decrease in large-message bandwidth and a slight increase in small-message latency can be seen. We emphasize, however, that the prototype code has not been tuned to any noteworthy extent; we expect to eliminate most of this performance gap with further effort.

5.3 Application Impact

We measured the impact of our changes on scalable applications by examining the performance and memory consumption behavior of certain NAS Parallel Benchmarks [2] and the Sequoia AMG benchmark that are representative of application behavior. All of these benchmarks exhibit fairly scalable communication patterns; that is, the number of communication partners remains flat

Table 1. Performance of selected NAS Parallel Benchmarks, version 3.3 run with 512 processes.

Benchmark	MPI	Time (s)		Memory/Process (kiB)	
cg.D.512	Trunk	536.77		5,149.2	
	Eager	520.55	(−3.02%)	5,144.7	(−0.09%)
	Lazy	556.82	(+3.74%)	4,588.2	(−10.89%)
mg.D.512	Trunk	18.69		5,154.2	
	Eager	19.19	(+2.68%)	5,154.3	(+0.00%)
	Lazy	19.49	(+4.28%)	4,602.3	(−10.71%)

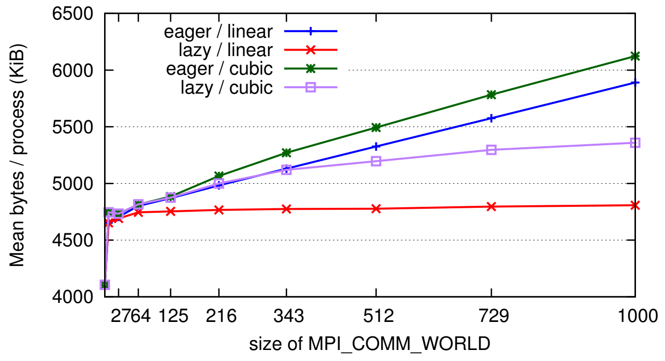

Fig. 3. Per process memory consumption in the prototype for the Sequoia AMG benchmark.

or increases slowly as job size increases. These codes are also well known and commonly used to represent the behavior of many real-world MPI numerical applications.

Table 1 lists the performance impact and average per-process memory consumption of our techniques when applied to the CG and MG class D NAS Parallel Benchmark. The benchmarks were run with the same three configurations from Figure 2. At this modest scale MPI memory consumption is reduced in the Lazy approach by approximately 550 bytes per process (≈11%), at a cost of less than 5% in performance. We did observe variability in the run times, despite great consistency in the memory consumption numbers, which we attribute to noise from the shared Infiniband network on this system.

Figure 3 shows per process memory consumption versus job size when running the Sequoia AMG benchmark [1] on the prototype with both the eager and lazy VC initialization strategies. The benchmark was configured to solve a Laplace-type problem[4] with two different three-dimensional processor layouts. The first layout was cubic (e.g., 36 processes organized as $P_x \times P_y \times P_z = 6\times6\times6$). The plot clearly shows a substantially slower-growing memory consumption

[4] AMG was run with the following options: `-laplace -n 25 25 25 -solver 4`.

curve for this case when lazy VC initialization is used. The second layout was entirely linear (e.g., $36 \times 1 \times 1$). Although unrealistic as a choice of typical application parameters, this layout has far fewer communication partners, which yields the expected almost entirely flat per-process memory consumption curve.

6 Conclusions

We have shown that an MPI implementation can be constructed so that memory use grows slowly as the number of processes increase and that the performance cost for a real application is low.

Acknowledgments

This work was supported by the U.S. Department of Energy, under Contract DE-AC02-06CH11357 and Award DE-FG02-08ER25835, and by the National Science Foundation Grant #0702182.

References

1. ASC Sequoia Benchmark Codes: AMG. `https://asc.llnl.gov/sequoia/benchmarks/#amg` (May 2011)
2. Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA (1995)
3. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Hoefler, T., Kumar, S., Lusk, E., Thakur, R., Träff, J.L.: MPI on millions of cores. Parallel Processing Letters 21(1), 45–60 (March 2011)
4. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: Proc. of 26th Int'l Symposium on Theoretical Aspects of Computer Science (STACS). pp. 111–122 (2009)
5. Chaarawi, M., Gabriel, E.: Evaluating sparse data storage techniques for MPI groups and communicators. In: Proc. of the 8th International Conference on Computational Science (ICCS). Lecture Notes in Computer Science, vol. 5101, pp. 297–306. Springer-Verlag (2008)
6. Kamal, H., Mirtaheri, S.M., Wagner, A.: Scalability of communicators and groups in MPI. In: Proc. of the ACM International Symposium on High Performance Distributed Computing (HPDC) (2010)
7. MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`
8. Stevens, R., White, A.: Report of the workshop on architectures and technologies for extreme scale computing. `http://extremecomputing.labworks.org/hardware/report.stm` (December 2009)
9. Träff, J.L.: Compact and efficient implementation of the MPI group operations. In: Proc. of the 17th European MPI Users' Group Meeting. Lecture Notes in Computer Science, vol. 6305, pp. 170–178. Springer-Verlag (2010)
10. uthash. `http://uthash.sourceforge.net/` (May 2011)