On the Reproducibility of MPI Reduction Operations

Pavan Balaji and Dries Kimpe

Mathematics and Computer Science Division, Argonne National Laboratory
{balaji, dkimpe}@mcs.anl.gov

***Abstract*—Many scientific applications go through a thorough validation and verification ("V&V") process to demonstrate that the computer simulation does, in fact, mirror what can be analyzed through physical experimentation. Given the complexity of and the time required for the V&V process, applications that have been validated and verified are typically conservative with respect to changes that might impact the reproducibility of their results. In the extreme case, some applications require bitwise reproducibility for their simulations. Thus, any change made to the application, the hardware, or the software on the system needs to preserve the bitwise reproducibility of the application. Such a constraint, however, can drastically affect the performance efficiency of the system in many ways. In this paper, we analyze the impact of such bitwise reproducibility on the performance efficiency of MPI reduction operations.**

***Index Terms*—MPI; Reduction Operations; Reproducibility; Performance; Multicore; Topology**

I. Introduction

The Message Passing Interface (MPI) is considered to be the de facto standard for parallel programming today. A vast number of scientific applications use MPI and rely on its functionality and performance in order to scale to the largest systems in the world. One specific genre of functionalities in MPI is the reduction class of operations, which includes **MPI_REDUCE**, **MPI_ALLREDUCE**, **MPI_SCAN**, and **MPI_EXSCAN**. Such functionality is heavily used in scientific applications today, often in the performance-critical path, making their performance and correctness an important expectation on MPI implementations.

Many scientific applications go through a thorough validation and verification process to demonstrate that they closely simulate physical phenomenon. Such applications are typically conservative with respect to changes that might impact the reproducibility of their results. In the extreme case, some applications require bitwise reproducibility for their simulations. Thus, any change made to the application, the hardware, or the software on the system needs to preserve the bitwise reproducibility of the application.

Why is bitwise reproducibility complicated? Many parallel scientific applications are not bitwise reproducible for two primary reasons. The first is floating-point arithmetic, which suffers from rounding errors and other inadequacies in the bit representation with a limited number of bits. Consequently, floating-point operations (such as SUM or PRODUCT) are commutative but not associative. Thus, the order in which they are applied makes a difference to the final outcome. The second reason is nondeterministic computational flow. In parallel applications where the progress of each process is not fully deterministic (e.g., if the computational order depends on the order in which messages are received), different runs of the same application can lead to different outcomes.

While cumbersome to achieve, bitwise reproducibility is still considered to be critical for some applications, sometimes just for contractual reasons (e.g., drug design or nuclear reactor design). Such reproducibility often is also useful for debugging purposes or for ensuring the reliability of output [15]. Such a constraint on bitwise reproducibility, however, can have a tremendous impact on the performance efficiency of the system in many ways. For example, fully avoiding such errors requires computations to use sequential executions and a fixed order of computations, since the hardware floating-point precision cannot be arbitrarily increased to accommodate application needs. These requirements are too expensive even on current systems and will be impractical on the next generation of systems. For example, MPI reduction operations that are meant for scalable computation on input data from a large number of processes have to guarantee a deterministic behavior if they are to be used by such applications.

In this paper, we analyze the impact of such bitwise reproducibility on the performance efficiency of MPI reduction operations. While the MPI standard does not guarantee that MPI reduction operations are bitwise reproducible, such behavior is strongly recommended by the standard for MPI implementations. Specifically, the MPI-3.0 standard states the following (page 175, lines 9–13):

It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of ranks.

Consequently, several MPI implementations provide the capability for such reproducibility, either by default or through an environment or configuration setting. In order to achieve this, however, the MPI implementation needs to be highly conservative with respect to the optimizations it can perform. Such a conservative approach can have a substantial impact on performance, especially with respect to the MPI implementation's ability to take advantage of the hardware topology in order to minimize data movement.

In this paper, we first analyze the algorithmic impact of hardware topology-based optimizations that MPI implementations can utilize, focusing on the reproducibility of results and the performance of MPI reduction operations. In addition to this detailed algorithmic analysis, we present performance numbers demonstrating the improvement in performance that such architectural topology-aware optimizations can bring.

```
MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
 IN     sendbuf             address of send buffer (choice)
 OUT    recvbuf             address of receive buffer (choice, significant only at root)
 IN     count               number of elements in send buffer (non-negative integer)
 IN     datatype            data type of elements of send buffer (handle)
 IN     op                  reduce operation (handle)
 IN     root                rank of root process (integer)
 IN     comm                communicator (handle)

int MPI_Reduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
               int root, MPI_Comm comm)
```

Fig. 1: Prototype of MPI_REDUCE

We note that reproducibility does not necessarily mean accuracy. If multiple runs of an application give the exact same result, it is still considered reproducible even if the result is not as accurate as what an infinite-precision system can achieve.

The rest of the paper is organized as follows. We provide a brief overview of various MPI reduction operations in Section II and the workings of topology-aware reduction mechanisms in Section III. In Section IV-B, we analyze the impact of the reproducibility constraints on MPI reduction operations with respect to performance. We present experimental results demonstrating such a performance impact, and we analyze their behavior in Section VI. Other literature related to this paper is presented in Section VII. We summarize our findings and present concluding remarks in Section VIII.

II. Overview of MPI Reduction Operations

MPI reduction operations form an important class of computational operations first introduced in MPI-1. Several new functions were added in this class in MPI-2 and MPI-3. MPI reduction operations fall into three categories:

1) Global Reduction Operations: **MPI_REDUCE**, **MPI_IREDUCE**, **MPI_ALLREDUCE** and **MPI_IALLREDUCE**.
2) Combined Reduction and Scatter Operations: **MPI_REDUCE_SCATTER**, **MPI_IREDUCE_SCATTER**, **MPI_REDUCE_SCATTER_BLOCK** and **MPI_IREDUCE_SCATTER_BLOCK**.
3) Scan Operations: **MPI_SCAN**, **MPI_ISCAN**, **MPI_EXSCAN**, and **MPI_IEXSCAN**.

The primary idea of these operations is to collectively compute on a set of input data elements in order to generate a combined output. For instance, consider the prototype of **MPI_REDUCE**, as illustrated in Figure 1. **MPI_REDUCE** is a collective function where each process provides some input data (e.g., an array of double-precision floating-point numbers). This input data is combined through an MPI operation, as specified by the "op" parameter. Most applications use MPI predefined operations such as summations or maximum value identification, although some applications also utilize reductions based on user-defined function handlers.

The MPI operator "op" is always assumed to be associative. All predefined operations are also assumed to be commutative. Applications, however, may define their own operations that are associative but not commutative.

The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, an MPI implementation can take advantage of associativity, or associativity and commutativity of the operations, in order to change the order of evaluation. Doing so may change the result of the reduction for operations that are not strictly associative and commutative, such as floating-point addition. With that caveat, however, the MPI standard strongly encourages implementations to implement reduction operations in a deterministic manner, which can reduce the extent to which they can take advantage of the associativity and commutativity of the operations and thus, in effect, fall back to the canonical rank-ordered evaluation of the reduction.

The primary reason for the loss of precision, and hence the nondeterminism in the result, comes from the order in which the data elements are reduced. For example, in the case of summation of input data values that cover a wide range, operations that combine very large values with very small values can lose data because of restrictions in the number of bits used in the representation. On the other hand, combining small values first allows us to do so without losing precision. Such contributions of small values from a large number of processes can have a large impact on the aggregate value contributed and thus on the overall result.

Since no such ordering of operation is described in MPI reduction operations, the loss of precision depends on the specific order in which the operations are performed. Consequently, if the order of operations changes, the loss of precision changes as well, making the result nondeterministic. Moreover, in the past few years, systems with millions of cores have been installed [11], [13]. As MPI applications scale to such large systems, the number of possible orderings that the MPI implementation can utilize increases as well, thus impacting the amount of nondeterminism possible. Further, since each process contributes one input data element for the reduction operation,[1] as the number of processes increases, the percentage impact that small values can have on the overall result would increase as well.

[1]Each process can contribute an array of elements for reduction. However, each element is reduced individually. Thus, with respect to loss of precision, only the number of processes contributing input data matters, and not the number of elements contributed by each process.

III. Topology-Aware Reduction

Most MPI implementations take advantage of hardware topology information in order to optimize data communication. This information includes point-to-point communication as well as collective communication operations, including reduction operations. In this section, we describe the algorithms used in the MPICH implementation of MPI for **MPI_REDUCE** and **MPI_ALLREDUCE**. The algorithms used for other reduction operations are similar and hence are not discussed in this paper.

A. *MPI_REDUCE*

When topology-awareness is not enabled, MPICH uses rank-order-based deterministic algorithms for the reduction operation. Specifically, if the message size is small (by default, less than 2 Kbytes) or if the number of processes in the communicator is not a power of two, MPICH uses a binomial tree algorithm based on the ranks. That is, the processes are considered to be in a binomial tree virtual topology, with the root of the reduction being the root of the tree. When the message size is large and the communicator size is a power of two, MPICH uses a reduce-scatter followed by a gather to the root.

When topology-awareness is enabled, MPICH first detects which processes reside on the same node. Using this information, it does a non-topology-aware reduction within the node for all nodes except the root node. Next, the temporary results of the first reduction on each node are further reduced onto the root node (again as a non-topology-aware reduction). Then, the root node reduces the internode result with the inputs from other processes on its node.

B. *MPI_ALLREDUCE*

When topology-awareness is not enabled and the number of processes is a power of two, MPICH uses a recursive-doubling algorithm for small message sizes or if the number of data elements is not a power of two or if the operation is user-defined. Otherwise, it uses a reduce-scatter followed by an allgather. If the number of processes is not a power of two, a few of the processes first do pairwise reductions to reduce the number of processes to a power of two. Then, they use the power-of-two algorithm described above.

When topology-awareness is enabled, MPICH first detects which processes reside on the same node. Then it does a non-topology-aware reduction within each node, followed by a reduction based on these results across nodes. It then broadcasts the final result to all processes.

C. *Discussion of Topology-Aware Algorithms*

Clearly, the algorithms used in the topology-aware reduction operations (both **MPI_REDUCE** and **MPI_ALLREDUCE**) are different from those in the non-topology-aware reduction operations. More important, the ordering of operations varies dramatically for different topologies. For example, the order of operations on a system with four nodes each with four cores is quite different from the order of operations on a system with eight nodes each with two cores. This makes the final result be heavily dependent on the hardware topology being used.

We note, however, that the topology-aware algorithms do not, by themselves, lose arithmetic precision any more than a non-topology-aware algorithm would. They only alter the order in which the reduction operations are done compared with that of a non-topology-aware algorithm or even a topology-aware algorithm on a different hardware topology. Thus, without information about the actual input data, we cannot claim that the topology-aware algorithms improve or degrade precision. We point out only that the hardware topology affects the reproducibility of the result.

IV. Consequences of Nondeterminism

Enabling nondeterministic behavior for reduction operations (for example, by making the algorithm depend on external factors such as the hardware topology) potentially has a number of consequences on the application and user. In this section, we analyze and discuss some of these consequences.

A. *Accuracy*

For mathematical operations, the order in which the reduction is performed can have a profound effect on the correctness and accuracy of the returned result. This is especially true when the encoding of the numerical value is not exact, as is often the case for floating-point numbers. However, even for integer encodings, typically assumed to be "exact" within a given range, changing the order of addition or subtraction can introduce overflow or underflow. An excellent discussion of these issues, including the accuracy and stability of reduction operations and how the order in which they are performed affects them, can be found in [9].

While the MPI standard encourages implementors to implement the reduction functionality in such a way that the result is the same given *the same arguments to the function (including the same order)*, it does not specify that this order needs to be in rank order. In fact, the standard explicitly states that an implementation is free to make use of associativity when possible. While in the strict mathematical sense, addition and subtraction are associative, when introducing rounding errors due to the floating-point storage format, this is no longer true. In practice, this means that programs that require the exact knowledge of the order in which the reductions are performed—typically for reasons of numerical stability or accuracy—cannot use the MPI reduction operations, whether the reduction algorithm is topology aware or not.

B. *Reproducibility*

The motivation behind the recommendation to implementors in the MPI standard is not to ensure accuracy but to ensure reproducibility. Reproducibility is important for many reasons. For example, it helps debugging. If the user can trust that MPI functions behave in a reproducible manner, and the rest of the program does not contain any nonreproducible behavior, changing output for the same input is a clear indication of program error. In addition, a reproducible program is deterministic. For the same set of inputs, the same output will be generated. Deterministic algorithms are easier to reason about, and automated software tools (such as correctness checkers) often require programs to be free of nondeterministic effects.

While topology-aware reduction operations should not affect accuracy (any more than the existing, reproducible algorithms), a topology-aware algorithm can have an effect on the reproducibility of the program, since the exact outcome of the reduction depends not only on the parameters passed to the reduce function on each rank but also on the underlying topology and the way each rank is mapped to that topology. We note, however, that if the user ensures that the mapping and topology do not change, the topology-aware algorithms are as reproducible as the topology-unaware algorithms.

Specifically, as described in [9], for recursive pairwise summation, the absolute error is provided by

$$\mathrm{abs(E)} = \mathrm{g(log_2(N))} \times \mathrm{SUM(abs(X}_i\mathrm{))} \quad (1)$$

(and)

$$\mathrm{g(N)} = (\mathrm{N} \times \mathrm{u}) \,/\, (1 - (\mathrm{N} \times \mathrm{u})) \quad (2)$$

where X_i are the numbers being reduced and u is the unit roundoff (relative error) for double precision (about 1e-16 for double precision).

V. Other Sources of Nondeterministic Behavior

When considering the consequences of the less deterministic behavior of topology-aware reduction algorithms, a discussion of other sources of nondeterminism is in order. After all, existing nondeterministic behavior in other components in the typical application workflow would limit complications caused by introducing a nondeterministic reduction operation. As discussed in Section IV-B, depending on the assumptions made by the user, nondeterministic behavior can potentially affect reproducibility as well. In this section, we discuss sources of nondeterminism originating outside the scope of the topology-aware algorithms and how these affect accuracy, reproducibility, or both.

In practice, few programmers write direct assembler code. Hence, the translation from a higher-level programming language to machine code might introduce nondeterministic results. The freedom given to the compiler depends on the programming language. For ANSI Fortran, the compiler is allowed to reorder expressions at will, as long as explicit parentheses added by the programmer are respected. In contrast, ANSI C and C++ are relatively strict, disallowing reassociation and forcing evaluation from left to right even in the absence of parentheses [1]. In addition, the same program can be compiled for different architectures, where differences in instruction set could affect the result, even when everything else (compiler, etc.) remains the same. For example, if an architecture supports *fused-multiply-add* (FMA) instructions (as is the case for most Intel, AMD, and ARM processors released in 2012 or later), which support computing $a + (b \times c)$ in one instruction, the default mode for the compiler is typically to enable the use of FMA unless explicitly disabled. While FMA typically improves accuracy by rounding the result only once (compared with two rounding operations without FMA), there is a chance that FMA could introduce exceptions or errors in certain situations [10], causing nondeterministic results. Clearly, if one wants absolute control over accuracy, the only way to be sure is to directly issue machine language instructions. In addition, in order to get reproducible execution, the same binary (including any dynamically loadable libraries) needs to be used.

The OpenMP parallelization directives [2] also support a reduction operation. As is the case for MPI, a tradeoff can be made between determinism, reproducibility, performance, and accuracy. For at least one compiler (Intel), the default is to use nondeterministic reductions. In order to get deterministic, reproducible results, the user has to explicitly force deterministic reductions. Additionally, static thread scheduling and explicitly determining the number of helper threads are required in order to get reproducible results [1].

While the MPI standard encourages library implementors to provide reproducible reduction functionality when the arguments and their order remain constant, one can implement multiple reduce algorithms and choose between them based on specific properties of the input arguments, such as the number of ranks in the communicator or the size of the base type. Doing so does not violate the recommendation of the standard; and given the performance advantage (certain algorithms show better performance for a specific range of parameters), many implementations have chosen to do so. Many of these implementations expose the parameters that determine when a certain algorithm is used. For example, the Open MPI [6] implementation of MPI enables per-installation tuning of these parameters [3]. In the STAR-MPI library, collective routines automatically self-tune during application execution, effectively introducing nondeterministic reduce behavior even when keeping arguments to the reduce function constant [4].

Certain parallel machines, such as the IBM Blue Gene/L [7] and later generations, offer hardware acceleration of reduce operations. However, the hardware typically has more restrictions than does a software algorithm, causing these hardware collectives to be activated only when a strict set of circumstances is fulfilled, falling back to one of the software algorithms otherwise. For example, when using a subset of the machine, there are many ways to embed a lower-dimensional subset into a higher-dimensional network topology. Only a few of these mappings will support hardware collective acceleration. In practice, this means that unless a user runs on exactly the same set of nodes, an MPI reduction will not be reproducible, even if all other factors (such as the compiler and the application binary) remain the same.

Considering these examples, and given the amount of software and hardware involved in running a modern application, ensuring strict reproducibility and absolute control over accuracy is a very hard goal to obtain.

VI. Analytical and Experimental Results

In this section, we present the results of our evaluation of the impact of topology on the performance of MPI reduction operations. Two sets of evaluations are presented. The first set represents an analytical analysis of the loss of precision in reduction operations. These are based on analytical analysis discussed in Section IV-B and are presented in Section VI-A. The second set of evaluations represents an experimental analysis of the performance impact of topology-aware reductions; the results are presented in Section VI-B.

A. Analytical Analysis

In this section, we present an analytical analysis of the error bounds of topology-aware reduction operations. While topology-aware algorithms do not, by themselves, lose arithmetic precision any more than a non-topology-aware algorithm would, they alter the order in which the reduction operations are done, causing nondeterminism in the result. In this section, we analytically showcase the extent of nondeterminism possible with varying topologies.

As described in Section IV-B, the maximum error (and hence the maximum amount of nondeterminism) depends on the number of input values and thus, in the case of the reduction operation, the number of processes in the system. In our experiment, we varied the number of processes in the system. For double-precision arithmetic (unit roundoff around 1e-16), the denominator in Equation IV-B tends to 1 for most realistic system sizes today. This makes the Equation IV-B proportional to the unit roundoff and, for a given input data set, increasing logarithmically with the number of processes in the system.[2]

For small system sizes, the relative error compared with the input data elements is not too high, at around 1e-16. However, the error increases with increasing system size and reaches 2e-15 for a million processes. Given that several systems have already crossed the million-process mark, such errors are not unreasonable on real systems today.

B. Experimental Analysis

For the experimental analysis, we used a 320-node system, with each node consisting of two Intel Xeon X5550 quad-core CPUs (total of 2,560 cores). The nodes are connected with Mellanox QDR InfiniBand HCAs. Our implementation is based on the mpich git master as of July 8, 2013.

Both **MPI_REDUCE** and **MPI_ALLREDUCE** were evaluated, operating on a varying-size vector of **MPI_DOUBLE**, using the **MPI_SUM** operation. For the experiments, a number of topologies, ranging from one core per node to eight cores per node, are evaluated, with the latter matching the real hardware topology of the machine. The default (topology-unaware) algorithm is equivalent to the topology-aware algorithm assuming a topology of a single core per node. The topology-aware reduction is evaluated on communicators ranging from 8 ranks up to 2,048 ranks. For each communicator size, the base type of the underlying reduction is varied from 1 up to 1,048,576 double-precision floating-point numbers (with each double-precision number requiring 8 bytes of storage). The base type size in the graphs indicates number of floating-point values, not the number of bytes.

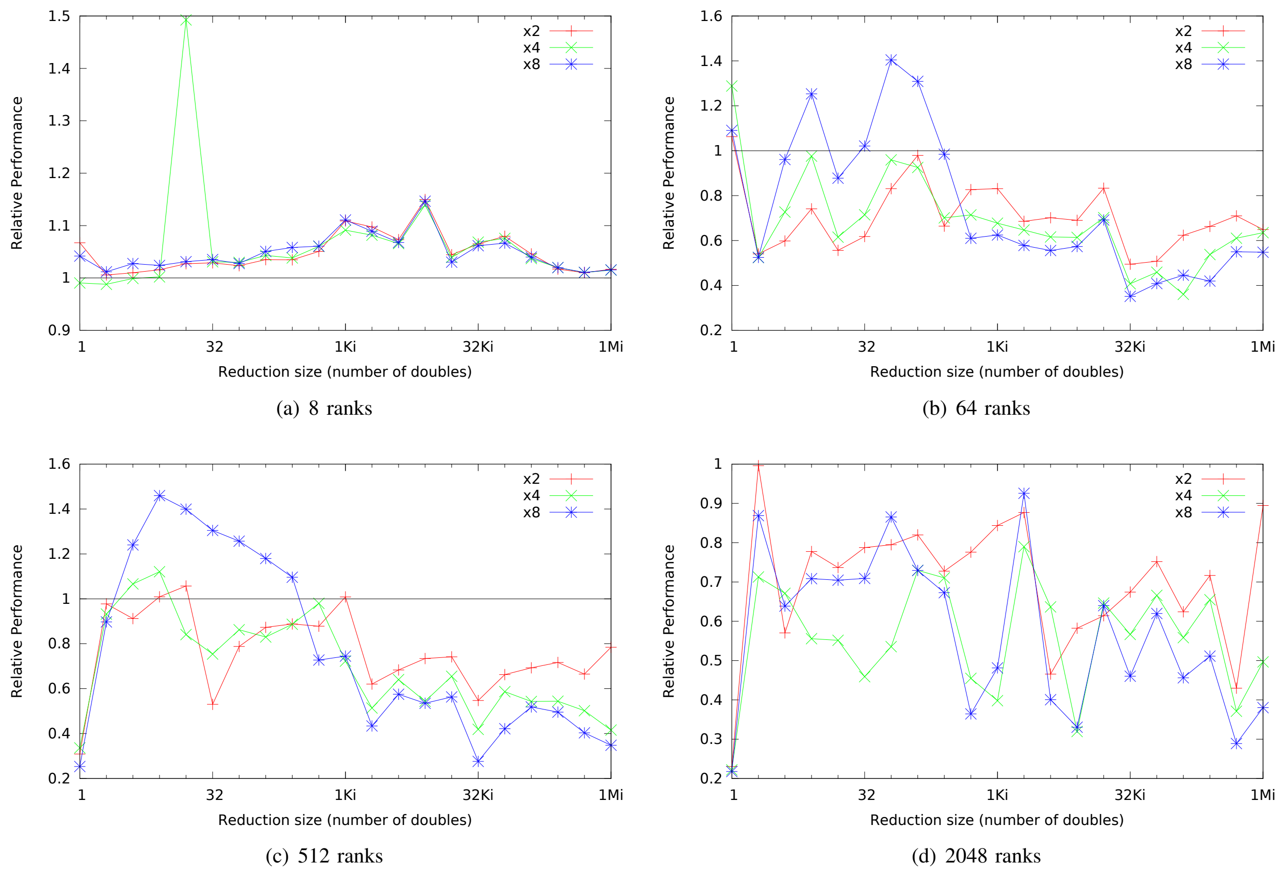
The experiments involved running the collective operation in a loop for 10,000 iterations. Each experiment was repeated 10 times for statistical confidence in the results.

1) Impact of Topology on MPI_REDUCE: Figure 2 shows the performance of **MPI_REDUCE** for different datatype counts on the x-axis. Each datatype is an **MPI_DOUBLE**, so the actual message size would be a factor of 8 higher (in bytes) than the unit on the x-axis. The legends used (x2, x4, and x8) indicate the relative difference in performance for different topologies. For example, "x2" refers to the factor of difference in performance when a topology of two cores per node is considered, compared with the topology-independent reduction performance. Note that neither the actual hardware topology nor the number of processes used changes in the experiment. Only the hardware topology exposed to the MPI reduction algorithm is artificially tweaked to appear as if there are multiple nodes with two cores each (for "x2") or multiple nodes with four cores each (for "x4") or eight cores each (for "x8"). The graphs indicate the performance relative to the topology-unaware algorithm. Thus, values less than 1 signify that the topology-aware algorithm performs better.

As shown in the figure, the difference in performance is minimal for 8 ranks (Figure 2(a)). This is expected because each physical node has 8 cores and all MPI ranks physically reside on the same node. Thus, the impact of topology in the reduction algorithm itself would not have much impact. As the system size increases, however, the impact of topology on the reduction performance continues to increase.

On our test system, for two nodes or more (i.e., when there is nonheterogeneous topology), starting at a base type size of 512 double-precision floating-point numbers (or 4 Kbytes), the topology-aware reduction outperforms the non-topology-aware reduction. For base types of 32,768 floating-point numbers, the topology-aware algorithm typically executes 2.5 to 5 times faster than the default algorithm. With some exceptions, the topology-aware algorithm typically increases its lead on the topology-unaware algorithms as the number of ranks increases. This situation is clearly visible for base type sizes of 512k elements and above. In this case, the runtime of the topology-aware algorithm decreases from approximately 65% of the time of the default algorithm at 16 ranks to approximately 35% at 1,024 ranks. In fact, for 2,048 cores (Figure 2(d)), topology-aware reductions can have as much as a fourfold improvement in performance.

[2]The increase in error is only described but not plotted as a graph because of the triviality of the graph. It is proportional to the unit tradeoff increasing logarithmically with the system size.



(a) 8 ranks

(b) 64 ranks

(c) 512 ranks

(d) 2048 ranks

Fig. 2: MPI Reduce: Impact of Topology on Performance

2) Impact of Topology on MPI_ALLREDUCE: Figure 3 shows the performance of **MPI_ALLREDUCE** for different datatype counts on the x-axis. Again, each datatype is an **MPI_DOUBLE**. For **MPI_ALLREDUCE**, the results look similar, albeit amplified because of the larger volume of communication. The data shows the effect of matching the assumed topology to the actual hardware topology, with a clear performance improvement when going from “x2” to “x8.” The performance improvement for **MPI_ALLREDUCE** is less dependent on the base type size and typically hovers around 50%.

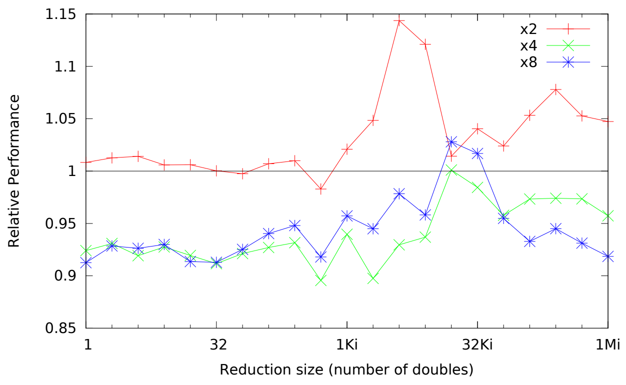
When considering scalability, unlike the topology-aware reduce algorithm, the allreduce algorithm is less sensitive to the base type size. Specifically, it shows that for a constant base type size, the performance typically stabilizes quickly at twice the throughput of the topology-unaware algorithm, independent of the number of ranks used. In all cases, matching the underlying hardware topology results in the best **MPI_ALLREDUCE** performance.

3) Impact of System Size: Figure 4 shows the impact of system size (on the x-axis) on the performance of **MPI_REDUCE** for different message sizes. We notice that for small message sizes (e.g., 4 Kbytes in Figure 4(a)), the impact of topology is minimal. This result is expected because topology mostly impacts data movement and for small message sizes this cost is small. Furthermore, as the system size increases, we notice that the improvement in performance generally increases, because the amount of data traffic crossing node boundaries is higher when topology is not considered.

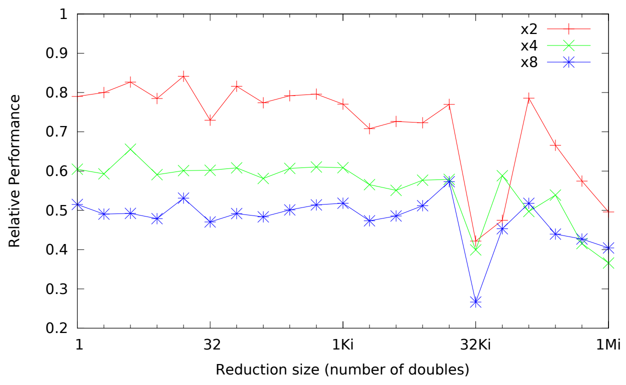
For large data (e.g., 8 Mbytes in Figure 4(d)), the performance impact is more profound, with topology-aware reductions outperforming non-topology-aware reduction by fourfold in some cases.

VII. Related Work

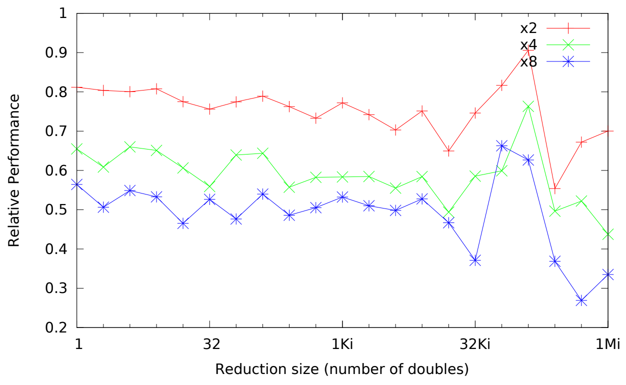
A large body of literature has investigated the performance of various collective communication operations in MPI. For example, in [14], the authors study different collective communication algorithms and their relative tradeoffs for performance. In [16], the authors investigate the performance improvements achievable by taking the hardware topology into account, specifically by splitting collective operations within the node and across nodes. The authors of [8] further improve such capabilities by utilizing shared-memory communication within the collective operations themselves. In [12], the authors utilize NUMA awareness to design new algorithms that can improve performance for such architectures. In [5] the authors describe an approach where the collective communication algorithms can be automatically tuned based on the system architecture.



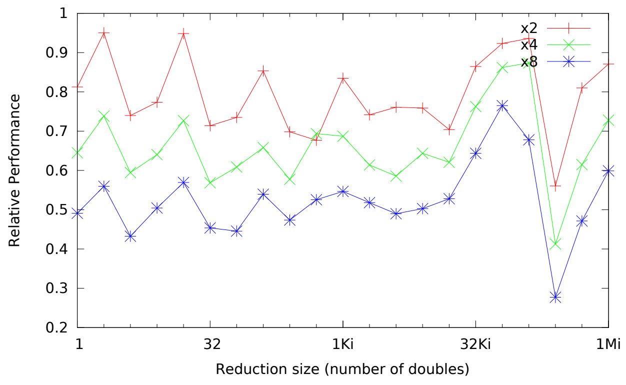
(a) 8 ranks



(b) 64 ranks



(c) 512 ranks



(d) 2048 ranks

Fig. 3: MPI Allreduce: Impact of Topology on Performance

While all these papers study the performance of collective operations in various environments, none of these measures the impact of collective operations as topology varies. For MPI reduction operations, the variation of topology can have a profound effect on the reproducibility of the results. Thus, understanding how these operations perform for different topologies is critical for applications to make the appropriate tradeoffs in reproducibility vs. performance.

VIII. Conclusions and Future Work

In this paper, we investigated the tradeoff between reproducibility and performance in collective MPI algorithms, specifically focusing on MPI reduction operations. As an example, we presented the performance of topology-aware **MPI_REDUCE** and **MPI_ALLREDUCE** operations. A discussion of existing sources of nondeterministic behavior (outside of MPI) created a proper framework for balancing the possible performance advantages against the increase in complexity triggered by change in reproducibility. We compared the topology-aware reduction algorithms with versions that do not take advantage of the topology information, using up to 2,048 cores with varying base type sizes. We found that, in many cases, by allowing the MPI library to optimize the reduction by considering the underlying hardware topology, we were able to achieve a significant performance improvement (up to fourfold).

We plan to extend our study to include further sources of nondeterministic behavior, such as jitter in collective operations. In addition, we will extend our research beyond MPI and investigate the performance-reproducibility tradeoffs in other programming models, such as OpenMP.

Acknowledgment

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

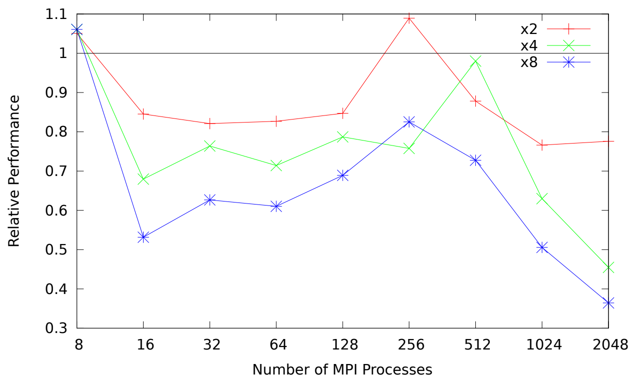
References

[1] Martyn J Corden and David Kreitzer. Consistency of floating-point results using the Intel compiler or why doesnt my application always give the same answer. Technical report, Technical report, Intel Corporation, Software Solutions Group, 2009.

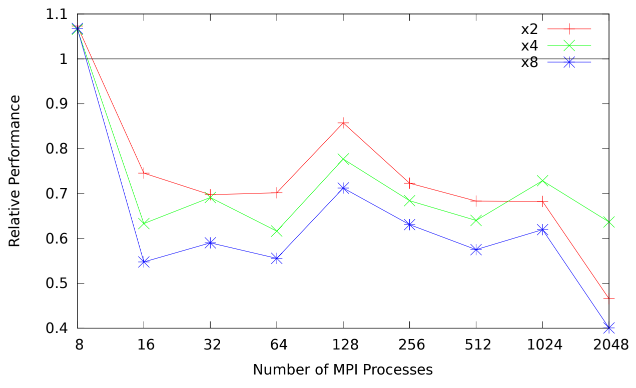
[2] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[3] Graham E Fagg, Jelena Pjesivac-Grbovic, George Bosilca, Thara Angskun, J Dongarra, and Emmanuel Jeannot. Flexible collective communication tuning architecture applied to Open MPI. In *Euro PVM/MPI*, 2006.

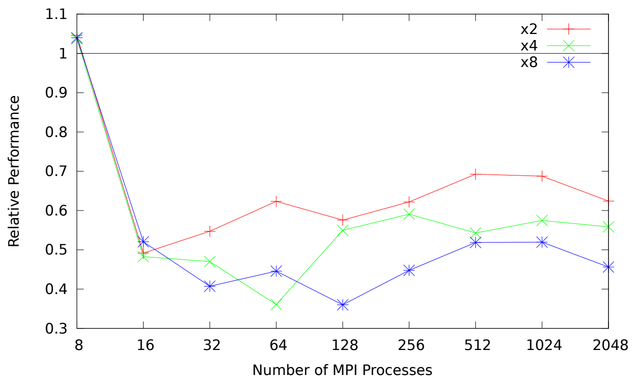
[4] Ahmad Faraj, Xin Yuan, and David Lowenthal. STAR-MPI: self tuned adaptive routines for MPI collective operations. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 199–208. ACM, 2006.



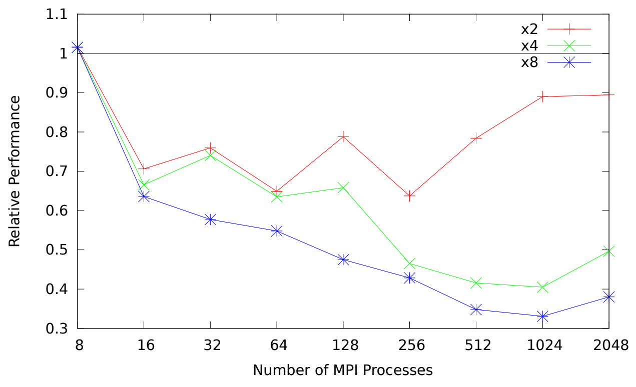
(a) 4 Kbytes



(b) 32 Kbytes



(c) 1 Mbyte



(d) 8 Mbytes

Fig. 4: MPI Reduce: Impact of System Size

[5] Ahmad Faraj, Xin Yuan, and David Lowenthal. STAR-MPI: Self Tuned Adaptive Routines for MPI collective operations. In *Proceedings of the 20th annual International Conference on Supercomputing*, ICS '06, pages 199–208, New York, NY, USA, 2006. ACM.

[6] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[7] Alan Gara, Matthias A Blumrich, Dong Chen, GL-T Chiu, Paul Coteus, Mark E Giampapa, Ruud A Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V Kopcsay, et al. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2.3):195–212, 2005.

[8] Richard L. Graham and Galen Shipman. MPI support for multi-core architectures: optimized shared memory collectives. In *Proceedings of the European PVM/MPI Users Group Meeting (Euro PVM/MPI)*, Sept. 2008.

[9] Nicholas J Higham. *Accuracy and Stability of Numerical Algorithms*. Number 48. SIAM, 1996.

[10] William Kahan. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754:94720–1776, 1996.

[11] Lawrence Livermore National Laboratory. Sequoia: IBM Blue Gene/Q supercomputer. http://en.wikipedia.org/wiki/IBM_Sequoia.

[12] S. Li, T. Hoefler, and M. Snir. NUMA-aware shared memory collective communication for MPI. In *Proceedings of the ACM International Conference on High Performance Distributed Computing (HPDC)*, June 2013.

[13] National University of Defense Technology. Tianhe-2 supercomputer. http://en.wikipedia.org/wiki/Tianhe-2.

[14] Rajeev Thakur and William Gropp. Improving the performance of collective pperations in MPICH. In *Proceedings of the 10th European PVM/MPI Users Group Meeting (Euro PVM/MPI)*, Sept. 2003.

[15] Oreste Villa, Daniel Chavarra-mir, Vidhya Gurumoorthi, Andrs Mrquez, and Sriram Krishnamoorthy. Effects of floating-point non-associativity on numerical computations on massively multithreaded systems.

[16] Hao Zhu, David J. Goodell, William Gropp, and Rajeev Thakur. Hierarchical collectives in MPICH2. In *Proceedings of the European PVM/MPI Users Group Meeting (Euro PVM/MPI)*, Sept. 2009.

Government License