A Statistical Analysis of Error in MPI Reduction Operations

Samuel D. Pollard
Computer and Information Science
University of Oregon
Eugene, OR
Email: spollard@cs.uoregon.edu

Boyana Norris
Computer and Information Science
University of Oregon
Eugene, OR

***Abstract*—This work explores the effects of nonassociativity of floating-point addition on Message Passing Interface (MPI) reduction operations. Previous work indicates floating-point summation error is comprised of two independent factors: error based on the summation algorithm and error based on the summands themselves. We find evidence to suggest, for MPI reductions, the error based on summands has a much greater effect than the error based on the summation algorithm. We begin by sampling from the state space of all possible summation orders for MPI reduction algorithms. Next, we show the effect of different random number distributions on summation error, taking a 1000-digit precision floating-point accumulator as ground truth. Our results show empirical error bounds that are much tighter than existing analytical bounds. Last, we simulate different allreduce algorithms on the high performance computing (HPC) proxy application Nekbone and find that the error is relatively stable across algorithms. Our approach provides HPC application developers with more realistic error bounds of MPI reduction operations. Quantifying the small—but nonzero—discrepancies between reduction algorithms can help developers ensure correctness and aid reproducibility across MPI implementations and cluster topologies.**

***Index Terms*—Floating-point arithmetic, message passing interface, parallel programming, reduction tree, roundoff error, summation order.**

I. Introduction

Message Passing Interface (MPI) [5] is the industry standard for distributed, high performance computing (HPC) applications. With MPI, collective operations assume an associative operator; however, floating-point arithmetic is in general nonassociative. This creates a predicament: on one hand, the nondeterminism of parallel operations is a cornerstone of the scalability of parallel computations. On the other hand, bitwise reproducibility is much more difficult to preserve if nondeterministic operation ordering is permitted with floating-point operations. To make matters worse, some small discrepancies are acceptable in many computational models, such as iterative solvers for linear systems [26]. Thus, the need for rigorous analysis of nonassociativity in HPC applications is twofold: both to ensure reproducibility where needed and to determine when these errors are spurious.

With the flexibility of the MPI standard, results are not guaranteed to be the same across different network topologies or even across different numbers of processes [28]. But we must not paint too pessimistic of a picture. In reality, the hard work of MPI developers has resulted in overall stable algorithms, and popular implementations such as OpenMPI and MPICH evaluate to the same result when the function is applied to the same arguments appearing in the same order [2]. However, this order is not necessarily *canonical* and so may differ from the serial semantics developers might expect.

A. Hypothesis

Our original hypothesis for this work was the following: selecting between different reduction schemes and reduction algorithms will cause a significant change in accuracy. Our first case studies indicate the opposite—reduction algorithms have little or no effect on the final result. Instead, the dynamic range of the summands is much more significant. We predict this is true because MPI reduction algorithms by nature strive to create a balanced reduction tree for efficiency. As formulated by Espelid [10], summation error (and in turn, MPI reduction error) consists of two independent parts: the initial error (based on the range of the inputs) and the algorithm error. Though there exist pathological orderings in any case to give huge algorithm errors, the optimal way to minimize summation error is to minimize the magnitudes of intermediate sums [10]. For many inputs, this goal is supplementary to creating balanced reduction trees.

B. Contributions

Our work analyzes error of MPI reduction algorithms several different ways. In this work, we:

- generate samples from every possible reduction strategy permitted by the MPI standard (Section II);
- calculate previously-established analytical bounds and probabilistic estimators as they apply to some sample input probability distributions (Section III);
- show experimental results for different probability distributions and summation algorithms as they relate to the MPI standard (Section IV);
- investigate the correlation between reduction tree height and summation error (Section IV-C); and
- consider the effect allreduce algorithms have on the accuracy of the proxy application Nekbone (Section V).

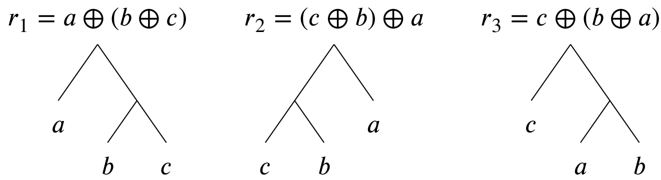

Fig. 1. With the commutative but nonassociative operator $\oplus$, $r_1 = r_2$ but $r_2 \neq r_3$.

II. State Space of MPI Reduction Operations

The goal of our work is to explore the implications of assuming associativity of floating-point operations with respect to MPI reduction operations. We first explain the concepts behind MPI reduction algorithms, then describe the state space of different ways these reductions can be performed.

A. Reduction Operations

Conceptually, an MPI reduction operation works by repeatedly applying a binary operation to a given input, thereby reducing its size and returning an accumulated value. More concretely, one use of `MPI_Reduce` is to compute the sum of all elements of an array in parallel, and transmit the sum to a single process.

A reduction operation is typically parameterized by its associativity. For a one-dimensional array, the natural choices are left or right-associative. Imperative, serial implementations typically imply left-associativity since that matches C and Fortran specifications for + and * and is strongly idiomatic. Recall that floating-point addition and multiplication are commutative but nonassociative operations. With MPI there is no fixed associativity, so many results are permitted for an operation such as `MPI_Reduce`.

This paper focuses on floating-point addition. By assuming associativity and commutativity, MPI's definition of a reduction operation permits many different ways to reduce an array of values. For example, Fig. 1 gives three different ways to reduce a three-element array. Each reduction strategy forms a *reduction tree*.

Previous work used randomly shuffled arrays to check floating-point summation error empirically [4]. We expand this method by also randomizing the reduction tree. While selecting summation order and associativity is not a new idea [14], we provide more results as they specifically relate to MPI. The state space we describe is all possible reduction trees and could also be applied to other parallel programming paradigms. Next, we analyze the possible state space of how collective operations can be performed.

B. Quantifying the State Space of Possible Reductions

The MPI 3.1 standard [22] has the following description:

> The operation op is always assumed to be associative. All predefined operations are also assumed to be commutative…However, the implementation can take advantage of associativity, or associativity and commutativity, in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating-point addition.

For custom operations, the issue is similar:

> If `commute = false`, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, taking advantage of the associativity of the operation. If `commute = true` then the order of evaluation can be changed, taking advantage of commutativity and associativity.

A *rank* in MPI terminology refers to one MPI process. Characterizing these two descriptions gives us two different families of possible results for a reduction operation. We compare these two families with the most straightforward serial implementation as well as previous work.

Throughout, we assume a one-dimensional array A of floating-point values of length n. We use A_k to denote array access with $k \in \{1, 2, \ldots, n\}$. Henceforth, we refer to the combination of permuting A along with a particular reduction tree as a *reduction strategy* of A.

We point out a slight clarification of our use of *random* throughout this paper. In reality, in reduction algorithms the order or associativity is not *random* but *unspecified*. We say *random* simply for notational consistency; in Section IV we generate random reduction strageties according to these families.

Now we present four families of reduction algorithms: two described by the MPI standard and two others.

1) Canonical. This is left-associative and defines a single possible reduction order.
2) FORA (**F**ixed **O**rder, **R**andom **A**ssociation). This represents the case when `commute = false`. The operations may be parenthesized, but the order is unchanged. For example, in Fig. 1, r_1 is an acceptable sum but r_2 and r_3 are not. This is a well-known combinatorial problem and there are C_n possible ways to parenthesize n values, where
$$C_n = \frac{(2n)!}{(n+1)!n!}$$
is the nth Catalan number [19, §7.2]. Each parenthesization corresponds to a binary tree, but for convenience we call these *associations*.
3) RORA (**R**andom **O**rder, **R**andom **A**ssociation). This is the default behavior when using `MPI_Reduce`. Operations may be reordered assuming both commutativity and associativity. Counting the possible number of different equivalent reduction trees is a less well-known, but still solved, combinatorial problem [6]. At first glance, this would appear to have C_n different reductions for each of the $n!$ possible permutations of A. However, because of commutativity, many of these reduce to the same answer.

Let g_n be the number of possible different reduction trees. As it turns out,

$$g_n = (2n-3)!! \quad (1)$$

where !! is the double factorial (in this case on odd integers). That is, $(2n-3)!! = 1 \times 3 \times 5 \times \cdots \times 2n-3$.

4) ROLA (**R**andom **O**rder, **L**eft **A**ssociative). This is not precisely described as part of the MPI standard but is used in previous work [4] and could still arise for algorithms where the array's order is nondeterministic, but reduction is still done in canonical order. Here, the array may be shuffled but the reduction is left-associative. There are $n!/2$ different possible orderings. To see this, note there are $n!$ permutations, and by commutativity only the first two elements can be rearranged; the rest are fixed by left-associativity.

For the last three cases the growth rate of the number of possible reduction orders is exponential; exhaustively checking results is infeasible which is why we generate reduction strategies randomly. Comparing the last three cases, we have

$$C_n < \frac{n!}{2} < g_n \quad (2)$$

for $n > 4$.

We discuss further these four cases and how they relate to the MPI standard. By the pigeonhole principle[1], ROLA cannot generate every possible value permitted in RORA. For example, $(bd)(ac)$ can be constructed via RORA but not via ROLA because we cannot convert every left-associative reduction tree into a balanced binary tree without associativity.

To reiterate, case 1) is canonical, case 2) (FORA) is precisely the possible state space for custom MPI operations when `commute = false`, and case 3) (RORA) is the state space for the default MPI case. For comparison to previous work, we include the case 4) (ROLA) in the subsequent analysis.

C. Generating Samples

We begin by generating random binary trees—because every possible association of a sequence of nonassociative binary operations can be described by a binary tree. Thus, when performing a reduction on n elements, we wish to sample from the space of every binary tree with n leaves. To do this we use Algorithm R (Rémy's procedure) [19].

For the FORA case, we generate a random binary tree. For the RORA case, we first shuffle the array, then generate a random binary tree by which to sum the array. For the ROLA case, we shuffle the array, then accumulate in a left-associative manner. Some examples of random reduction trees are given in Fig. 1.

[1]There are g_n distinct values for ROLA and $n!/2$ for RORA. Since $n!/2 < g_n$, there must be at least one reduction strategy that occurs in ROLA but not in RORA.

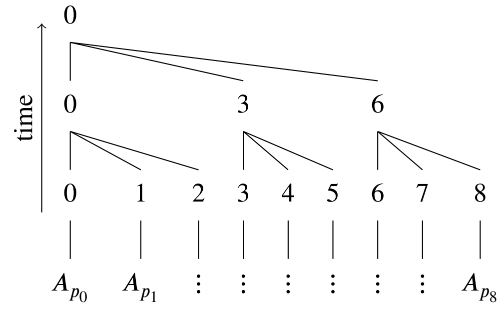

Fig. 2. One possible reduction algorithm. The array A is split up across 9 MPI Ranks. Each p_i indicates contiguous chunks of A. The final result is obtained by sending partial sums "up" the tree.

D. The Nondeterminism of Reduction Algorithms

We describe the entire state space of possible reduction strategies, but realistically, not every reduction strategy will occur for a given topology and reduction algorithm. For more sophisticated reduction algorithms such as Rabenseifner's algorithm [27], it is not clear how to limit the state space.

In addition, it is not immediately obvious where nondeterminism, given a particular algorithm, can arise. We show an example in Fig. 2 for visualization. In this case, nondeterminism can arise either from messages arriving in different order for the intermediate sums, or the array being distributed differently according to topology. Depending on the internal algorithm of the reduction, one may then be in any of the four cases. Another place for nondeterminism is in how the array is distributed across nodes. For topology-aware reduction algorithms, it may be the case that nodes are not in canonical order in the reduction tree, which is permitted by the MPI standard.

Finally, notice that *within* a rank, the values will probably be reduced in the most straightforward way—left-associative. We do not assume this, but can easily model it by using an array whose size is equal to the number of nodes and whose elements are the partial sums within a rank.

E. Generating Random Floating-Point Values

In most cases, we use the Marsaglia-Multicarry [21] pseudorandom number generator to generate uniform distributions with a fixed seed across experiments. We denote a uniform distribution over the half-open interval $[a, b)$ as $\mathrm{U}(a, b)$.

Uniform random number generators for floating-point values typically generate an integer in some range $[0, K)$, then divide by K to get a value in $[0, 1)$. However, if $K < 2^{1022}$ then it is impossible to generate every floating-point value in $[0, 1)$. Notably, no subnormal numbers (for double precision, numbers in $[0, 2^{-1022})$) are generated. We alleviate this by generating a special distribution which we call *subn*. The process for generating these is simple: generate 62 random bits, then clear the highest two. This ensures the exponent is in the range $[-1023, 0]$ and so every floating-point value in $[0, 2)$ can be generated. Sampling from this distribution gives

an expected value of 0.00282 and looks roughly exponentially distributed, with rate parameter $\lambda \approx 1/0.00282 \approx 354$.

III. Analytical Error Bounds and Estimators

Analytical error of floating-point summation in the general case can be pessimistic. We wish to see how pessimistic. First, we introduce some helpful notation. Let ϵ be machine epsilon, or the upper bound on relative rounding error from a floating-point operation. That is, for some real operator op and its corresponding floating-point operator $\odot$,

$$x \text{ op } y = (x \odot y)(1+\delta) \text{ and } |\delta| \le \epsilon. \tag{3}$$

For our experiments, we use IEEE-754 *binary64* format so $\epsilon = 2^{-53}$. We focus on floating point addition, notated as $\oplus$ to distinguish from real number addition. We put $\oplus$ above the summation symbol to denote floating-point sum with an unspecified reduction strategy like so: $\sum^{\oplus}$.

One last bit of notation: let S_A be the true value of the sum of every element of A. A well established result by Wilkinson [31] bounds the absolute error of floating-point summation as:

$$\left| \sum_{k=1}^{\oplus\, n} A_k - S_A \right| \le \epsilon(n-1) \sum_{k=1}^{n} |A_k| + O(\epsilon^2). \tag{4}$$

This bound can be refined with some assumptions. Existing statistical analysis by Robertazzi and Schwartz [23] derives expected estimators of the relative error of floating point summation in special cases. If you assume

1) The values are positive and either uniformly or exponentially distributed (for all k, $A_k \sim \text{U}(0, 2\mu)$ or $\exp(1/\mu)$);
2) floating-point addition errors are independent, distributed with mean 0 and variance σ_e^2; and
3) the summation ordering is random;

then the relative error is approximated by

$$\frac{1}{3}\mu^2 n^3 \sigma_e^2. \tag{5}$$

Assuming (3) holds and summation error is uniformly distributed in $[-\epsilon/2, \epsilon/2]$ yields

$$\sigma_e^2 = \frac{1}{12}\epsilon^2. \tag{6}$$

These equations are not particularly useful in isolation, but we use this estimator to verify our empirical results in the following section.

IV. Empirical Results

We compare the reduction strategies in Section II for reduction along with some baselines. We list how each is generated here:

i) Canonical (left-associative). The code is essentially `s=0.0; for(k=0;k<n;k++) s+=A[k];` for IEEE-754 double precision.
ii) FORA (**F**ixed **O**rder, **R**andom **A**ssociation). Generates case 2) reduction strategies. This is done by generating a random reduction tree but leaving the array in its original order.
iii) RORA (**R**andom **O**rder, **R**andom **A**ssociation). Generates case 3). This is done by shuffling the array, then generating a random reduction tree and reducing over that.
iv) ROLA (**R**andom **O**rder, **L**eft **A**ssociative). This is done by shuffling the array, then reducing left-associative. This is for comparison with previous work [4].
v) MPFR. We use the Multiple Precision Floating-Point Reliable (MPFR) library by taking the randomly-generated double-precision numbers as exact then convert to 1,000-digit (3,324-bit) MPFR floats [13] and reduce using MPFR's correctly-rounded operations in left-associative order. We use this as the ground truth.

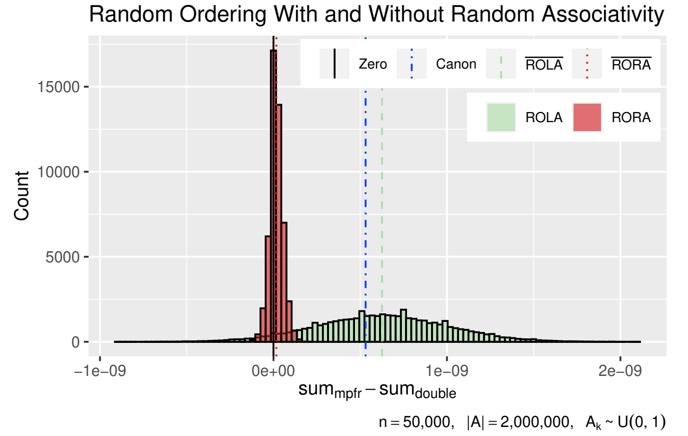

Fig. 3. On average, left-associative summation results in a larger error than random associativity. Left-associative, in MPI terms, would be a canonical ordering. A bar indicates the arithmetic mean of all samples.

Using 3,324 bits for MPFR may seem excessive, but a proposal by Kulisch describes a 4,288-bit accumulator to compute exact dot products for double-precision floats [20]. Multiplication requires more bits to account for the product of very large and very near-zero values, so for summations 3,324 digits are sufficient.

A. Distribution of Summation Errors

In Figures 3, 4, and 5 we present a set of histograms showing the error of different MPI reduction schemes. All of these are histograms with a sample size of 50,000, but the x-axes are different measurements of error.

In Fig. 3, we compare not the absolute or relative error, but simply the *difference* between the true value (sum_{mpfr}) and the computed value ($\text{sum}_{\text{double}}$). We see here that random ordering *and* random association, on average, provides a far tighter error bound compared to enforcing left-associativity. Not only that, but enforcing left-associativity on a shuffled array (ROLA) underestimates the true sum. Higham and Mary show this is expected: as the partial sums get sufficiently large, adding a small positive number will not change the sum; notationally, for sufficiently large S and small x, $S \oplus x = S$ [16]. Other distributions have similar behavior, where ROLA gives a more

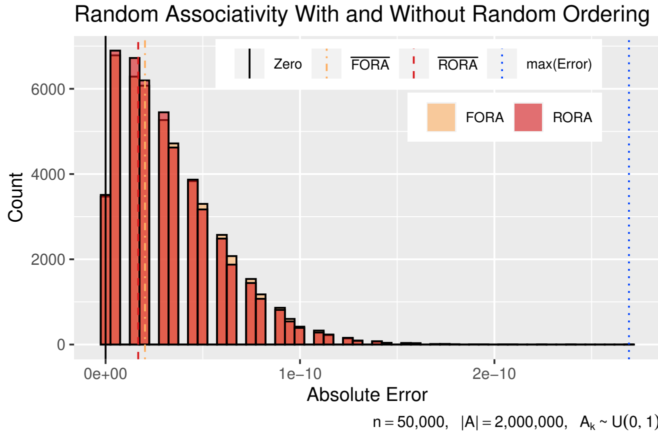

Fig. 4. Here we see shuffling the array has little effect on the summation error. The number of bins was chosen to be exactly the number of unique different values. The apparent missing values is *not* an artifact of plotting but rather floating-point values which occurred infrequently (or just once).

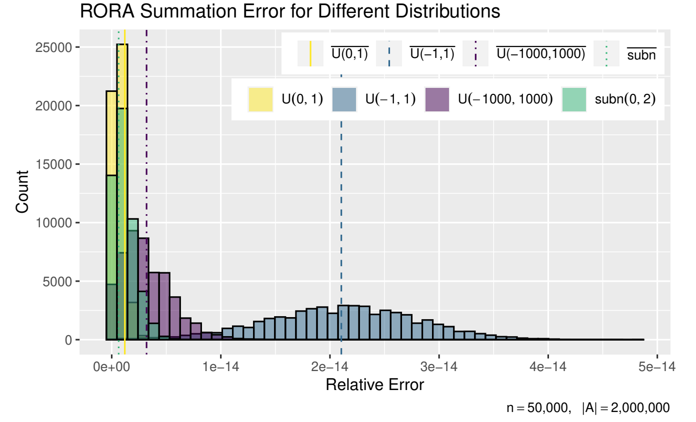

Fig. 5. We notice a wide variation of errors between different distributions with $U(-1,1)$ having the largest average relative error. We measure relative error as $|\text{sum}_{\text{double}} - \text{sum}_{\text{mpfr}}|/\text{sum}_{\text{mpfr}}$.

biased sum compared to RORA. This provides evidence that MPI reduction error may not be as severe as indicated in the work of Chapp et al [4]. For tabulation of the vertical lines in Fig. 3, see Table I.

Looking at Fig. 3, we can also compare our refinement of different floating-point summation. The similarity of ROLA (case iv, used by Chapp et al.) and RORA (case iii, everything permitted by the MPI standard). The smaller error of RORA indicates MPI summation can be quite robust if the inputs are randomly distributed similar to our approach.

We compare the distributions for FORA and RORA separately in Fig. 4 because they look similar. This indicates that the order in which elements are added has a smaller effect compared to the associativity. This makes sense in the context of our random number generation: previous work indicates summation error is dependent on the magnitudes of the intermediate sums [31]. Randomly shuffling the array would not affect the magnitude of intermediate sums much since the summands are independently and identically distributed (iid). Further research is needed to determine whether this holds for input which is not iid.

We point out some interesting things about Fig. 5. For one, the greatest relative error among everything we sampled is with the $U(-1,1)$ distribution. One possible explanation is that numbers close together have a higher chance of having large cancellation; this is the classic example of floating-point nonassociativity. That is, if $x \approx y$ then $x \ominus y$ is close to zero. Then, if for another intermediate summand $z \gg (x \ominus y)$ more error is exposed when computing $z \oplus (x \ominus y)$.

We tabulate some important values (displayed as vertical lines in Figs. 4, 5, and 3) alongside their analytical relative error bounds in Table I.

B. Comparing Empirical and Analytical Bounds

We calculate the "Robertazzi" row in Table I by substituting our experimental values into (5) and (6). Hence, for $U(0,1)$ and $n = 2 \times 10^6$,

$$\frac{1}{3}(0.5)^2(2\times 10^6)^3\left(\frac{1}{12}\epsilon^2\right) \approx 6.848\times 10^{-16}.$$

This estimator is very close to our observed results (See $U(0,1)$ $\overline{\text{RORA}}$ in Table I), but Robertazzi's work only applies for summation of either uniform or exponentially distributed positive numbers.

TABLE I
TABLE OF SIGNIFICANT VALUES FOR OUR EMPIRICAL RESULTS. SMALLER IS BETTER.

Distribution	Measurement	Relative Error	Note
$U(-1,1)$	$\overline{\text{RORA}}$	2.104×10^{-14}	
$U(-1,1)$	max(RORA)	4.824×10^{-14}	
$U(-1,1)$	$\overline{\text{ROLA}}$	8.358×10^{-12}	max empirical
$U(-1,1)$	Canonical	6.124×10^{-12}	
$U(-1,1)$	Analytical	7.951×10^{-7}	
$U(0,1)$	$\overline{\text{RORA}}$	6.702×10^{-16}	
$U(0,1)$	max(RORA)	4.073×10^{-15}	
$U(0,1)$	$\overline{\text{ROLA}}$	1.282×10^{-14}	
$U(0,1)$	Canonical	1.062×10^{-14}	
$U(0,1)$	Analytical	1.776×10^{-8}	
$U(0,1)$	Robertazzi	6.848×10^{-16}	
$U(-1000,1000)$	$\overline{\text{RORA}}$	3.194×10^{-15}	
$U(-1000,1000)$	max(RORA)	2.084×10^{-14}	
$U(-1000,1000)$	$\overline{\text{ROLA}}$	3.711×10^{-14}	
$U(-1000,1000)$	Canonical	7.503×10^{-15}	
$U(-1000,1000)$	Analytical	7.951×10^{-7}	max analytical
subn	$\overline{\text{RORA}}$	6.383×10^{-16}	min empirical
subn	max(RORA)	4.385×10^{-15}	
subn	Canonical	6.945×10^{-14}	
subn	Analytical	1.713×10^{-7}	min analytical
subn	Robertazzi	4.206×10^{-24}	not achievable
	machine ϵ	1.110×10^{-16}	double-precision

For subn, we get an impossibly small expected relative error since it is much smaller than machine ϵ. This is to be expected, however, since subnormal numbers lack the same precision as

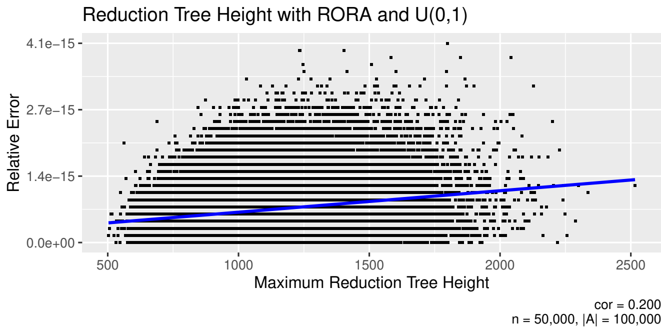

Fig. 6. The bands are a result of the gap between floating-point values.

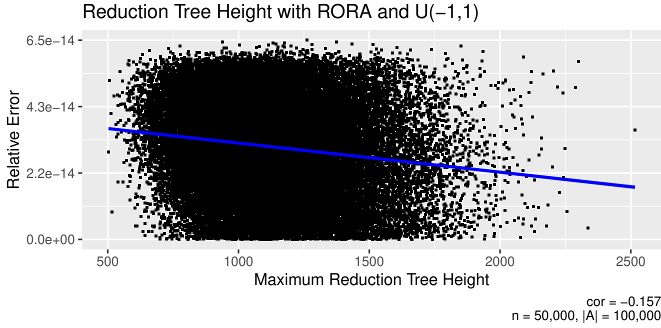

Fig. 7. The $U(-1, 1)$ distribution has a larger error than $U(0, 1)$.

normal numbers, and serves as more evidence that floating-point summation can go awry for numbers very close to zero (in the case of double precision, values smaller than 2^{-1022}).

For the array size of our experiments (2 million elements), we see the analytical bounds being particularly pessimistic: they are between 6 and 7 orders of magnitude *larger* than our observed results.

C. Reduction Tree Height and Error

When trying to answer the question of how to make our reduction strategy (recall this is the combination of array ordering and reduction tree shape), we ask: does reduction tree height affect summation error? We looked at the maximum height of each reduction tree for our experiments. Fig. 6 shows a very weak correlation, with correlation coefficient $\rho = 0.2$. We see a *negative* correlation with the $U(-1, 1)$ distribution in Fig. 7. However, we still plot these for completeness' sake.

Now that we have seen some empirical examples from manufactured arrays, we compare our results with a more realistic use-case.

V. Nekbone and SimGrid: A (Robust) Case Study

SimGrid is an ambitious project to simulate MPI applications [3]. We use two of the many features of SimGrid in this work: its simulation of network topology and its implementation of a large number of MPI reduction algorithms across different MPI implementations including MPICH, OpenMPI, Intel MPI, and MVAPICH [29]. Table II describes each of these algorithms. We do not use every allreduce algorithm that

TABLE II
Descriptions of allreduce algorithms used to generate Fig. 8. "smp" stands for symmetric multiprocessing—inter-node communication is separate from intra-node [29].

Mnemonic	Description
default	naïve reduce then broadcast
impi	use intel mpi selector
mpich	use MPICH selector
mvapich2	use MVAPICH2 selector
mvapich2_rs	rdb for small messages, else reduce-scatter
mvapich2_two_level	smp intra-node, mpich inter-node
ompi	use OpenMPI selector
ompi_ring_segmented	OpenMPI ring algorithm
rab	Rabenseifner
rab2	variation of rab (alltoall then allgather)
rab_rsag	variation of rab (reduce-scatter-allgather)
rdb	recursive doubling
redbcast	reduce then broadcast
smp_binomial	binomial tree
smp_binomial_pipeline	binomial tree with 4096 byte pipeline
smp_rdb	recursive doubling
smp_rsag	reduce-scatter-allgather

SimGrid implements because some do not work for processor counts which are not a power of two and our simulated topology is a 72-node fat tree cluster.

Nekbone is a proxy application for the Nek5000 physics simulation code [11]. Nek5000 is a powerful HPC application for solving various computational fluid dynamics problems [12]. Nekbone is a simplified version of Nek5000 containing only some key computational elements by solving a three-dimensional Poisson equation using the conjugate gradient iterative solver. We run Nekbone's default settings: 101 iterations of conjugate gradient using polynomials of order 10. We simulate hardware with one MPI rank per node. Our results are for 9,216 elements and we present the residuals after the 101st iteration.

We plot the residuals using different allreduce algorithms in Fig. 8. To better see the small changes between allreduce algorithms, we define zero as the residual of SimGrid's default allreduce algorithm. One algorithm in particular resulted in a residual an order of magnitude smaller every time: smp_rsag. If this algorithm were plotted along with the rest, then the distinction between all other algorithms would vanish. We visualize this difference by tabulating smp_rsag's residual against the default and least accurate residuals in Table III.

The outlier smp_rsag has smaller error across the various different simulated network topologies we used: 16 and 72-node 2D tori and 16 and 72-node fat trees, as well as different problem sizes. Further, executing Nekbone on a single node with 16 MPI ranks natively using OpenMPI yielded the same result for each of OpenMPIs seven allreduce algorithms. Thus, we predict the cause is the interaction of Nekbone and SimGrid but further investigation is needed.

Even with this outlier, there are just four different results for all SimGrid algorithms. Comparing this to the combinatorial number of acceptable MPI summation orders (1) shows there is need for more nondeterminism when running simulations

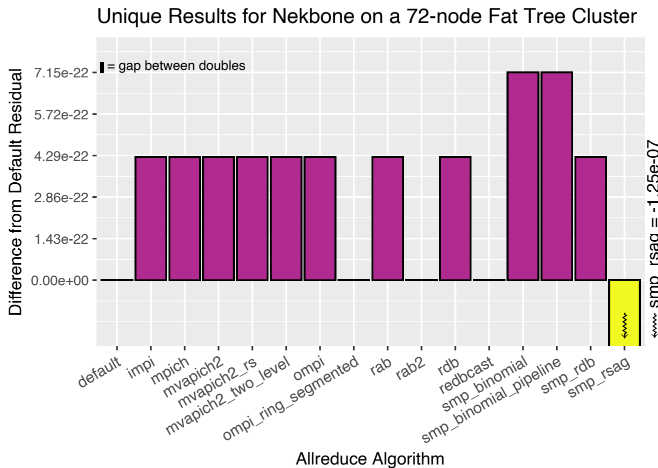

Fig. 8. Each bar represents the same residual for the conjugate gradient step of Nekbone across 50 runs. We annotate the distance between two consecutive double-precision floats at the scale of the residual. We cut off the y-axis for otherwise observable differences between other algorithms and smp_rsag would vanish. This shows four unique results among 16 allreduce algorithms.

TABLE III
THE SMP_RSAG ALGORITHM IS AN OUTLIER. SEVERAL ALGORITHMS MATCH THE DEFAULT AND WORST, AS INDICTED IN FIG. 8.

Allreduce Algo. Rank	Residual
Best (smp_rsag)	$1.616\,306\,278\,792\,575 \times 10^{-8}$
Default	$14.082\,603\,491\,982\,575 \times 10^{-8}$
Worst	$14.082\,603\,491\,982\,647 \times 10^{-8}$

to get a more complete idea of the space of potential results of HPC codes.

VI. Related Work

Concern over the accuracy of floating-point summation is as old as floating-point arithmetic itself [31]. Further developments with analytical and statistical bounds for various numerical algorithms is also a well-established area [15] and is broadly contained under the field of numerical analysis. Tools such as FPTaylor [25] and Daisy [7] use sophisticated methods such as symbolic Taylor expansions or SMT solvers, to calculate tight bounds on floating point expressions. However, rounding errors as they apply to MPI reduction operations is less widely studied.

Our main influence is work by Chapp et al. on the error of MPI collective operations [4]. This work provides empirical evidence of floating-point error effects depending on the amount of concurrency as well as reduction-tree shape. Their work advocated a need for dynamic selection between summation algorithms. We refine Chapp's model of MPI reduction error by considering random associations. We find for simulated input—both of our work looks at $U[-1000, 1000]$ floats—random associations results in a lower expected summation error.

Other work regarding MPI [30] looks into MPI collective errors as they apply specifically to the convergence of the conjugate gradient method applied to a power grid analysis.

While our work focuses on analyzing error from unmodified HPC applications, other researchers take this further and strive for bit-level reproducibility. One prominent project in this domain is an algorithm for parallel reproducible summation, [8] and more generally the Reproducible BLAS project [9]. We wish to analyze MPI error compared to ReproBLAS in the future. A different project seeking bit-level reproducibility has been undertaken by Arteaga et al. [1]

Common approaches to reduce summation error include Kahan's compensated summation [18] and prerounding [24]. Both of these can decrease summation error but have performance impacts. Neither is provided by any commonly-used MPI implementation.

SimGrid has also been used to check reproducibility by Hoffeins et al. [17] However this focuses on dynamic loop scheduling whereas we look at only the MPI reduction algorithm itself.

VII. Conclusion and Future Work

Our work begins a more detailed analysis of summation error with MPI reduction algorithms. We looked the errors for four different families of reduction strategies for a simple `MPI_Reduce` as well as the space of possible solutions for one HPC proxy application using `MPI_Allreduce`, Nekbone. We showed that, at least for some common distributions, the reduction tree shape has a greater effect on the summation error than the order of the array. We also saw that left-associativity typically results in larger error compared to generating a random ordering and reduction tree. This occurs because the distributions we used result in summands of roughly the same magnitude and left-associativity causes partial sums to be much larger than the summands. We demonstrated that analytical error bounds can vastly overestimate summation error. The probability of each partial sum resulting in the worst-case error is not realistic for most problem domains. So, our statistical analysis produces more practically useful expected error.

The problem of analyzing complex HPC applications for error remains unsolved. As mentioned in Section I-A, our experiments with independent, identically distributed random numbers result in relatively predictable behavior. However, distributions of floating-point numbers in HPC codes are far from random. For example, a singular matrix effectively cannot be generated by randomly generating its elements. A promising future direction would be to generate both more realistic and pathological random floating-point numbers.

The problem of generating realistic reduction strategies is another interesting direction of research. The most precise description of MPI reduction state space would depend on both network topology and reduction algorithm. However, some refinements to our model could be made: for example, it is unlikely any MPI reduction algorithm would have a

large tree height; parallel reduction algorithms strive to complete in $O(\log(n))$ parallel steps. Rémy's procedure generates trees that have an asymptotically different height: on average $4\sqrt{n/\pi}$ [19]. We investigated this somewhat and did not find a correlation between the height of the reduction tree and summation error (the correlation coefficient was between 0.2 and -0.16 for the distributions we sampled). However, it remains to be seen whether generating realistic trees has a significant effect on error.

Beyond this, MPI programs typically have highly-structured reduction schemes. It would be interesting to investigate multiple parallelism schemes and their effect on results. For example, a vector with 10^7 elements may only be reduced across a few dozen nodes, and inter-node summation may be nondeterministic or deterministic depending on the shared memory parallelism scheme. This limits the state space compared to our random generation via Rémy's procedure.

One challenge we note in this work on reproducibility is how hard it can be to get different answers! Small problem sizes, process counts, and simple examples usually result in identical answers. On one hand, this is evidence of the great work by the MPI and SimGrid developers: most MPI implementations are deterministic when run on the same machine with the same input. However, since the standard does not guarantee reproducibility, we wish to explore the entire state space of acceptable answers for a given application by varying the topology and input space to expose potential discrepancies when running at a larger scale or on a different architecture. This could be done by modifying MPI reduction algorithms to inject intentional nondeterminism. Another direction could focus on amplifying errors by using highly numerically unstable input. Both of these strategies are not generalizable to all applications but may give insight into the exact conditions upon which an HPC program can fail.

ACKNOWLEDGEMENT

The authors would like to thank Augustin Degomme and the SimGrid Team for their help with SimGrid, Dylan Chapp for explaining his research, and Jackson Mayo for his insightful comments.

REFERENCES

[1] A. Arteaga, O. Fuhrer, and T. Hoefler, "Designing bit-reproducible portable high-performance applications," in *28th International Parallel and Distributed Processing Symposium*, ser. IPDPS. IEEE, 2014, pp. 1235–1244. [Online]. Available: https://htor.inf.ethz.ch/publications/img/arteaga-fuhrer-hoefler-reproducible-apps-ipdps14.pdf

[2] P. Balaji and D. Kimpe, "On the reproducibility of MPI reduction operations," in *10th International Conference on High Performance Computing and Communications & International Conference on Embedded and Ubiquitous Computing*, ser. HPCC/EUC. Zhangjiajie, China: IEEE, 2013, pp. 407–414.

[3] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014. [Online]. Available: http://hal.inria.fr/hal-01017319

[4] D. Chapp, T. Johnston, and M. Taufer, "On the need for reproducible numerical accuracy through intelligent runtime selection of reduction algorithms at the extreme scale," in *IEEE International Conference on Cluster Computing*. Chicago, IL, USA: IEEE, Sep. 2015, pp. 166–175.

[5] L. Clarke, I. Glendinning, and R. Hempel, "The MPI message passing interface standard," in *Programming Environments for Massively Parallel Distributed Systems*, K. M. Decker and R. M. Rehmann, Eds. Basel: Birkhäuser Basel, 1994, pp. 213–218.

[6] M. Dale and J. Moon, "The permuted analogues of three catalan sets," *Journal of statistical planning and inference*, vol. 34, no. 1, pp. 75–87, Jan. 1993.

[7] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, "Daisy - framework for analysis and optimization of numerical programs (tool paper)," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Thessaloniki, Greece: Springer International Publishing, 2018, pp. 270–287.

[8] J. Demmel and H. D. Nguyen, "Parallel reproducible summation," *IEEE Transactions on Computers*, vol. 64, no. 7, pp. 2060–2070, 2015.

[9] J. Demmel, P. Ahrens, and H. D. Nguyen, "Efficient reproducible floating point summation and BLAS," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-121, Jun. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html

[10] T. O. Espelid, "On floating-point summation," *SIAM Review*, vol. 37, no. 4, pp. 603–607, 1995. [Online]. Available: https://doi.org/10.1137/1037130

[11] P. Fischer and K. Heisey, "NEKBONE: Thermal hydraulics mini-application," Tech. Rep., 2013, available at https://github.com/Nek5000/Nekbone.

[12] P. Fischer and S. Kerkemeier, "NEK: a fast and scalable high-order solver for computational fluid dynamics," 2020, available at https://nek5000.mcs.anl.gov/.

[13] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "Mpfr: A multiple-precision binary floating-point library with correct rounding," *Transactions on Mathematical Software*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[14] N. J. Higham, "The accuracy of floating point summation," *SIAM Journal of Scientific Computing*, vol. 14, no. 4, pp. 783–799, 1993.

[15] ——, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[16] N. J. Higham and T. Mary, "A new approach to probabilistic rounding error analysis," *SIAM Journal on Scientific Computing*, vol. 41, no. 5, pp. A2815–A2835, 2019. [Online]. Available: https://doi.org/10.1137/18M1226312

[17] F. Hoffeins, F. M. Ciorba, and I. Banicescu, "Examining the reproducibility of using dynamic loop scheduling techniques in scientific applications," in *IEEE International Parallel and Distributed Processing Symposium Workshops*, ser. IPDPSW. Lake Buena Vista, FL, USA: IEEE, 2017, pp. 1579–1587.

[18] W. Kahan, "Pracniques: Further remarks on reducing truncation errors," *Communications of the ACM*, vol. 8, no. 1, p. 40, Jan. 1965. [Online]. Available: https://doi.org/10.1145/363707.363723

[19] D. E. Knuth, *The Art of Computer Programming: Generating All Trees; History of Combinatorial Generation*. Boston, MA, USA: Addison-Wesley, 2006, vol. 4 Fascicle 4.

[20] U. Kulisch, "Very fast and exact accumulation of products," *Computing*, vol. 91, no. 4, p. 397–405, Apr. 2011. [Online]. Available: https://doi.org/10.1007/s00607-010-0131-y

[21] G. Marsaglia and A. Zaman, "A new class of random number generators," *The Annals of Applied Probability*, vol. 1, no. 3, pp. 462–480, 1991. [Online]. Available: http://www.jstor.org/stable/2959748

[22] Message Passing Interface Forum, "MPI: A message-passing interface standard: Version 3.1," MPI Forum, Knoxville, TN, United States, Tech. Rep., 2015.

[23] T. G. Robertazzi and S. C. Schwartz, "Best "ordering" for floating-point addition," *Transactions on Mathematical Software*, vol. 14, no. 1, p. 101–110, Mar. 1988. [Online]. Available: https://doi.org/10.1145/42288.42343

[24] S. M. Rump, "Ultimately fast accurate summation," *Journal of Scientific Computing*, vol. 31, no. 5, pp. 3466–3502, 2009.

[25] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," in *International Symposium on Formal Methods*, ser. FM 2015, N. Bjørner and F. de Boer, Eds. Oslo, Norway: Springer International Publishing, 2015, pp. 532–550.

[26] Z. Strakoš and P. Tichý, "On error estimation in the conjugate gradient method and why it works in finite precision computations," *Electronic*

Transactions on Numerical Analysis, vol. 13, pp. 56–80, 2002. [Online]. Available: https://www2.karlin.mff.cuni.cz/~strakos/download/2002_StTi.pdf

[27] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49—-66, Feb. 2005. [Online]. Available: https://doi.org/10.1177/1094342005051521

[28] The Open MPI Development Team, *MPIReduce(3) man page*, 4th ed., Software in the Public Interest, Jun. 2020. [Online]. Available: https://www.open-mpi.org/doc/v4.0/man3/MPI_Reduce.3.php

[29] The SimGrid Team, "SMPI: Simulate MPI applications," 2020, available at https://simgrid.org/doc/latest/app_smpi.html#mpi-allreduce.

[30] O. Villa, V. Gurumoorthi, and S. Krishnamoorthy, "Effects of floating-point nonassociativity on numerical computations on massively multithreaded systems," in *CUG 2009 Proceedings*. Cray User Group, 2009, pp. 1–11. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.458.2473

[31] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Mineola, NY, USA: Dover Publications, Inc., 1963.

APPENDIX

PROOF OF (2)

We proceed by proving the limit of the ratios diverge. Observe:

$$\lim_{n\to\infty} \frac{C_n}{n!/2} = \lim_{n\to\infty} \frac{\frac{(2n)!}{(n+1)!n!}}{n!/2} = \lim_{n\to\infty} \frac{2(2n)!}{(n+1)!} = \infty.$$

Next, we find it more convenient to use $n+1$ instead of n for the second inequality. So

$$\lim_{n\to\infty} \frac{g_{n+1}}{(n+1)!/2} = \lim_{n\to\infty} \frac{2(2n-1)!!}{(n+1)!}.$$

Since $2n-1$ is odd, we have

$$(2n-1)!! = \frac{(2n-1)!}{(2(n-1))!!}.$$

And further, since $2(n-1)$ is even, we can factor out a 2 from each term of $(2(n-1))!!$:

$$(2n-1)!! = \frac{(2n-1)!}{(2(n-1))!!} = \frac{(2n-1)!}{2^{n-1}(n-1)!}.$$

Putting this all together,

$$\begin{aligned}\lim_{n\to\infty} \frac{2(2n-1)!!}{(n+1)!} &= \lim_{n\to\infty} \frac{\frac{2(2n-1)!}{2^{n-1}(n-1)!}}{(n+1)!} \\ &= \lim_{n\to\infty} \frac{\frac{(2n-1)!}{2^{n}(n-1)!}}{(n+1)n(n-1)!} \\ &= \lim_{n\to\infty} \frac{(2n-1)!}{2^{n}(n+1)n} = \infty\end{aligned}$$

because the factorial function grows faster than 2^n. □

ARTIFACT DESCRIPTION

All software used to generate the figures and results from this paper are available under the GNU GPLv3 license at

- https://github.com/sampollard/reduce-error

using tag `correctness-2020`.

The data used to generate the results in this paper are available under the GNU GPLv3 license at

- https://dx.doi.org/10.5281/zenodo.4047699

Instructions to re-generate the dataests are included in the `README.md` provided in either repository.

A. *Hardware Information*

Experiments were performed on two Linux nodes:

1) Artemis
   - Ubuntu 18.04.4 LTS running Linux 4.15.0-55-generic
   - Two Intel Xeon Gold 6148 CPUs at 2.40GHz
   - 384GiB DDR4 RAM at 2666MHz
2) Talapas (hpcf.uoregon.edu/content/talapas
   - Only single nodes were requested
   - `lsb_release` is `core-4.1-amd64` running Linux version `3.10.0-957.27.2.el7.x86_64`
   - Two Intel Xeon E5-2690v4 CPUs at 2.60Ghz
   - 128GB DDR RAM
   - IBM GPFS file system

B. *Software Dependencies*

- Lmod environment modules 7.7 to manage environment variables.
- Shell scripts were run using Bash 4.4.20.
- Data analysis and figure generation were performed with `R` 4.0.2 and ggplot2 version 3.3.2.
- Spack 0.13.4 to manage Boost and MPFR
- Boost 1.72.0
  - Boost was loaded with `module load boost-1.72.0-gcc-7.5.0-q725eoa`
- Simgrid 3.25.1, commit hash `4b7251c4ac80`
- GNU MPFR 4.0.2
  - MPFR was loaded using `module load mpfr-4.0.2-gcc-7.5.0-fveqzlf`
- GCC and OpenMPI
  - GCC 7.5.0 and OpenMPI 4.0.3 on Artemis
  - GCC 7.3.0 and OpenMPI 2.1 on Talapas