Numerical Reproducibility and Accuracy at ExaScale

(Invited Paper)

James Demmel
Mathematics Department and CS Division
University of California at Berkeley
Berkeley, CA 94720
demmel@eecs.berkeley.edu

Hong Diep Nguyen
EECS Department
University of California at Berkeley
Berkeley, CA 94720
hdnguyen@eecs.berkeley.edu

I. Introduction

Given current hardware trends, ExaScale computing (10^{18} floating point operations per second) is projected to be available in less than a decade, achieved by using a huge number of processors, of order 10^9. Given the likely hardware heterogeneity in both platform and network, and the possibility of intermittent failures, dynamic scheduling will be needed to adapt to changing resources and loads. This will make it likely that repeated runs of a program will not execute operations like reductions in exactly the same order. This in turn will make reproducibility, i.e. getting bitwise identical results from run to run, difficult to achieve, because floating point operations like addition are not associative, so computing sums in different orders often leads to different results. Indeed, this is already a challenge on today's platforms.

Reproducibility is of interest for a number of reasons. First, it is hard to debug if runs with errors cannot be reproduced. Second, reproducibility is important, and sometimes required, for some applications. For example, in climate and weather modeling, N-body simulation, or other forward unstable simulations, a very small change in results at one time step can lead to a very different result at a later time step. Reproducibility is sometimes also required for contractual reasons where both sides need to agree on the results of the same computation.

We note that reproducibility and accuracy are not synonymous. It is natural to consider just using a standard algorithm at higher precision, say double the precision of the summands ("working precision"). It is true that this makes the probability of nonreproducibility much lower than using working precision. But it does not guarantee reproducibility, in particular for ill-conditioned inputs, or when the result is close to half-way between two floating point numbers in the output precision. And ill-conditioned inputs, i.e. a tiny sum resulting from a lot of cancellation, may be the most common case in many applications. For example, when solving $Ax = b$ using an iterative method, the goal is to get as much cancellation in the residual $r = Ax - b$ as possible.

Our goal is to present an algorithm for summations with the following properties: (1) It computes a reproducible sum independent of the order of the summands, how they are assigned to processors, or how they are aligned in memory. (2) It makes only basic assumptions about the underlying arithmetic (IEEE standard). (3) It scales as well as a performance-optimized, non-reproducible implementation, as n (number of summands) and p (number of processors) grow. (4) The user can choose the desired accuracy of the result. In particular, getting a reproducible result with about the same accuracy as the performance optimized algorithm should only be a small constant times slower, but higher accuracy is possible too.

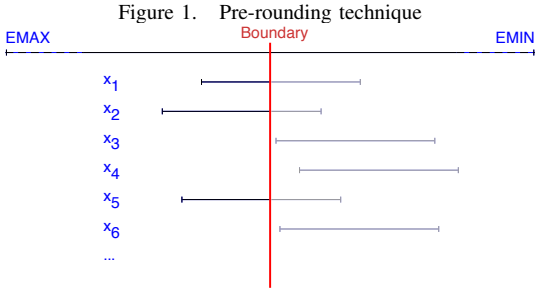
Communication, i.e. moving data between processors, is the most expensive operation on computers today, much more expensive than arithmetic, and hardware trends will only make this performance gap larger at exascale. This means that to achieve goal (3) above, our algorithm may only do a single reduction operation across the machine, and each operand in the reduction tree may only be a small constant factor bigger than a single floating point number. Furthermore, we can make no assumption about the size or shape of the reduction tree. We note that this makes our algorithm attractive in a cloud computing (eg map-reduce) environment too.

In a related paper [2] we introduced two other algorithms for reproducible summation, that require two reduction operations instead of one. In Figure 4 we compare the performance of all three of our algorithms (the one in this paper and the two from [2]) with a performance-optimized nonreproducible summation, on a large parallel machine for $n = 2^{20}$ and p varying from 1 to 2048. For $p \geq 512$ the algorithm in this paper was only 15% slower than the performance-optimized summation. For smaller p our two previous algorithms were sometimes faster. Overall, if we choose our fastest algorithm for each p, we are never more than 2.6x slower than the performance-optimized summation.

II. Proposed Solution

First we discuss some simple but inadequate ways to achieve reproducibility. The first solution is to use a deterministic computation order by fixing the number of processors as well as the data assignment and reduction tree shape. On today's large machines, let alone at ExaScale, such a deterministic computation will lead to substantial communication overhead due to synchronization. Moreover, a deterministic computation order also requires a fixed data order, not achieving goal (1).

Figure 1. Pre-rounding technique



Second, reproducibility can be obtained by eliminating rounding error, to ensure associativity. This can be done using exact arithmetic or correctly-rounded algorithms. It will however increase substantially the memory usage as well as the amount of communication when applied to more complicated operations such as matrix multiplication. Third, fixed-point arithmetic can also be used, but at a cost of limited argument range.

Instead of these, we propose a technique, called pre-rounding, to obtain deterministic rounding errors.

A. Pre-rounding technique

The main idea is to round input values to a common base according to some boundary (Figure 1) so that there will be no rounding error in summing the remaining (leading) parts. Then the error depends only on input values and the boundary, not on the intermediate results, which depend on the order of computation. Thus the computed result will be reproducible so long as the boundary is reproducible.

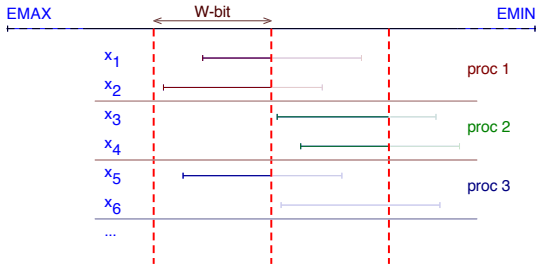
On one hand, the boundary cannot be too small, in order to avoid rounding errors when summing the leading parts. On the other hand, the boundary should be as small as possible in order to maximize the number of significant bits taking part in the summation, i.e. maximizing the accuracy of the computed result. The boundary can be determined from the maximum absolute value of all the input data, which is always reproducible since the maximum operation is associative. The splitting process can be done by directly manipulating the exponent and mantissa of the floating-point number, or by using the splitting algorithm proposed by Rump [4]. In [2], we used a reproducible boundary proportional to $M = c \cdot N \cdot \max|x_i|$ where c is a small constant (≈ 1) depending on which algorithm being used for splitting.

However, the use of maximum absolute value requires extra communication, in particular an extra reduction operation, eg a call to `MPI_Allreduce` with reduction operator `MAX`. This violates our goal of using a single reduction operation. Furthermore, the maximum may not be easily available, for example in the blocked versions of `dgemm` and `trsm` [1].

B. 1-Reduction technique

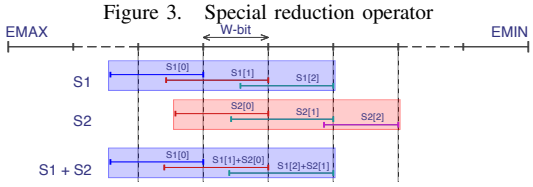
In order to avoid the a second reduction operation, we propose a way to precompute boundaries independently of input data, for example of form $2^{i \cdot W}$ for a carefully chosen W. Using precomputed boundaries requires no communication, and each processor can use its own local boundary, which is the largest one to the right of the leading bit of the local maximum absolute value.

Figure 2. 1-Reduction technique



Since each processor can have a different boundary, during the reduction we only sum intermediate values corresponding to the same boundary. Otherwise, only the intermediate value corresponding to the bigger boundary will be propagated up the reduction tree. Though it can be done easily by using two separate reductions, one for the maximum boundary and one for the summation of partial sums exceeding this boundary, we only want to use one reduction. So we combine these two reductions into one, where we compute both the largest boundary encountered, as well as the partial sum of the values exceeding this boundary. This is depicted in Figure 3 for the case of the 3-fold algorithm which will be explained in the next section. Therefore we call this technique *1-Reduction*.

Figure 3. Special reduction operator



C. Accuracy

The accuracy of the pre-rounding technique depends on the error of each pre-rounding, which in turn depends on the boundary. Since each trailing part that is thrown away in the pre-rounding technique can be bounded by $Boundary$, the total error is bounded by $N \cdot Boundary$. In the 1-Reduction technique, the boundaries are precomputed and do not depend on input data. In the worst case, the maximum absolute value of input data can be only slightly larger than the boundary. Therefore the best error bound that we can get for the 1-Reduction technique is $N \cdot \max|x_i|$, which is much too large.

In order to attain any desired accuracy, instead of keeping only the first leading part of each input datum, we can retain $k > 1$ segments of leading bits. This is analogous to the k-fold technique that has been used in multiple compensated summation algorithms. The same idea can also be found in the cascading accumulator algorithm proposed by Malcolm [3] where each input datum is broken into multiple segments, each containing

only a portion of significant bits, but not according to any precomputed boundary.

Using the k-fold technique, the computation's error is now determined by the minimum boundary being used. Suppose the gap between two consecutive precomputed boundaries is W bits. Then the error bound of the k-fold 1-Reduction algorithm is:

$$absolute\ error < N \cdot 2^{(1-k)\cdot W} \cdot \max|x_i|.$$

This means that the accuracy of 1-Reduction can be tuned by choosing k according to the accuracy requirement of each application.

In practice for the summation of double precision floating-point numbers we use $k = 3$ and $W = 40$. W is a tuning parameter that depends on the desired accuracy, and the relative costs of basic arithmetic versus the (very rare) carry propagation from one segment to another.

Given W, the absolute error will be bounded by

$$N \cdot 2^{-80} \cdot \max|x_i| = N \cdot 2^{-27} \cdot \texttt{macheps} \cdot \max|x_i|,$$

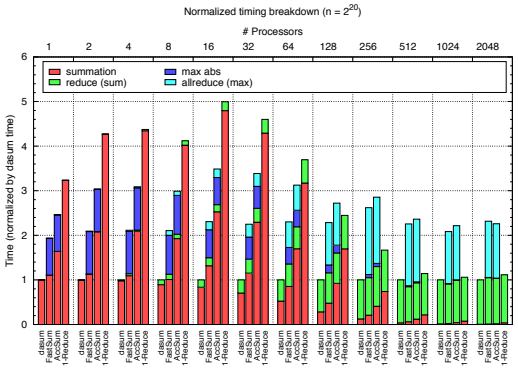
which is at least 2^{-27} times smaller than the error bound of the standard summation algorithm $(N-1) \cdot \texttt{macheps} \cdot \sum_1^N |x_i|$.

III. Experimental results

Preliminary results show that the proposed 1-Reduction technique works both correctly and with performance close to that of performance-optimized nonreproducible summation. Figure 4 summarizes experimental results on Hopper, a Cray XE6 (`www.nersc.gov/systems/hopper-cray-xe6`), for the summation of 2^{20} IEEE double precisions floating-point numbers. For each number of processors $p \in (1, 2, \ldots, 2048)$, we measured the run-time of 4 algorithms: The first uses the optimized non-reproducible `dasum` function in the ACML library to compute the local sum, and then MPI-Reduce to compute the final result. Therefore the running time consists of two parts: the local summation computing time which is colored red and the reduction time which is colored green. The total running time of this algorithm is always normalized to 1 in the figure. The second and third run-times are for the two algorithms Reproducible Sum and Fast Reproducible Sum presented in [2], which both use 2 reduction steps and 2 extraction steps ($k = 2$). Both these algorithms compute the local maximum absolute value (blue part) and communicate to reduce and broadcast the global maximum absolute value to all processors (cyan part). The fourth run-time is for the 1-Reduction technique using $k = 3$ extraction steps. Similar to normal summation, the run-time consists of only two parts: the local computing time (red) and the reduction (green). to reduce the final result.

For small numbers of processors, the running time of each algorithm is dominated by the local computing time (including computing the maximum). Therefore the reproducible summation algorithms are 2 or more times slower than the normal sum in terms of running time.

Figure 4. Performance of 1-Reduction technique on Hopper machine



Nevertheless, for large numbers of processors the running time of all algorithms tends to be dominated by communication time. Consequently the running time of each algorithm is close to proportional to the number of reductions it uses. Therefore, performance-optimized nonreproducible summation and 1-Reduction are both about twice as fast as the other two algorithms. Note that, using k-fold technique with $k = 3$, results computed by all three reproducible algorithms are of much better accuracy than results computed by the performance-optimized sum.

IV. Conclusions and future work

The proposed 1-reduction technique attains reproducibility for parallel heterogeneous computing environments. Reproducibility is guaranteed regardless of the number of processors, reduction tree shape, data assignment, data partitioning, etc. Since no extra inter-processor communication is needed, this technique is well suited for highly parallel environments including ExaScale and cloud computing, where communication time tends to dominate computing time. It can also be applied in data-streaming applications since the boundaries can be updated on-the-fly whenever new values come.

Though currently we have only demonstrated this technique for summation, it can be applied to other operations that use summation as a reduction operator, for example dot product, matrix-vector multiplication, and matrix-matrix multiplication. We also plan to apply this technique to prefix-sum as well as higher level BLAS and linear algebra routines.

References

[1] BLAS (Basic Linear Algebra Subprograms), `http://www.netlib.org/blas/`.

[2] James Demmel and Hong Diep Nguyen. Fast Reproducible Floating-Point Summation. In *21st IEEE Symposium on Computer Arithmetic*, Austin, Texas, USA, April 2013.

[3] Michael A. Malcolm. On accurate floating-point summation. *Commun. ACM*, 14(11):731–736, November 1971.

[4] Siegfried M. Rump. Ultimately fast accurate summation. *SIAM Journal on Scientific Computing (SISC)*, 31(5):3466–3502, 2009.