Research | October 01, 2018

Reproducible BLAS: Make Addition Associative Again!

By James Demmel, Jason Riedy, and Peter Ahrens

A few years ago, a commercial finite element software developer expressed interest in a reproducible parallel sparse linear equation solver [5], i.e., one that can yield bitwise identical results from multiple runs. Some customers and civil engineers had requested this type of solver at the behest of lawyers, who required that civil engineers' earthquake safety analysis code (and approval) of building designs remain consistent during multiple runs.

This motivated me to email 100 or so faculty members at the University of California, Berkeley affiliated with the school's Computational and Data Science and Engineering Program to ask whether they knew that one could no longer expect reproducibility from run to run. The majority of responses were similar. "How will I debug my code if I can't reproduce errors?" many people asked. But there were some unique replies as well. "I know better, I do error analysis to trust my results," one colleague said; of course, these skills (for which that colleague won a Turing Award) are not so common. Another coworker articulated a different concern. "I do fracture analysis with large-scale simulations to look for rare events, and then need to reproduce these rare events exactly to conduct further analysis of why they occurred," he said. "How will I do my science if I can't reproduce them?" A third colleague—whose software is used by the United Nations to detect illegal underground nuclear testing—expressed a further dilemma. "What if my software says 'They did it,' and then 'They didn't do it'?" he said. "That will be a political mess!"

Since then, attendees at numerous supercomputing conferences have discussed the need for reproducibility and proposed ways to achieve it [7]. A number of hardware and software vendors, as well as two standards committees, have begun addressing this challenge.

Of course, there are many reasons why a code may be nonreproducible. The user could link in a different math library, change compiler optimization flags, or write parallel code with race conditions or even "`random_number_seed = get_time_of_day()`." Here we address the more mathematically challenging issue of floating-point addition's nonassociativity due to roundoff, i.e., adding numbers in different orders yields *slightly* different answers, such as $(1 + 10^{20}) - 10^{20} = 0 \neq 1 = 1 + (10^{20} - 10^{20})$.

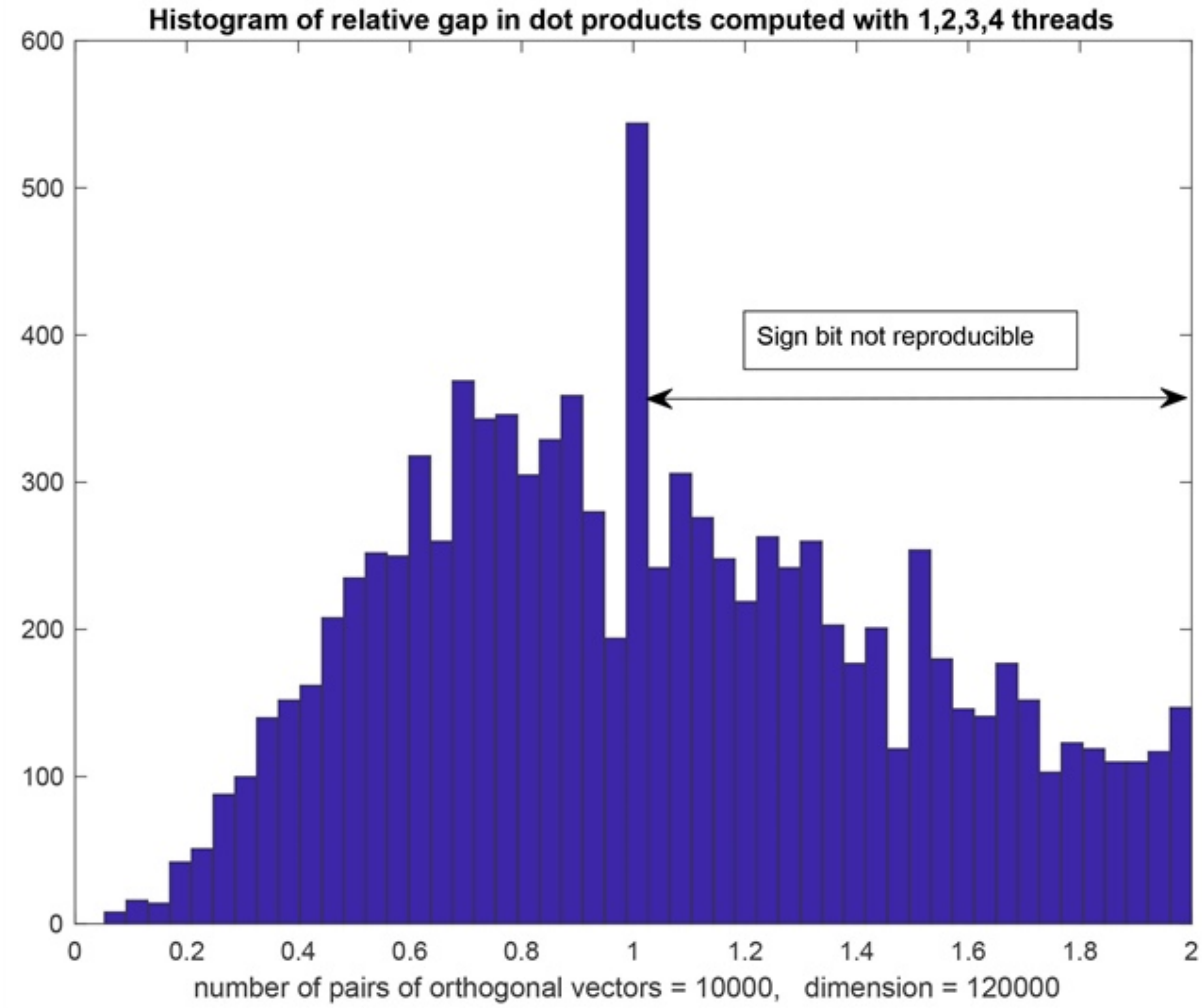
On a parallel machine with a variable (e.g., dynamically allocated) number of processors, a dynamic execution schedule, or even a sequential machine where diverse data alignments may cause a different use of single instruction, multiple data instructions, the order of summation can easily vary between runs or subroutine calls in the same run. This nonreproducibility appears wherever one has computed a parallel sum, from simple cases such as the `OpenMP` “`reduction(+, ...)`” clause to widely-used libraries like the `Basic Linear Algebra Subprograms` (`BLAS`). Nonreproducibility in the `BLAS` affects higher-level applications and the libraries that use them, including most linear algebra packages.

A number of vendors (including Intel and NVIDIA) have responded to this challenge with versions of their libraries that provide reproducibility under limited circumstances using the same summation order, while others (like ARM) offered new hardware instructions to support reproducible summation. Some have suggested correctly-rounded (and thus reproducible) summation techniques [2, 3], which require very large accumulators to exactly represent the entire floating-point sum. Our approach uses just six floating-point numbers to represent a “reproducible accumulator” and produces the same reproducible result when used in any reduction tree over summands in any order. To our knowledge, our method is the first to satisfy all of the following design goals:

1. Achieve reproducible summation independent of summation order using only standard floating-point operations
2. Provide tunable accuracy at least as good as “standard” summation
3. Perform one read-only pass over the summands and one reduction operation in parallel
4. Reproducibly handle overflow, underflow, and other exceptions
5. Enable tiling optimizations common in the `BLAS` and higher-level libraries like `LAPACK` and its parallel extensions using as little memory as possible
6. Provide primitives so parallel runtimes and higher-level environments can implement reproducible linear algebra operations.

Our technique attains these goals and costs $7n$ standard floating-point additions to sum n numbers into a reproducible accumulator consisting of six floating-point values (plus n `max()` and n `abs()` operations). We can extend our work on communication lower bounds and communication-avoiding algorithms to show that linear algebra’s communication costs increase proportionally to the square root of the reproducible accumulator’s size; thus, smaller is better [4].

One can easily describe the algorithm at a high level, although the details and proofs are more complicated [4]; we divide the range of all floating-point exponents into “bins” of a certain fixed width W ($W = 40$ bits for double precision numbers). At execution time, we divide the mantissa bits of each summand into their respective bins and separately sum all the bits within each bin. Since the bits in one bin roughly share a common exponent, summation within a bin behaves like fixed point arithmetic and is exact and reproducible. Furthermore, we only retain the few (typically three) bins corresponding to the highest exponents of any summands seen so far, and discard or omit computation of the rest; this limits both the work and size of the reproducible accumulator and introduces an error with a much smaller bound than is standard. At the end of summation, we can convert the reproducible accumulator back to a standard floating-point number using a few more operations.



We illustrate how parallel summation with a different number of processors can lead to nonreproducibility. We generate many pairs of random orthogonal unit vectors and compute their dot products in MATLAB with four different summation orders, corresponding to 1, 2, 3 or 4-way parallel summation. For each set of four dot products *s* (1:4), which would be identical if summed reproducibly, we compute the relative gap = (max_i s(i) – min_i s(i))/max_i abs(s(i)), or 0 if all *s*(*i*)= 0. The relative gap lies in the range [0, 2] and exceeds 1 if not even the sign bit is reproducible. The histogram depicts relative gaps for 10,000 dot products of dimension 120,000. In about half of the cases, not even the sign bit is reproducible. Figure courtesy of James Demmel.

An initial implementation of the `Reproducible BLAS` (`ReproBLAS`) using this algorithm is freely available. This implementation is sequential but vectorized for many Intel architectures. `ReproBLAS` provides primitives for parallel runtimes with an example summation operation for message passing interface (MPI) reductions. Use of these operations in `MPI_Reduce` delivers reproducible reductions like dot product regardless of the reduction tree's shape. The reductions remain reproducible even if computed redundantly (in possibly different orders) to remove a final, global broadcast.

Wide use of `ReproBLAS` naturally depends on performance cost. We compare `ReproBLAS` performance to routines in Intel's Math Kernel Library (MKL) on an Intel Sandy Bridge (i7-2600)-based system. For vectors that fit in the processor's top-level (L1) cache, a double-precision dot product (ddot) runs 3.3 to 4.2 times slower than the optimized MKL ddot — a worst-case scenario for ddot. The performance penalty is around $2.6\times$ for vectors that are too long for any cache. Implementing the inner loop in two hardware instructions [6] decreases the performance gap to $1.8\times$. Finally, the slowdown was just $1.2\times$ when comparing the speed of summing 1M numbers spread across 1K processors of a Cray XC30; this is because the required time for arithmetic is small compared to the reduction's network latency time. When data transfer dominates execution time, reproducibility could become the default. More performance comparisons are available in [4], including higher-level `BLAS`.

This work is influencing the possible directions of two ongoing standards committees: the IEEE 754 Standard for Floating-Point Arithmetic, which is up for renewal in 2018, and the `BLAS` standard, which is considering a variety of possible extensions — including a reproducible version of the `BLAS`. It turns out that our algorithm's inner loop uses a slight variation of a well-known operation, often called two-sum, that takes summands x and y and precisely returns both their rounded sum (call it h for "head") and the exact rounding error (call it t for "tail"), i.e., $h = \mathrm{round}(x + y)$ and $t = x + y - h$. Researchers have long used two-sum (and variants) to simulate extra-precise arithmetic, like double-double and compensated summation.

Implementing two-sum can take from three to seven standard floating-point operations, depending on prior (application-dependent) information about the relative magnitudes of x and y. Standardizing two-sum's behavior permits hardware implementations in one or two instructions. But we cannot use the conventional IEEE 754 rounding directions. In order to maintain reproducibility while supporting existing uses, new operations must break ties in a value-independent manner. Rounding to nearest but breaking ties toward zero works; this mode already exists for decimal arithmetic but not for binary works. In July 2018, the IEEE 754 committee voted to send a standard—including two-sum and related operations—to the next level of the standards process.

Implementing these operations as one or two hardware instructions reduces reproducible summation's arithmetic cost from $7n$ to $3n$ or $4n$. Removing performance as a bottleneck widens its appeal. Meanwhile, the `BLAS` standard group is working on a portable specification and names for both this and other algorithms [1].

Future work (possibly for many!) includes completion of the sequential and parallel `ReproBLAS` and exploration of its role in making higher-level linear algebra libraries and environments reproducible.

An expanded list of references is available online.

References

[1] Batched BLAS. (n.d.). Retrieved from http://icl.utk.edu/bblas/.

[2] Chohra, C., Langlois, P., & Parello, D. (2016). Reproducible, Accurately Rounded and Efficient BLAS. In F. Desprez, P.-F. Dutot, C. Kaklamanis, L. Marchal, K. Molitorisz, L. Ricci, ..., J. Weidendorfer (Eds.), *Euro-Par 2016: Parallel Processing Workshops* (pp. 609-620). Grenoble, France: Springer International Publishing.

[3] Collange, S., Defour, D., Graillat, S., & Iakymchuk, R. (2015). Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Para. Comput., 49*, 83-97.

[4] Demmel, J., Ahrens, P., & Nguyen, H.D. (2016). *Efficient Reproducible Floating Point Summation and BLAS* (Electrical Engineering and Computer Sciences). Berkeley, CA: University of California, Berkeley.

[5] Diethelm, K. (2010). Reproducible parallel solvers for linear systems. *NA Digest, 10*(36). Retrieved from http://www.netlib.org/na-digest-html/10/v10n36.html#1.

[6] Dukhan, M., Vuduc, R., & Riedy, J. (2016). Wanted: Floating-Point Add Round-off Error Instruction. *ISC High Performance Workshops*. Frankfurt, Germany. Preprint, *arXiv:1603.00491*.

[7] Leeser, M., Ahn, D.H., & Taufer, M. (2015). *SC15 Birds of a Feather (BoF): Reproducibility of High Performance Codes and Simulations Tools, Techniques, Debugging*. Retrieved from https://gcl.cis.udel.edu/sc15bof.php.

James Demmel is a professor of electrical engineering and computer sciences (EECS) and mathematics at the University of California, Berkeley and current chair of the Department of EECS. In his spare time he avoids communication. Jason Riedy is a senior research scientist in the Georgia Institute of Technology's College of Computing. His research ranges from high-performance data analysis and novel computing architectures to urban honey bees. Peter Ahrens is a third-year computer science graduate student and Department of Energy Computational Science Graduate Fellow at the Massachusetts Institute of Technology. His research focuses on writing algorithms to automatically optimize linear algebraic operations.