# Computational Reproducibility in Production Physics Applications

Robert W. Robey
Eulerian Codes Group, Computational Physics Division
Los Alamos National Laboratory
brobey@lanl.gov

## ABSTRACT

Computational reproducibility is becoming a leading concern for the Exascale computing era. As the problem sizes grow, the degree of parallelism and asynchronous, out-of-order computing will also need to increase, but with this parallelism also comes potentially less reproducibility of results. To deal with these issues, new parallel algorithms and techniques have also started to emerge. This paper details some of the techniques used in production physics codes and their observed benefits.

## Categories and Subject Descriptors

G.1.0 [**Mathematics of Computing**]: Numerical Analysis—*Computer arithmetic, error analysis, multiple precision arithmetic, numerical algorithms, parallel algorithms*

## General Terms

Algorithms

## Keywords

Reproducibility, global sums

## 1. INTRODUCTION

There is an increasing need for computational reproducibility in large production calculations due to three causes. The first driver is computational predictability of real-world problems without resorting to testing. This is a cornerstone of the DOE Accelerated Strategic Computing (ASC) program first established in 1995 with a shift away from testing. To achieve this goal, the quality of the simulations needs to improve and this begins with reproducibility and correctness. A second driver is the changing hardware with increasing parallelism, asynchronous operation, and out-of-order computation. These changes in hardware can cause the calculational results to vary. The last driver is the increasing size of calculations. This effect is shown in Figure 1 for the Leblanc shock tube problem with increasing number of cells. Using a long double summation variable with 80 bits of precision or 2 extra digits helps, but it is not enough to eliminate the error.

The reproducibility problem is dominated by the error of global sums across the entire computational mesh. Perhaps the best example of a reproducibility error is a solver based on reducing the residual error across all the cells using a global sum of the residual error. On a different number of processors, the solver can take a different number of iterations and thereby amplify the differences in the results. One technique to avoid this problem is to use the maximum residual instead.

The root cause of this error in the global sums is that finite precision addition is not associative. Simply, if the order of the calculations is different, the result of the calculation will be different. And the order of the calculation is different when the number of processors change or when calculations operate out-of-order for performance reasons. For many years, the emphasis has been to control the order of operations to reduce these reproducibility errors.

A few years ago we realized that there was another way to look at this issue. Infinite precision addition is associative. Thus, another way to deal with the global sum error is to use more precision in the summation. This is a more tractable approach for parallel programs where a sort is too expensive to even consider.

So now that we have two ways to deal with the summation errors, we need to assess their effects. Enforcing order will improve the consistency of the results, but the accuracy of the answer is not improved. The newer approach of increasing precision not only covers a wider range of reproducibility problems, but gives a more accurate answer as well. The accuracy generally improves from 7 digits with standard precision to 15 digits with enhanced precision in a typical computation.

## 2. BACKGROUND

The topic of enhanced precision sums goes back to the early days of computing hardware with CPUs that had varying floating point representations. W. Kahan was one of the first to address precision techniques in 1965 [6] and was a key contributor to the early IEEE floating point standards. A more formal analysis on finite precision floating point addition was given by Knuth in 1969 [7]. See Theorem B in Section 4.2.2 for a technique to collect the error for both terms, while Theorem C corresponds to the Kahan technique of collecting the error for one term. Thus when using

*NRE2015,* November 15-20 2015, Austin, TX, USA
Copyright is held by the owner/author(s).
ACM 978-1-4503-4007-6/15/11
http://dx.doi.org/10.1145/2831239.2831242.

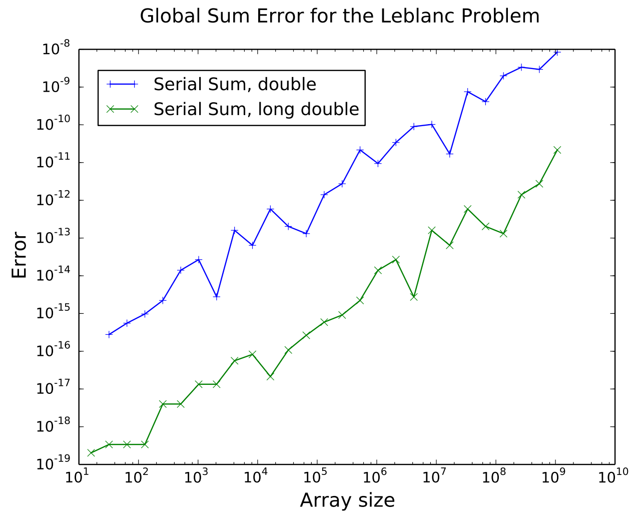

**Figure 1: Growth in global sum error with increasing problem size.**

the simpler Kahan algorithm, it is necessary to know which term will generally be greater.

In the mathematical community, implementations of general purpose extended precision libraries were first developed by Bailey around 1995 [3] with the multiple precision MP-FUN package. Eventually that led to the inclusion of a quad precision package in recent C, C++ and Fortran compilers.

The use of enhanced precision sums specifically for addressing the reproducibility of parallel global sums was introduced by Robey, Robey, Aulwes in 2011 [10, 9]. Cleveland, et al. [4] followed in 2013 with a technique to obtain reproducibility in a Monte Carlo calculation using enhanced precision sums along with rounding/truncation. Anderson [1] addressed the reproducibility of the parallel dot product.

## 3. EFFICIENT TECHNIQUES

In this section we review some of the techniques used in production codes where efficiency is an important consideration. These techniques and algorithms are still being developed and adapted for the challenges of the emerging hardware.

### 3.1 Pair-wise summation

The pair-wise sum of a sequence of numbers is an elegant way to enforce order in summations while also summing into smaller, similarly-sized accumulators. This pair-wise sum is repeated until there is only one remaining number which is the sum. This technique nearly eliminates the increasing error due to increasing array sizes in this study. It was first implemented at LANL by M. Gittings (SAIC) in the Rage code [5]. Shown in Listing 1 is an implementation used in this study.

**Listing 1: Pair-wise Summation**

```
double *pwsum = (double *)malloc(ncells/2*sizeof(
      double));

long nmax = ncells/2;
for (long i = 0; i<nmax; i++){
  pwsum[i] = var[i*2]+var[i*2+1];
}

for (long j = 1; j<log2(ncells); j++){
  nmax /= 2;
  for (long i = 0; i<nmax; i++){
    pwsum[i] = pwsum[i*2]+pwsum[i*2+1];
  }
}
double sum = pwsum[0];

free(pwsum);
```

### 3.2 Enhanced Precision Sums

#### *Long Double*

The simplest enhanced precision datatype is an 80-bit long double type available on some systems where the numeric registers are 80 bits. The storage size is two doubles when it is written to main memory. In C, the accumulator variable just needs to be declared *long double* as shown in Listing 2. Run-time is nearly the same since the operation is already done in 80 bits and just the rounding/truncation operation changes. As shown in Figure 1, the error is reduced substantially. But some error still remains and in addition the portability of this technique is limited.

**Listing 2: Long Double Summation**

```
long double ldsum = 0.0;
for (long i = 0; i < ncells; i++){
  ldsum += var[i];
}
```

#### *Kahan Sum*

The Kahan sum uses a double-double type consisting of two standard doubles. The truncation term is stored as a correction to add back in with the next term. An enhanced precision data type is defined and operated on as shown in Listing 3. Some of the other methods also use the esum_type composed of two doubles for convenience and that will not be repeated in their listings.

**Listing 3: Kahan Sum**

```
struct esum_type{
  double sum;
  double correction;
};

double corrected_next_term, new_sum;
struct esum_type local;
local.sum = 0.0; local.correction = 0.0;

for (long i = 0; i < ncells; i++) {
  corrected_next_term = var[i] + local.correction;
  new_sum = local.sum + local.correction;
  local.correction = corrected_next_term - (
        new_sum - local.sum);
  local.sum = new_sum;
}
double sum = local.sum + local.correction;
```

#### *Knuth Sum*

The Knuth sum is similar to the Kahan sum, but with corrections carried for each of the terms in the sum. For running sums like those here or in the dot product, the Knuth sum does not give any additional precision to the results. The coding is a little more complex as is shown in Listing 4.

**Listing 4: Knuth Sum**

```
double u, v, upt, up, vpp;
struct esum_type local;
local.sum = 0.0; local.correction = 0.0;

for (long i = 0; i < ncells; i++) {
  u = local.sum;
  v = var[i] + local.correction;
  upt = u + v;
  up = upt - v;
  vpp = upt - up;
  local.sum = upt;
  local.correction = (u - up) + (v - vpp);
}
double sum = local.sum + local.correction;
```

*Quad Double*

The quad double uses the new __float128 datatype introduced in the GCC 4.6.4 compiler and also in many other compilers. The accumulator is declared with the quad datatype in a similar way as was done with the long double as shown below in Listing 5. A second version of the routine was written with a quad precision energy array.

**Listing 5: Quad Double Summation**

```
__float128 qdsum = 0.0;
for (long i = 0; i < ncells; i++){
  qdsum += (__float128)var[i];
}
```

## 3.3 Rounding/Truncation

Cleveland, et al. [4] introduced the idea of truncating the number of significant digits with a proper rounding to gain complete reproducibility. While this can simply be done by limiting the digits printed out, we do it here by truncating the binary number. The number of digits is dependent on the summation technique and the input data dynamic range. Shown in Listing 6 is an implementation with the standard double summation. The number of digits needs to be input to the routine. Seven digits was used for the double with truncation routine, and ten for the long double.

**Listing 6: Summation with Rounding/Truncation**

```
double sum = 0.0;
for (long i = 0; i < ncells; i++){
  sum += var[i];
}
int n = (int)log10(sum);
double mult = pow((double)10.,(double)(digits-n));
sum = round(sum*mult)/mult;
```

## 3.4 MPI Implementation

In parallel, the enhanced precision sum should also be done within the MPI reduction operation. In order to do this we create an MPI Operation as was shown in Robey [10]. A slightly modified version is shown in Listing 7. This version uses the Kahan sum on processor since it is the most appropriate for a running sum where it is known which term is generally the largest. The MPI Op uses the Knuth sum since we don't know which term will be the largest and we should accumulate corrections for both terms. This will be more important as the number of MPI ranks grows. In the MPI_Op_Create call, the current code shows the commute flag set to 1 (true). This allows the sum to be reordered. Setting it to false would force the sum to be done in the order of the MPI ranks, which would give more reproducibility. What we really want here is to force the operations to occur as a fixed order tree similar to the pair-wise serial algorithm. This is what we hope to get with the commute flag set to true, but some experimentation and study of current MPI implementations is still needed.

**Listing 7: Parallel Enhanced Precision Sum with MPI Op**

```
double parallel_mpiop_enhanced_precision_sum(
    double *var) {
  double u, v, upt, up, vpp;
  struct esum_type local, global;
  MPI_Datatype MPI_TWO_DOUBLES;
  MPI_Op KNUTH_SUM;
  local.sum=0.0; local.correction=0.0;
  for(uint i=ncells; i<isize+ncells; i++){
    corrected_next_term = var[i] + local.
        correction;
    new_sum = local.sum + corrected_next_term;
    local.correction = corrected_next_term - (
        new_sum - local.sum);
    local.sum = new_sum;
  }
  MPI_Type_contiguous(2, MPI_DOUBLE, &
      MPI_TWO_DOUBLES);
  MPI_Type_commit(&MPI_TWO_DOUBLES);
  MPI_Op_create((MPI_User_function *)knuth_sum, 1,
      &KNUTH_SUM);
  MPI_Allreduce(&local, &global, 1,
      MPI_TWO_DOUBLES, KNUTH_SUM, MPI_COMM_WORLD);
  MPI_Op_free(&KNUTH_SUM); MPI_Type_free(&
      MPI_TWO_DOUBLES);
  return(global.sum);
}
void knuth_sum(struct esum_type *in, struct
    esum_type *inout, int *len, MPI_Datatype *
    MPI_TWO_DOUBLES) {
  double u, v, upt, up, vpp;
  u = inout->sum;
  v = in->sum+(in->correction+inout->correction);
  upt = u + v;
  up = upt - v;
  vpp = upt - up;
  inout->sum = upt;
  inout->correction = (u - up) + (v - vpp);
}
```

## 3.5 OpenMP Implementations

An OpenMP implementation of the regular double summation and the Kahan Sum were written. The Kahan Sum implementation is a little more complicated, so it is shown in Listing 8.

**Listing 8: OpenMP Implementation of Kahan Sum**

```
double sum = 0.0;
#pragma omp parallel reduction(+:sum)
{
  double corrected_next_term, new_sum;
  struct esum_type local;
  local.sum = 0.0; local.correction = 0.0;
#pragma omp for
  for (long i = 0; i < ncells; i++) {
    corrected_next_term= var[i] +local.correction;
    new_sum = local.sum + local.correction;
    local.correction = corrected_next_term - (
        new_sum - local.sum);
    local.sum = new_sum;
  }
  sum += local.correction;
#pragma omp barrier
  sum += local.sum;
}
```

| Method | Error | Run-time |
|---|---|---|
| Double | -1.99e-09 | 0.116 |
| Double w/trunc | 0.0 | 0.120 |
| Long Double | -1.31e-13 | 0.118 |
| Long Double w/trunc | 0.0 | 0.116 |
| Kahan Sum | 0.0 | 0.406 |
| Knuth Sum | 0.0 | 0.704 |
| Pair-wise Sum | 0.0 | 0.402 |
| Quad double | 5.55e-17 | 3.010 |
| Full Quad double | -4.81e-27 | 2.454 |
| OpenMP double | 2.465e-10 | 0.047519 |
| OpenMP Kahan | 1.388e-16 | 0.062805 |

**Table 1: Precision and Run-time results for various global sum techniques.**

## 4. RESULTS

A test program, globalsums.c, was written to look at CPU summation errors. The Leblanc shock tube problem was used as a test case since the starting condition has a large dynamic range. We use the version for the energy variable with the high value at 1.0e-1 and the low value at 1.0e-10. Significant errors were obtained with double precision and long double precision techniques as shown in Figure 1. The rest of the techniques were able to reduce the error relative to the exact answer to zero, or around 1.0e-16 or lower, as shown in Table 1. These results are for a problem size of 134,217,728 cells since that is at the upper range of what can be run on a single CPU in real problems. Note that the non-quad methods have no way of representing an error less than 1.0e-16 in their numeric representation. The truncation methods are applied to both the sum and the exact answer and are also successful at reducing the error to zero. Run-times are reasonable with about a factor of three for Kahan and pair-wise summations and six for the Knuth sum. The quad precision methods are much more expensive as is expected. The full quad double takes a quad precision input array and gets remarkable precision while the version operating on regular doubles is apparently limited by the accuracy of the conversion from double to quads. The OpenMP implementations of the double and Kahan sums give slightly different errors than the serial versions. Note that the error for the OpenMP doubles decreases a little due to the multiple accumulators, one for each thread.

## 5. IMPACTS

In the short time these techniques have been used in various computational physics codes, there have been both expected and unexpected benefits from the improved quality of the global sums.

### 5.1 Computational Reproducibility

The initial driver for implementing a global sum was to get the same results, independent of the number of processors. The improvement in the global sums reduced the variation in the total mass and other global sum outputs. The net impact was that the output values became consistent to almost 15 digits from the original 7 digits, reducing the differences in the outputs from runs with different numbers of processors.

### 5.2 Parallel Programming Correctness

Another benefit of the better global sums is that it enables the detection of more parallel programming bugs such as a missing halo update that would cause the code to use outdated values. The magnitude of these errors is often below the error in the global sum. The net impact of this was the ability to produce a higher quality parallel production code.

### 5.3 Demonstrate Conservation Properties

Another critical benefit is that the greater consistency of the global summation allows us to show that critical conserved quantities are properly conserved. The numerical equations are generally posed in conservation form because the Lax-Wendroff theorem [8] then guarantees a weak solution to the equations. Benefits of this are that the shock speeds will generally be correct and the solution will be robust. But this is assuming that the coding of the equations is properly done. The improved global sums allows us to demonstrate that mass and energy are properly conserved and that we should actually get the simulation properties guaranteed by the conservation form and the Lax-Wendroff theorem. If you care that you have correctly implemented the system of equations and that your code is more robust and does not crash as often. you should be using the higher accuracy global sums.

### 5.4 Resiliency through an Invariant Property

An unexpected benefit of the improved global sums is having a high quality invariant property. In recent work by Atkinson, et al. [2], the total mass in the shallow water Adaptive Mesh Refinement (AMR) CLAMR mini-app, was used to detect memory errors proactively during a simulation run. The research project initially began by inserting memory errors randomly into the processor during calculation and observing how they impacted runs. This required some measure of the correctness of the run. At first, the full results were compared at each time-step at the end of the run. Looking for other simpler ways of detecting errors, the total mass was checked during the run. If the sum varied too much, it was an indicator that the simulation had a problem. It was found that the total mass measure was an early detector of many memory errors including both silent data errors and crashes. Based on this, the team implemented an automatic restart capability if the total mass varied more than a set amount. Since the detection would often be before the code crashed and the recovery could be done immediately rather than having to resubmit the job. This also reduces the need for user intervention for fault recovery.

## 6. CONCLUSION

We have shown that there are a few viable approaches to obtain better quality and more consistent global sums that are appropriate to include in production simulation codes. The additional runtime costs are generally pretty modest, and when running with MPI in a typical explicit Eulerian calculation will not even be noticeable.

## 7. REFERENCES

[1] A. Anderson. Achieving numerical reproducibility in the parallelized floating point dot product. 2014.

[2] B. Atkinson, N. DeBardeleben, Q. Guan, R. Robey, and W. M. Jones. Fault injection experiments with the clamr hydrodynamics mini-app. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 6–9. IEEE, 2014.

[3] D. H. Bailey. A fortran 90-based multiprecision system. *ACM Transactions on Mathematical Software (TOMS)*, 21(4):379–387, 1995.

[4] M. A. Cleveland, T. A. Brunner, N. A. Gentile, and J. A. Keasler. Obtaining identical results with double precision global accuracy on different numbers of processors in parallel particle monte carlo simulations. *Journal of Computational Physics*, 251:223–236, 2013.

[5] M. Gittings. private communication, 2015.

[6] W. Kahan. Further remarks on reducing truncation errors. *Communications of the ACM*, Vol. 8(1):40, 1965.

[7] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley Press, 1969. Chap. 4.

[8] P. Lax and B. Wendroff. Systems of conservation laws. *Communications on Pure and Applied Mathematics*, 13(2):217–237, 1960.

[9] R. W. Robey. Reproducibility for parallel programming. Reproducibility Workshop, Supercomputing Conference, 2013.

[10] R. W. Robey, J. M. Robey, and R. Aulwes. In search of numerical consistency in parallel programming. *Parallel Computing*, 37(4):217–229, 2011.