In Search of Numerical Consistency in Parallel Programming

Robert W. Robey[a,*], Jonathan M. Robey[b], Rob Aulwes[c]

[a] *Los Alamos National Laboratory[1], X-Computational Physics Division, XCP-2, MS T086, Los Alamos, NM 87545*
[b] *University of Washington, Applied Math Department, Guggenheim Hall #414, Box 352420, Seattle, WA 98195-2420 and Los Alamos National Laboratory[1], Computational Physics and Methods, CCS-2, Los Alamos, NM 87545*
[c] *Los Alamos National Laboratory[1], Applied Computer Science, CCS-7, MS T080, Los Alamos, NM 87545*

Abstract

We present methods that can dramatically improve numerical consistency for parallel calculations across varying numbers of processors. By calculating global sums with enhanced precision techniques based on Kahan or Knuth summations, the consistency of the numerical results can be greatly improved with minimal memory and computational cost. This study assesses the value of the enhanced numerical consistency in the context of general finite difference or finite volume calculations.

Keywords: Kahan summation, Knuth summation, reproducibility, numerical consistency, parallel programming, finite difference, finite volume

1. Introduction

Development of parallel numerical software is hampered by numerical differences when running on different numbers of processors. Different sums of conserved values create a doubt whether the programming is correct or if the difference is due to the numerical precision of the global summation. Simply summing the data in a different order such as by row first instead of by column first can give different results. While within the error tolerance of the numerical method and the finite precision arithmetic of the computer processor, these numerical differences create difficulties in software development and mask errors

*Corresponding author
Email addresses: brobey@lanl.gov (Robert W. Robey), robeyj@u.washington.edu (Jonathan M. Robey), rta@lanl.gov (Rob Aulwes)

[1] Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396.

in the software programming. The numerical differences also give a distorted view of the numerical accuracy of the numerical method.

For the purpose of this paper, numerical consistency is defined as follows.

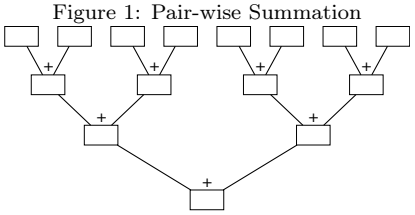
Numerical Consistency (for parallel programming) – getting the same result independent of the number of processors, or to a lesser extent, from cycle-to-cycle for conserved quantities.

Numerical consistency is independent of numerical accuracy or numerical correctness. A parallel simulation on different numbers of processors can have numerically consistent results and still be incorrect or inaccurate.

Several higher precision summation techniques have been developed for other applications. The most obvious method is to sort the data in ascending magnitude and then sum. With sorting, the numerical consistency of the sum can be made exactly the same in any condition. Note that there is still a truncation error to the sum, but it is the same every time. However, a sort operation is computationally expensive and is even more so for a parallel calculation. Sorting on processor would bring some benefit, but is still not practical due to the cost of the operation.

A second possible approach is to do a pair-wise summation as shown in Figure 1. This approach does a recursive pair-wise addition where pairs of numbers are added, reducing the size of the array with each iteration by a factor of two in a tree-like pattern until there is only one value. This gives a more accurate summation since at each step the magnitude of the summed value has the same number of contributors reducing the potential for truncation error. This method is also faster than the sort-based method.

Figure 1: Pair-wise Summation

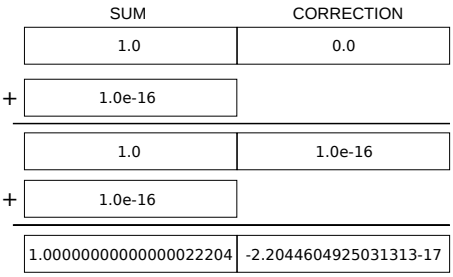


Truncation can also be reduced by increasing the precision of the underlying data representation from 64-bit to 128-bit, sometimes called a "double-double" technique. This method is effective, but again has a large memory and computational cost. It can be used in addition to the other methods and can be used selectively, reducing the extra memory and computational cost to only where it is most effective. An intermediate approach is to use a long-double in C which is an 80-bit type on x86 architectures and still is done in hardware to minimize performance impacts. However this is not a portable approach as discussed later.

A method developed by Kahan[**?**] in 1965 uses carry digits to bring along the truncated part of the addition operation. The Kahan summation is basically a whole second variable, in this case 64-bit, acting as multiple carry digits that carries the truncated part of the sum as shown in Figure 2. In the example in the figure, a small value is added to one, but it is too small

to change the sum. It is put into the correction term and added again in the next cycle along with the next value. When the next value is of the same small order of magnitude, it is just large enough to change the last bit of the sum. The remainder continues to be carried in the correction term.

Figure 2: Kahan Carry



Knuth developed a more formal error analysis for finite precision arithmetic as given in Theorem B in Chapter 4, Volume 2 of The Art of Computer Programming[**?**]. It takes a similar form to the Kahan summation with a carry digit, but it has a few more terms than the Kahan summation.

There have been previous studies on the use of more accurate summations. One was for the purpose of comparing simulations of surface sea height results[**?**] in global climate modeling simulations. In this study, He and Ding also suggest the more general use of more accurate algorithms, though that is not the central theme. They looked at both Kahan summations and a double-double technique. Ozawa and Miyazake[**?**] published a study on more accurate parallel summation techniques where they developed a parallel recursive doubling technique along the lines of the pair-wise summation discussed previously. The Smith and Margetts[**?**] study looked at the effect of global sum errors on iterative solvers. In some cases their analysis showed that the lack of consistency in the global sum played a major role in the variability of the calculations on different number of processors.

The emergence of advanced hybrid architectures with their stronger single precision performance relative to double precision are causing renewed interest in methods that utilize the single precision performance without incurring numerical errors. These architectures include graphical processing units (GPU) and the Cell accelerators in the Roadrunner machine at Los Alamos National Laboratory that was the first system to break a petaflop. Anderson, et. al. and Sainio found that they had to implement Kahan summation to improve accuracy in Quantum Monte Carlo methods and cosmological simulations that were adapted for the GPUs[**? ?**]. Weber, et. al. did a case study of GPU programming and found that using Kahan summation on the GPU would be faster than double-precision on current Nvidia GPUs [**?**].

Even with traditional architectures, there is evidence that numerical errors can affect results in parallel scientific applications. Wang and Borland implemented a Kahan summation in Pelegant, a scientific application for electron accelerator simulations in order to validate their parallel version[**?**]. We show our own example of how enhanced precision sums can help reveal parallel programming errors later in this article.

2. Theory/Calculation

The global sum operation is the greatest source of consistency error in parallel programs. We will reduce this consistency error by replacing the global sum with a higher precision method. The methods chosen are the Kahan summation and the Knuth summation due to their simplicity, low additional cost and their added precision. The computer code for the Kahan summation[**?**] is shown in Listing 1. The variable naming is adapted from Manning.

Listing 1: Serial Kahan Summation

```
double serial_kahan_sum(double **var)
{
  double sum, correction, corrected_next_term, correction, new_sum
  sum=0.0;
  correction=0.0;
  for(int j=0; j<jsize; j++){
    for(int i=0; i<isize; i++){
      corrected_next_term = var[j][i] - correction;
      new_sum = sum + corrected_next_term;
      correction = (new_sum - sum) - corrected_next_term;
      sum = new_sum;
    }
  }
  return(sum);
}
```

The Knuth sum is based on Theorem B [**?**] and provides a scheme similar to the Kahan summation to add two numbers and retain a correction term. The symbols $\oplus$ and $\ominus$ are used to represent finite-precision addition and subtraction, respectively. The symbols $+$ and $-$ are reserved for exact arithmetic. Theorem B states

$$\begin{aligned} u+v &= (u \oplus v) + ((u \ominus u^{'}) \oplus (v \ominus v^{''})) \\ u^{'} &= (u \oplus v) \ominus v \\ v^{''} &= (u \oplus v) \ominus u^{'} \end{aligned}$$

From Theorem B, it is noted that

$$(u \ominus u^{'}) \oplus (v \ominus v^{''})$$

is the correction term. If Kahan summation is applied when adding just two numbers, than the correction term is

$$((u \oplus v) \ominus u) \ominus v$$

Comparing Kahan's correction term to Knuth's correction term, we see that the correction term in the Kahan summation leaves out the term $u \oplus u''$ (swapping u for v). We therefore expect better precision with Knuth summation.

The Knuth summation algorithm is shown in Listing 2.

Listing 2: Serial Knuth Summation

```
double serial_knuth_sum(double **var)
{
  double sum, c, u, v, upt, up, vpp;

  sum = 0.0;
  c = 0.0;
  for(int j=0; j<jsize; j++){
    for(int i=0; i<isize; i++){
      u = sum;
      v = var[j][i] + c;
      upt = u + v;
      up = upt - v;
      vpp = upt - up;
      sum = upt;
      c = (u - up) + (v - vpp);
    }
  }
  return(sum);
}
```

To convert these to a parallel global sum, the sums and corrections on each processor can be added using an MPI_Allreduce. To facilitate the algorithm coding, a sum + correction pair is placed in a C data type using a positive correction sign. This data type is shown in Listing 3. The esum_type will be used in all of the following routines.

Listing 3: Struct for esum_type

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

Combining all of these changes results in a simple parallel Kahan summation as shown in Listing 4.

Listing 4: Simple Parallel Kahan Summation

```
double simple_parallel_kahan_sum(double **var)
{
  double corrected_next_term, new_sum;
  struct esum_type local, global;

  local.sum=0.0;
  local.correction=0.0;
  for(int j=0; j<jsize; j++){
    for(int i=0; i<isize; i++){
      corrected_next_term = var[j][i] + local.correction;
      new_sum = local.sum + corrected_next_term;
      local.correction = corrected_next_term - (new_sum-local.sum);
      local.sum = new_sum;
    }
  }

  MPI_Allreduce(&local, &global, 2, MPI_DOUBLE, MPI_SUM,
      MPI_COMM_WORLD);
```

```
  return(global.sum);
}
```

A similar process gives the simple parallel Knuth summation shown in Listing 5.

Listing 5: Simple Parallel Knuth Summation

```
double simple_parallel_knuth_sum(double **var)
{
  double u, v, upt, up, vpp;
  struct esum_type local, global;

  local.sum = 0.0;
  local.correction = 0.0;
  for(int j=0; j<jsize; j++){
    for(int i=0; i<isize; i++){
      u = local.sum;
      v = var[j][i] + local.correction;
      upt = u + v;
      up = upt - v;
      vpp = upt - up;
      local.sum = upt;
      local.correction = (u - up) + (v - vpp);
    }
  }
  MPI_Allreduce(&local, &global, 2, MPI_DOUBLE, MPI_SUM,
      MPI_COMM_WORLD);

  return(global.sum);
}
```

The next two variants use a custom MPI data type (MPI_Datatype) and a custom MPI operation (MPI_Op). MPI_Allreduce provides the capability to define a custom reduce operation and the custom reduce defined is the Kahan or the Knuth summation. An MPI_Datatype is used to encapsulate the local sum and the local correction term.

The first of the two variants using the MPI_Op is the Kahan summation on the local processor and a Kahan summation placed into a custom MPI_Op. The function used for this operation is shown in Listing 6. The custom MPI_Datatype and custom MPI_Op can be done once during the run rather than every time the global summation is calculated.

Listing 6: Kahan MPI Operation

```
double parallel_mpiop_kahan_sum(double **var)
{
  int commutative = 1;
  double corrected_next_term, new_sum;
  struct esum_type local, global;
  MPI_Datatype MPI_TWO_DOUBLES;
  MPI_Op KAHAN_SUM;

  local.sum=0.0;
  local.correction=0.0;
```

```
  for(int j=0; j<jsize; j++){
    for(int i=0; i<isize; i++){
      corrected_next_term = var[j][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 *)kahan_sum, commutative, &
      KAHAN_SUM);

  MPI_Allreduce(&local, &global, 1, MPI_TWO_DOUBLES, KAHAN_SUM,
      MPI_COMM_WORLD);

  MPI_Op_free(&KAHAN_SUM);
  MPI_Type_free(&MPI_TWO_DOUBLES);

  return(global.sum);
}

void kahan_sum(struct esum_type *in, struct esum_type *inout, int *
    len, MPI_Datatype *MPI_TWO_DOUBLES)
{
  double corrected_next_term, new_sum;

  corrected_next_term = in->sum + (in->correction+inout->correction
      );
  new_sum = inout->sum + corrected_next_term;
  inout->correction = corrected_next_term - (new_sum - inout->sum);
  inout->sum = new_sum;
}
```

The next variant uses MPI_Allreduce to combine Knuth summation to compute the local sum, with a Knuth sum to perform a pairwise summation of the local sums across the processors. This implementation of Knuth's sum uses the custom MPI_Op function, where u and v are passed as an MPI_Datatype that stores either the result of an intermediate reduction or the local Kahan sum on a processor along with the Kahan correction term. The function computes $u \oplus \tilde{v}$ where

$$\begin{aligned} u &= u[0] \\ \tilde{v} &= v[0] \oplus (v[1] \oplus u[1]) \end{aligned}$$

along with the Knuth correction term and stores the sum and correction term back into u. The implementation is shown in Listing 7.

Listing 7: Knuth MPI Operation

```
double parallel_mpiop_knuth_sum(double **var)
{
  int commutative = 1;
  double u, v, upt, up, vpp;
```

```
  struct esum_type local, global;
  MPI_Datatype MPI_TWO_DOUBLES;
  MPI_Op KAHAN_SUM;

  local.sum=0.0;
  local.correction=0.0;
  for(int j=0; j<jsize; j++){
    for(int i=0; i<isize; i++){
      u = local.sum;
      v = var[j][i] + local.correction;
      upt = u + v;
      up = upt - v;
      vpp = upt - up;
      local.sum = upt;
      local.correction = (u - up) + (v - vpp);
    }
  }

  MPI_Type_contiguous(2, MPI_DOUBLE, &MPI_TWO_DOUBLES);
  MPI_Type_commit(&MPI_TWO_DOUBLES);
  MPI_Op_create((MPI_User_function *)kahan_sum, commutative, &
      KAHAN_SUM);

  MPI_Allreduce(&local, &global, 1, MPI_TWO_DOUBLES, KAHAN_SUM,
      MPI_COMM_WORLD);

  MPI_Op_free(&KAHAN_SUM);
  MPI_Type_free(&MPI_TWO_DOUBLES);

  return(global.sum);
}

void knuth_sum(struct esum_type *v_in, struct esum_type *u_inout,
    int *len, MPI_Datatype *MPI_TWO_DOUBLES)
{
  double u, v, upt, up, vpp;
  u = u_inout->sum;
  v = v_in->sum + (v_in->correction + u_inout->correction);
  upt = u + v;
  up = upt - v;
  vpp = upt - up;
  u_inout->sum = upt;
  u_inout->correction = (u - up) + (v - vpp);
}
```

In addition to the double-precision implementations, a long-double version of each of the sums was also tested. To fully understand the results of the long-double, it is necessary to take a closer look at what the long-double type is in C. The C standard only specifies that the long-double is greater than or equal to the precision of the double type. For the GCC compiler on the x86 architecture where these tests were run, the long-double does its mathematical operations with an 80-bit number and stores it as a 128-bit storage size to keep the data aligned. The x86 processor does its normal arithmetic operations on an 80-bit value and has 80-bit numeric registers. For the double type, these 80-bit values are truncated to 64-bits for storage. On other processors and compilers, the

long-double can be the same as an ordinary double or a 128-bit number. To get full 128-bit numeric operations, the GCC compiler added a __float128 type with the 4.3 version of the compiler for the x86 architecture.

3. Results

To examine the effects of truncation error, three problems were run with the Sapient 2D[?] compressible fluid dynamics code. Sapient solves the standard 2D Eulerian set of equations.

$$\begin{aligned}
\rho_t + (\rho u)_x + (\rho v)_y &= 0 && \text{(conservation of mass)} \\
(\rho u)_t + (\rho u^2 + p)_x + (\rho uv)_y &= 0 && \text{(conservation of x-momentum)} \\
(\rho v)_t + (\rho uv)_x + (\rho v^2 + p)_y &= 0 && \text{(conservation of y-momentum)} \\
E_t + [u(E+p)]_x + [v(E+p)]_y &= 0 && \text{(conservation of energy)} \\
p = (\gamma - 1)[E - .5(\rho u^2 + \rho v^2)] & && \text{(ideal gas equation of state)}
\end{aligned}$$

where ρ = density u = x velocity, v = y velocity, E = total energy, p = pressure, and γ = gamma or ratio of specific heats.

The numerical method is a centered difference Total Variation Diminishing (TVD) scheme. The scheme is a variant of the centered TVD method developed by Davis[?] and Yee [?]. It uses a Lax-Wendroff two-step formulation with a flux-correction term. In the following explanation, U is the conserved state variable at the center of the cell . This state variable, $U = (\rho, \rho u, \rho v, E)$ in the first term in the equations above. $F$is the flux quantity that crosses the boundary of the cell and is subtracted from one cell and added to the other. The remaining terms after the first term are the flux terms in the equations above with one term for the flux in the x-direction and the next term for the flux in the y-direction. The first step estimates the values a half-step advanced in time and space on each face, using loops on the faces.

$$\begin{aligned}
U^{n+\frac{1}{2}}_{i+\frac{1}{2},j} &= (U^n_{i+1,j} + U^n_{i,j})/2 + \frac{\triangle t}{2\triangle x}\left(F^n_{i+1,j} - F^n_{i,j}\right) \\
U^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} &= (U^n_{i,j+1} + U^n_{i,j})/2 + \frac{\triangle t}{2\triangle y}\left(F^n_{i,j+1} - F^n_{i,j}\right)
\end{aligned}$$

The second step uses the estimated values from step 1 to compute the values at the next time step in a dimensionally unsplit loop.

$$U^{n+1}_{i,j} = U^n_{i,j} - \frac{\triangle t}{\triangle x}(F^{n+\frac{1}{2}}_{i+\frac{1}{2},j} - F^{n+\frac{1}{2}}_{i-\frac{1}{2},j}) - \frac{\triangle t}{\triangle y}(F^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} - F^{n+\frac{1}{2}}_{i,j-\frac{1}{2}})$$

Step two also applies a correction term to limit the flux for a second order

solution using the following with the least compressive minmod limiter.

$$\begin{aligned} U_{i,j}^{n+1} &+= 0.5\nu(1-\nu)[1-\phi(r^+, r^-)] \\ \phi(r^+, r^-) &= [max(min(1, r^-, r^+), 0) \\ r_k^+ &= \frac{(\triangle U_{k-\frac{1}{2}}^n, \triangle U_{k+\frac{1}{2}}^n)}{(\triangle U_{k+\frac{1}{2}}^n, \triangle U_{k+\frac{1}{2}}^n)} \\ r_k^- &= \frac{(\triangle U_{k-\frac{1}{2}}^n, \triangle U_{k+\frac{1}{2}}^n)}{(\triangle U_{k-\frac{1}{2}}^n, \triangle U_{k-\frac{1}{2}}^n)} \end{aligned}$$

All problems were run on a 1280x1280 mesh for 1000 cycles with a conservation check every 20 cycles. For this study the total mass and total energy are calculated every cycle. The maximum relative conservation error in multiples of machine precision is used for the comparison as shown in Equation 1.

$$\frac{(\frac{CurrentTotal}{OriginalTotal} - 1.0)}{DBL\_EPSILON} \tag{1}$$

$DBL\_EPSILON = 2.220446049e-16$ is from the C float.h header file, but it appears to be twice the actual machine precision in some cases, so half of this epsilon is used or $DBL\_EPSILON = 1.110223025e{-}16$. This machine epsilon is consistent with the IEEE 754-2008 standard.

All calculations were done on the first generation Roadrunner machine at Los Alamos National Laboratory. This system is composed of four dual-core AMD Opteron chips on an Infiniband interconnect running OpenMPI 1.3.2 and the code compiled with gcc version 3.4.5. The 4.3.3 version of gcc was also tried, but the optimizer caused the time measurements to be less coherent.

Total conserved values are often used to determine the correctness of the software and small differences across processors or even from cycle to cycle can make it difficult to determine whether the simulation is correct and can mask small errors such as the implementation of a boundary condition or the use of old ghost cell values that should have been updated. For implicit methods using solvers such as conjugate gradient solvers, the summation error can trigger different numbers of solver iterations and cause further difficulties comparing runs with different numbers of processors and even make parallel performance scaling difficult to measure.

The Leblanc problem, sometimes called "the shock-tube from hell", has a range of several orders of magnitude in the state values. This should present some opportunity for truncation error while summing up the state variables over the mesh. The precise definition used in this case is two separate regions as shown in Equation 2.

$$[\rho, u_x, u_y, E] = \begin{cases} [1.0, 0.0, 0.0, 0.1] & 0 \leqq x < 1/3 \\ [1.0e-03, 0.0, 0.0, 1.0e-10] & 1/3 \leqq x \leqq 1 \end{cases} \tag{2}$$

The data decomposition is along the y-axis which is the non-varying direction, yielding similar magnitude local sums on each processor. The standard sum-

mations show large errors. Using long-doubles decreases the error, but there are still small errors remaining. This is because the long-double is only 80 bits instead of the 128 bits obtained by the sum and correction terms in the Kahan and Knuth implementations. When a full double-double 128-bit sum was tested, it dropped the error to values closer to the most accurate methods. Adding the enhanced precision sums to the calculation is enough to drop the error to near machine precision. The magnitudes across the processors are similar, so the variants of the sums shown in the columns are for the serial part of the sum. The Knuth MPI_Op Long method is also shown as a reference of a more precise summation and essentially represents the error in the numerical simulation itself. It should be noted that the methods based on double precision are printed out in double precision and the long-double based methods are printed out in long-double precision which is why they can be represented in fractional amounts whereas the doubles can only be shown to the closest integer. These results are shown in Tables 1 and 2.

Table 1: Max Total Mass Error in Machine Epsilons for Horizontal Leblanc Problem

Proc	Max Mass Error				
	Normal	Long-Double	Kahan	Knuth	Knuth MPI_Op Long
1	-76643	-9.17	0	0	0.646
2	-62566	-6.29	0	0	0.646
4	24750	3.86	0	0	0.646
8	19044	-7.61	0	0	0.646
16	7740	-1.22	0	0	0.646
32	-2342	1.22	0	0	0.646
64	-2439	-0.584	0	0	0.646
128	-675	0.688	0	0	0.646
256	228	0.803	0	0	0.646

Table 2: Max Total Energy Error in Machine Epsilons for Horizontal Leblanc Problem

Proc	Max Energy Error				
	Normal	Long-Double	Kahan	Knuth	Knuth MPI_Op Long
1	-55438	-30.71	-2	-2	0.248
2	15810	-29.64	-2	-2	0.248
4	15124	2.20	-2	-2	0.248
8	1824	3.05	-2	-2	0.248
16	4750	-4.06	-2	-2	0.248
32	1300	-2.00	-2	-2	0.248
64	-1285	0.216	-2	-2	0.248
128	-462	0.229	-2	-2	0.248
256	-115	0.309	-2	-2	0.248

The second test problem used is identical to the Leblanc problem described

above, but swapping axes so the shock travels across processor boundaries. Each processor will have local sums of different magnitudes. This is intended to highlight the differences in the cross-processor sums. The maximum errors for each of the methods as shown in Tables 3 and 4 are near machine precision for all of the variants.

Table 3: Max Total Mass Error in Machine Epsilons for Vertical Leblanc Problem

Proc	Normal	Kahan	Knuth	Kahan MPI_Op	Knuth MPI_Op
1	145008	0	0	0	0
2	65482	2	2	2	2
4	-57653	2	2	2	-1
8	8862	2	2	2	2
16	8666	2	2	2	2
32	4768	2	2	2	2
64	1722	2	2	2	2
128	-1357	2	2	-3	2
256	366	2	2	2	2

Table 4: Max Total Energy Error in Machine Epsilons for Vertical Leblanc Problem

Proc	Normal	Kahan	Knuth	Kahan MPIop	Knuth MPIop
1	-129805	-2	-2	-2	-2
2	-50496	2	2	2	2
4	81030	2	2	2	2
8	-10811	2	2	-2	-2
16	-10297	-2	-2	2	2
32	2402	2	2	-2	-2
64	-1949	-2	-2	-2	-2
128	702	2	2	-2	-2
256	-398	2	2	2	2

These methods can be converted to long-doubles by using the C long data type and the MPI_LONG_DOUBLE data type on systems where the C compiler and MPI library support these types. Gcc and OpenMPI do support these, making it relatively easy to convert the sum routines to long-doubles. All input variables were cast to long-doubles and all local variables were declared long-double.

The maximum effect of the cross-processor sums should occur for the vertical Leblanc problem on 256 processors. Table 5 gives an in-depth look at the max relative error $\frac{CurrSum-OrigSum}{OrigSum}$ for all of the cross-processor variants.

If we take the Knuth MPI_Op long value as the correct sum, then the two double MPI_Op numbers for mass show a small improvement over the simple Kahan and Knuth sums. The long-double looks acceptable, but from Tables 1 and 2, we see that the long-double gives a much larger error on one processor

Table 5: Relative Error

yLeblanc 256 proc	Mass	Energy
Normal Double	4.061632959e-14	-4.420042831e-14
Kahan simple	3.329207344e-16	2.084925864e-16
Knuth simple	3.329207344e-16	2.084925864e-16
Kahan MPI_Op	1.664603672e-16	2.084925864e-16
Knuth MPI_Op	1.664603672e-16	2.084925864e-16
Normal Long Double	4.389091713e-17	4.530234421e-17
Kahan Simple Long	7.176977745e-17	2.758861861e-17
Knuth Simple Long	7.176977745e-17	2.758861861e-17
Kahan MPI_Op Long	7.168849798e-17	2.748681559e-17
Knuth MPI_Op Long	7.168849798e-17	2.748681559e-17

and then declines to significantly smaller values by 16 to 32 processors. Since our goal is to get a consistent value independent of processor count, the long-double is not attractive. This is due to the long-double being only an 80-bit operation on the tested platform. Also, the long-double implementation will have different precision depending on the C long-double type on that particular platform causing larger differences between platforms and compilers. The MPI_Op still shows a very small change from the simple Kahan and Knuth algorithms in the long versions. Table 5shows no difference between Kahan and Knuth sums in any of the pairs. Some difference between the two is observed on individual cycle values, but it is not enough to show up in the maximum relative error shown here.

The third test problem, referred to as the Shock-in-a-Box problem, is truly a 2D test problem, something lacking from the other two test problems used. This problem is intended to be representative of the general simulation case where shocks are traveling in all directions. The problem definition is as given in Equation 3.

$$[\rho, u_x, u_y, E] = \begin{cases} [5.0, 0.0, 0.0, 50.0] & \text{8-cell by 8-cell center region} \\ [1.0, 0.0, 0.0, 2.5] & \text{surrounding region} \end{cases} \tag{3}$$

This produces a circular shock that travels both within and across processor boundaries. The results for this problem are shown in Tables 6 and 7. The errors are again near machine precision for all of the enhanced precision sum techniques.

Table 6: Max Total Mass Error in Machine Epsilons for the Shock-in-a-Box Problem

Proc	Normal	Long-Double	Simple		MPI op	
			Kahan	Knuth	Kahan	Knuth
1	-68009	63.017	0	0	0	0
2	-51316	33.374	-2	-2	-2	-2
4	-32740	-3.325	2	2	2	2
8	2586	-3.004	2	2	-2	-2
16	2360	-2.500	2	2	-2	-2
32	1968	-1.621	2	2	-2	-2
64	1078	0.134	2	2	-4	-4
128	-726	-1.031	2	2	-2	-2
256	192	-0.998	2	2	-2	-2

Table 7: Max Total Energy Error in Machine Epsilons for the Shock-in-a-Box Problem

Proc	Normal	Long-Double	Simple		MPI_Op	
			Kahan	Knuth	Kahan	Knuth
1	-109551	-39.019	-2	-2	-2	-2
2	-14492	-25.695	-2	-2	0	0
4	23502	-3.837	-2	-2	-2	-2
8	2488	6.148	2	2	-2	-2
16	-6060	0.652	2	2	-2	-2
32	-635	-1.737	2	2	-2	-2
64	1524	-0.581	2	2	-2	-2
128	166	-0.391	2	2	-2	-2
256	-385	-0.407	2	2	-2	-2

The variations of the data across the computational mesh is more representative of the random variation in a real problem. The normal summation still shows large errors that reduce with more processors. The magnitude of the error is much less for the long-double sums, but it is larger than for the enhanced summation methods. It also declines in magnitude with more processors. All of the enhanced summations show results near machine precision and nearly independent of the number of processors.

A better understanding of the errors for the other methods can be obtained from a graph of the error versus the iterations for two processors as shown in Figure 3. The normal double sum is not shown because even with a log scale axis, the error is too large in comparison to the other methods. The long-double is the largest of the errors shown. The double precision Kahan and Knuth MPI_Op are identical but an order of magnitude less error than the long-double due to the 128-bit data size of their sums. The simple Kahan and Knuth sums show a lot of chatter at the jumps in error and make the plot confusing so they have been left off. The lowest error curve is all of the long-double enhanced summations which nearly overlay each other. There

are some slight differences between the simple and MPI_Op methods, but they don't even show up at this scale. It can be seen from this view of the methods that there are three possible steps to reducing the summation error. The first uses an 80-bit data sum, the second an 128-bit sum in software using the Kahan or Knuth summation preferrably with a custom MPI_Op, and a third using a 160-bit sum in software using a combination of Kahan or Knuth sums and the extended precision of the hardware through the long-double data type.

Timings were taken for all of the methods on 64 processors. The finite difference algorithm takes about 51.50 seconds to run on that many processors. The standard sum algorithm for two variables every cycle takes about .347 seconds. Shown in Table 8 are the timing multipliers for each method. Based on these numbers, the enhanced precision sums will increase the run time by about 1% and the long-double versions by about 1.5% from the standard sum. A full double-double 128-bit timing took 60 times as long as the normal double-precision sum.

Figure 3: Comparison of Methods (2 proc)

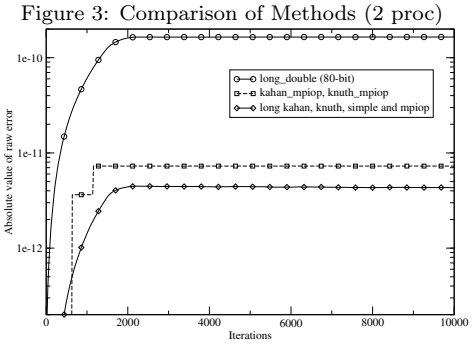


Table 8: Timings

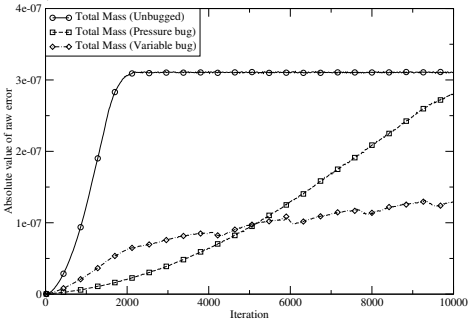
64 proc	Relative Time	
	Double	Long-Double
Normal	1.00	1.51
Simple Kahan	1.70	2.86
Simple Knuth	2.22	3.04
MPIop Kahan	1.74	2.90
MPIop Knuth	2.24	3.27

4. Motivating Example

To show how an error could be masked by normal summation, errors were deliberately inserted into Sapient. The problem used was the Shock-in-a-Box problem described earlier. The first bug, referred to on the graph as 'Pressure bug', was created by removing a pressure recalculation. The second bug, referred to as 'Variable bug' was generated by assigning values to the incorrect momentum variable. These errors are similar to actual errors made by one of the authors in programming this type of numerical model. The pressure bug causes a small boundary condition error resulting in a small effect on the total mass and energy. This boundary condition error is similar to the missing

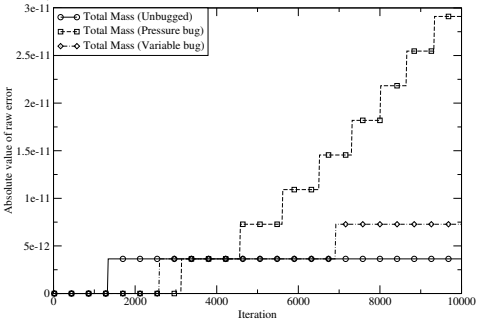
ghost zone updates from adjacent processors that is very common in parallel programming. The variable bug will have little or no effect on the total mass and energy. These two problems will be a stringent test of the ability of the enhanced-precision sums to expose errors of this magnitude. In Figure 4, a normal summation was used for the unbugged and two bugged runs. The error caused by the two bugs actually reduces the total error, perhaps because it is opposite in sign or effect, and had not become larger than the unbugged error by the end of the run. This masked the bug and would cause it to go undetetected. In Figure 5, a Kahan summation was used for the same three runs. This allowed the conservation error caused by the pressure bug to become larger than the unbugged conservation error. This demonstrates how without precision sums, small programming errors can easily go undetected[**?**]. The variable bug does not grow significantly larger than the unbugged case which reflects the little or no error in the total mass and energy resulting from this bug. This example of the masking effect of the global sum is for only one processor with the sum difference caused by the cycle-to-cycle sum consistency errors. The consistency error between one, two and more processors is greater (more variation in the sum order) and will mask more errors during parallel development.

Figure 4: Conservation Errors: Normal Sum (1 proc)



Many numerical algorithms take the result of the global sum and feed it back into the calculation. These types of calculations include particle motion, n-body calculations, implicit solvers, and adaptive mesh refinement to name just a few. When the global sum is used in this way, it greatly increases the consistency error between runs and becomes a far greater problem than the simple example here.

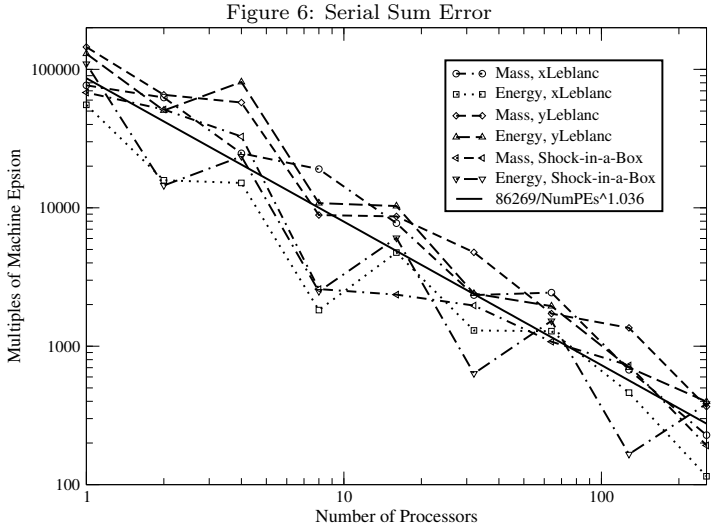
Figure 5: Conservation Errors: Kahan Sum (1 proc)



5. Discussion

To obtain more numerical consistency in a parallel program, it is important to understand the source of the consistency error. It is only the global sum operation that generally changes order in a parallel program. A local sum will mostly retain the same order and

Figure 6: Serial Sum Error



error, thus remaining consistent. There may be exceptions at ghost boundaries where the values of the adjacent processor are cached, but this is a small effect. The standard global sum error actually decreases as the problem is run on more processors. Plotting the serial sum error from the mass and energy totals for the three test problems and calculating a regression fit gives a function $Error = 86269/NumPEs^{1.036}$ as shown in Figure 6. This yields an estimated rule-of-thumb error reduction function of $Error = SerialError/NumPEs$ with multiple processors. This is from the same effect utilized in the pair-wise summation where small single values are not added into a large accumulated total. Still, it creates a problem because the answer is different than the one processor result. And the serial run is generally accepted as the "correct" answer, even though it has the largest numerical error, because of all the potential programming errors that can be introduced in parallelizing a program.

The local sum in the first Leblanc test problem will add in the largest values first and the smallest values last, maximizing the truncation error. It tests the ability of the Kahan summation and the Knuth sum to reduce the summation differences on processor and is insensitive to the particular technique used to sum across processor, because the magnitudes of the data across processors are similar. Results show that any of the Kahan or Knuth enhanced sums gives near machine precision results.

The second Leblanc test problem changes the data decomposition to get larger data magnitudes across processors. The variations among the parallel enhanced summations show a slight improvement of the MPI op based methods over the simple Kahan and Knuth summations. The improvement among the enhanced precision sum variants is small because the consistency errors are so small to start with. It should also be noted that the MPI reduction algorithms will naturally tend to a pair-wise type of summation to optimize the computation

and communication costs. With further inspection of the OpenMPI source code, we find that MPI_Allreduce uses recursive doubling for commutative reduction operations, which is effectively a pair-wise summation.

The Shock-in-a-Box problem shows that any of the enhanced precision sums gives near machine precision results. With a more random pattern of data, the consistency errors are still present in the standard and long-double sums. So the enhanced precision sums should improve the consistency results for typical problems as well as the more extreme cases in the first two test problems.

Real world implementations do intrude on this study. Some MPI algorithms, such as OpenMPI 1.3, will do a shared memory optimization on the 8-CPU node that may not be done in a pair-wise manner. It is important to specify the commutative flag in the custom MPI_Op to obtain the pair-wise operation. Also, on some CPUs, such as the x86, the serial operation may be done in an extended-precision 80-bit numeric register. Some of these coding variants may pull the data out of the numeric registers, giving unexplained reductions in precision in a method that theoretically should be better.

6. Conclusions

The results from this study show that Kahan summations should be a standard practice for global sums in parallel processing. The added computational cost is more than offset by the value of the more consistent results. With the communication and synchronization costs of the parallel global sum, the relative run-time cost is lower as the processor count is increased. With the global sums in this simulation being done for two variables every cycle, the additional computational cost is only one or two percent at most. If a dozen or more sums are done every cycle, for example in an iterative solver, the cost may be closer to 10%, so the cost must be evaluated for the particular circumstances in which it is being used.

From the addition of the enhanced precision summation techniques to the 2D Sapient hydrocode, it was immediately apparent that the mass and energy conservation of the numerical method was much better than previously thought based on the original mass and energy conservation checks. The mass and energy change reported was mostly due to the precision loss of the global sum and not in the state variables in the mesh. This observation that mass and energy are conserved to machine precision is important in that it is a necessary condition for showing that we have implemented the method correctly. Since the original equations were posed in conservation form this is a critical cross-check.

The greatest improvement in consistency is observed with the addition of the simple Kahan or MPI_Op sum over the normal summation method. The additional cost for either method is around half-a-percent in total runtime which is well worth the gain in consistency. In cases where the magnitudes of the sums and correction terms are vastly different, the MPI_Op should be better, so it is recommended to use the MPI_Op variant in Listing 6. In theory, the Knuth summation should be better than the Kahan sum and occasional differences with the Kahan summation are seen in the individual cycle output. But it is

not clear from this study that the difference is worth the additional cost of about a one half a percent over the Kahan summations. Perhaps other test problems may show greater improvement in results and thus demonstrate the value of the Knuth summation. The cost of the long-double is not as high as was originally expected at only one half a percent total runtime over their respective double variants but this is because the long-double is only 80 bits long and is done in hardware. The full 128-bit double-double is 60 times as expensive due to being implemented in software. Again, on these test problems, it is probably not worth the additional computational cost for even the long-double and more so for the 128-bit double-double. But there may also be test problems where this additional cost may prove worthwhile. Easy access to these types is not available in some languages such as FORTRAN, so there the solutions are limited to the Kahan and Knuth summation methods.

This study also suggests the need for a new FORTRAN intrinsic, *epsum*, which stands for extended precision sum. It should return a sum, or a sum and a correction. It also suggests a new MPI_Op, MPI_EPSUM, though more studies may be needed to provide a stronger justification and solidify the proper implementation.

What is remarkable at the conclusion of this study is that no limit can be seen for the numerical accuracy of the mass and energy conservation of the numerical method. The numerical algorithm essentially maintains conservation down to the last bit of the floating point representation used. Errors in the coding will be much easier to detect under these conditions. And perfect numerical consistency is potentially achievable and not so far out of reach as originally thought. Indeed these methods may already be at the minimum error precision and the one or two machine epsilon is partly due to the error measurement calculation and the underlying numerical method.

Acknowledgments

The persistent, tenacious pursuit of even the smallest differences in parallel implementations by our fellow members of the Eulerian Applications Group team members in XCP division at Los Alamos National Laboratory has been a key motivator in the development of these methods.

AppendixA. Appendices