Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications

Yun He and Chris H.Q. Ding
NERSC Division, Lawrence Berkeley National Laboratory
University of California, Berkeley, CA 94720, USA
YHe@lbl.gov, CHQDing@lbl.gov

Abstract

Numerical reproducibility and stability of large scale scientific simulations, especially climate modeling, on distributed memory parallel computers are becoming critical issues. In particular, global summation of distributed arrays is most susceptible to rounding errors, and their propagation and accumulation cause uncertainty in final simulation results. We analyzed several accurate summation methods and found that two methods are particularly effective to improve (ensure) reproducibility and stability: Kahan's self-compensated summation and Bailey's double-double precision summation. We provide an MPI operator MPI_SUMDD to work with MPI collective operations to ensure a scalable implementation on large number of processors. The final methods are particularly simple to adopt in practical codes: not only global summations, but also vector-vector dot products and matrix-vector or matrix-matrix operations.

Keywords: Reproducibility, Climate Models, Double-Double Precision Arithmetic, Self-Compensated Summation, Distributed Memory Architecture.

1 Introduction and Motivation

One of the pressing issues for large scale numerical simulations on high performance distributed memory computers is numerical reproducibility and stability. Due to finite precisions in computer arithmetics, different ordering of computations will lead to slightly different results, the so called rounding errors. As simulation systems become larger, more data variables are involved. As simulation time becomes longer, teraflops of calculations are involved.

All these indicate that accumulated rounding errors could be substantial. One manifestation of this is that final computational results on different computers, and on the same computer but with different number of processors will differ, even though calculations are carried out using double precision (64-bit).

It is important to distinguish between the high precision required in internal consistent numerical calculations and the expected accuracy of the final results. In climate model simulations, for example, the initial conditions and boundary forcings can seldom be measured more accurately than a few percent. Thus in most situations, we only require 2 decimal digits accuracy in final results. But this does not imply that 2 decimal digits accuracy arithmetic (or 6-7 bits mantissa plus exponents) can be employed during the internal intermediate calculations. In fact, double precision arithmetic is usually required.

In this paper, we are concerned with the numerical reproducibility and stability of the final computational results, and how the improvements in the internal computations could affect the final outcome. Numerical reproducibility and stability are particularly important in long term (years to decades and century long) simulations of global climate models. It is known that there are multiple stable regions in phase space (see Figure 1) that the climate system could be attracted to. However, due to the inherent chaotic nature of the numerical algorithms involved, it is feared that slight changes during calculations could bring the system from one regime to another. One common but expensive solution is to run the same model simulation with identical initial conditions with small perturbation at the level far below the observational accuracy for several times, and take the ensemble average.

In a scenario climate experiment, first a controlled run of model simulation with standard parameters is produced. Then a sensitivity run of model simulation with slightly changed parameters, say CO_2 concentration, is produced. The difference between these two runs is computed and the particular effect, say the average surface temperature, is discerned. Without numerical stability, the difference between the controlled run and the sensitivity run could be due to the difference in different regimes, not due to the fine difference in input parameters which is the focus of the scenario study.

On distributed memory parallel computer (MPP) platforms, an additional issue is the reproducibility and stability with regard to number of processors utilized in the simulations. One simulation run on 32 processors should produce "same" or "very close" results as the same simulation run on 64 processors. Sometimes, different number of processors could be involved during different periods of a long term (10 simulated years or more) simulation due to computer resources availability, further demanding reproducibility of the simulation.

Numerical reproducibility of the final results are determined by the internal computations during intermediate steps. Obviously, there are a large number of areas involved in the numerical rounding error propagation. In climate models, one major part is the dynamics,

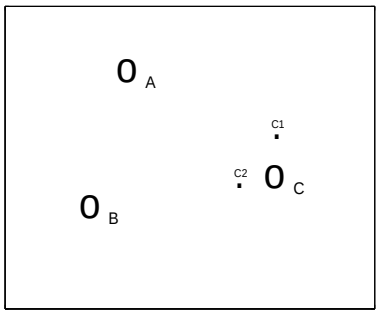

Figure 1: An illustrative diagram of several stable regimes centered around A, B, and C in the multi-dimensional phase space at a simulated time far beyond initial period (spin-up). By reproducibility and stability, we mean that results of the same simulation running on different computers or on same computer but with different number of processors remain close inside one regime, say C1 and C2. (The "exact" or "absolute" reproducibility, i.e., identical numerical results on different computers, or on different number of processors, or even on different compiler versions or optimization levels, is impossible in our view, and is not our goal.) In contrast, non-reproducibility indicates the simulation results change from one regime to another regime on different computers.

i.e., finite difference solution to the primitive equation. On MPP, stencil updates proceed almost identically as in sequential implementation except the update order changes slightly. If the first order time stepping scheme is used, the order remains unchanged.

In barotropic component of an ocean model [6, 19, 20, 24], an elliptic equation is solved using a global linear equation via a conjugate gradient (CG) method. In atmospheric data assimilation [5, 7], a correlation equation is also solved via a conjugate gradient method. The key parameters α, β calculated through the global summation (dot product between global vectors) appear to be sensitive to the different summation orders on different number of processors. A slight change on the least significant bits will accumulate quickly into the significant bits in α, β over several iterations, leading to a different CG trajectory in the multi-dimensional space, and a slightly different solution (see, e.g., [11, 22]). Although the self-correcting nature of the conjugate gradient method ensures that these different solutions on different number of processors are equally correct with regard to solving the linear equation, this inherent difference generating nature could be significant. It is feared that the multiplicative effects on the small differences in each time step will eventually lead the system into another stable regime (say from C to A in Figure 1), instead of merely wandering around the stable regime C (shifting from C1 to C2).

Another example is the spectral transforms used in many atmospheric models [8, 13], where global summations are very sensitive to the summation order. (In this case, there is

no self-correcting mechanism as in the CG method mentioned above.) There are many other parts in climate models where summation of global arrays is used. In general, experience in distributed memory parallel computing indicates global summation appears to be the most sensitive with regard to rounding error, a fact also known in numerical analysis [15, 21].

1.1 Current Approach

The commonly used method to resolve the global summation related reproducibility problem is to use serialized implementation, e.g., in some ocean models [6, 12]. In this scheme, elements of distributed array are sent to one designated processor, often processor 0, and are summed up in a fixed order on the designated processor. The result is then broadcasted back to all the relevant processors.

This scheme guarantees the reproducibility but at the extra costs of communication. The designated processor has to receive $P-1$ messages, each from other $P-1$ processors, in a time linear to P. Clearly this scheme does not scale well to large number of processors. In addition, this serialization causes unwanted complexity in the implementation for a simple summation.

For large arrays, it is more reasonable that each processor adds the subsection of the array in a fixed order and only sends its local summation to the designated processor. Although the local orders are the same for each processor, the global order of summation will be different when different number of processors are used, thus the final global summation result will not be reproducible.

Finally, even though the global summation result is reproducible, the result could often be inaccurate since double precision arithmetic is usually not adequate for the data with large cancellation. The accumulation of these rounding errors could be severe for long time simulations during repeated arithmetic operations. This affects numerical stability for certain applications.

1.2 New Approach

Here we look into a new approach that both guarantees (or substantially improves) reproducibility and achieves scalability and efficiency on distributed platforms.

The basic idea is that non-reproducibility is directly caused by the rounding errors involved in the intermediate arithmetic operations. Therefore instead of fixing the summation order, if we could reduce rounding errors very significantly (sometimes eliminate) with more accurate arithmetic, we could achieve reproducibility also. On a computer with infinitely accurate arithmetics, there is no rounding error. A simple and trivial solution is to use higher

precision, say 128-bit precision arithmetics in the relevant data arrays. However, most computer platforms we know do not support 128-bit precision arithmetic. The only exception is Cray PVP (SV1, C90) line of computers, where the 128-bit precision is supported and implemented in software, resulting in huge (factor of 10 or more) performance degradation. The real challenge here is to find a simple and practical method that can effectively improve the numerical reproducibility and stability.

In this paper, we examined several accurate summation methods, in particular, the fixed-point and the multi-precision arithmetics, the self-compensated summation (SCS) and doubly-compensated summation (DCS) methods, and the double-double precision arithmetics. We first examined these methods in sequential computer environment, i.e., on a single processor. We found two effective methods: the self-compensated summation and double-double precision summation. Then these promsing methods are examined in the parallel environment. We provide necessary functions to work together with MPI communication library to utilize these accurate methods in parallel environment in a scalable way. We also note that the local summation results from self-compensated summation and double-double precision summation can be summed accross processors in a simple and unified way with the MPI_SUMDD operator we provided. All the major problems associated with the serialized implementation, non-scalability and implementation complexity, are therefore resolved in our new approach.

Furthermore, the final results in the new approach will be more accurate due to accurate arithmetics, compared to the current serialized approach. This improves numerical *stability* and could be of critical importance for some applications.

In summary, our main concern in this paper is to find simple and practical methods to improve the numerical reproducibility between runs on different number of processors on the same distributed memory computer, and also between runs on different computers. We approach this task by using more accurate (than standard double precision) methods in some of the key steps in parallel applications. Although the overall final computation results will be undoubtedly more accurate, that is an added benefit, but is not our primary goal.

2 Sea Surface Height Data

Sea surface height (SSH) is the first serious difficulty we encountered in an ocean circulation model development [6]. The variable stores the average sea surface height from the model simulation, which can later be compared with satellite data. Repeating the same simulation on different number of processors leads to different results. This difficulty, along with issues mentioned above, motivated this work. This simple problem serves as a good example for the main ideas of global summation in parallel environment. All methods are tested against

this SSH data in this paper on Cray T3E at NERSC (We also used synthetic data in this work, and the results are very similar).

The SSH variable is a two-dimensional sea surface volume (integrated sea surface area times sea surface height) distributed among multiple processors. At each time step, the global summation of the sea surface volume of each model grid is needed in order to calculate the average sea surface height. The absolute value of the data itself is very large (in the order of 10^{10} to 10^{15}), with different signs, while the result of the global summation is only of order of 1. Running the model in double precision with different number of processors generate very different global summations, ranging from -100 to 100, making the simulation results totally meaningless.

We saved the SSH data from after one-day simulation as our test data. The 2D array is dimensioned as `ssh(120,64)`, with a total of 7680 double precision (64-bit) numbers. Other sizes and simulation time will not affect the conclusions reported here (Both the codes used here and the SSH data can be downloaded from our website [14]).

For the 2D SSH array, the most natural way of global summation is to use the following simple codes:

```
do j = 1, 64      ! index for latitude
do i = 1, 120     ! index for longitude
     sum = sum + ssh(i,j)                        Code (1)
end do
end do
```

We call this order the "longitude first" order, the result of summation is 34.4 (see Table 1). If we exchange the `do i` and `do j` lines, so that elements with different `j` index (while `i` index remain fixed) are summed up first (latitude first), we get 0.67, a totally different result! Sometimes in practice, we need to sum up the array in reverse order, such as `do i = 120, 1, -1`, denoted as "reverse longitude first", the result would change again, to 32.3.

These results are listed in Table 1. Clearly the results are summation order dependent on the sequential computer. In fact, we do not know what is the exact correct result, until later with other methods.

On a distributed memory computer platform, the 2D SSH array is decomposed into many subdomains among multiple processors. On different number of processors, the order of summation is not guaranteed, and the results are not reproducible!

The origin of the rounding error is due to finite representation of a floating point number. A simple example explains the idea best. In double precision, the following Fortran statement $S = 1.25 \times 10^{20} + 555.55 - 1.25 \times 10^{20}$ will get $S = 0.0$, instead of $S = 555.55$. The reason is that when compared to 1.25×10^{20}, 555.55 is negligibly small, or non-representable. In

Table 1: Results of the summation in sifferent natural orders with different methods in double precision

Order	Result
Longitude First	34.414768218994141
Reverse Longitude First	32.302734375
Latitude First	0.67326545715332031
Reverse Latitude First	0.734375
Longitude First SCS	0.3823695182800293
Longitude First DCS	0.3882288932800293
Latitude First SCS	0.37443733215332031
Latitude First DCS	0.32560920715332031

hardware, 555.55 is simply right-shifted out of CPU registers when the first addition is performed. So the intermediate result of the first addition is 1.25×10^{20}, which is then cancelled exactly in the subtraction step.

The sea surface height data is one of the worst cases and therefore serves as a good clear test case. For most variables, this dependency on summary ordering is less pronounced. However, this error could propagate to higher digits in the following iterations. Our goal is to find an accurate summation scheme that minimizes this rounding error.

3 Fixed-Point Arithmetic

The first method we investigate is a fixed point summation without loss of precision. It is a simple method and can be easily implemented (codes could be downloaded from our web site [14]). we wrote a code to first convert double precision floating point numbers of a global array into an array of integers, a `db2int()` function. Depending upon the dynamical range (maximum and minimum) and precision required, the integer representation chooses a proper fixed point (a scale factor) and one or a few integers to represent each floating point numbers.

These integers are then summed up using standard integer arithmetic, (and sum across the multiple processors using MPI_REDUCE with MPI_INTEGER data type) and are finally converted back to double precision numbers, rounding off all lower bits which are non-representable in double precision, by using the `int2db()` function.

This method is applied to the 3 number addition example and the correct result S=555.55 is obtained. We applied this method to the SSH data discussed above, and the summation

result is

$$\sum_{i,j} ssh(i,j) = 0.35798583924770355 \tag{1}$$

This result remains the same upon changing the summation orders, convincing us that it is the exact result. (The double-double precision discussed later gives the same result.)

This method requires the users to know the dynamical range before calling the `db2int()` conversion routine. A simple way is to find the maximum and minimum magnitudes of the array. Based on this information, an initialization routine will properly determine the scale factor (the fixed point) and the number of integers (32 bits or 64 bits) required to represent the floating point numbers.

In large simulation codes, however, finding the maximum and minimum in a large array distributed over multiple processors before each array summation is quite inconvenient in many situations. For this reason, we do not recommend this method. We strive to find methods which are simple to adapt in practical large scale coding.

4 Multi-Precision Arithmetic

The above fixed-point arithmetic is a simple example of a class of multi-precision arithmetic software packages which carry out numerical calculations at a pre-specified precision [15]. Brent's BMP [3] is the first complete Fortran package on the multi-precision arithmetics offering a complete set of arithmetic operations as well as the evaluation of some constants and special functions. Bailey's MPFUN [2] is a more sophisticated and more efficiently implemented complete package. He used this package to compute π to 29 million decimal digits accuracy! Several other packages are also available (see [15] for more discussions).

We looked into both BMP and MPFUN multi-precision packages and decided not to investigate them further because of the nontrivial practical coding efforts involved to adopt them in large simulation codes. At the same time we learned of the error-compensated summation methods and the double-double precision arithmetics which are much more easier to adopt. They seem to offer the practical solution we are looking for.

5 Self-Compensated Summation Methods

Kahan [16] in 1965 suggested a simple, but very effective method to deal with this problem. The idea is to estimate the roundoff error and store in an error buffer; this error is then added back in the next addition (For consistency, from now on subtraction will be called as addition, since subtraction = addition of a negative number). The computer implementation is as follows:

```
sum = a + b
error = b + (a - sum)
```

Using this self-compensated summation method in the previous example, the first addition would give 1.25×10^{20} and 555.55 as the `sum` and `error`.

The pair (`sum`, `error`) is returned as a complex number from the function `SCS(a,b)` [See Appendix A]. In the next addition, the `error` is first "compensated" or added back to one of the addends (numbers to be added); the same addition and error estimation are then repeated. Suppose we need to calculate d = a + b + c. The following two function calls give the final results:

```
(sum, error)   = SCS(a, b)                          Code (2)
(sum1, error1) = SCS(sum, c + error)
```

The final result is d = `sum1` with error = `error1`. The results of the error compensated summation is always a (`sum`, `error`) pair.

Priest [23] further improved the error-compensating method by repeated applications of SCS. Note that in the second SCS in Code (2) for addition of `c`, the addition of `error` to `c` could have substantial precision loss. The doubly-compensated summation is to use SCS again on this `c + error` to compensate the error for any possible loss of precision. This further application of SCS could be implemented as three SCS calls; the calculation of `d = a + b + c` is done by the following codes:

```
(sum, error)   = SCS (a, b)
(sum1, error1) = SCS (error, c)
(sum2, error2) = SCS (sum, sum1)                    Code (3)
(sum3, error3) = SCS (sum2, error1 + error2)
```

Here `error` from the first SCS is added back to `c` using SCS, which produces a second level error `error1` (called a second level error because it comes from an addition involving the `error`). The third line in Code (3) performs the same function as the second line in Code (2), which is essentially `sum + c` with another first level `error2` generated. These two errors are both compensated in the fourth line of Code (3). The final result is d = `sum3` with error = `error3`. See Appendix B for the Fortran implementation. This doubly-compensated summation method works the best when the original array is sorted in the decreasing magnitude order beforehand.

This concludes our intuitive discussions on the error-compensated methods. Good theoretical analyses about rounding error bounds of the methods could be found in [10, 15, 18].

We applied both SCS and DCS to the 2D sea surface height data, using the straight summation in Code (2). The results are listed in Table 1, "longitude first SCS", etc. Now all 4 orders give results with agreement on the first significant decimal digit; in fact they are quite close to the correct result in Eq.(1). This indicates very substantial improvements in rounding error reductions during the summations. We also note that DCS does not seem to outperform the much simpler SCS.

5.1 Sensitivity Regarding to Summation Order

To further study the sensitivity regarding to different summation orders and to see further difference in performance between SCS and DCS, we applied a number of different sorting orders on the SSH data. For simplicity, SSH data is treated as a simple 1D array of 7680 double precision numbers.

This 1D array is easily sorted in 6 different ways as explained in Table 2. They are summed up from the left to right. For an array of all positive numbers, the increasing order would be the best, since the smaller numbers are added first, and they would accumulate large enough not to be rounded off when added with big numbers. But for arrays with mixed positive and negative numbers, the decreasing magnitude order would be the best, especially when the absolute value of the summation is much smaller than the addends due to the cancellation of most addends. We also tried several other orders. The summation results are shown in Table 3.

Table 2: Seven different orders of the array tested for summation

Order	Sequential Order	Example
1	no sort	-9.9 5.5 2.2 -3.3 -6.6 0 1.1
2	increasing order	-9.9 -6.6 -3.3 0 1.1 2.2 5.5
3	decreasing order	5.5 2.2 1.7 0 -3.3 -6.6 -9.9
4	increasing magnitude order	0 1.1 2.2 -3.3 5.5 -6.6 -9.9
5	decreasing magnitude order	-9.9 -6.6 5.5 -3.3 2.2 1.1 0
6	positives reverse from order 2	-9.9 -6.6 -3.3 0 5.5 2.2 1.1
7	negatives reverse from order 2	-3.3 -6.6 -9.9 0 1.1 2.2 5.5

From these results we observe that (1) Using ordinary double precision summation, different sorted orders always lead to different results ranging from -73.6 to 34.4 , similar to those listed in Table 1. (2) In all 6 sorted orders (excluding Order 1 which is same as the Longitude First order in Table 1), DCS gives the identical results as SCS does. They are

Table 3: Results of the summation in different sorting orders with different methods in double precision [fixed point arithmetic and double-double precision arithmetics always give the same and correct answer as in Eq.(1)]

Order	Double Precision	SCS	DCS
1	34.414768218994141	0.3823695182800293	0.3882288932800293
2	-70.640625	0.359375	0.359375
3	-73.015625	0.359375	0.359375
4	13.859375	0.359375	0.359375
5	14.318659648299217	0.35798583924770355	0.35798583924770355
6	-36.254243895411491	0.35812444984912872	0.35812444984912872
7	-66.640625	0.359375	0.359375

always between 0.358 and 0.359, very close to the real result of 0.358 (less than 0.4% error). This indicates that sorting helps reduce the rounding error in SCS and DCS, but not much in the double precision arithmetic. (3) Among the 6 different orders, the one with decreasing magnitude order produces the correct answer of Eq.(1). Intuitively in this order, the numbers with similar magnitude but oppositive signs are added together first, and then the result is added to the accumulation without much loss of precision. This order is the recommended order in summation of large arrays [23, 15].

From these observations and results in Table 1, we conclude first that SCS and DCS always give results very close to the correct result (better results when the array is sorted), and show dramatic improvements over the straight double precision summation. Secondly, the differences between DCS and SCS are also very small: when array is sorted, there is no difference (Table 3); when array is unsorted, there is small difference (Table 1), but no indication that DCS does better than SCS. Therefore, from now on, we will concentrate on the simpler SCS.

So far, we have studied compensated methods that require no extra storage space (except 1-3 temporary buffer space) at the cost of extra additions. Although no exact tight error bounds can be given, they improve the accuracy very substantially. Another practical approach is to do truly higher precision arithmetic using existing double precision data representation and CPU arithmetic units.

6 Double-Double Precision Arithmetic

In double-double precision arithmetics, each floating point number is represented by two double-precision numbers. In fact, the bits in the second double precision number are ex-

tended mantissa of the first double precision number. The dynamic range of the double-double precision number remains the same as the double precision number. As an analogy to the self-compensated summation discussed above, one may think that the first double precision number as the summation result and the second double precision number as the estimated rounding error. A suite of Fortran 90 compatible codes are developed by Bailey [1] by using Knuth's [17] trick.

We used this double-double precision codes in the sea surface height data summation. It always gives the same correct result of Eq.(1), irrespective of the 9 different summation orders listed in Tables 1 and 3. This indicates the extra precisions (about 106 bits) in the double-double representation is highly effective in reducing the roundoff errors during the summation.

The double-double arithmetic employs slightly more arithmetic steps. The number of additions increases much more than in the self-compensated method. For a single addition, double-double arithmetic requires 11 double precision additions *vs.* 4 for the SCS. However, on cache-based processor architectures, the increased CPU arithmetic does not slow down calculation as much as the increased memory access required since we use two double precision numbers for one single double-double number. In practice, on wide range of processors, the double-double arithmetic only slows down by a factor of 2 while getting effective double-double precision.

The double-double arithmetics has also increased memory requirements, which is doubled. However, this is not a real problem in today's computers when DRAM is very plentiful. Furthermore, one can implement the basic arithmetic codes in a way that also requires no increase in memory requirement, similar to the self-compensated methods.

7 Summation Across Distributed Memory Processors

Among the above methods we have investigated so far, fixed-point/multi-precision and double-double arithmetics always give the same and correct result. However, as pointed out earlier, the fixed-point arithmetic requires the dynamical range of the data array before the summation which is not practical. The true multi-precision packages require too much effort to adopt regarding the conversion of the data format.

The self-compensated summation emerges as a very simple and effective method as it gives results very close to the exact one in the worst case of SSH data in 9 totally different summation orders (Tables 1 and 3). The doubly-compensated summation does not show significant improvement in accuracy over SCS but more arithmetic operations are involved. Double-double arithmetic is easy to implement. Thus, we will concentrate on only two summation methods in the distributed systems, i.e., the SCS method and double-double

precision arithmetics. DCS method can be adopted in same way as SCS.

In distributed memory environment, each processor will first do the summation on a subsection of the global array covered by the processor. One may use the self-compensated method or the double-double precision arithmetics. The results of this local summation are two double precision numbers: (`sum`, `error`) in the SCS and (`double, double`) in double-double arithmetic. In both cases, the result can be represented as a single complex number.

The issue here is how to sum up the pair of numbers on each processor in a consistent way to achieve high accuracy. We investigated a number of different approaches and found a unified and consistent approach.

7.1 Double-Double Precision Arithmetic

Consider the double-double arithmetic case first. If we simply use MPI to sum up the first double number on all processors together, and the second double numbers on all processors, we can use

```
MPI_REDUCE (local_sum, global_sum, 1, MPI_COMPLEX, MPI_SUM, ......)
```

Here the complex variable `local_sum` contains the local sums and `global_sum` is the result of the global summation. Note we use MPI_COMPLEX as the data type. During this procedure, the first double and second double numbers on different processors are summed up separately, using the usual double precision arithmetic, not the double-double arithmetic. Therefore the final results of first and second double numbers are not consistent and precision is lost during this procedure. Applying this to the SSH data, the final summation result changes on different processors, and is not exactly correct (see column 4 in Table 4).

A consistent way to sum up double-double numbers across multiple processors is to implement the double-double addition as an MPI operator, in the same functionality as MPI_SUM. This can be done by using the MPI operator creation subroutine MPI_OP_CREATE to create an operator MPI_SUMDD for the double-double summation. An implementation of MPI_SUMDD creation is given in the Appendix C. With the operator MPI_SUMDD, the consistent double-double arithmetic can be done in the same way as the normal complex double precision

```
MPI_REDUCE (local_sum, global_sum, 1, MPI_COMPLEX, MPI_SUMDD, ......)
```

With this approach, we always get the exact result of Eq.(1) on different number of processors, as expected (see column 5 in Table 4).

Table 4: Results from double precision summation and self-compensated summation with different number of processors in aatural order as in code (1). Here P means number of processors, MPI_SUM means to add sums and errors from self-compensated summation or first and second double numbers from double-double precision summation separately with double precision; MPI_SUMDD means to add local summations in double-double precision with a unified MPI operator across processors.

P	SCS		Double-Double	
	MPI_SUM	MPI_SUMDD	MPI_SUM	MPI_SUMDD
1	0.3882288932800293	0.3882288932800293	0.35798583924770355	0.35798583924770355
2	0.3745570182800293	0.3745570182800293	0.35798583924770355	0.35798583924770355
4	0.3804163932800293	0.3804163932800293	0.35798583924770355	0.35798583924770355
8	1.9995570182800293	0.3745570182800293	1.4829858392477036	0.35798583924770355
16	1.2169642448425293	0.3575892448425293	-0.78263916075229645	0.35798583924770355
32	2.1164088249206543	0.3351588249206543	-0.54826416075229645	0.35798583924770355
64	2.2617335319519043	0.3706812858581543	2.5547937005758286	0.35798583924770355

7.2 Global Summation Using Self-Compensated Method

Separating global summation of an array into local summation followed by summation across processors requires a slight change of the basic procedure of the self-compensated summation. Here on, say, 4 processors we have 4 (`sum`, `error`) pairs. In principle, we can just add up 4 `sum`s and add up 4 `error`s back for the final results. We can use collective function MPI_REDUCE with double precision operator MPI_SUM to sum all `sum`s and `error`s separately; and then add the sum of `sum`s and the sum of `error`s. In this way, the results are unpredictable (see column 2 in Table 4).

From our experience in the sequential environment, we need to sort the sums in decreasing magnitude order before summation. In order to sort the local `sum` and `error` across processors, we need to gather them to one processor, and apply the same sorting and SCS summation algorithm. The test results with different number of processors are all 0.35798583924770355, which is correct. However, as discussed in the introduction, this serialized communication method will not scale well to large number of processors.

Fortunately, we found that the MPI_SUMDD operator discussed above can be "directly" applied to the results of SCS method. Therefore the local summation results of SCS on different processors can be added up using MPI_REDUCE with the operator MPI_SUMDD. It will carry out the double-double precision arithmetic during the intermediate steps with the tree algorithms typically employed in MPI_REDUCE. This is sufficient to produce the exactly correct result in a scalable way.

The reason that the (`sum`, `error`) pairs, produced by the SCS (and DCS) method, can be used with MPI_SUMDD operator is that the `error` in the (`sum`, `error`) pair is exactly the non-representable portion of the double precision number `sum` [In other words, suppose on an imagined computer, the mantissa has 106 bits (which is effectively the precision offered by the double-double representation) to represent the exact summation result. On the real computer in IEEE format, a double precision number has mantissa of 53 bits. `error` would be the portion represented by those lower 53 bits beyond the upper 53 bits representable in the IEEE double precision number]. By design, this non-representable portion is exactly the second double precision number in the double-double number representation. Therefore, (`sum`, `error`) is already in the correct double-double representation. This reveals the intimate relationship between SCS method and double-double arithmetic.

This relation between SCS and double-double arithmetic could also be used in situations other than summation across processors. Suppose we have several subroutines or modules each producing a (`sum`, `error`) pair by employing SCS. Then these SCS results can be directly summed up using the double-double arithmetic without loss of precision. This technique will be very useful for maintaining modularity of large simulation codes.

Note that since DCS has the same representation of the final results as the SCS [`(sum3, error3)` in Code (3) *vs.* `(sum1, error1)` in Code (2)], this relationship holds for DCS too. In particular, one may use DCS on local summation, and then double-double arithmetic on global summation.

In summary, we found that SCS method can be effectively combined with double-double precision arithmetics to achieve accurate summation on a distributed global array.

8 Beyond Global Summation

In this paper, we illustrated the new approach with a simple SSH global summation example. We also studied the dot product between two global vectors to see the effects on orthogonalization between vectors; and the new method substantially improves the orthogonality. The dot product code is a simple modification of the global summation codes, but with a multiplication routine to perform multiplication of two double precision numbers in double-double precision accuracy and output the result in double-double precision. This code together with those listed in the Appendices are available for download from our web site [14].

Matrix-vector and matrix-matrix operations are repeated application of vector-vector dot product operations, and thus can be easily implemented with the above codes, without much memory overhead.

Also we note that there is an effort to develop XBLAS[4], extended precision BLAS using

the double-double precision arithmetics. With this suite of efficient codes, many intermediate results during matrix-vector operations can be promoted to double-double precision, therefore further improve the reproducibility and stability.

9 Practical Implications

Our main results are (a) SCS and double-double arithmetic are very effective; (b) The results of SCS are directly in double-double representation; (c) A simple MPI_SUMDD can carry out double-double arithmetic consistently during the summation across processors. Together with our experiences in this work and other large scale ocean and atmospheric model codes, we recommend the following steps in adopting the accurate arithmetics in real application codes to improve numerical reproducibility and stability:

(1) Select the parts or modules where global summation plays important roles, such as the conjugate gradient solution of the barotropic equations or the spectral transforms. The very simple self-compensated summation can be easily adopted. The double-double arithmetic can also be easily adopted to guarantee the summation accuracy. The results of different modules are summed up using double-double arithmetic. Results on different processors are summed up using MPI_REDUCE with MPI_SUMDD.

(2) Carry out test runs of the resulting codes on different computer platforms and on different number of processors to see if the numerical stability, especially reproducibility are improved. If the improvements are small in some measure, it might indicate that more parts of the codes should be considered for modification to adopt the SCS. If the improvements are substantial in some measure, either more parts of the codes should be considered for adoption, or the double-double arithmetic should be adopted on the selected codes for further accuracy enhancements.

Adoption of double-double arithmetics generally requires more memory and slight changes of codes. One technique we found useful is to think of both the (`sum, error`) in SCS and double-double number as a complex number (see our implementations listed in the Appendices). This is conceptually more consistent than simply using two double numbers in a row, and makes the modifications a little easier. As long as addition and subtraction are concerned, they can be carried out simply as complex numbers. Care must be taken, however, when multiplications and divisions are involved, which are fortunately not the major problems here.

Although we studied effects on summation due to sorted orders, and found sorting array does reduce rounding errors in SCS and DCS, we do not see it as a solution to the reproducibility problem: (1) sorting an array is much slower compared to summation; (2) sorting a distributed array across processors is even more time consuming and unnecessarily com-

plicated. As explained, we believe the SCS and double-double arithmetics could effectively resolve the problem.

10 Concluding Remarks

We studied the numerical reproducibility and stability issue on scientific applications, especially in climate simulations, on parallel distributed memory computers. We focus on the dominant issue that affects the reproducibility and stability: the global summation of distributed data arrays. We investigated accurate summation methods and found two particular effective methods which can also be easily implemented and scale well to large number of processors.

Our main point here is that numerical reproducibility and stability can be improved substantially by using high accuracy in some of the key steps in parallel computations. As a result, the final results will be slightly more accurate; This is a welcome benefit, but of secondary consideration. In fact, we do not propose to use higher precision on all parts of a simulation codes, unless it is shown to be absolutely necessary.

As we pointed out in Section 6, for a single addition, double-double arithmetic involves 11 double precision arithmetics *vs.* 4 for SCS *vs.* 1 for normal double precision summation. The run timings for these methods are not in the same ratio. For a particular scientific application such as climate simulation, global summation is usually takes a very small part (less than 1%), though critical, of the overall codes. Adopting accurate arithmetic methods will almost not affect the total timing.

To our knowledge, this work is the first systematic attempt to address the reproducibility from accurate arithmetic approach. Here we used the SSH data as a validation test of the improvement techniques. How well these techniques perform in practical codes when the variables affect dynamics? Are they adequate? All these require further investigations. We plan to examine the spectral transforms in the CCM atmosphere model [8, 13] and conjugate gradient solver in the POP ocean model [24]. Global summation is just an example we used in this work. High accurate arithmetics should be used in wherever reproducibility is an issue.

Acknowledgment. We thank David Sarafini for pointing to us Kahan's methods, David Bailey for the double-double arithmetic codes, Sherry Li for discussions on extended precision arithmetics, and Horst Simon for support. We also thank those who participated in the discussions on the reproducibility issue at climate modeling conference [25] and workshop [26]; those discussions motivated this work. This work is supported by the Office of Biological and Environmental Research, Climate Change Prediction Program, and by the

Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences, of the U.S. Department of Energy under contract number DE-AC03-76SF00098. This research uses resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

References

[1] D. H. Bailey. A Fortran-90 Suite of Double-Double Precision Programs. See web page at http://www.nersc.gov/~dhb/mpdist/mpdist.html.

[2] D. H. Bailey. Multiprecision Translation and Execution of Fortran Programs. *ACM Transactions on Mathematical Software*, 19(3):288–319, September 1993.

[3] R. P. Brent. A Fortran Multiple Precision Arithmetic Package. *ACM Transactions on Mathematical Software*, 4:57–70, 1978.

[4] J. Demmel, X. Li, D. Bailey, M. Martin, J. Iskandar, and A. Kapur. A Reference Implementation for Extended and Mixed Precision BLAS. In Preparation.

[5] C. H. Q. Ding and R. D. Ferraro. A Parallel Climate Data Assimilation Package. *SIAM News*, pages 1–12, November 1996.

[6] C. H. Q. Ding and Y. He. Data Organization and I/O in an Ocean Circulation Model. In *Proceedings of Supercomputing'99*, November 1999. Also LBL report number LBNL-43384, May 1999.

[7] C. H. Q. Ding, P. Lyster, J. Larson, J. Guo, and A. da Silva. Atmospheric Data Assimilation on Distributed Parallel Supercomputers. *Lecture Notes in Computer Science*, 1401:115–124. Ed. P. Sloot *et al.*, Springer, April 1998.

[8] J. Drake, I. Foster, J. Michalakes, B. Toonen, and P. Worley. Design and Performance of a Scalable Parallel Community Climate Model. *Parallel Computing (PCCM2)*, 21:1571, 1995.

[9] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Vol 1. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[10] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, March 1991.

[11] A. Greenbaum. *Iterative Methods for Solvong Linear Systems.* Frontiers in Applied Mathematics. Vol 17. SIAM. Philadelphia, PA, 1997.

[12] S. M. Griffies, R. C. Pacanowski, M. Schmidt, and V. Balaji. The Explicit Free Surface Method in the GFDL Modular Ocean Model. Submitted to *Monthly Weather Review*, 1999.

[13] J. J. Hack, J. M. Rosinski, D. L. Williamson, B. A. Boville, and J. E. Truesdale. Computational Design of NCAR Community Climate Model. *Parallel Computing*, 21:1545, 1995.

[14] Y. He and C. H. Q. Ding. *Numerical Reproducibility and Stability/NERSC Homepage.* See web page at http://www.nersc.gov/research/SCG/ocean/NRS.

[15] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM Press, Philadelphia, PA, 1996.

[16] W. Kahan. Further Remarks on Reducing Truncation Errors. *Comm. ACM*, page 40, 1965.

[17] D. E. Knuth. *The Art of Computer Programming.* Vol 2, Chap4, Arithmetic. Addison-Wesley Press, Reading, MA, 1969.

[18] D. Moore. *Class Notes for CAAM 420: Introduction to Computational Science.* Rice University, Spring 1999. See web page at http://www.owlnet.rice.edu/~caam420/Outline.html.

[19] *The NCAR Ocean Model User's Guide.* Ver 1.4. See web page at http://www.cgd.ucar.edu/csm/models/ocn-ncom/UserGuide1_4.html, 1998.

[20] R. C. Pacanowski and S. M. Griffies. *MOM 3.0 Manual.* GFDL Ocean Circulation Group, Geophysical Fluid Dynamics Laboratory, Princeton, NJ, September 1999.

[21] B. N. Parlett. *The Symmetric Eigenvalue Problem.* Classics in Applied Mathematics, 20. SIAM, Philadelphia, PA, 1997.

[22] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in Fortran: the Art of Scientific Computing.* 2nd Edition. Cambridge University Press, Cambridge, UK, 1992.

[23] D. M. Priest. Algorithms for Arbitrary Precision Floating Point Arithmetic. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations.* Ph.D. Thesis. Mathematics Dept. University of California, Berkeley, 1992.

[24] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Parallel Ocean General Circulation Modeling. *Physica*, D60:38, 1992. See web page at http://www.acl.lanl.gov/climate/models/pop.

[25] Second International Workshop for Software Engineering and Code Design for Parallel Meteorological and Oceanographic Applications. Scottsdale, AZ, June 1998.

[26] Workshop on Numerical Benchmarks for Climate/ Ocean/Weather Modeling Community. Boulder, CO, June 1999.

Appendices

Note: These codes are written on Cray T3E, where `real` is of 64-bit precision by default. On many other computers, `real` is of 32-bit by default and one needs to replace `real` by `real*8`, `complex` by `complex*16`, etc., in the implementation.

A. Self-Compensated Summation Method

```
C  This function returns the (sum,error) pair as a complex number.
      complex function SCS_sum (array, n)
      real array(n)
      complex cc, SCS
      cc = cmplx (0.0, 0.0)
      do i= 1, n
         cc = SCS (real(cc), imag(cc) + array(i))
      enddo
      SCS_sum = cc
      end

      complex function SCS (a, b)
      real a, b, sum
      sum = a + b
      SCS = cmplx (sum,  b - (sum - a))
      end
```

B. Doubly-Compenstated Summation Method

```
C  This function returns the (sum,error) pair as a complex number.
      complex function DCS_sum (array, n)
      real array(n)
      complex cc, SCS, c1, c2
      cc = cmplx (array(1), 0)
      do i= 2, n
         c1 = SCS (imag(cc), array(i))
         c2 = SCS (real(cc), real(c1))
         cc = SCS (real(c2), imag(c1) + imag(c2))
      enddo
      DCS_sum = cc
      end
```

C. Double-Double Precision Summation

```
C  This code calculates the summation of an array of real numbers
C  distributed on multiple processors using double-double precision.
      include 'mpif.h'
      real array(n)
      integer myPE, totPEs, stat(MPI_STATUS_SIZE), ierr
      integer MPI_SUMDD, itype
      external DDPDD
      complex local_sum, global_sum
      call MPI_INIT(ierr)
      call MPI_COMM_RANK( MPI_COMM_WORLD, myPE, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, totPEs, ierr )
C  operator MPI_SUMDD is created based on an external function DDPDD.
      call MPI_OP_CREATE(DDPDD, .TRUE., MPI_SUMDD, ierr)
C  assume array(n) is the local part of a global distributed array.
      local_sum = cmplx (0.0,0.0)
      do i = 1, n
         call DDPDD (cmplx(array(i), 0.0), local_sum, 1, itype)
      enddo
C  add all local_sums on each PE to PE0 with MPI_SUMDD.
C  global_sum is a complex number, represents final (sum, error).
```

```
      call MPI_REDUCE (local_sum, global_sum, 1, MPI_COMPLEX, MPI_SUMDD,
     &                 0, MPI_COMM_WORLD, ierr)
      call MPI_FINALIZE(ierr)
      end

C   Modification of original codes written by David H. Bailey.
C   This subroutine computes ddb(i) = dda(i) + ddb(i)
      subroutine DDPDD (dda, ddb, len, itype)
      implicit none
      real*8 e, t1, t2
      integer i, len, itype
      complex*16 dda(len), ddb(len)
      do i = 1, len
c   Compute dda + ddb using Knuth's trick.
      t1 = real(dda(i)) + real(ddb(i))
      e = t1 - real(dda(i))
      t2 = ((real(ddb(i)) - e) + (real(dda(i)) - (t1 - e)))
     &     +imag(dda(i)) + imag(ddb(i))
c   The result is t1 + t2, after normalization.
      ddb(i) = cmplx ( t1 + t2, t2 - ((t1 + t2) - t1) )
      enddo
      end
```