Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

An order-invariant real-to-integer conversion sum

Robert Hallberg *, Alistair Adcroft

NOAA Geophysical Fluid Dynamics Laboratory, 201 Forrestal Rd., Princeton, NJ 08540, USA
Atmospheric and Oceanic Sciences Program, Princeton University, Princeton, NJ 08540, USA

ARTICLE INFO

Article history:
Received 6 March 2012
Received in revised form 7 February 2014
Accepted 21 April 2014
Available online 30 April 2014

Keywords:
Order-invariant
Global sums
Parallel programming
Fixed-point
Reproducibility

ABSTRACT

This paper describes a technique for obtaining sums of floating point values that are independent of the order-of-operations, and thus attractive for use in global sums in massively parallel computations. The basic idea described here is to convert the floating point values into a representation using a set of long integers, with enough carry-bits to allow these integers to be summed across processors without need of carries at intermediate stages, before conversion of the final sum back to a real number. This approach is being used successfully in an earth system model, in which reproducibility of results is essential.

Published by Elsevier B.V.

1. Introduction

When writing scientific software for massively parallel computers, it is often desirable that the answers should be independent of the number of processors used or how a model's data is decomposed among these processors. Meeting this criterion often requires bitwise invariant arithmetic operations on real numbers. Oceanic or atmospheric models, for instance, are highly nonlinear, and changes even in the least significant bit of a floating point number can rapidly cascade up to leading order changes in local values (e.g., [1]). Other applications like molecular dynamics simulations are similarly sensitive to the accumulation of round-off errors in sums (e.g., [2]). However, naïve approaches for obtaining a processor-count invariant global sum of a real array, such as collecting that global array on a single processor, do not scale with processor count and can become extremely memory and communication intensive. This paper describes an approach for calculating a sum of real values that is invariant to the order in which the sum is taken, and introduces no loss of precision relative to the original values over a broad range of values.

Sums of floating-point numbers have truncation errors which make them non-associative. For example, with 32-bit IEEE real numbers [3], the result of $1 - 1 + 10^{-10}$ may give 10^{-10} or 0, depending on whether it is evaluated as $(1 - 1) + 10^{-10}$ or $1 + (-1 + 10^{-10})$, respectively. Even roundoff on sums of positive numbers can lead to significant errors; as an extreme example, naïve evaluation of $\sum_1^{10^{10}} 10^{-10}$ by simply adding each successive value to the running sum would lead to a value of roughly 8.4e−4 with 32-bit IEEE real numbers, at which point adding another 10^{-10} no longer changes the running sum. There are several cost-effective approaches for greatly reducing the magnitude of these errors and making it much less likely (but not impossible) that order-of-arithmetic would alter a global sum [4]. These include adding terms in an order determined by partially sorting them by the magnitude of their exponents [5], using 128-bit floating point intermediate sums

* Corresponding author at: NOAA Geophysical Fluid Dynamics Laboratory, 201 Forrestal Rd., Princeton, NJ 08540, USA. Tel.: +1 609 452 6508; fax: +1 609 987 5063.
E-mail addresses: Robert.Hallberg@noaa.gov (R. Hallberg), Alistair.Adcroft@noaa.gov (A. Adcroft).

http://dx.doi.org/10.1016/j.parco.2014.04.007
0167-8191/Published by Elsevier B.V.

to evaluate sums of 64-bit floating point numbers [6], or accumulating as a second floating point number an estimate of the truncation errors in each step of a sum, which can then be used as a correction for the next step [7]. The accumulation of the truncation errors can be extended to arbitrary precision, making order dependent round-off highly unlikely [8]. Alternately, a deterministically ordered reduction tree can be used to give truncation errors that are independent of domain decomposition, although the final answer does depend on the choice of tree structure [9]. Implementation of these approaches in parallel sums can require modifications to the standard global reduction routines, and even then there can be the possibility that the sums could be changed at the level of round-off, particularly when the global sum is a small residual compared to individual values [4].

By contrast, integer arithmetic has no truncation errors, and sums are associative. Integer sums across large numbers of processors exactly reproduce. However, integer sums are subject to overflow. The largest representable 64-bit signed integer is $2^{63} - 1 \approx 9.2\text{e}18$. If a large number, p, of 64-bit integers are summed, each must initially be less than $(2^{63} - 1)/p$ to avoid the possibility of overflow in the sum. These properties of integer sums are a primary motivation behind the technique described here.

2. Proposed technique

This paper proposes to represent real numbers with a series of integers, which can then each be summed, first within each processor, and then across a large number of processors without overflow, before conversion back to a real number. A real number, r, can be represented with a set of N 64-bit integers a_i, as

$$r = \sum_{i=0}^{N-1} a_i 2^{(i-\frac{N}{2})M} \tag{1}$$

where M is a positive integer that is less than 63. When two such representations are added together, each of the N integers can be added independently, with values larger than 2^M being carried and 1 added to the next integer. If each of the a_i are smaller in magnitude than 2^M, $(2^{63-M} - 1)$ such values can be added before there is a risk of overflow, followed by a collective carry to make the representation as a set of integers unique and invariant to the order of the sums. This approach essentially uses fixed-point arithmetic of arbitrary precision, but with a large enough number of carry bits to accommodate sums across a large number of processors, using standard routines for global sums of a standard integer data type.

A key objective in a massively parallel computation is to do the sum without any special treatment of incremental results. Ideally the sums should be able to occur across P processors without worrying about carrying values between the integers at intermediate stages of the calculation. For 64-bit integers, this is true if $P < 2^{63-M}$. Since choosing a larger value of M increases the information content of each integer, the right choice for M is determined by P, as illustrated in Table 1. For instance, if M is chosen as 46, up to 131,071 values can be added before overflow becomes an issue, and each of the integers has a range of $\pm 7.036 \times 10^{13}$. The range of representable values is determined by the product of N and M, as illustrated in Table 2. Setting $N = 6$ and $M = 46$ allows values with magnitudes between 2.87×10^{-42} and 3.48×10^{41} to be represented, covering more than the valid range for 32-bit IEEE floating-point numbers, although numbers smaller than 2.41×10^{-35} are represented at reduced precision. Scientific programs, such as ocean- and climate-models, typically use 64-bit floating point numbers in their calculations (usually for their greater precision, not their larger range). When summing 64-bit floating point numbers the same range applies as for 32-bit floating point numbers, but values smaller than 1.29×10^{-26} have reduced precision when $N = 6$ and $M = 46$. If nothing is known about the size of the values being summed, except that they are of reasonable magnitude, $N = 6$ would seem to be a reasonable choice; it permits, for instance, operations on values ranging from the S.I. mass of the electron (9.1×10^{-31} kg) to the S.I. mass of the sun (1.99×10^{30} kg). If the rough magnitude of the final sum is known a priori, it might be possible to rescale the floating-point numbers being summed, in which case $N = 2$ might give a perfectly adequate range and precision.

The algorithm for converting real numbers into the proposed set of integers is very simple, as illustrated by the following line of C-style pseudocode converting a floating point number r into the form described in (1):

$$for\ (i = N - 1; i >= 0; i - -)\ \left\{a_i = (\text{long\_integer})\left[r \times 2^{-(i-\frac{N}{2})M}\right];\ r = r - a_i \times 2^{(i-\frac{N}{2})M}\right\}. \tag{2}$$

Table 1
Maximum processor count (max PEs) and precision per integer as a function of the number of bits (M) used in the 64-bit signed integer representation.

M	Max PEs = $2^{63-M} - 1$	Precision per 64-bit integer = 2^M
43	1,048,575	8.796093×10^{12}
46	131,071	7.036874×10^{13}
49	16,383	5.629500×10^{14}
53	1023	9.007199×10^{15}
55	255	3.602880×10^{16}

Table 2
The range of representable positive real numbers ($2^{-MN/2}$ to $2^{MN/2}$) as a function of the number (N) of 64-bit signed integers used and the number of bits (M) in each integer that is used. Because the integer with the largest basis does not have any carry bits, the largest representable value for the "high-water-mark" in sums is increased by a factor of 2^{63-M} (up to $2^{MN/2+63-M}$, listed in parentheses). However, the individual values being summed can only safely use the original range extended by *floor* ($2^{63-M}/P$) and still have proper overflow error handling. By comparison, the largest representable 32-bit IEEE binary floating-point number is $\sim 3.403 \times 10^{38}$, while the largest representable 64-bit floating-point number is 1.798×10^{308}. The choice to center this range around 1 is arbitrary, and could be shifted to larger or smaller numbers to optimally suit a particular application.

M/N	2	4	6	38	44
43	1.137×10^{-13} to 8.796×10^{12} (9.223×10^{18})	1.292×10^{-26} to 7.737×10^{25} (8.113×10^{31})	1.469×10^{-39} to 6.806×10^{38} (7.236×10^{44})	1.144×10^{-246} to 8.740×10^{245} (9.164×10^{251})	1.681×10^{-285} to 5.948×10^{284} (6.237×10^{290})
46	1.421×10^{-14} to 7.039×10^{13} (9.223×10^{18})	2.019×10^{-28} to 4.952×10^{27} (6.490×10^{32})	2.870×10^{-42} to 3.486×10^{41} (4.567×10^{46})	7.939×10^{-264} to 1.260×10^{263} (1.651×10^{268})	2.279×10^{-305} to 4.389×10^{304} (5.753×10^{309})
49	1.776×10^{-15} to 5.630×10^{14} (9.223×10^{18})	3.155×10^{-30} to 3.169×10^{29} (5.192×10^{33})	5.605×10^{-45} to 1.784×10^{44} (2.923×10^{48})	5.509×10^{-281} to 1.815×10^{280} (2.974×10^{284})	$<10^{-310}$ to $>10^{310}$
53	1.110×10^{-16} to 9.007×10^{15} (9.223×10^{18})	1.233×10^{-32} to 8.113×10^{31} (8.378×10^{34})	1.368×10^{-48} to 7.308×10^{47} (7.483×10^{50})	7.291×10^{-304} to 1.372×10^{303} (1.404×10^{306})	$<10^{-310}$ to $>10^{310}$
55	2.775×10^{-17} to 3.603×10^{16} (9.223×10^{18})	7.704×10^{-34} to 1.298×10^{33} (3.323×10^{35})	2.138×10^{-50} to 4.677×10^{49} (1.197×10^{52})	$<10^{-310}$ to $>10^{310}$	$<10^{-310}$ to $>10^{310}$

In practice, the powers of 2 can be pre-computed and stored, so each conversion involves $2N$ real multiplies and N real subtracts. The addition of each new value to a running sum requires an additional N integer adds, and can occur as a part of the same loop as above. The other steps, such as carrying the values larger than $2M$ or conversion back to a floating point number, can occur after a large number of values have been accumulated, and do not contribute significantly to the computational cost of the algorithm.

The conversion of the final sum back to a floating point number follows directly from (1), but with one subtlety that has to be taken into account to get order-invariant sums. The representation in (1) is not unique, in that while each of the a_i are required to be in the range $-2^M < a_i < 2^M$ after carrying, 2^M could be added to (or subtracted from) a_i and 1 subtracted from (or added to) a_{i+1} to represent the same value. The result after any sum may have both positive and negative values among the a_i. The different representations of the same result could give different roundoff error in taking the sum (1) to give the final floating point result, To counter this, the integer series representation of the final result has to be converted into a standard unique form, such as requiring that all of the a_i be of the same sign, before conversion back to a floating point number using (1).

The results from taking sums with the approach described here are not only invariant to the order in which the sum occurs, they also give the *right* answer in all cases where there is not overflow. For instance, $1 - 1 + \varepsilon$ is always ε and $\sum_{i=1}^{Z} 1/Z$ is always 1 (provided that ε and $1/Z$ are within the range of values that can be represented in the first place). The only point at which there is any loss of information is the original conversion of the real numbers to the integer sets and in the conversion of the final sum back to a floating-point number. The sums of the integer sets themselves are exact. The final floating-point result is completely independent of the order of arithmetic.

The robustness of the sums that are calculated with this technique also has clear utility, for instance in verifying the numerical conservation of integrated quantities in ocean or atmospheric models (such as total mass of water or total mass of salt) that should be conserved according to the underlying equations. With a simple floating point global sum, conservation can only be evaluated to within roundoff relative to the globally summed quantity (at best). However, by taking the difference directly between the integer sums from successive states, before conversion back to a real number, conservation can be verified to within roundoff relative to the largest individual element. In idealized test cases with the Generalized Ocean Layered Dynamics (GOLD) ocean model [10], this approach led to a reduction of up to 3 orders of magnitude in the diagnosed non-conservation of water or salt (to 1 part in 10^{18}), relative to the naïve approach (the solutions themselves were identical). This added diagnostic accuracy can be useful for detecting subtle bugs or demonstrating algorithmic improvements. In an ice-sheet model with solutions that depend on global sums in its conjugate-gradient elliptic solver, this technique gives results that are invariant to the number of processors used (D. Goldberg, pers. comm.).

3. Computational performance in a test case

Summing a series of real numbers with this approach is more computationally expensive than simply adding the real numbers, although in an actual application like a coupled climate model there are relatively few global sums and the added expense is almost undetectable in overall model runtime. The conversion of each real number to the set of integers takes $2N$ real multiplies and N real adds, while adding to the running sum takes another N integer adds, as compared with a single real

Table 3
Wall-clock CPU time in seconds required to do global sums of a 360 × 180 array of 64-bit real numbers 24,000 times on dedicated nodes of a Cray XT6 (which has 24 cores per node) using standard MPI calls for data reduction or collection, averaged over 5 trials each (except the 648-core case using collection of whole arrays, which is for a single trial).

Technique	Wall-clock time on 72 cores	Wall-clock time on 648 cores
Direct non-reproducing sum	0.98 ± 0.04	2.22 ± 0.18
Order-invariant real-to-integer conversion sum, $N = 3$	10.4 ± 0.3	5.0 ± 0.4
Order-invariant real-to-integer conversion sum, $N = 6$	21.1 ± 0.7	8.1 ± 0.4
Reproducing sum by collection of whole arrays to a single processor	3247 ± 74 (Extrapolated from 240 sums)	1,800,000 (Extrapolated from 24 sums)

add with a simple sum of the original real numbers. On computationally bound codes the proposed approach will be between $2N$ and $4N$ times as expensive as simply taking an order-sensitive sum of real numbers (it could be $2N$ since one of the real multiplies, the real add and the integer add can occur within the same clock cycle on modern computers, something single-processor tests support); on memory or communication bound codes this factor can be much smaller. The timings of a test of these scalings in parallel applications are shown in Table 3. In a test summing a 64,800 point array decomposed over 72 processors, the integer conversion sum with $N = 3$ takes 10.5 times longer than simply adding the real numbers, while with $N = 6$ it takes 21 times as long. For the same test with 648 processors, the ratios are just 2.2 for $N = 3$ and 3.5 for $N = 6$, because the time used for the communication across processors is similar between approaches. However, when it matters that the sum is independent of processor count or domain decomposition, this is not the relevant comparison. Since the order of integer addition does not affect the result, with the proposed approach values can be summed independently on each processor, and then only a single one of these integer sets needs to be communicated and added across processors in an arbitrary order. By contrast, an exactly reproducing sum across processors would take either a communication of all values to a single processor (which may be very memory and communication intensive, as illustrated in the last row of Table 3), repeated communication of partial sums across processors (which may have large communication latency costs), or doing the global sum via a deterministic reduction tree structure to enforce a specified order of operations (which can involve communication costs, load imbalances, or redundant calculations [9]). Many of these costs typically will not scale well with increasing numbers of processors; any of these costs can dwarf the added costs of converting to the integer sets or summing the sets, especially on larger numbers of processors. Compared with these high parallelization costs, the approach described here may be an attractive alternative for applications where sums that are certain to be invariant to processor count or domain decomposition are required.

Acknowledgment

We would like to thank Dan Goldberg for his patient help in testing our implementation of these ideas in his ice-sheet model.

References

[1] J.M. Rosinski, D.L. Williamson, The accumulation of rounding errors and port validation for global atmospheric models, SIAM J. Sci. Comput. 18 (1997) 552–564.
[2] M. Taufer, O. Padron, P. Saponaro, S. Patel, Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs, in: Proceeedings of the IEE/ACM IPDPS, 2010.
[3] IEEE, IEEE standard for floating point arithmetic, IEEE Std. 754-2008, 2008, doi: 10.1109/IEEEESTD.2008.4610935.
[4] R.W. Robey, J.M. Robey, R. Aulwes, In search of numerical consistency in parallel programming, Parallel Comput. 37 (2011) 217–229.
[5] J. Demmel, Y. Hida, Accurate and efficient floating point summation, SIAM J. Sci. Comput. 25 (2003) 1214–1248, http://dx.doi.org/10.1137/S1064827502407627.
[6] Y. He, C. Ding, Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications, J. Supercomput. 18 (2001) 259–277.
[7] W. Kahan, Further remarks on reducing truncation errors, Commun. ACM 8 (1965) 40.
[8] S.M. Rump, Ultimately fast accurate summation, SIAM J. Sci. Comput. 32 (2009) 3466–3502, http://dx.doi.org/10.1137/080738490.
[9] O. Villa, D. Chavarría-Miranda, V. Gurumoorthi, Andrés Márquez, S. Krishnamoorthy, Effects of floating-point non-associativity on numerical computations on massively multithreaded system, in: Proceedings of the Cray User Group, CUG 2009 Meeting, 2009.
[10] R. Hallberg, A. Adcroft, Reconciling estimates of the free surface height in Lagrangian vertical coordinate ocean models with mode-split time stepping, Ocean Model. 29 (2009) 15–26, http://dx.doi.org/10.1016/j.ocemod.2009.02.008.