2016 IEEE International Parallel and Distributed Processing Symposium

Order-Invariant Real Number Summation: Circumventing Accuracy Loss for Multimillion Summands on Multiple Parallel Architectures

Patrick E. Small, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta
Collaboratory for Advanced Computing and Simulations
Dept. of Computer Science, Dept. of Physics & Astronomy, Dept. of Chemical Engineering & Materials Science
University of Southern California
Los Angeles, CA 90089-0242, USA
{patrices, rkalia, anakano, priyav}@usc.edu

***Abstract*—Achieving reproducibility of scientific results in parallel computing is both a challenge and a source of active research. A significant contribution to non-reproducibility is rounding error introduced into calculations by the non-associativity of floating point addition. Scientific applications that rely on accumulation of many small values, such as climate and N-body simulations, are susceptible to this type of error. This paper proposes a variant of an existing fixed-point method for real number summation that yields sums with perfect precision, and which are invariant to summation order and system architecture. The new method improves upon the existing technique by exhibiting improved performance for large numbers of summands, introducing tunable fractional precision to place precision where it is needed, and eliminating the aliasing problem of the original method. The proposed technique is described and its performance is demonstrated in the OpenMP, MPI, CUDA, and Xeon Phi parallel programming environments. In particular, the proposed method outperforms the previous state-of-the-art for larger problems involving over one million summands at high precision. With the anticipated convergence of exascale high-performance computing and big data analytics on hybrid architectures, computational reproducibility will become an even more difficult problem than it is today. Use of numerical techniques such as the method proposed here can help to mitigate the impact of error and variation within simulations at these large scales.**

Keywords-reproducibility; rounding error; high-precision arithmetic; summation; fixed-point arithmetic

I. INTRODUCTION

A fundamental problem with performing real number arithmetic on a computer is that floating-point addition is non-associative. Precision limitations with the representation of floating-point numbers force a small rounding error to be introduced into the sum each time an addition is performed [1]. Over many consecutive additions, this rounding error accumulates and can become very large. In effect, the sum becomes a random walk across the space of possible rounding error.

The significance of this error is often overlooked, as given a deterministic summation order for a set of summands, a consistent total sum will always be returned by the computer. However, even this sum contains rounding error - it is simply hidden from view. With the adoption of large-scale parallel computing, this inherent round-off error is unmasked and manifests itself as small perturbations in global sums that are performed in parallel. In scientific codes, this can be a significant problem [12]. At a minimum, the error makes exact reproduction of computation results across simulations difficult to achieve. At worst, error is compounded in each time step until the simulation results are meaningless.

Proposed methods for minimizing rounding error involve either application of general-purpose arbitrary precision arithmetic, error compensation techniques, alternative real number representations arising from computable analysis, hardware solutions, high-precision intermediate sums, or combinations of these approaches [5]. An example implementation of general-purpose arbitrary precision arithmetic is the GNU Multi-Precision Library [9]. Arbitrary precision can perform nearly any type of arithmetic operation at potentially infinite precision, yet it requires extensive computational and memory resources which can be prohibitive in a parallel computing environment.

Error compensation methods include utilizing error-free transformations to track accumulated error during the summation process [6-8, 13, 15, 16, 19, 21], or manipulating the summation order to minimize error such as with pairwise summation (see also [12]). The error-free transformations can yield a significant reduction in rounding error with very good performance, although they typically cannot completely eliminate the problem. As the number of operands may be very high and distributed across many different processors, ordered summation approaches are prohibitive at large scales. Although computable analysis techniques can be computationally expensive, they can often

1530-2075/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPS.2016.41

IEEE computer society

Figure 1: Observed standard deviation for the sum of sets of *n* semi-random numbers. Each set was constructed to yield exactly zero on a computer with infinite precision.

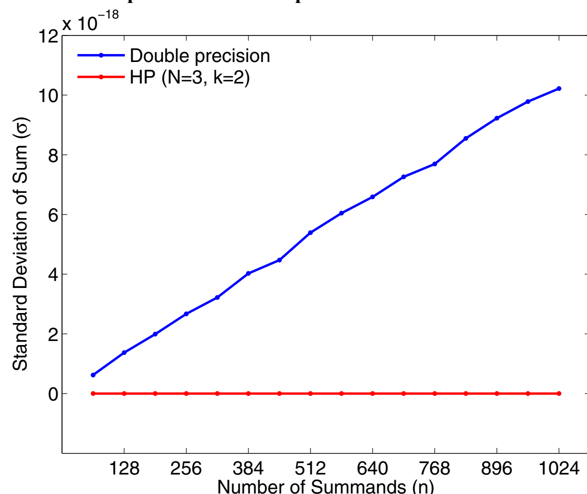
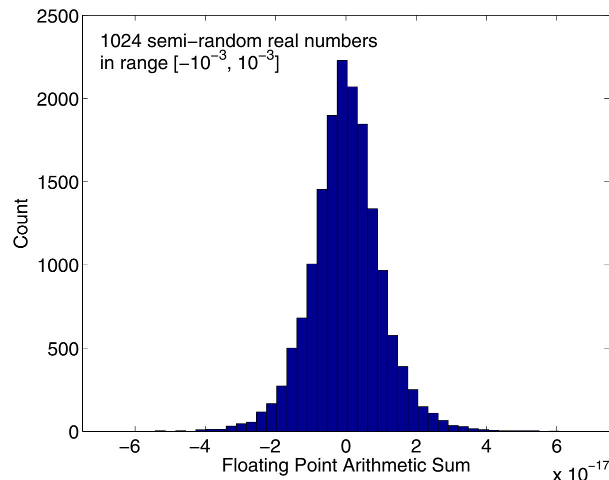


Figure 2: Distribution of 16384 floating-point sums consisting of 1024 semi-random numbers, with each trial adding the operands in random order.



be extended to support mathematical operations beyond addition [2-4, 17]. Proposed hardware solutions strive to overcome the limitations of IEEE floating-point arithmetic by redefining the real number representation in the hardware layer. The universal number format (unum) is a recent example of work in this area [10]. The challenge with this approach is that significant resources are needed to design and field the new hardware.

The remaining class of techniques involves performing the addition with a very high precision intermediate sum and subsequently rounding down to a lower precision final result [11, 12]. Higher precision intermediate sums can exhibit zero rounding error when given sufficient memory to represent the sum. However, some properties of the operands must be known a priori, an example being dynamic range when using a fixed point high-precision accumulator. In addition to their potential accuracy, these techniques benefit from being conceptually simple and easy to implement across a wide range of computer architectures.

The method proposed in this paper, called the High-Precision (HP) method, is a high-precision intermediate sum technique that is derived from Hallberg [11]. Whereas the Hallberg method utilizes carry minimization in order to increase performance, the HP method instead attempts to maximize information content. The HP method yields comparable performance at the same level of precision while eliminating the storage overhead and aliasing of values in the original approach. As is its predecessor, it is order invariant and can be implemented efficiently in virtually any parallel environment.

The remaining sections begin with a discussion of the rounding problem in quantitative terms as well as the original Hallberg method of summation. Then the proposed HP method is described, and its performance is evaluated against both the Hallberg method and double precision addition in multiple parallel computing environments.

II. Background

A. Rounding Error

To demonstrate the extent of the rounding problem, a test was constructed to identify the accumulated rounding error after summing sets of n small random floating point values with random summation orders, for various values of $n = \{64, 128, ..., 1024\}$. Each set of semi-random numbers was generated in such a way that their sum must be zero on a computer with infinite precision.

For each set of size n, $n/2$ random double precision values in the range [0.0, 0.001] were generated. The remaining $n/2$ values were selected to be negatives of the first $n/2$ values, their purpose being to cancel the sum of the first $n/2$ values. With each set of numbers generated in this manner, 16384 trials were conducted for each set in which all the numbers were arranged in a random order and summed together using standard floating point arithmetic. The residual sums for the trials were recorded, producing a distribution of sums for each set. Despite the inclusion of both a random number and its complement in a particular set, the test is protected from catastrophic cancellation effects by the random ordering of operands in each trial. It is unlikely that total cancellation will occur until the last value is added, at which point the operation is complete and the loss of significance is irrelevant. Forcing the true sum to be zero allows us to compute accurate statistics describing the distribution of sums, as the statistics calculation itself is subject to round-off error.

Listing 1: C style code for conversion of a real number r from double precision to HP integers a_i, with translation to two's complement notation.

```
dtmp = fabs(r) * 2^(64*(N-k-1));
isneg = (r < 0.0);
for (i = 0; i < N-1; i++) {
   itmp = (uint64_t)dtmp;
   dtmp = (dtmp – (double)itmp) * 2^64;
   a[i] = (isneg) ? ~itmp + (dtmp <= 0.0) : itmp;
}
a[N-1]  = (isneg) ? ~(uint64_t)dtmp + 1 : (uint64_t)dtmp;
```

Listing 2: C style code for addition of two HP numbers of the form *a* = *a* + *b*.

```
a[N-1] = a[N-1] + b[N-1];
co = (a[N-1] < b[N-1]);
for (i = N-2; i >= 1; i--) {
   a[i] = a[i] + b[i] + co;
   co = (a[i] == b[i]) ? co : (a[i] < b[i]);
}
a[0] = a[0] + b[0] + co;
```

Figure 1 illustrates the standard deviation of the residual sums found after executing the test for n = {64, 128, …, 1024}. The observed error in the sum increases linearly with the number of additions performed. If the n summands in each set were uncorrelated with each other, we would expect the error to increase relative to $\sqrt{n}$. Since one-half of the elements are complements of the remainder, however, this likely introduces a bias to the rounding direction which pushes the accumulated error towards the worst case.

This set selection strategy was chosen to mimic the force accumulation process that is typical of many N-body atomic simulations. There is an accumulation of forces or displacements at each time step within these applications, each contribution consisting of a small positive or negative floating point value. Thus, Figure 1 illustrates that scientific applications which rely on reductions of a large number floating point values, such as N-body simulations, are highly susceptible to floating point rounding error.

Figure 2 illustrates the extent of the spread of the final sums for the case n = 1024 by plotting the distribution of values. We see that the histogram describes a normal distribution whose mean is approximately zero (corresponding to the true sum of zero) and whose standard deviation matches that shown in Figure 1.

Note that Figure 1 also shows results produced using the HP method with parameters N = 3 and k = 2 to perform the summation (see section III below for the definition of the parameters). The HP method achieved perfect precision on these data sets and correctly computed the final sum as zero for all test cases. This proposed technique is described in the following section.

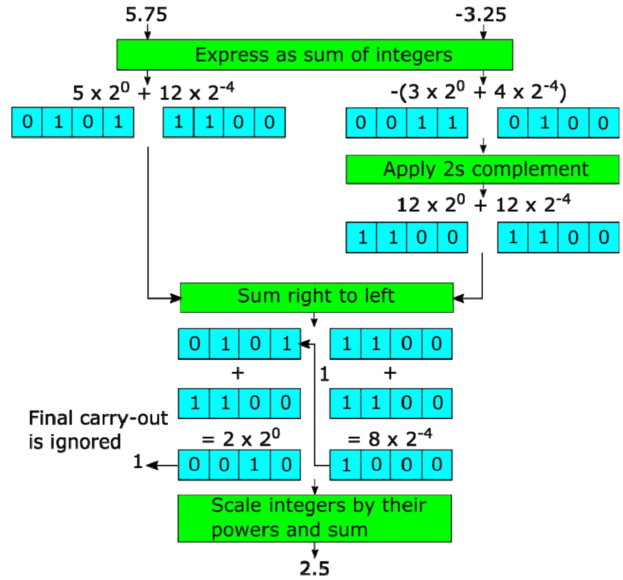
B. *Hallberg Order Invariant Sum*

As described in [11], a real number r can be represented with a set of N 64-bit signed integers, a_i, as follows:

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

Where M is a positive integer that is less than 63. Two real numbers expressed in this form may be added together by summing their N corresponding integers independently.

Figure 3: Example addition of two floating point numbers using the HP method.



When the result of any individual integer addition exceeds 2^M, there is a carry out, and one must be added to the next integer in the set. This is analogous to the carry process in base-10 arithmetic with pen-and-paper. By choosing $M < 63$, however, one can delay or even avoid performing this carry. Selecting M in this manner sets aside a number of bits within each integer to hold the carry that may be generated during addition. The number of carries that can be accommodated in this buffer is $2^{(63-M)} - 1$. Thus, if it is known how many numbers are to be summed, it is possible to select an M that guarantees no carry will be performed during the summation. This serves to reduce the number of integer additions required to add two numbers to N.

However, there are negative consequences to utilizing this representation. The first problem is the overhead - not all bits in the integers serve to provide real-number precision. Each signed integer has a sign bit, and 63 - M carry bits are dedicated to book-keeping purposes. A second problem is aliasing, where multiple integer representations could represent the same real number. A normalization process is required when the summation is complete and the sum is converted back to a real number. Another undesirable quality of this method is that the user must know a priori the expected number of summands. Otherwise, a catastrophic overflow may occur, or an expensive carryout detection and normalization process needs to be conducted at runtime which defeats the purpose of this format.

The next section discusses the alternative approach (based on this method) which addresses all of these concerns, while simultaneously exhibiting better performance at large scales.

Table 1: Maximum range and smallest representable number for the HP method with varying N and k.

N	k	Bits	Max Range	Smallest
2	1	128	$\pm 9.223372 \times 10^{18}$	5.421011×10^{-20}
3	2	192	$\pm 9.223372 \times 10^{18}$	2.938736×10^{-39}
6	3	256	$\pm 3.138551 \times 10^{57}$	1.593092×10^{-58}
8	4	512	$\pm 5.789604 \times 10^{76}$	8.636169×10^{-78}

Table 2: Parameters N and M used with the Hallberg method to achieve near equivalency with the 512-bit HP method.

N	M	Precision Bits	Maximum Summands
10	52	520	≤ 2048
12	43	516	≤ 1 M
14	37	518	≤ 64 M

III. Algorithm

A. Concept

As mentioned previously, the HP method is a variation of the technique described in the previous section. The new method modifies equation (1) so that a real number r is expressed in this alternative form:

$$r = \sum_{i=0}^{N-1} a_i 2^{64(N-k-i-1)}\,, \qquad (2)$$

The integers a_i are unsigned 64-bit values, and all bits are allocated to store the real number with the exception of bit 63 of integer 0. The parameter k is the number of 64-bit unsigned integers to assign to the fractional portion of r where $0 \le k \le N$, thus N-k integers are allocated to represent the whole number component. This tunable parameter allows the user to distribute the total precision among the whole and fractional components.

Negative real numbers are accommodated by first determining the integer representation of its absolute value according to equation (2), then converting that integer representation into two's complement notation. The conversion involves flipping all of the bits of the N unsigned integers, adding one to integer N - 1, and propagating any carries back through to integer 0. Thus, only one bit (in integer 0) is used to track sign regardless of precision at the cost of some additional processing during conversion.

If we restrict the real number under consideration to double-precision, then both the conversion of that number and translation to two's complement notation can be performed in a single pass through the array of integers as shown in Listing 1. The **for** loop utilizes a look-ahead strategy for determining if there will be a carry-in for the current block based on the remainder of the real number at each step. Any non-zero remainder will absorb the added one and will not allow it to propagate to higher magnitude blocks.

Adding two numbers in HP format is a simple process. The corresponding integers of the two integer sets are summed, starting at integer position N - 1 and proceeding to position 0. Any carryout from integer pair i is added to the sum of integer pair i - 1. Since the numbers are stored in two's complement notation, addition of positive and negative numbers is identical. Overflow detection of the sum is accomplished by comparing the signs of the summands with the sign of the sum. Negative summands with a positive sum, or positive summands with a negative sum indicate overflow has occurred. Listing 2 describes the addition process with two sets of HP integers a and b in pseudo-code, and Figure 3 illustrates the entire process for two floating point numbers.

Converting a number from HP format to double precision is the inverse of the algorithm shown in Listing 1.

B. Properties

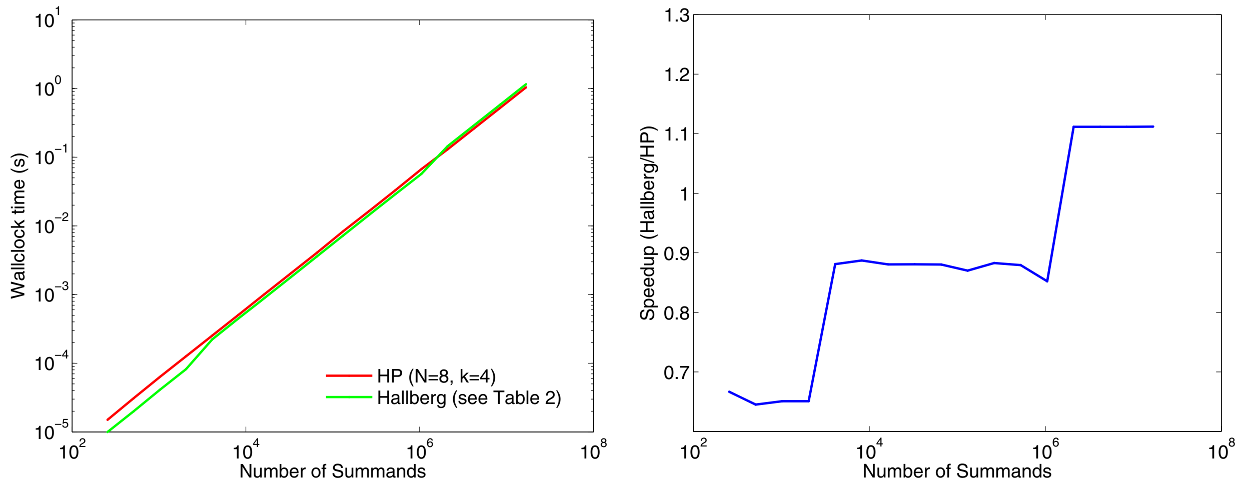
The HP method is a fixed-point representation for real numbers. Juxtaposed with floating-point, HP sacrifices total range for increased precision within the more limited range it supports. Furthermore, this precision is constant over the full range of representable numbers. Table 1 lists the maximum range and smallest representable number for several choices of N and k. The following subsections discuss additional properties of this format.

1) Overflow and Underflow: Operations on real numbers in the HP format are subject to overflow and underflow just as in floating-point arithmetic. Overflow may occur at three points, depending on the relative precisions of the double precision and HP numbers. The first point where overflow is possible is during conversion from double precision to HP. If the magnitude of the double precision number falls outside the maximum range supported by the HP data type, then overflow will occur. The second point where overflow may occur is during the addition of two HP numbers (two very large positive numbers, for example). Lastly, if the HP maximum range exceeds that of double precision, overflow may occur when an HP number is translated back to double precision. All three of these situations can be easily detected at runtime. Underflow may occur during double-to-HP conversion and vice versa, for the same reasons as described above.

2) Atomicity: A high-precision real number representation must be able to operate in a parallel environment for it to be successful. Ideally, it should support the ability to atomically add one number to another in a multi-threaded environment. The HP method can guarantee atomicity of addition using only the compare-and-swap (CAS) synchronization primitive, which is supported in most major compilers as well as CUDA. An atomic adder can be constructed with carry out detection using only CAS. If we express HP addition as $a = a + b$ with a assumed to be the global variable, only one atomic addition with this adder is required for each of the N pairs of integers to be summed. The remaining operations are thread local. Atomicity of addition using CAS is demonstrated with the CUDA performance test described later in this paper.

3) Invariance of Sum: The greatest strength of the HP method is that given sufficient precision with appropriate selection of the N and k parameters, the sum of any quantity

Figure 4: Left: Runtime comparison of HP and Hallberg techniques for up to 16*M* real numbers. Right: Relative speedup of the HP method versus the Hallberg method.



of operands is guaranteed to be invariant, both with respect to the order of the summation and to the architecture on which the addition is performed. Figure 1 illustrates the success of the HP method with N = 3 and k = 2 in computing the final sum of 1024 real numbers in the range [0.0, 0.001] with perfect accuracy. This is possible because both the HP and Hallberg methods reduce real number addition to integer addition, which is fully associative and implemented identically across all architectures. Thus, it is possible to add a sequence of real numbers separately on an Intel CPU and on an Nvidia GPU, for example, and derive the same result in both cases.

IV. PERFORMANCE

A. Comparison with Hallberg Summation

To compare the Hallberg and HP approaches, we may first examine the number of operations required to add a floating point number to a running sum. As stated in [11], the Hallberg method requires 2N FP multiplications and N FP additions to convert a real number to an integer representation, and N integer additions to sum that value with the intermediate sum. The HP method factors out one of the multiplications from the conversion loop (Listing 1) to yield only N FP multiplications along with N FP additions. However, it also requires 3N integer arithmetic/logic (ALU) operations in the worst case (when the number is negative). Adding this converted number to an intermediate sum (Listing 2) requires 4(N - 1) arithmetic/logic operations.

Juxtaposing these raw operation counts alone indicates that the Hallberg should outperform the HP method, yet we find in practice this is not the case for three subtle reasons. First, the latency of floating point multiplication instructions is typically much greater than that of ALU instructions on modern processors [14]. The HP method performs half as many of these expensive operations, trading them for less expensive ALU operations. In addition, many modern processors have more than one ALU and thus the HP method can benefit from ALU instruction concurrency. Lastly, the HP method is a more compact representation since nearly all bits are dedicated to representing precision. The consequence of this compactness is that fewer integer blocks, N, are required to represent the desired precision and this representation occupies less memory. This compactness serves to reduce main memory access latency.

With these differences in mind, it is possible to directly compare the relative performance by establishing a precision equivalency between them. Figure 4 shows the relative performance of the two methods in summing a set of n random real numbers in the range [-2^{191}, 2^{191}] with the smallest such number being $\pm 2^{-223}$, for n = {128, …, 16M}. The HP representation was configured with parameters N = 8 and k = 4 to provide 511 precision bits, while the Hallberg N and M parameters were selected to provide nearly equivalent precision for each n as shown in Table 2. For small numbers of summands, the Hallberg method slightly outperforms the HP method, which is expected as very few bits are reserved for carry storage at these scales and it avoids performing any carry operations as designed. However, the HP method gradually overtakes its counterpart for summand counts in excess of 1M. Thus, the information content maximization strategy of HP can be just as effective as carry minimization.

Formalizing this analysis further, we can show that the relative performance is expected to continue to improve as more operands are included in the sum. Since both techniques operate on a sequence of N 64-bit integer blocks, let us consider their run times as a function of N. Let us also assume that we can bound the cost of converting and adding one integer block to an intermediate sum as taking constant time since both methods require a fixed number of arithmetic operations. The HP and Hallberg costs per integer block will be c_p and c_b, respectively. Then the execution times for the HP method, T_p, and the Hallberg method, T_b, can be expressed as:

Figure 5: Left: Runtime comparison of HP, Hallberg, and double precision global summation for 32*M* numbers in OpenMP. Right: Strong scaling efficiency for the three methods.

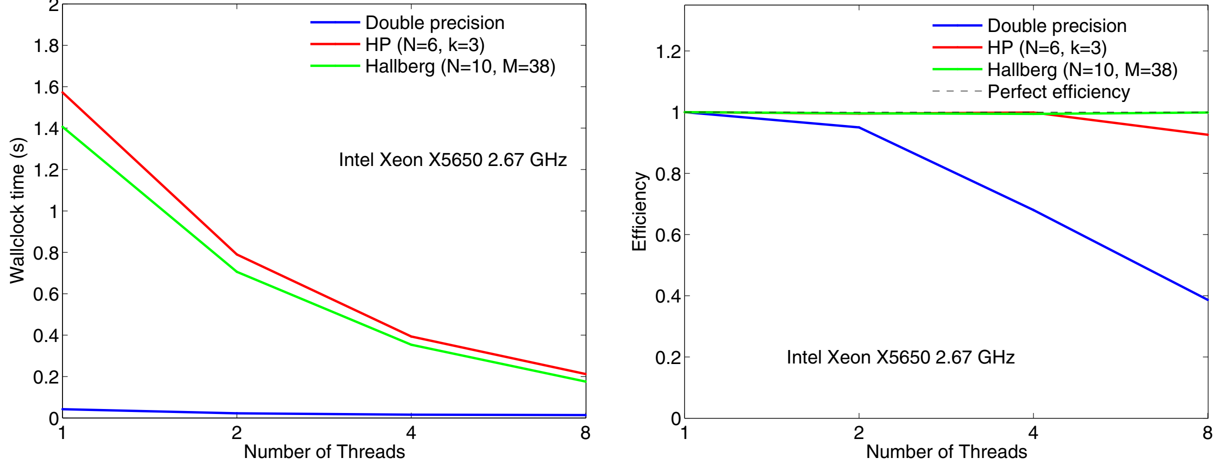
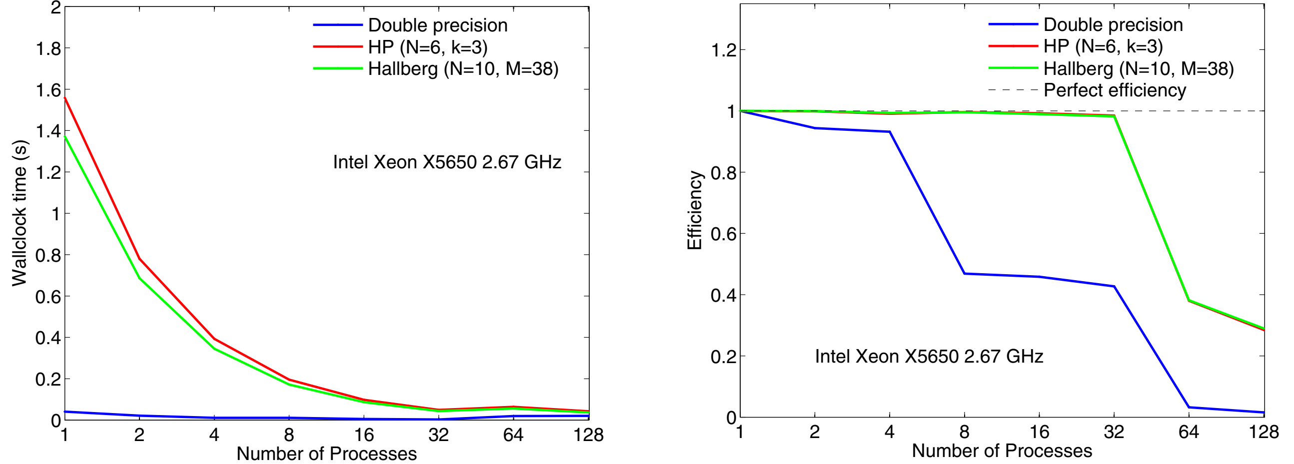


Figure 6: Left: Runtime comparison of HP, Hallberg and double precision global summation for 32*M* numbers in MPI. Right: Strong scaling efficiency for the three methods.



$$T_p = c_p N_p = c_p \left\lceil \frac{b+1}{64} \right\rceil \qquad T_b = c_b N_b = c_b \left\lceil \frac{b}{M} \right\rceil, \tag{3}$$

The parameters N_p and N_b are the number of integer blocks required by each method to support a particular precision. In the case of the HP method, one bit must be added to the precision bit count b to account for the single sign bit, and the number of integer blocks is this value divided by 64. The number of blocks for the Hallberg method is determined by dividing the desired precision, b, by the precision bits per block parameter, M. Ceilings are necessary since there must be an integral number of blocks.

Since the speedup factor of HP versus Hallberg, S, is $S = T_b/T_p$, we have:

$$S = \frac{c_b \left\lceil \frac{b}{M} \right\rceil}{c_p \left\lceil \frac{b+1}{64} \right\rceil}, \tag{4}$$

Converting the ceilings to an inequality and simplifying this expression yields:

$$S \geq \left(\frac{c_b}{c_p} \right) \frac{64b}{M(b+65)}, \tag{5}$$

By limiting the precisions under consideration to $b > 64$, we can bound the term $b/(b+65) \geq 1/2$ which yields the following lower bound on the speedup as a function of M:

$$S \geq \left(\frac{32c_b}{c_p} \right) \frac{1}{M}, \tag{6}$$

Therefore, given a fixed precision b, we would expect the speedup factor S to increase as M is reduced to accommodate more summands. Also note from equation (5) that the speedup has a weak dependency on the number of precision bits b. The speedup is also expected to improve slightly with increased precision for a fixed M, which is confirmed in the following section where the relative runtime performance is examined using a lower precision.

Figure 7: Left: Runtime comparison of HP, Hallberg, and double precision global summation for 32*M* numbers in CUDA. Right: Strong scaling efficiency for the three methods.

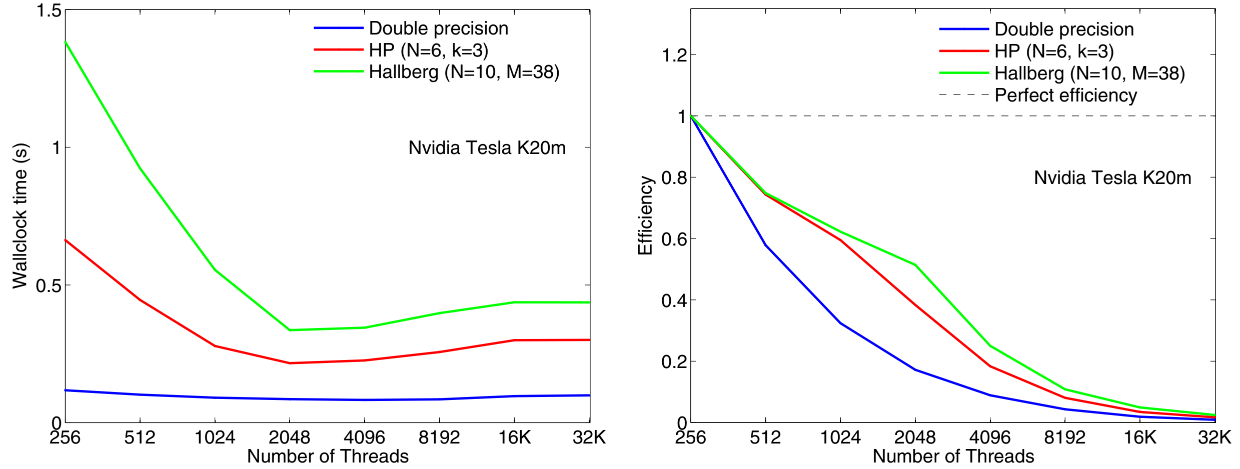
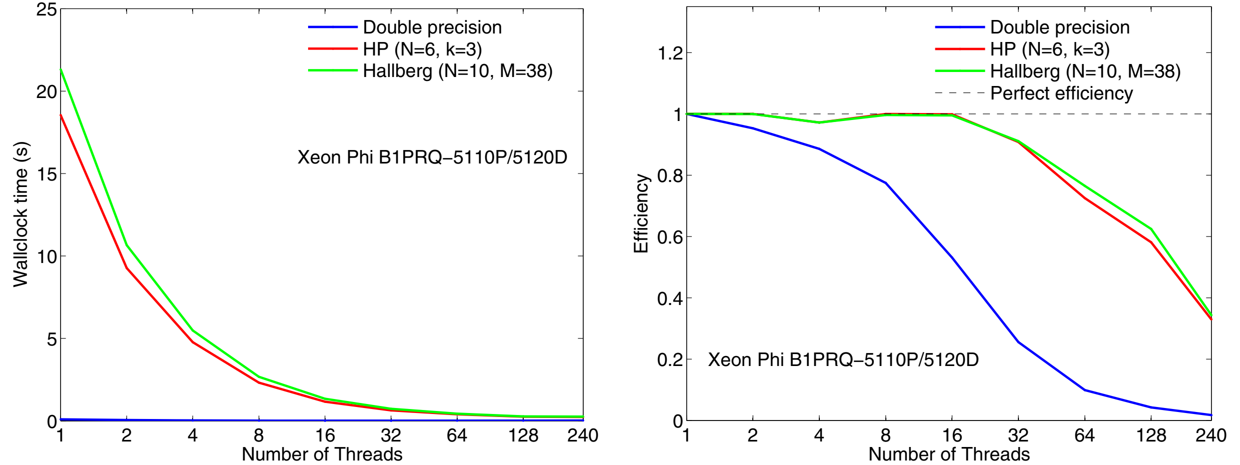


Figure 8: Left: Runtime comparison of HP, Hallberg, and double precision global summation for 32*M* numbers on the Xeon Phi. Right: Strong scaling efficiency for the three methods.



B. Comparison with Double Precision Summation

To evaluate the scaling behavior of the HP method versus that of conventional double precision summation, a strong scaling analysis was performed using a simple global summation in the OpenMP, MPI, CUDA, and Xeon Phi parallel programming environments for varying counts of processing elements (PE). All four global summation implementations create an array of $n = 2^{25} \approx 32\text{M}$ random double precision numbers in the range [-0.5, 0.5] and distribute them across p PEs, with the smallest such number being $\pm 2^{-95}$. A reduction of the local array slice is performed by each PE followed by a global reduction, using standard floating point arithmetic, the HP method with parameters $N = 6$ and $k = 3$, and the Hallberg method with parameters $N = 10$ and $M = 38$. These parameters were chosen to achieve precision equivalency between the two techniques. The OpenMP and MPI implementations were compiled with the GNU C compiler while the Xeon Phi implementation was compiled with the Intel C/C++ compiler. The CUDA implementation was compiled using the Nvidia C/C++ compiler. The full optimization compiler flag (-O3) was set for all builds.

The execution time of these reductions was recorded and averaged over 10 trials. In the case of OpenMP, MPI, and the Xeon Phi, each PE computes a local partial sum of n/p values, and the master PE reduces the p partial sums into a final result. In the case of MPI, this necessitated the creation of a custom MPI data type and **MPI_Op** operation to support reduction with **MPI_Reduce()**. The Xeon Phi benchmark used the heterogeneous offload programming model to distribute the summands to the PEs and compute the partial sums. The CUDA implementation differs slightly from the general approach used by the previous three methods, instead having all p threads simultaneously accumulate results into 256 partial sums using atomic operations, where the partial result used by each thread t is selected by (t modulus 256). The 256 partial sums are then

copied back to the host where the final sum is calculated. This was done to showcase the method's support for atomic operations.

Figures 5, 6, 7, and 8 illustrate the runtime and efficiency results for the OpenMP, MPI, CUDA, and Xeon Phi tests, respectively. First focusing on the OpenMP and MPI results (Figures 5 and 6), we observe that HP requires approximately 37-38x more time in the single PE case as double precision summation on Dual Hex-core Intel Xeon X5650 2.67 GHz CPUs. However, we see that this increased cost is amortized effectively as the number of PEs increases and becomes negligible in the limit. As real scientific applications are much more complex than a simple summation process, we expect HP arithmetic to add a small overhead to the existing calculation.

The CUDA results shown in Figure 7 are more interesting. The slowdown introduced by HP summation is observed to be at most 5.6x for the Nvidia Tesla K20m GPU, while the Hallberg method suffers a much greater slowdown, and performance for all methods plateaus beyond 2048 threads. The plateau is caused by thread saturation as the Tesla K20m supports a maximum of 2496 concurrent threads.

The relative GPU performance, which is better than the OpenMP and MPI cases, can be explained by noting that our global sum application is dominated by global memory accesses and the presence of atomic operations. With the HP method, the addition of a summand to a partial sum requires, at a minimum, reads of seven 64-bit words from global memory (one for the double precision summand, six for the HP partial sum) and writes of six words. The Hallberg method requires eleven reads and ten writes. Meanwhile, double precision requires a read of two words (summand and partial sum) and one write. These are minimums since compare-and-swap may necessitate many more reads to complete the addition.

Thus, if we assume that performance is determined purely by memory operations alone, we would expect the HP method to require at least 4.3x more time than double precision. This is consistent with the observed results, although the effect of the atomic updates cannot be ignored. Recall there are only 256 partial sums that are shared among all threads using atomic operations. This is a point of contention that serves to limit throughput. The HP method suffers slightly less in this regard since three threads may lock an HP partial sum simultaneously (one for each integer in the data type) versus only one thread for a double precision float. The result of this increased concurrency is that the HP method performs slightly better than the predicted 4.3x factor as the thread count is increased to a large multiple of 256.

The Xeon Phi benchmarking results in Figure 8 show that both high-precision methods incur a very high cost versus double-precision arithmetic for the single thread case, likely due to effective use of SIMD and vectorization by the Intel compiler for native double precision, but this cost is amortized as threads are added. The runtimes for all three summation methods are dominated by the data transfer times between the host CPU and device for high thread counts.

Two other observations can be made when examining these performance results in aggregate. The first observation is that the break-even point for the HP method performance relative to the Hallberg method is not constant for all levels of precision. The HP and Hallberg parameters used in these tests were specifically chosen not only to provide an additional data point to the analysis of the previous section, but to illustrate the claim that the number of summands needed to achieve performance parity drops as precision is increased. This particular choice of method parameters for 384 bits of precision and 32M summands yielded relative performance bounded within a small constant factor across all four architectures. More operands, and thus a lower Hallberg M value selection, would be needed for the HP method to consistently outperform the Hallberg method at this precision.

The second observation is that the actual performance of the two techniques are dependent not only on the choice of method parameters, but also on the compilers used to build the code and architectures upon which they run. Identical implementations of the HP and Hallberg methods were used in the OpenMP and Xeon Phi, for example. Yet, simply employing different compilers (GNU versus Intel) and parallel architectures yielded significantly different performance characteristics. This illustrates the difficulties in designing algorithms which yield high performance on any architecture, as well as the weakness of bounding algorithm complexity by arithmetic operations counts alone, which is the traditional technique used to compare the various high-precision floating point methods. As the CUDA GPU results show, memory latencies and memory access patterns are relevant, and accurately accounting for these resource utilizations can greatly complicate the asymptotic analysis.

V. Final Remarks

This research presented a new computational method for adding large numbers of floating point values to produce an invariant sum. The sum is invariant both to the order of summation and underlying architecture, due to the method's reliance on integer arithmetic. The method is simple enough to allow atomic updates of values, and it was successfully demonstrated in several parallel environments in common use today.

With the anticipated convergence of exaflops high-performance computing and exabyte big data analytics on hybrid architectures [18, 20], global reduction of a very large set of floating point data is expected to become a norm. In this setting, computational reproducibility will become an increasingly more important and difficult problem that it is today. Use of numerical techniques such as the method proposed here can help mitigate the impact of error and variation within simulations and data analytics at these extreme scales.

One flaw with this technique is the reliance on the user knowing the range of real numbers to be summed, and tailoring the HP parameters N and k appropriately to ensure enough precision exists. An opportunity for future research is to extend the HP method to adaptively adjust precision at

runtime to accommodate any range of real numbers that may be encountered.

ACKNOWLEDGMENT

This research was partially supported by the U.S. Department of Energy grant DE-SC0014607 and the USC Annenberg Graduate Fellowship Program. Computation for the work described in this paper was supported by the University of Southern California's Center for High-Performance Computing (hpc.usc.edu).

REFERENCES

[1] IEEE, IEEE Standard for Floating-Point Arithmetic, Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008.

[2] J. Blanck, Exact real arithmetic using centred intervals and bounded error terms, The Journal of Logic and Algebraic Programming, 66(1):50-67, January 2006.

[3] H.-J. Boehm, R. Cartwright, M. Riggle, and M. O'Donnell, Exact real arithmetic: a case study in higher order programming, ACM Symposium on Lisp and Functional Programming, pp. 162–173, 1986.

[4] K. Briggs, Implementing exact real arithmetic in python, C++ and C, Theoretical Computer Science, 351(1):74-81, 2006.

[5] S. Collange, D. Defour, S. Graillat, R. Iakymchuk, Numerical reproducibility for the parallel reduction on multi- and many-core architectures. 2015, <hal-00949355v3>.

[6] J. Demmel and H.D. Nguyen, Fast reproducible floating-point summation, 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA, April 2013.

[7] J. Demmel and H.D. Nguyen, Numerical accuracy and reproducibility at exascale, 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA, April 2013.

[8] J. Demmel and H.D. Nguyen, Parallel reproducible summation, IEEE Transactions on Computers, 64(7):2060-2070, 2015.

[9] T. Granlund and the GMP development team, GNU MP: The GNU Multiple Precision Arithmetic Library, 5.0.5 edition, 2012. http://gmplib.org/.

[10] J. Gustafson, The End of Error: Unum Computing, Chapman and Hall/CRC, 2015.

[11] R. Hallberg and A. Adcroft, An order-invariant real-to-integer conversion sum, Parallel Computing, 40(5-6):140-143, 2014.

[12] Y. He and C. H.Q. Ding, Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications, Journal of Supercomputing, 18:259-277, March 2001.

[13] N.J. Higham, The accuracy of floating point summation, SIAM Journal on Scientific Computing, 14:783-799, 1993.

[14] Intel, Intel 64 and IA-32 Architectures Optimization Reference Manual, September 2014. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architecturesoptimization-manual.pdf.

[15] W. Kahan, Pracniques: Further remarks on reducing truncation errors. Communications of the ACM, 8(1):40, January 1965.

[16] D.E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd ed., Addison-Wesley, 1997.

[17] V. Ménissier-Morain, Arbitrary precision real arithmetic: design and algorithms, The Journal of Logic and Algebraic Programming, 64(1):13-39, July 2005.

[18] B.H. Obama, U.S. Presidential Executive Order — Creating a National Strategic Computing Initiative, July 29, 2015.

[19] D.M. Priest, Algorithms for arbitrary precision floating point arithmetic. In Proceedings of the 10th Symposium on Computer Arithmetic, pp. 132-145, IEEE Computer Society Press, 1991.

[20] D.A. Reed and J. Dongarra, Exascale Computing and Big Data: The Next Frontier, Communications of the ACM, 58(7):56-68, 2015.

[21] S.M. Rump, Ultimately fast accurate summation, SIAM Journal on Scientific Computing, 31(5):3466-3502, 2009.