Parallel Reproducible Summation

James Demmel
Mathematics Department and CS Division
University of California at Berkeley
Berkeley, CA 94720
demmel@eecs.berkeley.edu

Hong Diep Nguyen
EECS Department
University of California at Berkeley
Berkeley, CA 94720
hdnguyen@eecs.berkeley.edu

***Abstract*—Reproducibility, i.e. getting bitwise identical floating point results from multiple runs of the same program, is a property that many users depend on either for debugging or correctness checking in many codes [10]. However, the combination of dynamic scheduling of parallel computing resources, and floating point nonassociativity, makes attaining reproducibility a challenge even for simple reduction operations like computing the sum of a vector of numbers in parallel. We propose a technique for floating point summation that is reproducible independent of the order of summation. Our technique uses Rump's algorithm for error-free vector transformation [7], and is much more efficient than using (possibly very) high precision arithmetic. Our algorithm reproducibly computes highly accurate results with an absolute error bound of $n \cdot 2^{-28} \cdot macheps \cdot \max |v_i|$ at a cost of $7n$ FLOPs and a small constant amount of extra memory usage. Higher accuracies are also possible by increasing the number of error-free transformations. As long as all operations are performed in to-nearest rounding mode, results computed by the proposed algorithms are reproducible for any run on any platform. In particular, our algorithm requires the minimum number of reductions, i.e. 1 reduction of an array of 6 double precision floating point numbers per sum, and hence is well suited for massively parallel environments.**

I. Introduction

Given current hardware trends, computing performance is improved by using more processors, for example from multi-core platform to many-core platform, as well as distributed-memory systems, and more recently the cloud computing (eg map-reduce) environment. Exascale computing (10^{18} floating point operations per second) is projected to be available in less than a decade, achieved by using a huge number of processors, of order 10^9. Given the likely hardware heterogeneity in both platform and network, and the possibility of intermittent failures, dynamic scheduling will be needed to adapt to changing resources and loads. This will make it likely that repeated runs of a program will not execute operations like reductions in exactly the same order. This in turn will make reproducibility, i.e. getting bitwise identical results from run to run, difficult to achieve, because floating point operations like addition are not associative, so computing sums in different orders often leads to different results. Indeed, this is already a challenge on today's platforms.

Reproducibility is of interest for a number of reasons. First, it is hard to debug if runs with errors cannot be reproduced. Second, reproducibility is important, and sometimes required, for some applications. For example, in climate and weather modeling, N-body simulation, or other forward unstable simulations, a very small change in results at one time step can lead to a very different result at a later time step. Reproducibility is sometimes also required for contractual reasons where both sides need to agree on the results of the same computation.

We note that reproducibility and accuracy are not synonymous. It is natural to consider just using a standard algorithm at higher precision, say double the precision of the summands ("working precision"). It is true that this makes the probability of nonreproducibility much lower than using working precision. But it does not guarantee reproducibility, in particular for ill-conditioned inputs, or when the result is close to half-way between two floating point numbers in the output precision [9]. And ill-conditioned inputs, i.e. a tiny sum resulting from a lot of cancellation, may be the most common case in many applications. For example, when solving $Ax = b$ using an iterative method, the goal is to get as much cancellation in the residual $r = Ax - b$ as possible.

Our goal is to present an algorithm for summations with the following properties: (1) It computes a reproducible sum independent of the order of the summands, how they are assigned to processors, or how they are aligned in memory. (2) It makes only basic assumptions about the underlying arithmetic (a subset of the IEEE Standard 754-2008 specified below). (3) It scales as well as a performance-optimized, non-reproducible implementation, as n (number of summands) and p (number of processors) grow. (4) The user can choose the desired accuracy of the result. In particular, getting a reproducible result with about the same accuracy as the performance optimized algorithm should only be a small constant times slower, but higher accuracy is possible too.

Communication, i.e. moving data between processors, is the most expensive operation on computers today such as distributed memory systems and cloud computing environments, much more expensive than arithmetic, and hardware trends will only make this performance gap larger on future architectures, including Exascale. This means that to achieve goal (3) above, our algorithm may only do a single reduction operation across the machine, and each operand in the reduction tree may only be a small constant factor bigger than a single floating point number. Furthermore, we can make no assumption about the size

or shape of the reduction tree.

In a related paper [4] we introduced two other algorithms for reproducible summation, that require $(3\mathtt{K}-1)n$ FLOPs [1] and $4\mathtt{K}n$ FLOPs respectively where n is the input size and $\mathtt{K}$ is the number of passes of the error-free transformation that can be chosen by the user in order to obtain the required accuracy. Both these algorithms produce reproducible results with an error bound of order

$$(\mathtt{K} + c_1 \cdot n \cdot \epsilon + c_2 \cdot \mathtt{K} \cdot n^{\mathtt{K}+1} \cdot \epsilon^{\mathtt{K}-1}) \cdot \epsilon \cdot \max|v_i| \quad (1)$$

where ϵ is the machine epsilon, and c_1, c_2 are small constants. Meanwhile the error bound of a standard summation algorithm is provided by the following theorem.

Theorem 1 (Error bound of standard sum)**.** *[5, Section 4.2] Let s be result computed by a standard summation algorithm. Then*

$$|s - \sum_{i=1}^{n} v_i| < (n-1) \cdot \epsilon \cdot \sum_{i=1}^{n} |v_i| + \mathcal{O}(\epsilon^2). \quad (2)$$

The goal of this paper is to provide techniques to compute a reproducible sum which is almost of the same accuracy as the conventional sum, i.e. having an error bound of the same order as (2) or better.

The error of both algorithms presented in [4] depends strongly on the input size, i.e. proportional to $n^{\mathtt{K}+1}$. Therefore the assumption $n \cdot \epsilon \ll 1$ in some cases would not be sufficient to obtain reasonably accurate results. When n is small so that $n^{\mathtt{K}} \cdot \epsilon^{\mathtt{K}-1} \leq 1$, then the error bound provided by (1) is at most about $c_2 \cdot \mathtt{K} \cdot n \cdot \epsilon \cdot \max|v_i|$ which can be better then the standard summation's error bound. Nonetheless, if n is so large that $n^{\mathtt{K}} \cdot \epsilon^{\mathtt{K}-1} \gg 1$ for a given $\mathtt{K}$, then the error bound (1) will be $\gg c_2 \cdot \mathtt{K} \cdot n \cdot \epsilon \cdot \max|v_i|$ which can be much worse than the standard summation's error bound. As long as $n \cdot \epsilon < 1$, this can be remedied by increasing $\mathtt{K}$ at a cost of lowering performance.

More importantly, both these two algorithms require two reduction operations. It means that they could not scale as well as an optimized non-reproducible summation when the number of processors p grows. In this paper, we introduce a new technique to obtain reproducibility which both exhibits an improved error bound that is proportional to n instead of $n^{\mathtt{K}+1}$, and requires only one reduction operation in order to minimize communication when running on large-scale systems. As will be seen in Section V, the slowdown of our proposed algorithm in comparison with the performance-optimized Intel MKL[2] library is only 20% in terms of running time when both algorithms are run on 1024 processors.

The paper is organized as follows. Section II presents some notation and numerical analysis that will be used throughout the paper. Section III discusses some related work as well as our previously proposed pre-rounding technique to obtain reproducibility. Section IV presents our new technique to compute the reproducible sum using only one reduction operation. Section V contains some experimental results and Section VI contains conclusions as well as future work.

[1] Floating-point Operations

II. Notation and background

Denote by $\mathbb{R}, \mathbb{F}, \mathbb{Z}$ the set of real numbers, floating-point numbers and integers respectively.

In this paper we assume that the floating-point arithmetic in use complies with the IEEE 754-2008 standard with five rounding modes: rounding to nearest even, rounding to nearest with tie-breaking away from 0, rounding toward 0, rounding toward $+\infty$, and rounding toward $-\infty$. To distinguish them from the two rounding to nearest modes, the last 3 rounding modes are called directed roundings.

Let $f = s \times 2^e \times m \in \mathbb{F}$ be a floating-point number represented in IEEE 754 format, where $s = \pm 1$ is the sign, $\mathtt{emax} \geq e \geq \mathtt{emin}$ is the exponent (usually referred to as $\mathtt{exponent}(f)$), p is the precision, and $m = m_0.m_1 m_2 \ldots m_{p-1}, m_i \in \{0, 1\}$, is the significand of f. f is said to be normalized if $m_0 = 1$ and $e > \mathtt{emin}$. $f = 0$ if all $m_i = 0$ and $e = \mathtt{emin}$. $f \neq 0$ is said to be subnormal if $m_0 = 0$ and $e = \mathtt{emin}$.

The unit in the last place, denoted by $\mathtt{ulp}(f)$, represents the spacing between two consecutive floating-point numbers of the same exponent e: $\mathtt{ulp}(f) = 2^e \times 2^{1-p} = 2^{e-(p-1)}$.

The unit in the first place, denoted by $\mathtt{ufp}(f)$, represents the first significant bit of a floating-point number: $\mathtt{ufp}(f) = 2^e$. Obviously we have: $\mathtt{ufp}(f) \leq |f| \leq 2\mathtt{ufp}(f) - \mathtt{ulp}(f)$ if f is normalized and $f \neq 0$.

Machine epsilon ϵ is the spacing between 1 and the next smaller floating-point number: $\epsilon = 2^{-p}$.

The unit roundoff $\mathbf{u}$ is the upper bound on the relative error due to rounding. $\mathbf{u} = \epsilon$ for rounding to nearest, and $\mathbf{u} = 2\epsilon$ for directed rounding.

$\mathtt{fl}(\cdot)$ denotes the evaluated result of an expression in floating-point arithmetic.

For any normalized floating-point number $f \in \mathbb{F}$, and $r \in \mathbb{R}$ we have the following properties:

(a) $|r - \mathtt{fl}(r)| < \mathtt{ulp}(\mathtt{fl}(r)) < 2\epsilon|\mathtt{fl}(r)|$,
(b) $|r - \mathtt{fl}(r)| \leq \mathbf{u} \times |\mathtt{fl}(r)|$ and $|r - \mathtt{fl}(r)| \leq \mathbf{u} \times |r|$,
(c) $\frac{1}{2}\epsilon^{-1}\mathtt{ulp}(f) = \mathtt{ufp}(f) \leq |f| < \epsilon^{-1}\mathtt{ulp}(f) = 2\mathtt{ufp}(f)$.

Another way to represent f is $f = M * \mathtt{ulp}(f)$, $M \in \mathbb{Z}$. $2^{p-1} \leq |M| < 2^p$ for normalized numbers and $0 < |M| < 2^{p-1}$ for subnormal numbers. Moreover, for any $n \in \mathbb{Z}$ and $|n| < 2^p = \epsilon^{-1}$ then $n * \mathtt{ulp}(f) \in \mathbb{F}$. Let $x \in \mathbb{R}$ be a real number, denote $x\mathbb{Z} = \{n * x \mid n \in \mathbb{Z}\}$. If $a, b \in x\mathbb{Z}$ then $a \pm b \in x\mathbb{Z}$, letting us deduce the following lemmas.

Lemma 1. *Let $f, g \in \mathbb{F}$. If $|g| \geq |f|$ then $g \in \mathtt{ulp}(f)\mathbb{Z}$.*

Lemma 2. *Let $f, x, y \in \mathbb{F}$ where $x, y \in \mathtt{ulp}(f)\mathbb{Z}$. If $|x + y| < \epsilon^{-1}\mathtt{ulp}(f)$ then $x + y \in \mathbb{F}$, i.e. $x + y$ can be computed exactly.*

Let $x, y \in \mathbb{F}$. If there is no overflow or underflow in evaluating $\mathtt{fl}(x \circ y)$, $\circ \in \{+, -, \times, /\}$, then

$$\mathtt{fl}(x \circ y) = (x \circ y)(1 + \delta), \quad |\delta| \leq \mathbf{u} \quad (3)$$

which is the standard model for numerical analysis of an algorithm.

Overflow and underflow can impact the reproducibility. A floating point computation can overflow or not depending on the order of evaluation. Consider for example the two following computations which basically evaluate the same mathematical expression: (1) $(\texttt{MAX\_DOUBLE} - \texttt{MAX\_DOUBLE}) + \texttt{MAX\_DOUBLE}$ and (2) $(\texttt{MAX\_DOUBLE} + \texttt{MAX\_DOUBLE}) - \texttt{MAX\_DOUBLE}$, where MAX_DOUBLE is the maximum value that is representable by a double precision floating point number. Meanwhile there is no overflow as well as no rounding error in evaluating (1), an overflow clearly occurs in evaluating (2). This leads to nonreproducibility.

If subnormal numbers are supported then underflow does not impact the reproducibility. However, in case of no subnormal number support underflow can lead to nonreproducibility depending on the order of evaluation as can be seen in the two following mathematically equal expressions: (1) $(\texttt{MIN\_DOUBLE} + 1.5 * \texttt{MIN\_DOUBLE}) - \texttt{MIN\_DOUBLE}$ and (2) $\texttt{MIN\_DOUBLE} + (1.5 * \texttt{MIN\_DOUBLE} - \texttt{MIN\_DOUBLE})$, where MIN_DOUBLE is the minimum postive value that can be represented by a normal double precision floating point number. No rounding error and no underflow occurs in evaluating (1), hence (1) gives $1.5 * \texttt{MIN\_DOUBLE}$. However, underflow does occur in evaluating (2) and leads to a different computed result which is MIN_DOUBLE.

Therefore, special handling is required for overflow and underflow [4]. For all the numerical analyses throughout the paper, we assume that there is no overflow or underflow during execution.

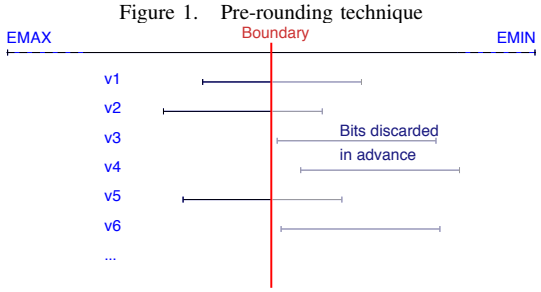
III. Pre-rounding technique

First we discuss some simple but inadequate ways to achieve reproducibility. The first solution is to use a deterministic computation order by fixing the number of processors as well as the data assignment and reduction tree shape. On today's large machines, let alone at Exascale, such a deterministic computation will lead to substantial communication overhead due to synchronization. Moreover, a deterministic computation order also requires a fixed data order, not achieving goal (1).

Second, reproducibility can be obtained by eliminating rounding error, to ensure associativity. This can be done using exact arithmetic [8] or correctly-rounded algorithms. It will however increase substantially the memory usage as well as the amount of communication when applied to more complicated operations such as matrix multiplication. Third, fixed-point arithmetic can also be used, but at a cost of limited argument range.

Instead of these, we proposed a technique, called pre-rounding, to obtain deterministic rounding errors. The same technique is found in [6], [7] to compute an accurate floating-point summation. We will first present two algorithms which can also be found in [4]: Algorithm 2 which costs $4n$ FLOPs in to-nearest rounding mode, and Algorithm 3 which costs only $3n$ FLOPs but must be performed in directed-rounding mode. Then we will present a new algorithm, Algorithm 5, which is performed in to-nearest rounding mode and costs in total $3n$ floating point additions and n floating point OR-bit operations. On some platform, for example on Intel Sandy Bridge processors, the practical cost of Algorithm 5 is equivalent to only $3n$ FLOPs.

Figure 1. Pre-rounding technique



The main idea of the pre-rounding technique is to round input values to a common base according to some boundary (Figure 1) so that there will be no rounding error in summing the remaining (leading) parts. Then the error depends only on input values and the boundary, not on the intermediate results, which depend on the order of computation. Thus the computed result will be reproducible so long as the boundary is reproducible.

The splitting process can be done by directly manipulating the exponent and the mantissa of the floating-point number, or by using the splitting algorithm proposed by Rump [6], [7], which is similar to the FastTwoSum algorithm proposed by Dekker [3].

Algorithm 1 Extract scalar: $[S, q, r] = \texttt{Split}(M, x)$

Require: $M, x \in \mathbb{F}, M \geq |x|$. All operations are performed in to-nearest rounding mode.

1: $S = \texttt{fl}(M + x)$
2: $q = \texttt{fl}(S - M)$
3: $r = \texttt{fl}(x - q)$

Ensure: $x = q + r$ and $S + r = M + x$

Algorithm 1 is called an error-free transformation in to-nearest rounding mode because it computes the sum of $S = \texttt{fl}(M + x)$ together with its rounding error $r = (M + x) - \texttt{fl}(M + x)$. In other words the floating-point number x has been split into two parts q and r which contains respectively the leading bits and the trailing bits of x.

Theorem 2 (Algorithm 1 in rounding to nearest [4])**.** *Let $S, q, r \in \mathbb{F}$ be the results computed by Algorithm 1. Then $q = S - M$, $x = q + r$, $M + x = S + r$, $q \in \frac{1}{2}\texttt{ulp}(M)\mathbb{Z}$, and $|r| \leq \frac{1}{2}\texttt{ulp}(S) \leq \texttt{ulp}(M) \leq 2\mathbf{u}M$.*

According to Theorem 2, the extracted leading part q is a multiple of $\frac{1}{2}\texttt{ulp}(M)$, meanwhile the magnitude of the trailing part r is bounded by $\texttt{ulp}(M)$. It means that the first bit of r might overlap with the last significant bit of q. In order to have a disjoint extraction, a stronger assumption on the value of M is needed.

Corollary 1 (Disjoint splitting)**.** *Let $S, q, r \in \mathbb{F}$ be the results computed by Algorithm 1. If $\mathtt{ulp}(S) = \mathtt{ulp}(M)$ then $q \in \mathtt{ulp}(M)\mathbb{Z}$, and $|r| \leq \frac{1}{2}\mathtt{ulp}(M)$.*

Proof: The proof is straightforward from Theorem 2 and that $S \in \mathtt{ulp}(M)\mathbb{Z}$. ∎

Corollary 2. *If $M = 3 \cdot 2^k, k \in \mathbb{N}$ and $|x| \leq 2^k - \mathtt{ulp}(M)$ then $x = q + r$, $q \in \mathtt{ulp}(M)\mathbb{Z}$, and $|r| \leq \frac{1}{2}\mathtt{ulp}(M)$.*

Proof: Since $|x| \leq 2^k - \mathtt{ulp}(M)$ we have $2^{k+1} + \mathtt{ulp}(M) \leq M + x \leq 2^{k+2} - \mathtt{ulp}(M)$. Therefore $2^{k+1} + \mathtt{ulp}(M) \leq S \leq 2^{k+2} - \mathtt{ulp}(M)$. This means $\mathtt{ulp}(S) = \mathtt{ulp}(M)$. The proof follows from Corollary 1. ∎

Given an input vector $v = [v_1, \ldots, v_n]$, each element v_i is split using the same M into the leading part $q_i \in \mathtt{ulp}(M)\mathbb{Z}$ and the trailing part $r_i \leq \frac{1}{2}\mathtt{ulp}(M)$. According to Lemma 2 if M is big enough so that $\sum_1^n |q_i| < M$ then the sum of leading parts q_i can be computed exactly. In other words, the input vector v has been transformed exactly into $T = \sum_1^n q_i$ and another vector of trailing parts $r = [r_1, \ldots, r_n]$ where $\sum_1^n v_i = T + \sum_1^n r_i$.

According to Corollary 2 we have $|q_i| = |v_i - r_i| \leq |v_i| + |r_i| \leq |v_i| + \frac{1}{2}\mathtt{ulp}(M)$. So $\sum_1^n |q_i| \leq \sum_1^n |v_i| + n \cdot \frac{1}{2}\mathtt{ulp}(M) \leq n \cdot (\max|v_i| + \frac{1}{2}\mathtt{ulp}(M))$. Since $\mathtt{ulp}(M) = 2\epsilon \cdot \mathtt{ufp}(M)$, if $\mathtt{ufp}(M) \geq n \cdot \max|v_i|/(1 - n * \epsilon)$ then $\mathtt{ufp}(M) \geq \sum_1^n |q_i|$. This leads to the Algorithm 2 [4].

Algorithm 2 Error-free vector transformation: $[T, r] = \mathtt{ExtractVector}(M, v)$

Require: v is a vector of n floating-point numbers, $M = 1.5 \cdot 2^k, k \in \mathbb{Z}$ and $2^k \geq n \cdot \max|v_i|/(1 - n \cdot \epsilon)$. All operations are performed in rounding to nearest even mode.

1: $T = 0$
2: **for** $i = 1$ to n **do**
3: $\quad [S, q_i, r_i] = \mathtt{Split}(M, v_i)$
4: $\quad T = \mathtt{fl}(T + q_i)$
5: **end for**

Ensure: T is the exact sum of high order parts q_i, r is the vector of low order parts r_i.

Since the sum of high order parts $T = \sum_1^n q_i$ is exact, it does not depend on the order of evaluation. Therefore the result computed by the Algorithm 2 will be reproducible so long as M is reproducible. This can be achieved by using $M = 1.5 \cdot 2^{\lceil \log_2(\delta) \rceil}$ where $\delta = n \cdot \max|v_i|/(1 - n \cdot \epsilon)$ which is always reproducible since the maximum operation is associative.

As observed by Rump in [7], M does not need to be the same for each iteration. We can instead use the updated value $M_i = \mathtt{fl}(M_{i-1} + v_i)$ to split the subsequent element v_{i+1}. Since $M_{i-1} + v_i$ is computed without rounding error, the sum of leading parts $\sum_1^n q_i$ can be obtained by a simple subtraction $M_n - M_0$. According to [4], in order to obtain reproducibility beside the requirement of reproducible M, two additional requirements must also be satisfied:

- All M_i must have the same unit in the last place. This can be satisfied by using $M = 3 \cdot 2^k$ where $2^k > n \cdot \max|v_i|/(1 - 2n \cdot \epsilon)$.
- All operations must be performed in a directed-rounding mode to avoid the midpoint issue (the computed result is exactly halfway between two consecutive floating-point number).

This leads to the second improved algorithm for error-free vector transformation, which is reproducible as long as M is reproducible.

Algorithm 3 Error-free vector transformation: $[T, r] = \mathtt{ExtractVector2}(M, v)$

Require: v is a vector of n floating-point numbers, $M = 3 \cdot 2^k, k \in \mathbb{Z}$ and $2^k \geq n \cdot \max|v_i|/(1 - n \cdot \epsilon)$. All operations are performed in the same directed rounding mode.

1: $M_0 = M$
2: **for** $i = 1$ to n **do**
3: $\quad [M_i, q_i, r_i] = \mathtt{Split}(M_{i-1}, v_i)$
4: **end for**
5: $T = \mathtt{fl}(M_n - M_0)$

Ensure: T is a reproducible sum of all elements of v_i, r is the vector of low order parts of v_i.

Algorithm 3 requires only $3n$ FLOPs, which is 25% fewer than Algorithm 2. Nonetheless, Algorithm 3 requires the use of one directed rounding mode which is not always preferable. First, the default rounding mode on most processors is to nearest even. Second, in some cases the to-nearest rounding mode is obligatory as will be seen in the Section IV. According to [4] the use of directed rounding mode avoids the midpoint issue which leads to non-deterministic rounding in to-nearest-even rounding mode. In fact, this issue can be avoided by simply setting the last bit of x to 1. This lead to a new algorithm which is presented in Algorithm 4 to split a floating point number.

Algorithm 4 Extract scalar: $[S, q, r] = \mathtt{Split2}(M, x)$

Require: $M, x \in \mathbb{F}, M > |x|$. All operations are performed in to-nearest rounding mode. $\mathtt{MASK} = \mathtt{0\ldots01}$ is a bit mask which is of the same precision as x and whose bit representation is full of 0 except for the last bit which is 1.

1: $\bar{x} = x$ OR MASK ▷ Set last bit of x to 1
2: $S = \mathtt{fl}(M + \bar{x})$
3: $q = \mathtt{fl}(S - M)$
4: $r = \mathtt{fl}(x - q)$

Ensure: $x = q + r$ and $S + r = M + x$

Theorem 3. *Let $S, q, r \in \mathbb{F}$ be the results computed by Algorithm 4. Then $q = S - M$, $x = q + r$, $M + x = S + r$, $q \in \frac{1}{2}\mathtt{ulp}(M)\mathbb{Z}$, and $|r| \leq \mathtt{ulp}(M) + \mathtt{ulp}(x)$.*

Proof: Let $d = \bar{x} - x$. Since $\bar{x}$ is computed by setting the last bit of x to 1, it is easy to see that $d \in \{-\mathtt{ulp}(x), 0, \mathtt{ulp}(x)\}$ and $\mathtt{ulp}(\bar{x}) = \mathtt{ulp}(x)$. Therefore $|\bar{x}| \leq M$. Let $\bar{r} = \mathtt{fl}(\bar{x} - q)$, according to Theorem 2

we have $q = S - M$, $\bar{x} = q + \bar{r}$, $q \in \frac{1}{2}\mathtt{ulp}(M)\mathbb{Z}$, and $|\bar{r}| \le \frac{1}{2}\mathtt{ulp}(S) \le \mathtt{ulp}(M)$.

Since $M \ge |x|$, we have $M \in \mathtt{ulp}(x)\mathbb{Z}$. Hence $S = \mathtt{fl}(M + \bar{x}) \in \mathtt{ulp}(x)\mathbb{Z}$, $q = S - M \in \mathtt{ulp}(x)\mathbb{Z}$, $\bar{r} = \bar{x} - q \in \mathtt{ulp}(x)\mathbb{Z}$, and $x - q \in \mathtt{ulp}(x)\mathbb{Z}$. Let's consider two cases of the value of $\bar{r}$:

1) $|\bar{r}| < 2\mathtt{ufp}(x)$. This implies $|\bar{r}| \le 2\mathtt{ufp}(x) - \mathtt{ulp}(x)$. $|x - q| = |\bar{x} - d - q| = |\bar{r} - d| \le |\bar{r}| + \mathtt{ulp}(x) \le 2\mathtt{ufp}(x)$. Therefore $r = \mathtt{fl}(x - q) = x - q$ and so $S + r = M + x$ as desired.
2) $|\bar{r}| \ge 2\mathtt{ufp}(x)$. Therefore $\bar{r} \in 2\mathtt{ulp}(x)\mathbb{Z}$, and $\epsilon^{-1}\mathtt{ulp}(x) = 2\mathtt{ufp}(x) \le |\bar{r}| \le \mathtt{ulp}(M)$. Since $q \in \frac{1}{2}\mathtt{ulp}(M)\mathbb{Z}$, we have $q \in 2\mathtt{ulp}(x)\mathbb{Z}$. Hence $\bar{x} = q + \bar{r} \in 2\mathtt{ulp}(x)\mathbb{Z}$, i.e. $\bar{x} \in 2\mathtt{ulp}(\bar{x})\mathbb{Z}$. Since the last bit of $\bar{x}$ is 1, this case is invalid.

Therefore we always have $r = x - q$, i.e. $x = q + r$. Consequently $|r| \le |\bar{r}| + |d| \le \mathtt{ulp}(M) + \mathtt{ulp}(x)$. ∎

Corollary 3. *Let $S, q, r \in \mathbb{F}$ be the results computed by Algorithm 4. If $\mathtt{ulp}(S) = \mathtt{ulp}(M)$ and $\mathtt{ulp}(x) < \frac{1}{2}\mathtt{ulp}(M)$ then $q \in \mathtt{ulp}(M)\mathbb{Z}$, and $|r| \le \frac{1}{2}\mathtt{ulp}(M)$.*

Proof: Let $\bar{r} = \bar{x} - q$. According to Corollary 1 we have $q \in \mathtt{ulp}(M)\mathbb{Z}$ and $|\bar{r}| \le \frac{1}{2}\mathtt{ulp}(M)$. Since $\mathtt{ulp}(\bar{x}) < \frac{1}{2}\mathtt{ulp}(M)$ and the last bit of $\bar{x}$ is 1, $M + x$ cannot be a midpoint. This means that $|(M+x) - \mathtt{fl}(M+x)| < \frac{1}{2}\mathtt{ulp}(\mathtt{fl}(M + x))$, i.e. $|\bar{r}| < \frac{1}{2}\mathtt{ulp}(M)$. Since $\mathtt{ulp}(x) < \frac{1}{2}\mathtt{ulp}(M)$, $\frac{1}{2}\mathtt{ulp}(M) \in \mathtt{ulp}(x)\mathbb{Z}$. Therefore $|\bar{r}| + \mathtt{ulp}(x) \le \frac{1}{2}\mathtt{ulp}(M)$. Moreover $|r| = |\bar{r} - (\bar{x} - x)| \le |\bar{r}| + \mathtt{ulp}(x)$. Consequently $|r| \le \frac{1}{2}\mathtt{ulp}(M)$. The proof is complete. ∎

Corollary 4. *Let $M_1, M_2, x \in \mathbb{F}$, and $[S_1, q_1, r_1] = \mathtt{Split2}(M_1, x)$, and $[S_2, q_2, r_2] = \mathtt{Split2}(M_2, x)$. If $2\mathtt{ulp}(x) < \mathtt{ulp}(M_1) = \mathtt{ulp}(M_2)$ then $q_1 = q_2$ and $r_1 = r_2$.*

Proof: Let $\bar{r}_1 = \bar{x} - q_1$, and $\bar{r}_2 = \bar{x} - q_2$. From the proof of Corollary 3 we have $q_1 \in \mathtt{ulp}(M_1)\mathbb{Z}$, $q_2 \in \mathtt{ulp}(M_2)\mathbb{Z}$, and $|\bar{r}_1| < \frac{1}{2}\mathtt{ulp}(M_1)$, $|\bar{r}_2| < \frac{1}{2}\mathtt{ulp}(M_2)$. Since $q_1 + \bar{r}_1 = q_2 + \bar{r}_2$, it is easy to deduce that $q_1 = q_2$. Therefore $r_1 = x - q_1 = x - q_2 = r_2$. ∎

Setting last bit of v_i can be done using an OR-bit operation. Algorithm 4 costs 4 FLOPs, counting OR as a FLOP. Nonetheless, as will be seen in Section V, on some processors for example the Intel Sandy Bridge the floating point OR-bit operation can be executed in parallel with the floating point addition operation. In that case, the OR-bit operation is not counted, and so the cost of Algorithm 4 is only 3 FLOPs.

Corollary 4 means that, if $\mathtt{ulp}(M)$ is reproducible then the splitting is reproducible. This leads to Algorithm 5 to exactly transform an input vector, which is reproducible when M is reproducible.

Theorem 4. *Let $S \in \mathbb{F}, r \in \mathbb{F}^n$ be results computed by Algorithm 5. Then $(S - M) + \sum_1^n r_i = \sum_1^n v_i$, $|q_i| \le 2^{k - \lceil \log_2 n \rceil}$, and $\sum_1^n |r_i| \le \frac{n}{2} \cdot \mathtt{ulp}(M)$.*

Proof: Since $6 \cdot 2^k \le M < 7 \cdot 2^k$, we have $\mathtt{ulp}(M) = 2^{k+3} \cdot \epsilon^{-1} < 2^{k - \lceil \log_2 n \rceil}$ and $M \le 7 \cdot 2^k - \mathtt{ulp}(M)$.

Algorithm 5 Error-free vector transformation: $[S, r] = \mathtt{ExtractVector3}(M, v)$

Require: v is a vector of size $n \ll \epsilon^{-1}$. $M \in \mathbb{F}$: $6 \cdot 2^k \le M < 7 \cdot 2^k, k \in \mathbb{Z}$ and $\max|v_i| \le 2^{k - \lceil \log_2 n \rceil} = \mathtt{ufp}(M) \cdot 2^{-2 - \lceil \log_2 n \rceil}$. All operations are performed in to-nearest rounding mode.

```
M_0 = M
for i = 1 to n do
    [M_i, q_i, r_i] = Split2(M_{i-1}, v_i)
end for
S = M_n
```

Ensure: $S - M$ is a reproducible sum of leading parts of v_i, r is the vector of low order parts of v_i.

We will prove by induction that $M_i - M = \sum_1^i q_j$, $\mathtt{ulp}(M_i) = \mathtt{ulp}(M)$, and $|q_i| \le 2^{k - \lceil \log_2 n \rceil}$ for all $1 \le i \le n$. This is true for $i = 0$. Suppose this is true for $1 \le i < n$ we will prove it is also true for $i + 1$. We have $M_i \pm 2^{k - \lceil \log_2 n \rceil} = M + \sum_1^i q_j \pm 2^{k - \lceil \log_2 n \rceil} \rightarrow |M_i + 2^{k - \lceil \log_2 n \rceil} - M| \le (i + 1)2^{k - \lceil \log_2 n \rceil} \le 2^k$. In addition, we have $6 \cdot 2^k \le M \le 7 \cdot 2^k - \mathtt{ulp}(M)$. Therefore $5 \cdot 2^k \le M_i \pm 2^{k - \lceil \log_2 n \rceil} \le 8 \cdot 2^k - \mathtt{ulp}(M)$. It is easy to see that $M_i \pm 2^{k - \lceil \log_2 n \rceil} \in \mathtt{ulp}(M)\mathbb{Z}$, so $M_i \pm 2^{k - \lceil \log_2 n \rceil} \in \mathbb{F}$ and $\mathtt{ulp}(M_i \pm 2^{k - \lceil \log_2 n \rceil}) = \mathtt{ulp}(M)$. Since $|v_{i+1}| \le 2^{k - \lceil \log_2 n \rceil}$ we have $M_i - 2^{k - \lceil \log_2 n \rceil} \le M_i + v_{i+1} \le M_i + 2^{k - \lceil \log_2 n \rceil}$. Moreover $M_{i+1} = \mathtt{fl}(M_i + v_{i+1})$, therefore $M_i - 2^{k - \lceil \log_2 n \rceil} \le M_{i+1} \le M_i + 2^{k - \lceil \log_2 n \rceil}$. By consequence $\mathtt{ulp}(M_{i+1}) = \mathtt{ulp}(M)$, and $|q_{i+1}| = |M_{i+1} - M_i| \le 2^{k - \lceil \log_2 n \rceil}$.

According to Corollary 3 we have $v_j = q_j + r_j$, $q_j \in \mathtt{ulp}(M)\mathbb{Z}$ and $|r_j| \le \frac{1}{2}\mathtt{ulp}(M)$ for all $1 \le j \le i$. Therefore $\sum_1^n |r_i| \le \frac{n}{2} \cdot \mathtt{ulp}(M)$, and $\sum_1^n v_i = \sum_1^n q_i + \sum_1^n r_i = M_n - M + \sum_1^n r_i$, i.e. $(S - M) + \sum_1^n r_i = \sum_1^n v_i$. The proof is complete.

∎

According to Theorem 4 $\mathtt{ulp}(M_i) = \mathtt{ulp}(M)$ for all $i = 1, \ldots, n$. Therefore Algorithm 5 is reproducible as long as M is reproducible. Algorithm 5 costs $4n$ FLOPs, counting OR-bit operation as a FLOP. If the OR-bit operation is not counted then Algorithm 5 costs $3n$ FLOPs. For clarity, the cost of Algorithm 5 will be written as $3n$ FAdd $+ n$ FOr, where FAdd stands for floating point addition, and FOR stands for an OR-bit operation of two floating point numbers.

Algorithms 2, 3, and 5 can be used to compute a reproducible sum of a given input vector, regardless of the order of computation as well as the platform on which the algorithm is run. The only requirement is that all operations must be performed in the same rounding mode, namely to-nearest rounding mode for Algorithm 2 and 5 and directed-rounding mode for Algorithm 3.

These algorithms have two main drawbacks. First, the absolute errors of these algorithms depend strongly on the size of input vector. For both algorithms, the absolute error is bounded by $E = c \cdot n \cdot \mathtt{ulp}(M) \approx c \cdot n \cdot \epsilon \cdot M \approx c \cdot n^2 \cdot \epsilon \cdot \max|v_i|$, where c is a small constant. This error bound can be improved by using the K-fold technique,

which will increase the running time substantially when n is large. The reader can refer to [4] for implementations of Algorithm 2 and 3 using the K-fold technique, i.e. applying K passes of error-free transformation to extract K leading parts of each input floating point instead of just 1 leading part to improve the computed accuracy. If the trailing parts are not computed then the cost of the K-fold version of Algorithm 2 and 3 will be $(4\mathtt{K}-1)n$ FAdd and $(3\mathtt{K}-2)n$ FAdd respectively.

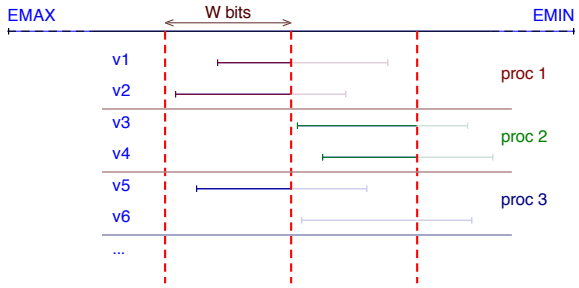
Second, all these three algorithms rely on $\max|v_i|$ to determine the extraction factor M. The use of maximum absolute value requires extra communication, in particular an extra reduction operation, eg a call to MPI_Allreduce with reduction operator MAX. This violates our goal of using a single reduction operation. Furthermore, the maximum may not be easily available, for example in the blocked versions of dgemm and trsm [1].

In order to avoid a second reduction operation and to improve the error bound of reproducible summation, we propose a new technique called 1-Reduction which will be presented in the next section.

IV. 1-Reduction technique

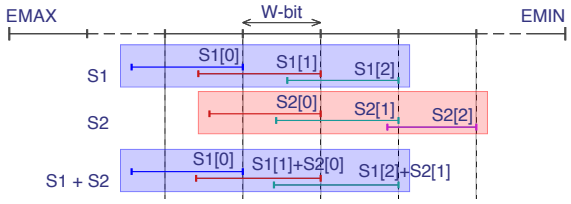
In order to avoid having to compute the global maximum absolute value, we propose a way to precompute boundaries independently of input data, for example of form $2^{i\cdot \mathtt{W}}$ for a carefully chosen W. Using precomputed boundaries requires no communication, and each processor can use its own local boundary, which is the largest one to the right of the leading bit of the local maximum absolute value.

Figure 2. 1-Reduction technique



Since each processor can have a different boundary, during the reduction we only sum intermediate values corresponding to the same boundary. Otherwise, only the intermediate value corresponding to the bigger boundary will be propagated up the reduction tree. Though it can be done easily by using two separate reductions, one for the maximum boundary and one for the summation of partial sums exceeding this boundary, we only want to use one reduction. So we combine these two reductions into one, where we compute both the largest boundary encountered, as well as the partial sum of the values exceeding this boundary. This is depicted in Figure 3 for the case of the 3-fold algorithm which will be explained in the next section. Therefore we call this technique *1-Reduction*.

Figure 3. Special reduction operator



The 1-Reduction technique can also be described by means of binning. A bin corresponds to a chunk of W adjacent bits along the exponent range. Each bin is characterized by an index i, which means that the bin covers the bits from position $i\cdot\mathtt{W}$ to position $(i+1)\cdot\mathtt{W}-1$ in the exponent range. Each input floating-point value will then be put in a number of contiguous bins. The values inside each bin will be summed separately. During the process of computation, including both local computation and result propagation along the reduction tree, only the K left-most bins (those with the highest indices) will be kept. This is similar to the K-fold technique.

It is important to note that cancellation is not taken into account during the computation, a bin will always be propagated even if its carrying value (sum of floating values that were put into that bin) is zero. It means that the left-most bin will always be determined by the maximum absolute value of the input numbers which is reproducible. Therefore results computed by the 1-Reduction technique will be reproducible provided that the aggregation of each bin is exact.

The binning process can be done by converting each input floating point number into an array of integers. However, this might require data movement between Floating Point Unit and Arithmetic Unit. Therefore we propose to perform the binning process based entirely on floating point arithmetic, using one of the algorithms presented in the previous section.

The following subsection will present how to use Algorithm 5 to implement the 1-Reduction technique. Algorithm 2 can also be used with some slight modification. Nonetheless, as will be explained in subsection IV-A, Algorithm 3 cannot be used directly due to the use of directed-rounding.

A. Algorithm

Using the extraction algorithms from Section III to perform the splitting, M will be chosen from precomputed values $M_{[i]} = 0.75\cdot\epsilon^{-1}\cdot 2^{i\cdot\mathtt{W}}$. Input floating-point numbers will be split according to boundaries $\mathtt{ulp}(M_{[i]}) = 2^{i\cdot\mathtt{W}}$.

For a scalar x the boundary $M_{[i]}$ will be chosen so that $2^{(i+1)\cdot\mathtt{W}} > 2|x| \geq 2^{i\cdot\mathtt{W}}$. In order to get the first k leading parts, x will be split using $M_{[i]}, M_{[i+1]}, \ldots, M_{[i+k-1]}$. Since $2^{(i+1)\cdot\mathtt{W}} > 2|x|$, i.e. $|x| < \frac{1}{2}\mathtt{ulp}(M_{[i+1]})$, then $fl(M_{[i+1]} + x) = M_{[i+1]}$ in to-nearest rounding mode. Therefore for all $j > i$ using $M_{[j]}$ to split x following either Algorithm 1 or Algorithm 4 will not change the value of x. It means that the using any sequence of extraction factors $M_{[j]}, M_{[j+1]}, \ldots, M_{[j+k-1]}$, $j \geq i$ the extracted part of x that is put into a certain bin will be reproducible

and depend only on the index of that bin. This property cannot be guaranteed if the operations are performed in a directed rounding mode since even if $\texttt{ulp}(M_{[j]}) \gg |x|$ it is still possible that $\texttt{fl}(M_{[j]} + x) \neq M_{[j]}$ in directed rounding.

Similarly for an input vector $v = [v_1, \ldots, v_n]$, $M_{[i]}$ is chosen so that $2^{(i+1)\cdot \texttt{W}} > 2 \cdot \max|v_i| \geq 2^{i\cdot \texttt{W}}$. In case of K-fold algorithm, $M_{[i]}, M_{[i+1]}, \ldots, M_{[i+k-1]}$ will be used.

In addition, in order for Algorithm 5 to be error-free, and hence be reproducible, it is required to satisfy that $\max|v_i| \leq \texttt{ufp}(M_{[i]}) \cdot 2^{-2-\lceil \log_2 n \rceil}$. Using precomputed boundaries, we only have $2 \cdot \max|v_i| < 2^{(i+1)\cdot \texttt{W}}$. Therefore in order to avoid overflowing the bin we have to ensure that $2 \cdot 2^{(i+1)\cdot \texttt{W}} \leq \texttt{ufp}(M_{[i]}) \cdot 2^{-2-\lceil \log_2 n \rceil}$. This means $n \leq 2^{-(i+1)\cdot \texttt{W}-1} \cdot \texttt{ufp}(0.75 \cdot \epsilon^{-1} \cdot 2^{i\cdot \texttt{W}}) = \epsilon^{-1} \cdot 2^{-\texttt{W}-2}$. In case of IEEE double precision, i.e. $\epsilon = 2^{-53}$, if the bin width is $\texttt{W} = 40$, then the maximum number of floating point numbers that can be summed without overflow is $n \leq 2^{11} = 2048$. Therefore, in order to accommodate for a input vector one needs to perform the carry-bit propagation after every $\texttt{NB} \leq \epsilon^{-1} \cdot 2^{-W-2}$ additions. It means that the algorithm needs to be implemented in a blocked fashion. It is worth noting that the maximum absolute value can also be evaluated in blocked fashion. Denote by $v_{[l:r]} = [v_l, \ldots, v_r]$ a block of input vector v from index l to r. Algorithm 6 is the pseudo-code for K-fold 1-Reduction reproducible summation. In total, Algorithm 6 costs n FMax $+ (3\texttt{K} - 2)n$ FAdd $+ \texttt{K}n$ FOr, where FMax stands for a floating point maximum absolute value operation.

B. Reproducibility

The reproducibility of Algorithm 6 can be proved by analyzing each code block of the algorithm.

The algorithm starts with K smallest bins of the exponent range and no carry bits. The main part of the algorithm is a `for` loop (Line 4) that iterates through the whole input vector v by blocks of NB elements. For each block, the local maximum absolute value of all the elements is first computed. This local maximum absolute value will then be used to update the list of K left-most bins (Lines 7 to 14). This `while` loop ensures that $m < 2^{\texttt{W}} \cdot \texttt{ulp}(S_1) = \texttt{ufp}(S_1) \cdot \epsilon \cdot 2^{\texttt{W}+1} \leq \texttt{ufp}(S_1) \cdot 2^{-2-\log_2 \texttt{NB}}$ which is the requirement for Algorithm 5 to be reproducible. Therefore S_1 always corresponds to the updated maximum absolute value of all the visited elements in v, which means that at the end of the algorithm S_1 will correspond to $\max_1^n |v_i|$. Hence S_1 is guaranteed to be reproducible.

The second `for` loop (Line 15 to 17) performs the vector transformation to extract the K leading parts corresponding to $S_k, k \in [1, \ldots, \texttt{K}]$, from each input floating point number. After each extraction, we always have that $|v_j| \leq \frac{1}{2}\texttt{ulp}(S_k) = \texttt{ufp}(S_{k+1}) \cdot \epsilon^{-1} \cdot 2^{-\texttt{W}-2}$, therefore all the K vector transformations on $v_{[i:\texttt{1N}]}$ are error-free. According to Theorem 4, these leading parts are added to S_k without any rounding error.

The final loop (Line 18 to 29) is to perform the carry-bit propagation, which helps to avoid overflow. Essentially, this carry-bit propagation part is to ensure that $1.5 \cdot \texttt{ufp}(S_k) \leq S_k < 1.75 \cdot \texttt{ufp}(S_k)$, which is needed for the vector transformation of subsequent blocks to be error-free and reproducible.

Algorithm 6 Sequential Reproducible Summation: $[S, C] = \texttt{rsum}(v, \texttt{K}, \texttt{W})$

Require: v is a vector of size n. $1 \leq W < -\log_2 \epsilon$ is the bin width. $M_{[i]} = 2^{i\cdot \texttt{W}}, \texttt{imin} \leq i \leq \texttt{imax}$ are precomputed. K is the number of bins to be kept. $\texttt{NB} \leq \epsilon^{-1} \cdot 2^{-\texttt{W}-2}$ is the block size. $S, C \in \mathbb{F}^{\texttt{K}}$

1: **for** $k = 1$ to K **do** ▷ Initialization
2: $S_k = M_{[\texttt{imin}+\texttt{K}-k]}, C_k = 0$
3: **end for**
4: **for** $i = 1$ to n step NB **do**
5: $\texttt{1N} = \min(i + \texttt{NB} - 1, n)$
6: $m = \max|v_{[i:\texttt{1N}]}|$ ▷ Local maximum absolute
7: **while** $m \geq 2^W * \texttt{ulp}(S_1)$ **do** ▷ Update
8: **for** $k = \texttt{K}$ to 2 **do**
9: $S_k = S_{k-1}$
10: $C_k = C_{k-1}$
11: **end for**
12: $C_1 = 0$
13: $S_1 = 1.5 * 2^W * \texttt{ufp}(S_1)$
14: **end while**
15: **for** $k = 1$ to K **do** ▷ Extract K first bins
16: $[S_k, v_{[i:\texttt{1N}]}] = \texttt{ExtractVector3}(S_k, v_{[i:\texttt{1N}]})$
17: **end for**
18: **for** $k = 1$ to K **do** ▷ Carry-bit Propagation
19: **if** $S_k \geq 1.75 * \texttt{ufp}(S_k)$ **then**
20: $S_k = S_k - 0.25 * \texttt{ufp}(S_k)$
21: $C_k = C_k + 1$
22: **else if** $S_k < 1.25 * \texttt{ufp}(S_k)$ **then**
23: $S_k = S_k + 0.5 * \texttt{ufp}(S_k)$
24: $C_k = C_k - 2$
25: **else if** $S_k < 1.5 * \texttt{ufp}(S_k)$ **then**
26: $S_k = S_k + 0.25 * \texttt{ufp}(S_k)$
27: $C_k = C_k - 1$
28: **end if**
29: **end for**
30: **end for**

Ensure: S_k, C_k are respectively the leading part and trailing part of the aggregation of values in the k-th left most bin.

The carry-bit is stored in C_k. By consequent, after each iteration the aggregated value of each bin will be determined by

$$\begin{aligned} T_k &= C_k \cdot 0.25 \cdot \texttt{ufp}(S_k) + S_k - 1.5 \cdot \texttt{ufp}(S_k) \\ &= 0.25 \cdot (C_k - 6) \cdot \texttt{ufp}(S_k) + S_k. \end{aligned}$$

In the worst case, the value of $|C_k|$ is bounded by $2 \cdot \lceil \frac{n}{\texttt{NB}} \rceil$. In order to avoid overflow, the precision needed to store C_k is $p_c = 2 + \lceil \log_2 \frac{n}{\texttt{NB}} \rceil$. In other words, if the precision used to store C_k is p_c then Algorithm 6 can compute a reproducible sum of up to $\texttt{NB} \cdot 2^{p_c - 2}$ elements. Moreover, NB is bounded by $\texttt{NB} \leq \epsilon^{-1} \cdot 2^{-\texttt{W}-2}$, the maximum number

of floating point numbers that can be summed reproducibly is $N_{max} = \epsilon^{-1} \cdot 2^{p_c - \mathtt{W} - 4}$. For example, if the bin width is set to $\mathtt{W} = 40$ and C_k is stored in double precision floating point numbers then the maximum number of double precision floating point that can be summed reproducibly is $N_{max} = 2^{53} \cdot 2^{53-40-4} = 2^{62} \approx 4e18$, which is sufficiently large enough for Exascale.

C. Accuracy

The accuracy of the pre-rounding technique depends on the error of each pre-rounding, which in turn depends on the boundary. Since each trailing part that is thrown away in the pre-rounding technique can be bounded by $Boundary$, the total error is bounded by $n \cdot Boundary$. In the 1-Reduction technique, the boundaries are precomputed and do not depend on input data. In the worst case, the maximum absolute value of input data can be only slightly larger than the boundary, i.e. $\max|v_i| \approx \frac{1}{2}\mathtt{ulp}(M)$. Therefore if only one bin is used to perform the summation, the best error bound that we can get for the 1-Reduction technique is $n \cdot \max|v_i|$, which is much too large.

In case of K-fold technique, the computation's error is now determined by the minimum boundary being used. Suppose the gap between two consecutive precomputed boundaries is W bits. Then the error bound of the K-fold 1-Reduction algorithm is:

$$\begin{aligned} absolute\ error &\le n \cdot \frac{1}{2}\mathtt{ulp}(S_{\mathtt{K}}) \\ &\le n \cdot \frac{1}{2}\mathtt{ulp}(S_1) \cdot 2^{(1-\mathtt{K})\cdot\mathtt{W}} \\ &< n \cdot 2^{(1-\mathtt{K})\cdot\mathtt{W}-1} \cdot \max|v_i|. \end{aligned}$$

This means that the accuracy of 1-Reduction can be tuned by choosing K according to the accuracy requirement of each application.

In practice for the summation of double precision floating-point numbers we use $\mathtt{K} = 3$ and $W = 40$. W is a tuning parameter that depends on the desired accuracy, and the relative costs of basic arithmetic versus the (very rare) carry propagation from one segment to another.

Given W, K, the absolute error will be bounded by

$$n \cdot 2^{-81} \cdot \max|x_i| = n \cdot 2^{-28} \cdot \epsilon \cdot \max|x_i|,$$

which is at least 2^{-28} times smaller than the error bound of the standard summation algorithm $(n-1) \cdot \epsilon \cdot \sum_1^n |x_i|$.

D. Reduction operator

As explained at the beginning of this section, in order to compute a reproducible global sum, we need to have a special reduction operator that takes into account the binning process.

Using Algorithm 6, the local result of each processor is a pair of two arrays $S, C \in \mathbb{F}^{\mathtt{K}}$, which satisfy $1.5 \cdot \mathtt{ufp}(S_k) \le S_k < 1.75 \cdot \mathtt{ufp}(S_k)$ for all $1 \le k \le \mathtt{K}$. Therefore, the reduction can be performed by implementing a special addition operation of two such pairs of arrays.

An implementation of such a special reduction operator is presented in Algorithm 7. First, the two input pairs are aligned to the maximum bin index (Lines 1 to 12). Then the aggregation is carried out by simply adding the corresponding mantissas and carries. Finally the carry-bit propagation (Lines 16 to 25) is performed to avoid overflow in subsequent reduction steps.

Algorithm 7 1-Reduction's special reduction operation $[S, C] = \mathtt{RepReduce}(S1, C1, S2, C3)$

Require: $S1, C1, S2, C2 \in \mathbb{F}^{\mathtt{K}}$. W is the bin width.

1: **if** $\mathtt{ulp}(S2_1) > \mathtt{ulp}(S1_1)$ **then**
2: $\quad [S, C] = \mathtt{RepReduce}(S2, C2, S1, C1)$
3: $\quad$ **return**
4: **end if**
5: **while** $\mathtt{ulp}(S2_1) < \mathtt{ulp}(S1_1)$ **do** ▷ Right Shifting
6: $\quad$ **for** $k = \mathtt{K}$ to 2 **do**
7: $\quad\quad S2_k = S2_{k-1}$
8: $\quad\quad C2_k = C2_{k-1}$
9: $\quad$ **end for**
10: $\quad S2_1 = 1.5 * 2^W * \mathtt{ulp}(S2_1)$
11: $\quad C2_1 = 0$
12: **end while**
13: **for** $k = 1$ to K **do** ▷ Aggregation
14: $\quad S_k = S1_k + (S2_k - 1.5 * \mathtt{ulp}(S2_k))$
15: $\quad C_k = C1_k + C2_k$
16: $\quad$ **if** $S_k \ge 1.75 * \mathtt{ufp}(S_k)$ **then** ▷ Carry Propagation
17: $\quad\quad S_k = S_k - 0.25 * \mathtt{ufp}(S_k)$
18: $\quad\quad C_k = C_k + 1$
19: $\quad$ **else if** $S_k < 1.25 * \mathtt{ufp}(S_k)$ **then**
20: $\quad\quad S_k = S_k + 0.5 * \mathtt{ufp}(S_k)$
21: $\quad\quad C_k = C_k - 2$
22: $\quad$ **else if** $S_k < 1.5 * \mathtt{ufp}(S_k)$ **then**
23: $\quad\quad S_k = S_k + 0.25 * \mathtt{ufp}(S_k)$
24: $\quad\quad C_k = C_k - 1$
25: $\quad$ **end if**
26: **end for**

Ensure: $[S, C]$ is the sum of the K corresponding left-most bins of both $[S_1, C_1]$ and $[S_2, C_2]$.

Using this special reduction operator, a global sum requires only one reduction over 2 arrays of K elements. In case of the 3-fold summation of double precision floating numbers, and C_k being stored in double precision, then the size of data structure to be reduced will be $2 * 3 * 8 = 48$ bytes.

V. Experimental results

In this section, we present some experimental results to check the reproducibility, the accuracy, as well as the performance (both sequential and parallel) of the proposed 1-Reduction technique.

A. Reproducibility and Accuracy

Input data are generated using a linear distribution, uniform distribution and normal distribution in $[-1, 1]$ for varying vector sizes $n = 10^3, 10^4, 10^5, 10^6$.

In order to check the reproducibility, for each input vector all elements are summed in different orderings: increasing, decreasing, increasing magnitude, decreasing

Table I
ACCURACY OF REPRODUCIBLE SUMMATION

Generator v_i	`rsum`	standard
`drand48()`	0	[-8.5e-15 , 1.0e-14]
`drand48()` $- 0.5$	1.5e-16	[-1.7e-13 , 1.8e-13]
$\sin(2.0 * \pi * i/n)$	1.5e-15	$[-1.0$, 1.0 $]$

magnitude and random permutation. The computed results are considered reproducible if and only if they are bit-wise identical for all considered data orderings. As expected, results computed using Algorithm 6 are reproducible for all input data. In contrast, results computed by the standard summation algorithm are rarely reproducible for different data ordering.

Table I summaries some test results of the summation of $n = 10^6$ double precision floating-point numbers. Computed results of both reproducible summation and standard summation for different data orderings are compared with result computed using quad-double precision. The first column shows the formula used to generate input vector. The second column shows the relative errors of results computed by Algorithm 6 using 3 bins of 40-bit width, which are always reproducible. The third column shows the range of relative errors of results computed by the standard summation algorithm.

As can be seen from Table I, in most cases results computed by Algorithm 6 have smaller relative errors than results computed by the standard summation algorithm. Especially in the third case, i.e. $v_i = \sin(2.0 * \pi * i/n)$, the sum would be identically zero in exact arithmetic. In finite arithmetic, nonetheless, since we do not have the exact π, the exact value of $\sum_1^n v_i$ is instead very close to zero. In this case, the standard summation summation could hardly obtain any correct significant bit in the computed result. Our proposed 1-Reduction technique, i.e. the `rsum` function, however computes a very accurate result which is correct to almost 15 decimal digits. This correlates with the fact that the error bound of Algorithm 6 is much smaller than that of the standard summation algorithm.

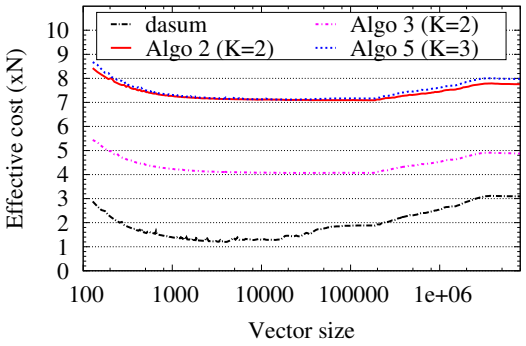
B. Performance

In order to check the performance of the proposed 1-Reduction technique we measured two test sets for sequential and parallel computing environments respectively.

The first test set measures the performance of the 3-fold version of Algorithm 5 (which is the building block of Algorithm 6) as well as the 2-fold versions of Algorithm 2 and Algorithm 3 for varying vector sizes $n = 100, \ldots, 10^7$ on a single Intel Sandy Bridge core at 2 GHz , with L1 Cache of 32K, L2 Cache of 256KB, and L3 Cache of 10MB. Each test is run with warmed-up cache (by repeating the function call a large number of times) to amortize the data loading time.

Recall that the theoretical cost of the 2-fold version of Algorithm 2 is $7n$ `FAdd`. The theoretical cost of the 2-fold version of Algorithm 3 is $4n$ `FAdd`. And the theoretical

Figure 4. Effective costs of Error-free transformation algorithms



cost of the 3-fold version of Algorithm 5 is $7n$ `FAdd` + $3n$ `FOr`.

Since the summation algorithm does not use any floating point multiplication, using SSE2 instructions the peak performance of a standard summation algorithm can be $P_{peak} = 2 \times 2.0 = 4.0$ GFLOPs. Figure 4 plots the effective cost of each algorithm, which is determined by $P_{peak}/(\#iterations * n/running\ time)$. For example a ratio 2.0 means that the effective cost of the algorithm is equivalent to $2n$ floating point additions.

As can be seen from Figure 4, when the vector size is large enough ($n > 1000$) to amortize the cost of loop control logic as well as of function calls, then the effective cost of Algorithm 3 (`K` $= 2$) and Algorithm 3 (`K` $= 2$) is only slightly higher than 7 and 4 respectively, which match closely to the FLOP counts of the corresponding algorithms. Nonetheless, the effective cost of Algorithm 5 (`K` $= 3$) is only ≈ 7, which is roughly equal to the effective cost of Algorithm 2 (`K` $= 2$). It means that in this case the `OR`-bit operation is not counted.

When the vector sizes get larger ($n > 10^6$), the input vector can no longer fit into the L3 cache which is of size 10MB and hence can only hold up to $1,310,720$ double precision floating point numbers. Therefore the data loading time becomes more important and increases the effective cost of each algorithm. From Figure 4, the effective costs of all the three reproducible summation algorithms both increase by 1 (from 3 to 4 for Algorithm 3, and from 7 to 8 for Algorithm 2, 5).

As a reference, consider the effective cost of function `dasum` from the Intel MKL library [2] to compute the sum of absolute values of input vector which is also plotted in Figure 4. When $n \in [10^3, 2 \cdot 10^4]$, the effective cost of `dasum` is close to 1, which means that the cost of the `dasum` function is almost the same as the cost of the standard summation algorithm. It suggests that the floating point absolute value operation does not contribute to the cost of the `dasum` function. Nonetheless, when $n > 2 \cdot 10^6$ the effective cost of `dasum` increases substantially from 1 to 3. Therefore the `dasum` function is more communication-bound than the other three reproducible summation algorithm.

The second test set checks the scalability of the proposed 1-Reduction technique in a

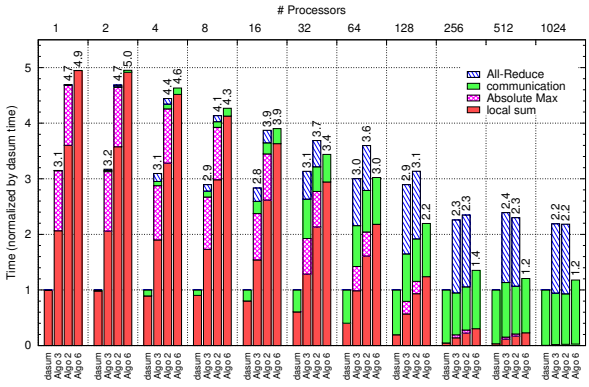
massively parallel environment. Figure 5 summarizes experimental results on Edison, a Cray XC30 (`www.nersc.gov/systems/edison-cray-xc30`), for the summation of 10^6 IEEE double precisions floating-point numbers. For each number of processors $p \in (1, 2, \ldots, 1024)$, we measured the run-time of 4 algorithms: The first uses the optimized non-reproducible `dasum` function in the Intel MKL library [2] to compute the local sum of absolute values, and then MPI-Reduce to compute the final result. Therefore the running time consists of two parts: the local summation computing time which is colored red and the reduction time which is colored green. The total running time of this algorithm is always normalized to 1 in the figure. The second and third run-times are for the two algorithms Fast Reproducible Sum and Reproducible Sum presented in [4], which are essentially based on Algorithm 3 and Algorithm 2 respectively with 2 vector transformation passes ($\mathtt{K} = 2$); both require 2 reduction steps. Both these algorithms compute the local maximum absolute value (magenta part) and communicate to reduce and broadcast the global maximum absolute value to all processors (blue part). The fourth run-time is for Algorithm 5 using $\mathtt{K} = 3$ extraction steps. Similar to normal summation, the run-time consists of only two parts: the local computing time (red) and the reduction (green) to reduce the final result.

For small numbers of processors, the running time of each algorithm is dominated by the local computing time (including computing the maximum). Therefore the reproducible summation algorithms are 3 or more times slower than the normal sum in terms of running time. Nevertheless, for large numbers of processors the running time of all algorithms tends to be dominated by communication time. Consequently the running time of each algorithm is close to proportional to the number of reductions it uses. Note that, the `AllReduce` collective function used to reduce and broadcast the global maximum absolute value is even more expensive than the `Reduce` function. Therefore, performance-optimized nonreproducible summation and 1-Reduction are both about twice as fast as the other two algorithms. For example when $p = 512, 1024$, the 1-Reduction technique is only 20% slower than the performance-optimized summation.

VI. Conclusions and Future Work

The proposed 1-Reduction technique attains reproducibility for parallel heterogeneous computing environments. Reproducibility is guaranteed regardless of the number of processors, reduction tree shape, data assignment, data partitioning, etc. Since no extra inter-processor communication is needed, this technique is well suited for highly parallel environments including ExaScale and cloud computing, where communication time tends to dominate computing time. It can also be applied in data-streaming applications since the boundaries can be updated on-the-fly whenever new values come. The 1-Reduction technique can be applied for example for heterogeneous

Figure 5. Performance of 1-Reduction technique on Edison machine



systems that involves both general-purpose CPUs and GPUs (Graphics Processing Unit). The order of evaluation changes radically when porting code from CPU to GPU, therefore it is much harder for a performance-optimized floating point library to guarantee reproducibility between CPU and GPU.

Though in this paper we have only demonstrated this technique for summation, it can be applied to other operations that use summation as a reduction operator, for example dot product, matrix-vector multiplication, and matrix-matrix multiplication. We also plan to apply this technique to prefix-sum as well as higher level BLAS and linear algebra routines.

Even though our proposed 1-Reduction technique works both correctly and with performance close to that of performance-optimized nonreproducible summation in massively parallel environment, it is still slower than a performance-optimized summation by a factor of 7 in terms of FLOP count. This slowdown cannot be easily amortized by communication cost in other computation-bound operations such as matrix-matrix multiplication. Therefore we are also investigating possibilities of hardware support to reduce that cost. Ideally, if we can combine both the bin indexing and the bin aggregation step into a single instruction then the slowdown of local summation would be reduced to 1. It means that we can obtain reproducibility at almost the same cost as a performance-optimized nonreproducible algorithm.

References

[1] BLAS - Basic Linear Algebra Subroutines, `htttp://www.netlib.org/blas/`.

[2] MKL - Intel® Math Kernel Library, `www.intel.com/software/products/mkl/`.

[3] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Methematik*, 18(3):224–242, 1971.

[4] James Demmel and Hong Diep Nguyen. Fast Reproducible Floating-Point Summation. In *21st IEEE Symposium on Computer Arithmetic*, Austin, Texas, USA, April 2013.

[5] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

[6] Siegfried M. Rump. Ultimately fast accurate summation. *SIAM Journal on Scientific Computing (SISC)*, 31(5):3466–3502, 2009.

[7] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Fast high precision summation. *Nonlinear Theory and Its Applications, IEICE*, 1(1):2–24, 2010.

[8] Stefan Siegel and Jürgen Wolff von Gudenberg. A long accumulator like a carry-save adder. *Computing*, 94(2-4):203–213, March 2012.

[9] Michela Taufer, Omar Padron, Philip Saponaro, and Sandeep Patel. Improving numerical reproducibility and stability in large-scale numerical simulation on GPUs. In *IPDPS*, pages 1–9, 2010.

[10] Oreste Villa, Daniel Chavarria-Miranda, Vidhya Gurumoorthi, Andres Marquez, and Sriram Krishnamoorthy. Effects of floating-point non-associativity on numerical computations on massively multithreaded systems. In *Cray User Group meeting, CUG*, 2009.