Designing Bit-Reproducible Portable High-Performance Applications*

Andrea Arteaga
ETH Zurich, Switzerland
andrea.arteaga@env.ethz.ch

Oliver Fuhrer
Federal Office for Meteorology and Climatology
MeteoSwiss, Zurich, Switzerland
oliver.fuhrer@meteoswiss.ch

Torsten Hoefler
ETH Zurich, Switzerland
htor@ethz.ch

***Abstract*—Bit-reproducibility has many advantages in the context of high-performance computing. Besides simplifying and making more accurate the process of debugging and testing the code, it can allow the deployment of applications on heterogeneous systems, maintaining the consistency of the computations. In this work we analyze the basic operations performed by scientific applications and identify the possible sources of non-reproducibility. In particular, we consider the tasks of evaluating transcendental functions and performing reductions using non-associative operators. We present a set of techniques to achieve reproducibility and we propose improvements over existing algorithms to perform reproducible computations in a portable way, at the same time obtaining good performance and accuracy. By applying these techniques to more complex tasks we show that bit-reproducibility can be achieved on a broad range of scientific applications.**

***Keywords*-determinism; reproducibility; parallelism; IEEE-754 standard;**

I. Introduction

The apparent paradox between the theoretical deterministic behavior of computers and the difficulty of reaching reproducible results is the consequence of many factors, some of which concern the increasing parallelism that has progressively taken place in every kind of computations. The goal of performing bit-reproducible computations is often regarded as difficult and prohibitively expensive [1]. As we show in this work, many relevant tasks can be solved in a reproducible fashion by building applications specifically designed to be deterministic, and the performance of these applications can be competitive with respect to their non-reproducible counterparts.

In this work, we define *reproducibility* as an application property. We call an application (bit-)reproducible[1] if it provides the same output when provided with the same input across different runs, which may be executed on different hardware, in different software environments, with a different data distribution or with a different numbers of processing elements.

Bit-reproducibility has obvious advantages for debugging and testing purposes. Being able to reproduce the exact behavior of an application on a different number of processors, a different hardware architecture or for several consecutive runs is often of key importance in order to locate and isolate bugs. Especially, when refactoring an application in a way that the results should not change, reproducibility can significantly ease testing. However, debugging is only a secondary use-case for us. Many applications being run on large, parallel high performance computing facilities simulate the behavior of complex and highly non-linear systems. Prominent examples can be found in molecular dynamics or weather and climate simulation. For example, for weather and climate systems a small difference in computation on the level of a rounding error may rapidly lead to a completely different evolution of the weather patterns [2]. This behavior of non-linear systems is well understood [3], [4] and does not per-se impose the concerned applications to be bit-reproducible in order to be useful. Nevertheless, there are use cases where bit-reproducibility is important for such applications.

One such use-case is relevant in meteorology: With the stagnating performance of storage subsystems it may soon become impractical to store all data produced by a climate simulation. Being able to reproduce the simulation in a heterogeneous computing environment and on a different number of processors is required in order to be able to produce a consistent set of diagnostics when analyzing the simulation. Also, with stagnating inter-node communication speeds with respect to increasing floating-point performance, it may become advantageous to replace communication with redundant computation. If the reproducibility of these computations is not guaranteed, the evolution of the simulation on different processors may no longer be consistent.

Bit-reproducibility is a semantic requirement of the operations performed by the floating-point unit. In the development of an application, all steps that define this semantics must be carefully performed with certain restrictions. The important factors are the designing of the algorithm, the implementation through a programming language, the compilation, the run-time environment and hardware. Although we occasionally refer to hardware features and compilation stages, the scope of this paper is the algorithmic/programming level. In this work, we demonstrate a set of tools that can be used to implement bit-reproducible floating-point applications. We show that, if implemented well, the proposed techniques have low or negligible over-

* this online version is slightly updated

[1] we use “bit reproducible” and “reproducible” interchangeably

heads such that they can be used in production settings. Our main contributions of this paper are:

- An empirical study of the performance impact of a reproducible portable implementation of standard transcendental functions in comparison to the native platform libraries. We show that our tuned reproducible implementation is (in the geometric mean) even faster than the platform libraries on five systems and for ten functions).
- The derivation of an algorithm for bit-reproducible sum reduction in a distributed-memory environment.
- A detailed performance study showing that our reproducible reduction performs better than existing schemes and comparable to non-reproducible reductions (maximum 10% slower in out-of-cache case).

II. Floating-point algebra

A. Background

The IEEE-754 [5] standard defines the format used to represent floating-point values, the rounding modes and the arithmetic operations, i.e. the sum $\oplus$, the subtraction $\ominus$, the multiplication $\otimes$, the division $\oslash$ and the square root `sqrt`. Floating-point numbers are represented as $x \cdot 2^E$, where $x \in [1, 2)$ is a number defined by m binary digits (*bits*), and E, $E_{min} \leq E \leq E_{max}$, is an integer number called *exponent*. m, E_{min} and E_{max} are defined by the format. Numbers that can be expressed in this notation are said to be *representable*. Rounding modes include the round-to-nearest, round-towards-zero, round-towards-positive-infinity and round-towards-negative-infinity and can be selected by changing the internal state of the floating-point unit (FPU). Every rounding mode defines a rounding function $fl(x)$ that produces a representable number out of a real number x. Every arithmetic operation is defined as the rounding of the exact result of the abstract arithmetic operation. For example, the sum of two numbers $a \oplus b$ is defined as $fl(a + b)$. In computations that require more than a single operation all the intermediate values are rounded. As a result, the operators $\oplus$ and $\otimes$ are not associative.

Example 1. Summing numbers with very different magnitude in floating-point arithmetic can lead to relatively high rounding errors. In the double precision format, the sum $2^{57} \oplus 1$ gives as result 2^{57}: the contribution of the second addend was exactly zero. Therefore, the expressions $(2^{57} \ominus 2^{57}) \oplus 1$ and $(2^{57} \oplus 1) \ominus 2^{57}$ give two completely different results (1 and 0, respectively), although they are equivalent in standard arithmetic.

Example 2. The values $a = 1.2$, $b = 3.3$, $c = 4.8$ are expressed in the `binary16` format [6] as: $a = 1.0011001101_2 \cdot 2^0$, $b = 1.1010011010_2 \cdot 2^1$, $c = 1.0011001101_2 \cdot 2^2$. The sum of these values can be computed as $(a \oplus b) \oplus c$ as well as $a \oplus (b \oplus c)$. The former gives the result $1.0010100110_2 \cdot 2^3$, while the latter gives $1.0010100111_2 \cdot 2^3$. These results differ because of the intermediate rounding.

Compilers play an important role in the definition of the floating-point semantics. To some extent, they are allowed to reinterpret the code provided by the user and to apply non-conservative optimizations. The documentation provided by the compiler vendor is an important reference for the programmer. The technical report provided by Intel [7] addresses both generic issues and issues related to the Intel Compiler.

In order to be able to perform a rough analysis of the rounding errors, we introduce the concepts of *machine epsilon* (ϵ) and *units in the last place* (`ulps`). The former is a number defined for every floating-point format (its value is 2^{-23} for the single precision and 2^{-52} for the double precision format) expressing the maximal relative rounding error. The *unit in the last place* identifies the difference between two consecutive representable normalized numbers. Since this difference depends on the exponent of the number, the *unit in the last place* is a function of the numbers. For example, $\mathtt{ulp}(1.0) = \epsilon$, $\mathtt{ulp}(x) = \mathtt{ulp}(-x) = \mathtt{ulp}\left(2^{\lfloor log_2(|x|) \rfloor}\right) = 2^{\lfloor log_2(|x|) \rfloor}\epsilon$, for all normal numbers.

Every operation that involves exactly one rounding can introduce a maximal absolute error equal to $\kappa \cdot \mathtt{max}((x))$, where x is the operand or result with largest magnitude and $\kappa \in \{1/2, 1\}$ depending on the rounding mode. For example, the error introduced by the sum $fl(1.4) \oplus fl(0.8)$ is bounded by $\mathtt{ulp}(fl(1.4) \oplus fl(0.8)) = 2\epsilon$, since the maximum exponent of all the operands and the result is 1 (the result is approximately 2.6). In Example 1 the maximal error of the sum is bounded by $\mathtt{ulp}(2^{57})$, since 2^{57} is the largest operand. The well-known article [8] addresses many important aspects of the computation in floating-point arithmetics.

In this work we assume that the state of the FPU is fixed. Although we sometimes provide more general rules, we also assume that the selected rounding mode is the default, i.e., round-to-nearest with ties to even.

B. Portable reproducible arithmetic functions

A possible source for non-determinism can stem from using external functions in computational parts of the code. If the vendor of the library does not specify its exact floating-point semantics, no assumptions can be made about the reproducibility of the computation. Besides the possible non-determinism embedded in the functions themselves, another possible issue is the usage of different libraries implementing the same interface. This can happen particularly if the same application runs on different systems, on which different implementations, or different versions of the same library could be installed. If a library for specific computations is needed, the programmer has to make sure that its implementation

is deterministic and that exactly the same implementation exists on all relevant architectures.

Different architectures have different features concerning, among others, the memory layout, the instruction set, the parallelism. However, one can rely on the fact that all computationally relevant architectures implement a common subset of the IEEE-754 [5], [6] standard, which makes it possible to perform internally deterministic computations.

Since floating-point computations are mostly non-associative, the order of the operations must be strictly defined. The IEEE-754 2008 [6] standard includes the new ternary instruction Fused-Multiply Add (FMA), that performs a multiplication followed by an addition involving just one rounding. Using this instruction, the expression $a \cdot b + c$ is evaluated as $fl(a \cdot b + c)$ instead of $fl(fl(a \cdot b) + c)$, thus obtaining more precise result with less effort. However, this introduces an ambiguity, since two semantically different operations $FMA(a, b, c)$ and $(a \otimes b) \oplus c$ are expressed with the same code in most programming languages. Different compilers, or different compiler flags can cause the same code to use the former or the latter operation, breaking the internal determinism. Many compilers will avoid FMA instructions when optimizing expressions with brackets like `(a*b)+c`, although the standards do not force this behavior.

Since the format of floating-point numbers is completely defined by the IEEE-754 standard, one can portably perform bit-wise operations on floating-point numbers. To check whether a double-precision number x has magnitude larger than 1, one can take its high-order 32 bits as x_{hi} and check $(x_{hi} \,\&\, 0\text{x}7\text{ff}00000) > 0\text{x}3\text{f}800000$. While this technique is portable, special attention must be paid if the code is to be ported on architectures with different endianness.

Many implementations of transcendental functions make use of lookup tables. Lookup tables avoid to perform some computation by storing reference values for specific inputs and using them as starting point for the computation of the result. Since the cost of computation with respect to memory access decreases on a regular basis, along with the fact that some architectures don't provide a suitable caching mechanism for storing such tables, implementations based exclusively upon computation are the best candidates for high-performance portable reproducible functions.

C. Case study: standard transcendental functions

We consider now the example of basic mathematical functions. C, C++ and Fortran standards establish the availability of a set of mathematical functions as part of the standard library, but do not define their return values. This allows the programmers to use these functions without knowing which implementation will be used, but it is not guaranteed what result will be provided and what maximum error these routines have. A possible solution to this problem would be the standardization of the return values of the mathematical functions. Proposals for defining [9], [10] and implementing [11] such functions exist but the state of the art is not affected by them.

NVIDIA provides, as part of the CUDA toolkit, a set of *device* functions that includes all the functions specified by the C standard. The documentation of the toolkit contains a list of the maximum errors for each function in terms of `ulps` [12]. Since these functions do not use lookup tables and are based just upon computation, they can be used as a template to construct a set of portable reproducible functions. The code provided by NVIDIA contains portions that are specifically designed for CUDA devices. For instance, the non-standard function `rsqrt` is occasionally used. One can substitute non-portable code with code performing standard operations without loss of accuracy — in this case $rsqrt(x) = 1.0/sqrt(x)$.

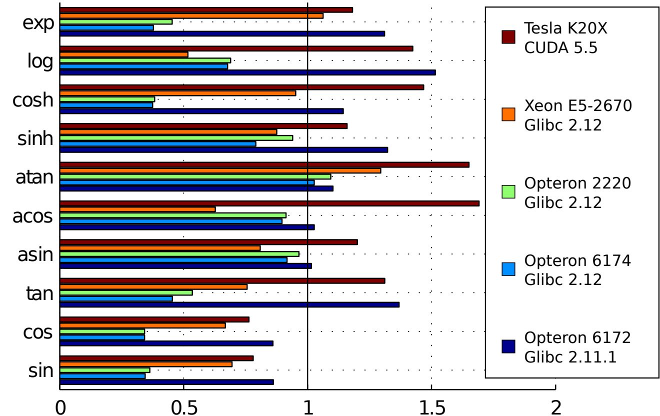

Figure 1. Performance penalty of double-precision reproducible transcendental functions based on CUDA library.

Figure 1 shows the performance loss of the reproducible implementation of transcendental functions with respect to the standard library installed on different systems, i.e., Glibc for the CPU and the CUDA toolkit for GPUs. The loss of performance is defined as the ratio of the time required by the deterministic implementation to the time required by the standard one on the corresponding platform to perform the task of evaluating the function on 2^{22} input values.

Figure 2 shows the geometric mean of the performance loss of all implemented functions for every architecture. The reproducible implementation based upon the CUDA library has been proven to be overall faster than many standard libraries. The geometric mean of the means presented in Figure 2 is 0.83, which is a performance improvement.

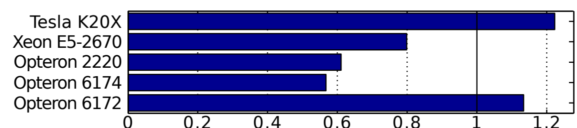

Figure 2. Geometric means over the set of transcendental functions shown in Figure 1

III. Reduction operations

We now discuss the more complex problem of reductions, which is used in many applications. Reductions are mathematically defined with a commutative and associative operator $\odot$ and a sequence v_i as:

$$T = v_1 \odot v_2 \odot \ldots \odot v_n.$$

The definition of the reductions does not provide a bracketing, as the evaluation order for commutative and associative operators is unimportant. This is not the case in floating-point arithmetic, whose operators $\oplus$ and $\otimes$ are non-associative (though they are commutative): the result of a floating-point reduction depends therefore on the bracketing. In order to make the result of such a reduction reproducible, the bracketing must be fixed. While this is feasible also in the distributed-memory case, it is difficult to implement efficiently. Although fixing the bracketing is the only generic method for performing reproducible reduction with a non-associative operator, a special approach for sum reductions allows to avoid this intrusive requirement. In the remaining part of this section we explain this approach and how it can be applied to obtain reproducible reductions without the need for a fixed bracketing.

A. Deterministic sum reduction

The method discussed in this section has been first explored by Rump [13] and by Demmel and Nguyen [14], [15]. We introduce the terminology and the approach that we will extend in the next section. We refer in particular to [14], where the whole method presented here is precisely derived and explained.

The technique is based on the fact that, by pre-rounding the values, it is possible to perform the floating-point sums so that no more rounding is involved. In order to apply a pre-rounding, a constant positive value M called *extractor* is used. M must be a power of two (negative exponents are allowed) and its magnitude must be larger than that of every partial sum of the values v_i. Demmel and Nguyen show that n values v_i can be reproducibly reduced in a floating-point format whose precision is ϵ, if

$$M \geq n \cdot |v_i| \,/\, (1 - 2n\epsilon) \quad \forall\, 1 \leq i \leq n. \tag{1}$$

The pre-rounding consists in summing separately every value v_i with the extractor M, subtracting M from the result and adding the result of the subtraction $q_i := (v_i \oplus M) \ominus M$ to the partial sum $T := \sum_{i=1}^{n} q_i$. We call the values q_i *contributions* to the sum. The following example shows the procedure applied on two values.

Example 3. We will reduce the `binary16` values $v_1 = fl(3.16) = 1.1001010010 \cdot 2^1$ and $v_2 = fl(1.73) = 1.1011101011_2 \cdot 2^0$ using the extractor $M = 16 = 1 \cdot 2^4$. In the sum $v_1 \oplus M$ the trailing two bits of the mantissa of v_1 do not pass any information, as they are insignificant to the result[2]. The result of the sum is thus $v_1 \oplus M = 1.0011001010_2 \cdot 2^4$. The result of the subtraction is $q_1 = (v_1 \oplus M) \ominus M = 1.1001010000_2 \cdot 2^1$, where the trailing 3 bits are zero as result of the cancellation. When the same procedure is performed with v_2, the result is $q_2 = (v_2 \oplus M) \ominus M = 1.1011110000_2 \cdot 2^0$. Both contributions q_i have enough trailing bits set to 0 to sum them without rounding errors: the value $q_1 \oplus q_2$ is $1.0011100100_2 \cdot 2^2$. The fact that the result has an exponent higher than those of the original values caused two trailing bits of the results to be discarded, but these bits were 0 as effect of the pre-rounding. The sum $q_1 \oplus q_2$ was thus free of rounding errors.

The procedure is supported by the fact that the contributions q_i are integer multiples of ϵM. There are no rounding errors because *a)* the sums are semantically equivalent to error-free integer sums, and *b)* this sum can be exactly contained by a floating-point value, since we chose an extractor which is large enough, following (1).

In the procedure shown in Example 3 some information is lost, since q_i is only an approximation of v_i. This information can be recovered by using the error-free subtraction $r_i := v_i \ominus q_i$. The *remainder* r_i contains exactly the part of v_i that has not been considered in the sum. The accuracy of the reduction can be improved by applying the same procedure explained above to the remainders, i.e., pre-rounding them using a suitable extractor and performing the error-free sum of the corresponding contributions. A suitable extractor M^2 for the remainders r_i is a power of two with the property $M^2 \geq n \cdot |r_i| \,/\, (1 - 2n\epsilon)\ \forall 1 \leq i \leq n$, analogously to (1). Since $|r_i| < \kappa\epsilon M$ — where $\kappa \in \{1, 1/2\}$ depends on the rounding mode — every power of two with $M^2 \geq n\epsilon M/(1 - 2n\epsilon)$ will be suitable. By defining

$$\alpha_n := 2^{\lceil log_2(n/(1-2n\epsilon)) \rceil},$$

one can set $M^2 = \alpha_n \kappa\epsilon M$, which is a power of two and satisfies the aforementioned requirement.

Example 4. Continuing on the same setting of Example 3, we can now compute the remainders $r_i = v_i - q_i$: $r_1 = 1.0_2 \cdot 2^{-8}$, $r_2 = -1.01_2 \cdot 2^{-8}$. As expected, both remainders satisfy $|r_i| \leq \kappa\epsilon M$ — in this case $\kappa = 1/2$, $M = 2^4$, $\epsilon = 2^{-10}$, $\kappa M\epsilon = 2^{-7}$. The extractor used to pre-round these remainders will be $M^2 = \alpha_n \kappa\epsilon M = 2^{-5}$.

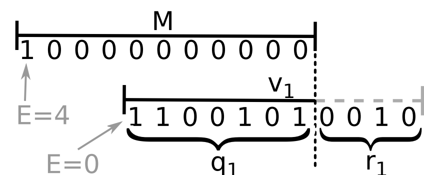

Figure 3. Pre-rounding of value v_1 in Examples 3 and 4. Extractor M has exponent 4, value v_1 has exponent 0.

[2] depending on the rounding mode. See [5] for detailed explanations.

We will call the pre-rounding of the values v_i and the sum of the corresponding contributions *first-level extraction*. The extractor that we have denoted with M until here will be more precisely identified by M^1. The corresponding contributions will be q_i^1 and the remainders r_i^1. The sum of the contributions q_i^1 will be identified by T^1. The extractor used to pre-round the remainders r_i^1 will be M^2, the corresponding contributions q_i^2, their sum T^2. The remainders of the pre-rounding operation will be r_i^2. The process of pre-rounding the remainders r_i^1 and summing the corresponding contributions will be called *second-level extraction*. One can add arbitrary many levels by pre-rounding the remainders of the previous level. The superscripts of the variables M, T, q, r and N (not yet introduced) will identify the level, not the exponent, for the remaining part of the section. The number of levels used is identified by k.

In order to implement this algorithm in a distributed-memory environment, a global reduce is needed at the beginning in order to choose the first-level extractor M^1 and another global reduce is needed at the end, in order to sum all the local results. Since all sums derived from an extraction are associative and thus order-independent, the final reduce does not have to be internally deterministic in order to provide a reproducible result. This is implemented in Algorithm 9 (Parallel K-Fold Reproducible sum, later referred to as *double-sweep algorithm*) proposed by Demmel and Nguyen in [14], where the mathematical background and derivation are explained in detail, along with the error estimate for the multi-level reduction: the maximal difference between the computed sum $T := \sum_{f=1}^{k} T^f$, where k is the number of levels, and the exact sum is

$$E_{rep,k} = 2k\kappa\epsilon\,|T| + \frac{8n}{1 - 8n\kappa\epsilon}\,(\kappa\epsilon)^2\,M^1 + \frac{2}{3}n\kappa\epsilon M^k. \quad (2)$$

B. Single-sweep deterministic sum

The double-sweep algorithm requires two communication steps, one at the beginning and one at the end, and latency hiding is impossible. With today's high-performance applications being designed to exploit to parallelism given by large number of computing processes, this need for communication can decrease the efficiency of the reduction. We design in this section an algorithm that avoids the first communication step and provides a completely reproducible and accurate result.

We base our design on initial ideas by Demmel and Nguyen [15]. Indeed, a technique similar to the one presented in this section has been published concurrently and independently by Demmel and Nguyen in [16]. In order to reduce the amount of computation required by the multi-level extraction, the authors of that paper make use of new instructions that fuse a bit-mask and a sum in a single floating-point operation, obtaining performance improvement where these instructions are available, e.g., on the MIC architecture, at the cost of a small accuracy loss. The results obtained by Demmel and Nguyen show performance degradation of up to 5x compared to a non-reproducible sum for small number of processes and a 20% penalty for large number of processes. We aim at reducing these penalties maintaining at the same time a good accuracy. However, the technique presented in [16] supports a much larger number of input values than the algorithm proposed in this work.

In the previous section we have made use of extractors whose magnitude was defined as a function of $v_{max} := |\max(v_i)|$. The sequence M^f was indeed defined as

$$\begin{aligned} M^1 &= \alpha_n 2^{\lceil log_2(v_{max})\rceil} \\ M^{f+1} &= \alpha_n \epsilon M^f. \end{aligned}$$

We define now a sequence N^f that does not depend on v_{max}, thus removing the need to compute it. In this sequence the starting value is the largest possible extractor, while the relationship between two consecutive extractors is the same that was used to define M^{f+1}:

$$\begin{aligned} N^1 &= 2^{E_{max}} \\ N^{f+1} &= \alpha_n \epsilon N^f. \end{aligned}$$

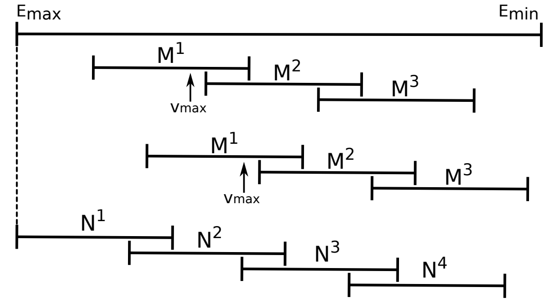

Figure 4. The sequence M^f used in the previous section is displayed in the two top examples. The last example shows the sequence N^f, which depends on the number of elements, but not on the magnitude of v_{max}.

Figure 4 shows the difference between the two definitions. The sequence M^f depends on the magnitude of v_{max} for the magnitude of the first extractor. The sequence N^f starts instead with a fixed N^1. These definitions differ remarkably from those presented in [16], where the ratio between two extractors is fixed a-priory and independently from the number of input values.

Definition 1. An extractor M *supports the value* v_i *among* n if

$$M \geq \alpha_n 2^{\lceil log_2(|v_i|)\rceil}. \quad (3)$$

In the following, when not otherwise stated, we will express that an extractor *supports* a value, or that a value *is supported*, implicitly meaning *among the number of values in the array*.

When comparing this definition with (1), which is a weaker requirement, it is clear that a value M supporting all the n elements of a vector is suitable for extracting them using the pre-rounding techniques explained in the previous

section.

The algorithm that we propose performs a single sweep through the vector and chooses dynamically the set of extractors used to pre-round the values. During the start-up, the lowest extractor N^f supporting the first value v_1 is found and the set of extractors $\{N^f, \dots, N^{f+k}\}$ is chosen for the multi-level pre-rounding. The algorithm keeps track of the maximal value supported by the largest extractor in this set. At every iteration a new value v_i is loaded and processed: if the value is still supported by the largest extractor in the set — i.e., if its absolute value is lower than or equal to the maximal supported value —, then the pre-rounding operation is performed in the same way we presented in the previous section. Otherwise, the set of extractors must be updated by discarding the lowest one and including a larger one at the beginning. The procedure must be repeated until the new largest extractor supports the value v_i.

Figure 5 shows this procedure. In this figure, and in the remaining part of the section, the current largest extractor is designated by M^1, and the set itself by $\{M^1, \dots, M^k\}$.

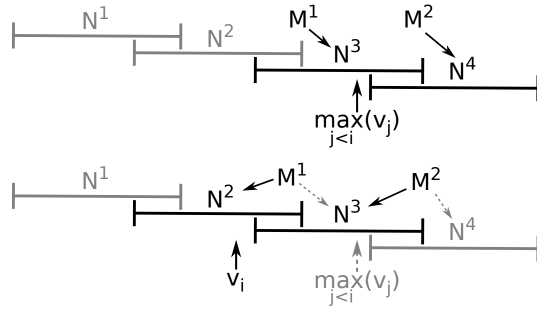

Figure 5. Update of the extractor set in a two-level reduction. The value v_i is supported by the current largest extractor N^3: a larger extractor N^2 enters the set, while the extractor N^4 is discarded.

The contributions extracted with every extractor M^f are summed into the corresponding partial sum T^f. When the procedure explained in the previous paragraph is used to include a new extractor, a new corresponding partial sum must also be initialized, while the last partial sum T^k is discarded. The initialization of the new leading partial sum T^1 is not trivial, as the following example shows.

Example 5. In the `binary16` format, we want to perform a two-level reduction the values $v_1 = fl(0.4) = 1.1001100110_2 \cdot 2^{-2}$ and $4 < v_2 < 5$ among a vector of n values, using the extractors $M^1 = 2^7$ and $M^2 = 2^2$. For a small n (e.g., $n = 20$), v_1 is supported by both extractors, while v_2 is supported by M^1 only. If v_1 is processed first, and the current maximal extractor is M^1, the first and second-level contributions and remainders of v_1 are $q_1^1 = 1.1_2 \cdot 2^{-2}, r_1^1 = 1.10011_2 \cdot 2^{-6}, q_1^2 = 1.1_2 \cdot 2^{-6}, r_1^2 = 1.1_2 \cdot 2^{-10}$. If instead the current largest extractor is M^2, the first-level contribution and remainder are $\tilde{q}_1^1 = 1.0011_2 \cdot 2^{-2}, \tilde{r}_1^1 = 1.1_2 \cdot 2^{-10}$ (the second-level contribution is in this case unimportant). Clearly, $\tilde{q}_1^1 = q_1^1 + q_1^2$ and $\tilde{r}_1^1 = r_1^2$, meaning that the total contributions of v_1 into the sum have been the same and the final remainder is the same. But they are distributed differently among the partial sums. This breaks the reproducibility if at some point a large value (e.g., v_2) requires that the current second-level partial sum is discarded. At that point, depending on the order of processing of the values, two possible results can be obtained: either q_1^2 is discarded, or it is not, because it is part of the current first-level contribution.

To solve the situation depicted in this example, we add one more larger level, the *zeroth level*. The extractor set contains one more entry, M^0, larger than M^1. For every processed value v_i, the contribution q_i^0 is computed and summed in the partial sum T^0. The first-level contribution q_i^1 is computed by extracting the same value v_i using M^1. Note that the zeroth-level remainder r_i^0 is not needed and thus not computed. When the procedure of choosing a new extractor is followed because an unsupported value is found, the new first-level partial sum is initialized to the value contained in the old zeroth-level: $T_{new}^1 = T^0$. The contributions redundantly contained in both the old zeroth and the first level are subtracted: $T_{new}^2 = T^1 - T^0$. The new zeroth-level partial sum is set to zero, since the contributions to this level by values supported by M^1 are exactly zero: $T_{new}^0 = 0$. To clarify the last sentence, we remind that there is an upper bound to the absolute value of v_i given by the fact that it is supported by the old M^1 (see Definition 1). Since $M^0 = \alpha_n \epsilon M_{new}^0$ and $v_i \le M^1 = \alpha_n \epsilon M^0 = (\alpha_n \epsilon)^2 M_{new}^0$, we can eventually bound the absolute value of v_i by $|v_i| < \frac{1}{2}\epsilon M_{new}^0$, which ensures that $v_i \oplus M_{new}^0 = M_{new}^0$, i.e., its contribution is zero. Lemma 1 (appendix[3]) contains a detailed proof of this fact.

Example 6. We apply the derived technique on Example 5. When v_1 is processed and the current largest extractor is M^2, the contribution of v_1 to M^1 and M^2 are computed as $\hat{q}_1^f$: $\hat{q}_1^1 = (v_1 \oplus M^1) \ominus M^1 = 1.1_2 \cdot 2^{-2}$, $\hat{q}_1^2 = (v_1 \oplus M^2) \ominus M^2 = 1.10011_2 \cdot 2^{-2}$. When v_2 is processed and a new extractor enters the computation, in order to recover the correct contributions of v_1 to M^1 and of the remainder r_1^1 to M^2, it is sufficient to do: $q_1^1 = \hat{q}_1^1 = 1.1_2 \cdot 2^{-2}$, $q_1^2 (= (r_1^1 \oplus M^2) \ominus M^2) = \hat{q}_1^2 - \hat{q}_1^1 = 1.1_2 \cdot 2^{-6}$. When compared these values with q_1^1 and q_1^2 those obtained in Example 5, one can see that they match perfectly.

If the array is distributed among p processes, each one owning n_p values, each process must perform a local computation without the need for an initial global reduction to find the maximum value. Since different processes can end the local reduction with different extractor sets, during the final reduction the partial results must be reduced so that the partial sums corresponding to the same level are summed

[3]*An identical version of this paper including the appendix can be downloaded at http://htor.inf.ethz.ch/sec/bitrep-ipdps.pdf.

Algorithm 1 Merge of partial sums: `MergeLevels`

Require: $(h_{max,p}, \underline{T}_p)$ and $(h_{max,q}, \underline{T}_q)$ are the resulting levels and partial sums coming from processes p and q.

1: $h_{diff} = h_{max,p} - h_{max,q}$
2: **if** $h_{diff} \geq 0$, **then**
3: $\quad T_p^f = T_p^f \oplus T_q^{f-h_{diff}}$ **for** $f = h_{diff}$ **to** k
4: **else**
5: $\quad T_p^f = T_p^{f+h_{diff}} \oplus T_q^f$ **for** $f = k$ **to** $-h_{diff}$
6: $\quad T_p^f = T_q^f$ **for** $f = -h_{diff} - 1$ **to** 0
7: **end if**
8: $h_{max,p} = \max\{h_{max,p}, h_{max,q}\}$

Ensure: $(h_{max,p}, \underline{T}_p)$ is the sum of both partial results

together. Algorithm 1, which addresses this issue, is used as operator within the standard `MPI_Reduce` to provide the correct global partial sums. The partial sums are eventually summed by the root process. Algorithm 2 implements the whole procedure.

C. Accuracy

The multi-level pre-rounding reduction with initial maximum computation uses the optimal set of extractors, i.e., whose largest extractor M^1 is the smallest extractor supporting v_{max}. The single-sweep algorithm does not know v_{max}. It fixes the sequence of suitable extractors N^f. At the end of the computation, the largest used extractor is at least as large as the optimal extractor M^1, i.e., the largest extractor of the optimal set. In the best case the largest used extractor is equal to the optimal extractor. It can not always be the case, since only a small number of extractors are part of the sequence N used by the single-sweep algorithm. Figure 6 shows two cases: in the first case the sequence N contains an extractor (N^{f+1}) which is slightly larger than the optimal extractor M^1; in the second case the first element of the sequence N which at least as large as M^1 is N^f, which is far larger than M^1. In the former case the accuracy of the single-sweep algorithm with k levels is similar to the accuracy of the double-sweep algorithm with the same number of levels. In the latter case the accuracy of the single-sweep algorithm is much worse because many significant bits are *wasted* as effect of the selection of a suboptimal largest extractor.

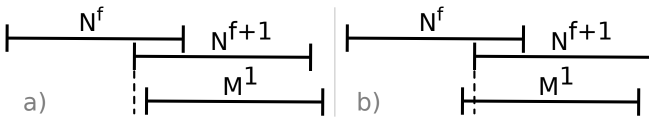

Figure 6. Sketch of *a)* good and *b)* bad case of relationship between optimal largest extractor M^1 and actually used largest extractor in the extractor set for the single-sweep algorithm (N^{f+1} resp. N^f).

Unfortunately, since the sequence N depends just on the number of values n, it is not possible to adjust the extractors to fit the situation better. This leads to possible

Algorithm 2 Multiple-level vector extraction with adaptive extractor on distributed-memory

Require: $\underline{v}$ is a vector of global size n, local size n_p

1: $\alpha = 2^{\lceil log_2(\frac{n}{1-2n\epsilon}) \rceil}$
2: $h = \texttt{argmin}_{f \geq 2} \{N^f : N^f \text{ supports } v_1 \text{ among } n\}$
3: $M^f = N^{h+f}$ for $f \in \{0, 1, \ldots, k\}$
4: $T^f = 0$ for $f \in \{0, 1, \ldots, k\}$
5: $v_{max} = M^1/\alpha_n$
6: **for** i = 1 to n_p in any sequential order **do**
7: $\quad$ **while** $|v_i| > v_{max}$, **do**
8: $\quad\quad$ **for** f = k to 3
9: $\quad\quad\quad M^f = M^{f-1}$; $T^f = T^{f-1}$
10: $\quad\quad$ **end for**
11: $\quad\quad M^2 = M^1$; $\quad T^2 = T^1 \ominus T^0$
12: $\quad\quad M^1 = M^0$; $\quad T^1 = T^0$
13: $\quad\quad M^0 = N^{h-2}$; $\quad T^0 = 0$
14: $\quad\quad h = h - 1$; $\quad v_{max} = M^1/\alpha_n$
15: $\quad$ **end while**
16: $\quad q^0 = (M^0 \oplus v_i) \ominus M^0$
17: $\quad T^0 = T^0 \oplus q^0$
18: $\quad r^0 = v_i$
19: $\quad$ **for** f = 1 to k
20: $\quad\quad q^f = (M^f \oplus r^{f-1}) \ominus M^f$
21: $\quad\quad T^f = T^f \oplus q^f$
22: $\quad\quad r^f = r^{f-1} \ominus q^f$
23: $\quad$ **end for**
24: **end for**
25: $T^1 = T^1 \ominus T^0$
26: `Reduce` $\left((h, [T^0, T^1, \ldots T^k]), \texttt{MergeLevels}\right)$
27: $t = 0$
28: **for** f = k to 0 **do**: $t = t \oplus T^f$; **end for**

Ensure: t is the deterministic sum of the k-level high-order parts of the values v_i

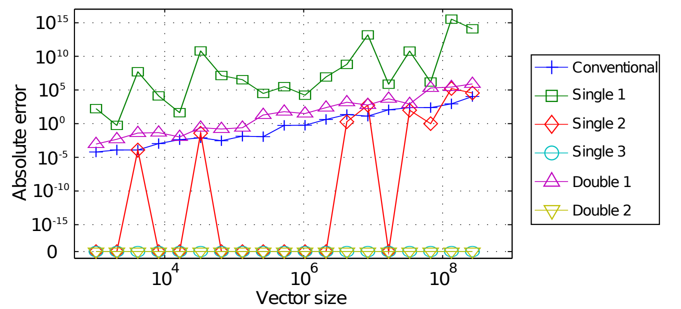

Figure 7. Absolute error comparison of conventional reduction, single-sweep and double-sweep reduction in double-precision with exponentially distributed input data ($\lambda = 10^{-8}$). *Single 1*, *Single 2* and *Single 3* labels refer to the single-sweep algorithm with corresponding number of levels. *Double 1* and *Double 2* labels refer to the double-sweep algorithm.

large oscillations of the actual accuracy and to a loose error bound. Indeed, the only a-priori bound is given by the fact that there is a maximal difference between the exponent of the largest used extractor and that of the optimal extractor. This difference is the difference between two levels. In other

words, the absolute error of the single-sweep algorithm with k levels is at most as large as the absolute error of the double-sweep algorithm with $k-1$ levels, which is given in (2). The reason for this is that the optimal extractor lies between the largest extractor actually used and the second-level used extractor. Therefore, the $(k+1)$-th actually used extractor is at most as large as the (k)-th extractor after the optimal one, i.e., the last extractor used by the k-th level double-sweep algorithm.. The actual absolute error strongly depends on the maximum value of the array, which in turn depends on the distribution of data. Different distributions lead to different absolute errors.

Figure 7 shows double-precision experimental results in which the absolute error of both algorithms and of the conventional algorithm are compared. We explore the error when the vector elements are exponentially distributed with $\lambda = 10^{-8}$ to maximize the error for reproducible algorithms. As expected, the accuracy of the single-sweep algorithm depends on the size of the vector and is not monotonic because a small variation of the size of the vector can determine an improvement or a worsening of the situation depicted in Figure 6. The accuracy of Single 2, i.e., the two-levels single-sweep algorithm, is approximately the same of Double 1, i.e., one-level double-sweep, in the worst cases; in the best cases its accuracy is complete (it is accurate to the least significant bit). Single 3 and Double 2 always returned the exact result in every experiment. The exact value is computed through a 4-level double-sweep algorithm, which gives always the exact result if the input values are non-negative, as the maximum absolute error is still less significant than the least significant bit of the result.

D. Performance

We now compare the performance of the distributed reduction using the conventional algorithm, the two-level double-sweep, and the three-level single-sweep with a theoretical model and experimental results. We select one more level in the single-sweep case in order to maintain at least the same accuracy as the double-sweep algorithm.

The conventional algorithm performs n_p flops locally to compute the sum, then an `MPI_Reduce`. The double-sweep algorithm with $k=2$ performs a local max reduction (one flop for the absolute value, one for max, $2n_p$ flops in total) followed by an `MPI_Allreduce`, then a local computation involving $8n_p$ flops followed by an `MPI_Reduce`. The single-sweep algorithm with $k=3$ performs instead a single `MPI_Reduce` operation after a local computation involving $16n_p$ flops. In the out-of-cache case, the main memory accesses are more relevant than the number of flops. The double-sweep algorithm reads the whole array twice ($16n_p$ bytes in double-precision), while the single-sweep algorithm reads it just once ($8n_p$ bytes). The given numbers are first-order approximations.

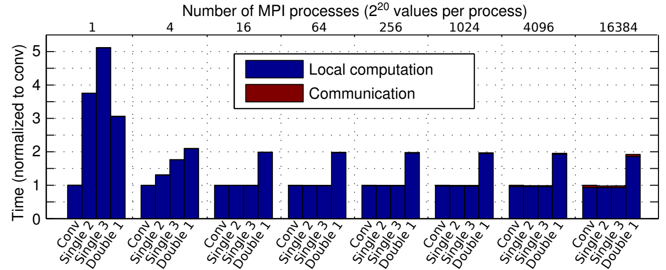

Figure 8. Performance comparison of conventional reduction performed with MKL (*Conv*), single-sweep reduction with two levels (*Single2*), with three levels (*Single3*) and double-sweep reduction with 1 level (*Double 1*) on varying number of processes, each owning 2^{20} double-precision values,

We define α as the communication latency, β as the main memory bandwidth, and γ as the FPU bandwidth (i.e., the number of floating-point operations per second). We assume that `MPI_Reduce` uses a binary tree communication pattern, that the number of communicated values (no more than $k+2$ values, i.e., usually no more than 5) is negligible and that the local computation and main memory prefetch into the cache can overlap. The time required by the double-sweep algorithm is thus $2\lceil log_2(p)\rceil \cdot \alpha + \max\{^{16n_p}/_{\beta}, ^{10n_p}/_{\gamma}\}$, while the time required by the single-sweep algorithm is $\lceil log_2(p)\rceil \cdot \alpha + \max\{^{8n_p}/_{\beta}, ^{16n_p}/_{\gamma}\}$. In the out-of cache case, the single-sweep algorithm is expected to be always faster than the double-sweep one, as it requires less communication and less main memory reads. In the in-cache case the single-sweep algorithm can be expected to be faster if $6\ ^{n_p}/_{\gamma} < \lceil log_2(p)\rceil \cdot \alpha$ (more accurate models may depend on CPU or GPU type).

We performed a test in double-precision format on the Piz Daint cluster[4]. Figure 9 shows the results of local computation, communication and overall performance on 16384 processes, whereas Figure 8 shows the performance of the reduction algorithms on a varying number of processes. The results show that the communication required by the double-sweep dominates the runtime in the in-cache case, while the single-sweep algorithm is balanced in both in-cache and out-of-cache cases. In this experiment the single-sweep algorithm is faster than the double-sweep algorithm without loss of accuracy and its overhead to the conventional algorithm is less than 10% for large arrays.

IV. Case studies

We now discuss how our developed tools can be applied to various science domains in order to develop reproducible parallel programs.

A. Linear algebra and applications thereof

Linear algebra contains a broad set of operations that form the basis for many computational applications. Constructing a reproducible library for linear algebra means moving an

[4]http://www.cscs.ch/computers/piz_daint/index.html

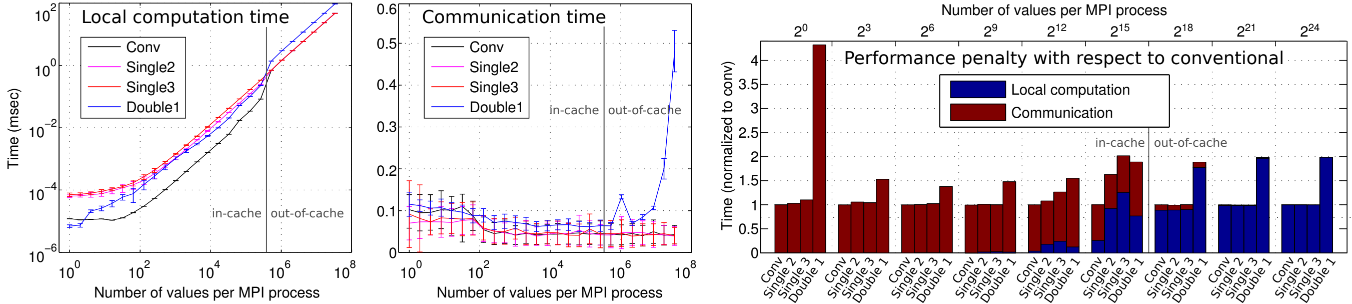

Figure 9. Performance comparison of conventional reduction performed with MKL (*Conv*), single-sweep reduction with two levels (*Single2*), with three levels (*Single3*) and double-sweep reduction with 1 level (*Double 1*) in double-precision format with 16384 MPI processes. The gray bar separates the in-cache and out-of-cache cases and highlightes the different performance behaviour due to the increasing relevance of the loading from the main memory.

important step towards the ability to construct reproducible scientific applications. The problems that can threaten reproducibility are mainly due to reductions:

- Basic linear algebra operations like scalar product, matrix-vector and matrix-matrix multiplications are essentially particular types of reductions. By applying the pre-rounding technique for reductions, these operations can be made reproducible also for the distributed case.
- Matrix decompositions can be performed by just making use of the mentioned basic operations. LU decompositions, e.g., can be implemented, as in the commonly used function `getf2`, as a sequence of rank-1 updates, which are in turn made reproducible by applying the already mentioned techniques. The `getrf` function decomposes instead the matrix in blocks and applies a sequence of `getf2` and other lower-level operations. This method can be made reproducible if the block size is fixed, which can be a restrictive requirement.
- Construction of sparse matrices, for example in the compressed row storage format (CRS), is usually performed by incrementally adding non-zero values. If several entries refer to the same position in the matrix, a reduction must be performed, which is the only issue that could compromise reproducibility.
- Sparse solvers like the preconditioned conjugate gradient rely on basic linear algebra (matrix-vector multiplication in particular) applied to a sparse matrix and fast solution of a system with the preconditioner matrix. All these operations can be made in deterministic as shown, which would automatically lead to reproducible solvers.

A broadly used technique to solve partial differential equations is the finite elements method. The steps needed to solve a finite element problem starting from a spatial discretization are *a)* the computation of the contributions of each basis function into the stiffness matrix A and into the right-hand side vector ϕ, *b)* the assembly of the stiffness matrix in a sparse format like CRS, *c)* the assembly of the right-hand side vector ϕ and *d)* the solution of the system $Ax = \phi$. To compute the contributions of the basis functions, a quadrature must be performed. As a quadrature is a weighted sum of the evaluation of the function to be integrated in a set of points, two requirements must be satisfied in order for point *a)* to be reproducible: the function must be deterministic and the reduction must be reproducible. The former is achieved by considering the methods explained in Section II, while the latter can be achieved by fixing the order of the weighted sum. Parallelism in usually not applied when computing single quadratures, and mesh points and basis functions have a reproducible order defined by the mesh. Fixing the sum order is thus feasible and has no overhead. We have shown that assembling a CRS matrix can be made reproducible even if the order in which the single values appear is not determined, which can be the case if a different data distribution is applied. As result, the matrix A and the right-hand side vector ϕ are fully reproducible. The solution of the system through a sparse solver was also treated previously, leading to the conclusion that the finite elements method can be implemented in a reproducible way.

B. Stencils

Other widely used methods involve computations performed on structured grids in form of stencils: every entry of a field is computed at every iteration by evaluating a function whose input are the values contained in other fields. The data is usually distributed among the participating computational units. Since the shape of the stencils are fixed, the portions of the fields needed to compute a specific value are limited and well-known. In order to update a value all the values that take place in the corresponding computation must be made available to the computational unit responsible for that value. If these values are non-local, a communication is performed.

The assumptions that ensure that a stencil computation is reproducible are *a)* that the functions used to update the values are deterministic and *b)* that all values needed to perform the update are consistently available. The latter is a requirement for the stencil computation to be consistent and is therefore assumed to be true in any case. For the former, we refer to Section II.

V. Related work

As already mentioned, work on reproducible reductions has been performed by Demmel and Nguyen [14], [15], [16]. An asynchronous parallel model to provide deterministic reductions has been developed by Budlimič et al. [17]. Internally deterministic algorithms have been developed by Blelloch et al. [18].

Sources of non-determinism in particular in MPI communications have been studied by Chiang et al. [19]. Other studies on non-determinism caused by parallelism have been performed by Bergan et al. [20], Bocchino et al. [21], Leiserson et al. [22], Olszewski et al. [23].

Jooybar et al. proposed [24] a GPU architecture capable of enforcing bit-reproducibility in computations exploiting at the same time the massive multi-threaded parallelism. Further work on IEEE-754 floating-point operations performed on GPU architectures with OpenCL has been performed by Leeser et al. [25].

Bit-reproducibility is a problem that needs attention at many levels. This work focused on the algorithmic/programming part. Our approach is thus different from that of many other works. The heterogeneity of these works on the same topic shows its importance and its vastness.

VI. Conclusions

We presented techniques and algorithms that solve the problem of achieving bit-reproducible results for specific tasks maintaining, or improving, accuracy and performance over conventional, non-reproducible counterparts. The research on this promising topic is active, but still in early stages. With our contribution, we pursued the goal of providing practical knowledge, which can be directly used to build applications that can benefit from bit-reproducibility. We released the code used in this work under the BSD license at the URL http://github.com/andyspiros/bitrep, in the hope that a collaborative effort to build a toolset for reproducible computations can be started. The practical knowledge that we provided can be also used at all levels, e.g., by library vendors to ensure the reproducibility of their products.

In an HPC context that is developing towards increasing heterogeneity, the ability to construct software products able to consistently and efficiently run on different platforms can make the difference between promising and successful efforts. In highly non-linear systems, in particular in climate science, the goal of achieving bit-reproducibility can be crucial in order to exploit current and future heterogeneous hardware platforms. From this point of view, the role of bit-reproducibility is bound to become increasingly important.

Acnkowledgments

We thank Hong Diep Nguyen for helpful discussions related to the details of the comparison of the single-sweep algorithm and the 1-Reduction in [15].

References

[1] R. Skålin and D. Bjørge, "Implementation and Performance of a Parallel Version of the HIRLAM Limited Area Atmospheric Model." *Par. Comp.*, vol. 23, no. 14, pp. 2161–2172, 1997.

[2] E. N. Lorenz, "The predictability of a flow which possesses many scales of motion," *Tellus*, vol. 21, no. 3, p. 289, 1969.

[3] J. Teixeira, C. A. Reynolds, and K. Judd, "Time step sensitivity of nonlinear atmospheric models: numerical convergence, truncation error growth, and ensemble design," *Journal of the atmospheric sciences*, vol. 64, no. 1, pp. 175–189, 2007.

[4] T. N. Palmer, "Predicting uncertainty in forecasts of weather and climate," *Progress in Physics*, vol. 63, no. 2, p. 71, 2000.

[5] "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Std 754-1985*, 1985.

[6] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, 2008.

[7] M. J. Corden and D. Kreitzer, "Consistency of floating-point results using the Intel compiler or why doesn't my application always give the same answer," Intel Corp., Tech. Rep., 2009.

[8] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[9] D. Defour, G. Hanrot, V. Lefèvre, J.-M. Muller, N. Revol, and P. Zimmermann, "Proposal for a Standardization of Mathematical Function Implementation in Floating-Point Arithmetic," *Numerical Algorithms*, no. 1-4, pp. 367–375, 2004.

[10] V. Lefevre and J. M. Muller, "Worst cases for correct rounding of the elementary functions in double precision," in *15th IEEE Symposium on Computer Arithmetic*, 2001, pp. 111–118.

[11] J. M. Muller, F. de Dinechin, C. Lauter *et al.* Correctly-Rounded mathematical library. [Online]. Available: http://lipforge.ens-lyon.fr/www/crlibm

[12] NVidia. CUDA C programming guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[13] S. M. Rump, "Ultimately fast accurate summation," *SIAM Journal on Scientific Comp.*, vol. 31, no. 5, p. 3466, 2009.

[14] J. Demmel and H. D. Nguyen, "Fast Reproducible Floating-Point Summation," in *21st IEEE Symp. on Computer Arithmetic*, ser. ARITH'13, 2013, pp. 163–172.

[15] ——, "Numerical reproducibility and accuracy at exascale," in *21st IEEE Symp. on Computer Arithmetic*, 2013, p. 235.

[16] ——, "Parallel Reproducible Summation," *Submitted paper*, 2014. [Online]. Available: http://www.eecs.berkeley.edu/ hdnguyen/public/papers/repsum.pdf

[17] Z. Budimlic, M. Burke, K. Knobe, R. Newton, D. Peixotto, V. Sarkar, and E. Westbrook, "Deterministic reductions in an asynchronous parallel language," *2nd WoDet*, 2011.

[18] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *Proc. of the 17th ACM Symp. on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012, pp. 181–192.

[19] W.-F. Chiang *et al.*, "Determinism and Reproducibility in Large-Scale HPC Systems." Workshop on Determinism and Correctness in Parallel Programming, 2013.

[20] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "CoreDet: a compiler and runtime system for deterministic multithreaded execution," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 53–64.

[21] R. Bocchino, V. Adve, S. Adve, and M. Snir, "Parallel programming must be deterministic by default," *First USENIX workshop on hot topics in parallelism (HOTPAR 2009)*, 2009.

[22] C. E. Leiserson, T. B. Schardl, and J. Sukha, "Deterministic parallel random-number generation for dynamic-

multithreading platforms," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 193–204, 2012.
[23] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," in *ACM Sigplan Notices*, vol. 44, no. 3. ACM, 2009, pp. 97–108.
[24] H. Jooybar, W. W. Fung, M. O'Connor, J. Devietti, and T. M. Aamodt, "GPUDet: a deterministic GPU architecture," in *Proc, of the 18th Intl. Conf. on Architectural Support for Prog. Lang, and Operating Systems*. ACM, 2013, pp. 1–12.
[25] M. Leesera, J. Ramachandranb, T. Wahlb, and D. Yablonskic, "OpenCL Floating Point Software on Heterogeneous Architectures – Portable or Not?" in *Workshop on Numerical Software Verification (NSV)*, 2012.

APPENDIX

This appendix provides additional proofs and algorithms that are not a core part of the paper. The appendix can thus be omitted from the review process. This text will be removed in the camera-ready version.

Algorithm 3 implements the double-sweep method. It is a modified version of Algorithm 9 in [14]. The modification consists by the fact that the theoretical algorithm proposed by Demmel and Nguyen performs actually $k+1$ sweeps: one sweep at the beginning to find the maximal value, then one sweep per level. The latter part can be trivially adapted to a single sweep to improve the memory access pattern. Another change is the extraction operation. As Demmel and Nguyen explain in detail, in order to target directed rounding modes, different kind of extraction operations can be performed. In their algorithm, they target directed roundings and extract the value v_i with $T^f = T^f \oplus v_i$. This approach does not work in the round-to-nearest with ties to even mode that we are targeting. We rely therefore on the method shown in the same paper, Algorithm 2.

Lemma 1. Given a value v_i supported by an extractor N^f among n (according to Definition 1), the contribution of v_i to the sum carried by extractor N^{f-2} is zero, under the assumption that n is small ($n\epsilon < 0.1$ is enough).

Proof: By definition $N^f = \alpha_n \epsilon N^{f-1} = \alpha_n \epsilon \left(\alpha_n \epsilon N^{f-2}\right) = \alpha_n^2 \epsilon^2 N^{f-2}$. We also remind the definition $\alpha_n = 2^{\lceil log_2(n/(1-2n\epsilon)) \rceil}$. Since $n\epsilon < 1/8$, $n/(1-2n\epsilon) \leq 2n \Rightarrow \alpha_n \leq 2^{\lceil log_2(2n) \rceil} \leq 2^{\lceil log_2(n)+1 \rceil} \leq 2^{\lceil log_2(n) \rceil + 1} \leq 2 \cdot 2^{\lceil log_2(n) \rceil} \leq 4 \cdot n$. We can now bound N^f by: $N^f \leq 16\epsilon^2 N^{f-2}$. From Definition 1: $|v_i| \leq N^f \leq 16\epsilon^2 N^{f-2}$. Clearly $16\epsilon < \frac{1}{4} \Rightarrow |v_i| \leq \frac{1}{4} N^{f-2}$. Therefore, $v_i \oplus N^{f-2} = N^{f-2}$. ∎

Algorithm 3 Multi-level vector extraction, adapted from [14], Algorithm 9

Require: $\underline{v}$ is the local part with size n_p of a global vector of size n, $k \geq 1$
1: $m = \mathtt{max}_{i \leq n_p} \{|v_i|\}$
2: $\overline{m} = \mathtt{AllReduce}(m, \mathtt{MAX})$
3: $\alpha_n = 2^{\lceil log_2(\frac{n}{1-2n\epsilon}) \rceil}$
4: $M^1 = \alpha_n \cdot 2^{\lceil log_2(\overline{m}) \rceil}$
5: $M^f = \alpha_n \epsilon M^{f-1}$ **for** $2 \leq f \leq k$
6: $T^f = 0$ **for** $1 \leq f \leq k$
7: **for** i = 1 to n in any sequential order **do**
8: $r^0 = v_i$
9: **for** f = 1 to k **do**
10: $S^f = (r^{f-1} \oplus M^f) \ominus M^f$
11: $T^f = T^f \oplus q^f$
12: $r^f = r^{f-1} - q^f$
13: **end for**
14: **end for**
15: $\mathtt{Reduce}\left(\left[T^1, \ldots T^f\right], \mathtt{SUM}\right)$
16: $t = \sum_{f=1}^{k} T^f$
Ensure: t is the deterministic sum of the k levels of high-order parts of the global vector