Iakymchuk, R.; Barreda, M.; Wiesenberger, M.; Aliaga, JI.; Quintana Ortí, ES. (2020). Reproducibility strategies for parallel preconditioned Conjugate Gradient. Journal of Computational and Applied Mathematics. 371:1-13. https://doi.org/10.1016/j.cam.2019.112697

Reproducibility Strategies for Parallel Preconditioned Conjugate Gradient

Roman Iakymchuk[a], Maria Barreda[b], Matthias Wiesenberger[c], José I. Aliaga[b], Enrique S. Quintana-Ortí[d]

[a]*Fraunhofer ITWM, Germany and KTH Royal Institute of Technology, Sweden*
[b]*Universitat Jaime I, Spain*
[c]*Technical University of Denmark (DTU), Denmark*
[d]*Universidad Politècnica de València, Spain*

Abstract

The Preconditioned Conjugate Gradient method is often used in numerical simulations. While being widely used, the solver is also known for its lack of accuracy while computing the residual. In this article, we aim at a twofold goal: enhance the accuracy of the solver but also ensure its reproducibility in a message-passing implementation. We design and employ various strategies starting from the ExBLAS approach (through preserving every bit of information until final rounding) to its more lightweight performance-oriented variant (through expanding the intermediate precision). These algorithmic strategies are reinforced with programmability suggestions to assure deterministic executions. Finally, we verify these strategies on modern HPC systems: both versions deliver reproducible number of iterations, residuals, direct errors, and vector-solutions for the overhead of only 29 % (ExBLAS) and 4 % (lightweight) on 768 processes.

Keywords: Reproducibility, accuracy, floating-point expansion, long accumulator, fused multiply-add, preconditioned Conjugate Gradient, High-Performance Computing.
2000 MSC: 65Y05, 65Y20, 65-05, 65K99, 68Q10, 68W10.

1. Introduction

Many current scientific and engineering problems involve solution of large and sparse linear systems of equations. Some traditional examples appear in circuit and device simulation, quantum physics, large-scale eigenvalue computations, nonlinear sparse equations, and all sorts of applications that include the discretization of partial differential equations (PDEs) [1]. For many problems (especially those associated with 3-D models), the size and complexity of these systems have turned iterative projection methods, based on Krylov subspaces, into a highly competitive approach compared with direct solvers [2]. For instance, the Conjugate Gradient (CG) method is one of the most efficient Krylov subspace-based algorithms for the solution of sparse linear systems when the coefficient matrix is symmetric positive definite (s.p.d.) [2]. Preconditioning is usually incorporated in real implementations of the method in order to accelerate the convergence of the method and improve its numerical features, obtaining the Preconditioned Conjugate Gradient (PCG) method.

One would expect that the results of the sequential and parallel implementations of PCG are identical, for instance, in the number of iterations, the intermediate and final residuals, as well as the output vector. However, in practice this is not often the case due to different reduction trees – the Message Passing Interface (MPI) implementations (libraries) [3] offer up to 14 different implementations for reduction –, data alignment, instructions used, etc. Each of these factors impacts the order of floating-point operations, which are commutative but not associative, and, therefore, violates reproducibility.

Ensuring identical and accurate outputs of computations, including the residuals/ errors, is in our view a way to ensure *robustness* and *correctness* of iterative methods. The robustness and correctness in this case have a threefold goal: *reproducibility* of the results with the *accuracy guarantee* as well as *sustainable (energy-efficient)* algorithmic solutions.

Email addresses: roman.iakymchuk@itwm.fhg.de (Roman Iakymchuk), mvaya@icc.uji.es (Maria Barreda), mattwi@fysik.dtu.dk (Matthias Wiesenberger), aliaga@icc.uji.es (José I. Aliaga), quintana@disca.upv.es (Enrique S. Quintana-Ortí)

In general, reproducibility is defined as the ability to obtain a bit-wise identical floating-point result from multiple runs of the code on the same input data. We enhance this definition by incorporating accuracy: hence, *reproducibility is the ability to obtain a bit-wise identical and accurate result for multiple executions on the same data*. However, the bit-wise reproducibility is a complex and often expensive task. In particular, this is due to the required modifications to the algorithm, its underlying parts like the BLAS (Basic Linear Algebra Subprograms) routines [4, 5], or both. Such modifications are necessary to preserve every bit of information (both result and error) [6] or, alternatively, to chop off some parts of data and operate on the remaining most-significant parts [7]. The later can also be viewed as switching to fixed-precision computations. Additionally, the bit-wise reproducibility can get costly with the overhead of at least 8 % for parallel reduction [6, 7], up to 2x-4x for matrix-vector product [8], and more than 10x for matrix-matrix multiplication [9].

In this paper, we aim to revisit reproducibility and make it lighter through reducing its negative impact on performance and minimizing changes to both algorithm and its building blocks. We argue and raise a question: *Can reproducibility of algorithms be ensured by design with both minimal changes to algorithms and almost negligible overhead?* Hence, our idea is to address those parts of algorithms that violate associativity – such as parallel reductions, possible replacements by compilers of $a * b + c$ in the favor of fused multiply-add (`fma`) operation, etc. – as well as to combine that with sequential executions of sub-blocks/ -routines. Such sequential execution of operations is reproducible under some constraints, for example the same initial conditions on the input data like data alignment.

We propose to verify this idea (both algorithmic and programmability) on a typical sparse linear algebra solver such as PCG and ensure its reproducibility on parallel distributed-memory systems. As a consequence, we also address the common issue of sparse iterative solvers – the accuracy while computing the residual – and propose to use solutions that offer reproducibility (and potentially correct-rounding) only while computing the corresponding dot products.

Hence, we construct our primary reproducible solution for PCG based on the ExBLAS approach, which is robust but costly, as well as derive its lightweight version, which is faster but less generic. While ExBLAS represents one of the classic computer arithmetic approaches [10, 11, 12, 13] that often rely upon conducting rigorous proofs of derived algorithms to ensure strict upper bound errors for all cases, which are considered by some of those authors as pessimistic estimates, we seek for more optimistic/ realistic estimates and, hence, solutions that would reflect and adjust algorithms to the problem at hand. For instance, the Kulisch long accumulator, which is the cornerstone algorithm of ExBLAS, is designed to handle severe (ill-conditioned) cases with very broad dynamic ranges, while in practice "100 bits suffice for many HPC applications" as noted by David Bailey at ARITH-21 [14]. This idea inspired the ARM team (Lutz, Burgess et al) to design a mini long accumulator with the limited range [15, 16]. Therefore, we foresee to explore this motivated-by-practice idea of moderately conditioned problems with moderate dynamic ranges in order to derive a lightweight algorithmic solution from ExBLAS. For that, we reply on a floating-point expansion (FPE) that is the other core algorithmic component in the ExBLAS approach.

In order to show applicability and feasibility of the aforementioned idea with the ExBLAS- and FPE-based strategies are employed in the flat MPI implementation of PCG, we use an example of a 3D Poisson's equation with 27 stencil points. *To sum up*: the FPE-based (we also call it Opt) solution is efficient and fast, but it is limited to cases where the condition number and/ or the dynamic range do not exceed certain limits, e.g. the dynamic range is below 10^{50}; the ExBLAS-based solution is reserved for cases where we do not have any information about the data condition number and the dynamic range.

This article is organized as follows. Section 2 reviews several aspects of computer arithmetic, in particular floating-point expansion and long accumulator, as well as the ExBLAS approach for accurate and reproducible computations. Section 3 introduces the PCG algorithms and describes in details its MPI implementation. We present strategies for ensuring reproducibility of PCG in Section 4 and evaluate corresponding implementations in Section 5. Finally, Section 6 reviews related work, while Section 7 draws conclusions and outlines future directions.

2. Background

At first, we briefly introduce the floating-point arithmetic that consists in approximating real numbers by numbers that have a finite, fixed-precision representation. These numbers are composed of a significand, an exponent, and a sign: $x = \pm \underbrace{x_0.x_1 \ldots x_{M-1}}_{mantissa} \times b^e$, $0 \le x_i \le b-1$, $x_0 \neq 0$, where b is the basis (2 in our case), M is the precision, and e stands for the exponent that is bounded ($e_{\min} \le e \le e_{\max}$).

The IEEE 754 standard [17], created in 1985 and then revised in 2008, has led to a considerable enhancement in the reliability of numerical computations by rigorously specifying the properties of floating-point arithmetic. This standard is now adopted by most processors, thus leading to a much better portability of numerical applications. The standard specifies floating-point formats, which are often associated with precisions like *binary16*, *binary32*, and *binary64*, see Table 1. Floating-point representation allows numbers to cover a wide *dynamic range* that is defined as the absolute ratio between the number with the largest magnitude and the number with the smallest non-zero magnitude in a set. For instance, `binary64` (double-precision) can represent positive numbers from 4.9×10^{-324} to 1.8×10^{308}, so it covers a dynamic range of 3.7×10^{631}.

Table 1: Parameters for three IEEE arithmetic precisions; quadruple (128 bits) is omitted.

Type	Size	Significand	Exponent	Rounding unit	Range
half	16 bits	11 bits	5 bits	$u = 2^{-11} \approx 4.88 \times 10^{-4}$	$\approx 10^{\pm 5}$
single	32 bits	24 bits	8 bits	$u = 2^{-24} \approx 5.96 \times 10^{-8}$	$\approx 10^{\pm 38}$
double	64 bits	53 bits	11 bits	$u = 2^{-53} \approx 1.11 \times 10^{-16}$	$\approx 10^{\pm 308}$

The IEEE 754 standard requires correctly rounded results for the basic arithmetic operations ($+, -, \times, /, \sqrt{}$, `fma`). It means that the operations are performed as if the result was first computed with an infinite precision and then rounded to the floating-point format. The correct rounding criterion guarantees a unique, well-defined answer, ensuring bit-wise reproducibility for a single operation. Several rounding modes are provided. The standard also contains the reproducibility clause that forwards the reproducibility issue to language standards. Emerging attention to reproducibility strives to draw more careful attention to the problem by the computer arithmetic community. It has led to the inclusion of error-free transformations (EFTs) for addition and multiplication – to return the exact outcome as the result and the error – to assure numerical reproducibility of floating-point operations, into the revised version of the standard. These mechanisms, once implemented in hardware, will simplify our reproducible algorithms – like the ones used in the ExBLAS [6], ReproBLAS [7], OzBLAS [18] libraries – and boost their performance.

There are two approaches that enable the addition of floating-point numbers without incurring round-off errors or with reducing their impact. The main idea is to keep track of both the result and the errors during the course of computations. The first approach uses EFT to compute both the result and the rounding error and stores them in a floating-point expansion (FPE), which is an unevaluated sum of p floating-point numbers, whose components are ordered in magnitude with minimal overlap to cover the whole range of exponents. Typically, FPE relies upon the use of the traditional EFT for addition that is `twosum` [19] (Algorithm 1) and for multiplication that is `twoprod` EFT [20] (Algorithm 2). The second approach projects the finite range of exponents of floating-point numbers into a long vector so called a long (fixed-point) accumulator and stores every bit there. For instance, Kulisch [21] proposed to use a 4288-bit long accumulator for the exact dot product of two vectors composed of `binary64` numbers; such a large long accumulator is designed to cover all the severe cases without overflows in its highest digit.

Algorithm 1: Error-free transformation for the summation of two floating-point numbers.

```
Input: $a, b$ are two floating-point numbers.
Output: $r, s$ are the result and the error, resp.
Function $[r, s] = twosum(a, b)$
    $r := a + b$
    $z := r - a$
    $s := (a - (r - z)) + (b - z)$
```

Algorithm 2: Error-free transformation for the product of two floating-point numbers.

```
Input: $a, b$ are two floating-point numbers.
Output: $r, s$ are the result and the error, resp.
Function $[r, s] = twoprod(a, b)$
    $r := a * b$
    $s := fma(a, b, -r)$
```

2.1. ExBLAS – Exact BLAS

The ExBLAS project [22] is an attempt to derive fast, accurate, and reproducible BLAS library by constructing a multi-level approach for these operations that are tailored for various modern architectures with their complex multi-level memory structures. On one side, this approach is aimed to be fast to ensure similar performance compared to the non-deterministic parallel versions. On the other side, the approach is aimed to preserve every bit of information before the final rounding to the desired format to assure correct-rounding and, therefore, reproducibility. Hence, ExBLAS combines together long accumulator and FPE into algorithmic solutions as well as efficiently tunes and implements them on various architectures, including conventional CPUs, Nvidia and AMD GPUs, and Intel Xeon Phi co-processors (for details we refer to [6]). Thus, ExBLAS assures reproducibility through assuring correct-rounding.

The corner stone of ExBLAS is the reproducible parallel reduction, which is at the core of many BLAS routines. The ExBLAS parallel reduction relies upon FPEs with the `twosum` EFT [19] and long accumulators, so it is correctly rounded and reproducible. In practice, the latter is invoked only once per overall summation that results in the little overhead (less than 8 %) on accumulating large vectors. Our interest in this article is the dot product of two vectors, which is another crucial fundamental BLAS operation. The EXDOT algorithm is based on the previous EXSUM algorithm and the `twoprod` EFT [20] (see Algorithm 2): the algorithm accumulates the result and the error of `twoprod` to same FPEs and then follows the EXSUM scheme. These and the other routines – such as matrix-vector product, triangular solve, and matrix-matrix multiplication – are distributed in the ExBLAS library (`https://github.com/riakymch/exblas`). Since EXDOT supports only GPUs, here we derive its distributed version with two FPEs underneath (one for the result and the other for the error) that are merged at the end of computations.

3. Algorithm(s)

We present here the PCG algorithm: its design and implementation with Message Passing Interface (MPI).

3.1. Preconditioned Conjugate Gradient Solver

We consider the linear system $Ax = b$, where the coefficient matrix $A \in \mathbb{R}^{n \times n}$ is sparse and symmetric positive definite (s.p.d.), with n_z nonzero entries; $b \in \mathbb{R}^n$ is the right-hand side vector; and $x \in \mathbb{R}^n$ is the sought-after solution vector. The algorithmic description of the classical iterative PCG is presented in Figure 1. The loop body consists of a sparse matrix-vector product (SpMV) (S1), three DOT products (S2, S6, and S8), three AXPY (-like) operations (S3, S4, and S7), the preconditioner application (S5), and a few scalar operations [1]. For simplicity, in our implementation of the PCG method we integrate the Jacobi preconditioner [2], which is composed of the elements in the diagonal of the matrix ($M := D = diag(A)$). In consequence, the application of the preconditioner is conducted on a vector and requires an element-wise multiplication of two vectors.

Compute preconditioner for $A \rightarrow M$
Set starting guess $x^{(0)}$
Initialize $z^{(0)}, d^{(0)}, \beta^{(0)}, \tau^{(0)}, l := 0$ (iteration count)

$r^{(0)} := b - Ax^{(0)}$
$\tau^0 :=< r^{(0)}, r^{(0)} >$
while $(\tau^{(l)} > \tau_{\max})$

Step	Operation	Kernel
S1 :	$w^{(l)} := Ad^{(l)}$	SpMV
S2 :	$\rho^{(l)} := \beta^{(l)} / < d^{(l)}, w^{(l)} >$	DOT product
S3 :	$x^{(l+1)} := x^{(l)} + \rho^{(l)} d^{(l)}$	AXPY
S4 :	$r^{(l+1)} := r^{(l)} - \rho^{(l)} w^{(l)}$	AXPY
S5 :	$z^{(l+1)} := M^{-1} r^{(l+1)}$	Apply precond.
S6 :	$\beta^{(l+1)} := < z^{(l+1)}, r^{(l+1)} >$	DOT product
S7 :	$d^{(l+1)} := (\beta^{(l+1)} / \beta^{(l)}) d^{(l)} + z^{(l+1)}$	AXPY-like
S8 :	$\tau^{(l+1)} := < r^{(l+1)}, r^{(l+1)} >$	DOT product
	$l := l + 1$	

end while

Figure 1: Formulation of the PCG solver annotated with computational kernels. The threshold $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution. In the notation, $< \cdot, \cdot >$ computes the DOT (inner) product of its vector arguments.

Compute preconditioner for $A \rightarrow M$
Set starting guess x
Initialize $z, d, \beta, \tau, l := 0$

$r := b - Ax, \tau :=< r, r >$
while $(\tau > \tau_{\max})$

Step	Operation	Communication
	$\beta' := \beta$	–
S1 :		
S1.1 :	$d \rightarrow e$	Allgatherv
S1.2 :	$w := Ae$	–
S2 :	$\rho := \beta / < d, w >$	Allreduce
S3 :	$x := x + \rho d$	–
S4 :	$r := r - \rho w$	–
S5 :	$z := M^{-1} r$	–
S6 :	$\beta := < z, r >$	Allreduce
S8 :	$\tau := < r, r >$	Allreduce
S7 :	$d := (\beta / \beta') d + z$	–
	$l := l + 1$	

end while

Figure 2: Message-passing formulation of the PCG solver annotated with communication.

3.2. Message-passing Parallel PCG Implementation

In this subsection we perform a communication analysis of a message-passing implementation of the PCG solver that operates in a distributed-memory platform. For clarity, hereafter we will drop the superindices that denote the iteration count in the variable names. Thus, for example, $x^{(l)}$ becomes x, where the latter stands for the storage space employed to keep the sequence of approximations $x^{(0)}, x^{(1)}, x^{(2)}, \ldots$ computed during the iterative process. We consider the following aspects in the analysis:

- The parallel platform consists of K processes (or MPI ranks), denoted as $P_1, P_2, \ldots, P_K$.
- The coefficient matrix A is partitioned into K blocks of rows ($A_1, A_2, \ldots, A_k$), where each process stores one row-block with the k-th *distribution block* $A_k \in \mathbb{R}^{p_k \times n}$ is stored in P_k, and $n = \sum_{k=1}^{K} p_k$.

- Vectors are partitioned and distributed in the same way as with the block-row distribution of A. For example, the residual vector r is partitioned as $r_1, r_2, \ldots, r_K$ and r_k is stored in P_k.
- The scalars $\alpha, \beta, \rho, \tau$ are replicated on all K processes.

Taking into account these previous considerations, we analyze the different computational kernels (*S1–S8*) that compose the loop body of a single PCG iteration in Figure 1.

Sparse matrix-vector product (S1): This kernel needs as input operands: the coefficient matrix A, which is distributed by blocks of rows, and the vector d, which is partitioned and distributed conformably with A.

Consequently, prior to computing this kernel, we need to obtain a replicated copy of the distributed vector d in all processes, denoted as $d \rightarrow e$; vector e is the only array that is replicated in all processes. We can recognize here a communication stage, but, after that, each process can then compute its local piece of the output vector w concurrently: $P_k : w_k := A_k\, e$. This kernel thus requires assembling the distributed pieces of d into a single vector e that is replicated in all processes (in MPI, for example via an `MPI_Allgatherv`). The computation can then proceed in parallel, yielding the result w in the expected distributed state with no further communication involved. At the end, each MPI process owns a piece of w.

DOT *products (S2, S6, S8):* The next kernel in the loop body is the DOT product in the step *S2* between the distributed vectors d and w. Here, each process can compute concurrently a partial result $P_k : \rho_k :=< d_k, w_k >$ and when all processes have finished this partial computation, these intermediate values have to be reduced into a globally-replicated scalar $\rho := \beta/(\rho_1 + \rho_2 + \cdots + \rho_K)$. We can apply the same idea to the DOT products in the steps *S6* and *S8*, yielding a total of three process synchronizations (in MPI, via `MPI_Allreduce`) since ρ, β, τ are globally-replicated.

AXPY*(-type) vector updates (S3, S4, S7):* The next kernel is the AXPY kernel in the step *S3*, which involves the distributed vectors x, d and the globally-replicated scalar ρ. The operations in the steps *S4* and *S7* follow the same idea because the scalars β, β' are globally-replicated. In these types of kernels, all processes can perform their local parts of the computation to obtain the result without any communication: $P_k : x_k := x_k + \rho\, d_k$.

Application of the preconditioner (S5): The kernel in the step *S5* consists of applying the Jacobi preconditioner M, scaling the vector r by the diagonal of the matrix. Therefore, it can be executed in parallel by all processes because each of them stores a different set of the diagonal elements (those related with the piece of the matrix that it stores) and the corresponding set of the vector r elements: $P_k : z_k := M_k^{-1} r_k$.

We can easily reduce the number of synchronizations in the loop body of the PCG solver if we re-arrange the operations so that the step S8 is pushed up next to the step S6. Thus, a simultaneous execution of these two reductions decreases the number of process synchronizations from 3 to 2 per iteration. The reorganized algorithm and communications are summarized in Figure 2.

4. Strategies for Reproducibility

In this section, we present two strategies for ensuring reproducibility of the PCG computations. The first strategy relies on the ExBLAS approach, while the second is based on so-called floating-point expansions (FPEs). Then, we demonstrate how both strategies that are reinforced with the programmability effort such as the explicit use of the `fma` instruction and careful re-arrangement of computations. The idea in a nutshell is to guarantee reproducibility on each iteration. Then, reproducibility of the PCG solver is obvious as a sequence of reproducible iterations.

4.1. ExBLAS-based Strategy

Since Section 2.1 provides an overview of the ExBLAS approach, here we exploit the ExBLAS parallel reduction in conjunction with the `twoprod` EFT to derive the distributed memory DOT product for PCG that is presented in Section 4.3 together with a lightweight reproducible alternative.

While delivering both correctly rounded and reproducible results, ExBLAS has two major drawbacks [6]. The first drawback is related to the required memory storage, which amounts for $n_t \times p + acc_s$, where n_t is the thread count, p is the size of floating-point expansion, and acc_s is the size of superaccumulator (2,098 bits for summation). The second drawback is the number of required operations: For an input vector of size n with dynamic range d, the cost of accumulation is

$$n \times \left\lceil \frac{d}{52} \right\rceil \times \frac{C_{fpe}}{VL} + n_t \times p \times VL \times C_{sa}, \text{ when } \ d < 52p,$$
$$n \times \left(p \times \frac{C_{fpe}}{VL} + C_{sa} \right) + n_t \times p \times VL \times C_{sa}, \text{ otherwise,}$$

where $C_{fpe} = 6$ flops, see Algorithm 1, is the cost of the expansion update, VL is the architecture-dependent vector length on SIMD architectures (4 with AVX and 1 on GPUs), and $C_{sa} = 16$ flops +2 indirect memory accesses is the cost of the long accumulator update. The right-hand side term is the cost of flushing expansions to long accumulators at the end of the summation and gets negligible as n increases. These two drawbacks can be observed for compute-intensive kernels, leading to large performance overheads [9]. However, these drawbacks are either hardly visible or relatively small on bandwidth- and memory-bound operations like dot product (reduction) and potentially PCG due to the possibility to saturate bandwidth and hide the cost of extra computations and memory needs.

4.2. Lightweight Strategy for Reproducibility

The ExBLAS drawbacks served us as a motivation to design a lightweight strategy for reproducible computations with accuracy (correct rounding) guarantees. Examining the PCG method for moderately conditioned but largely sparse matrices like the studied Poisson matrix, we come to the conclusion that the method can successfully accommodate accurate and reproducible computations using at least three-floating point numbers, meaning the FPE of size 3 (FPE3). According to Hida et al [23], FPE3 is capable to represent at least 159 bits of significant. In order to be robust and more generic, we design FPEs of size 8 and enabled their variable size; the size can be adjusted by user. This approach is complemented with the early-exit technique [6]: we stop propagating zero-errors in the FPE. From our experience, the early-exit technique significantly improves algorithm's performance.

Our main motivation for iterative solvers, where next iteration corrects the previous estimate, is to provide a good enough associativity-assuring approach since the properties provided by ExBLAS get demolished by the next computation/ iteration[1]. However, we want all computations on a single iteration to be reproducible. Therefore, we aim to make the FPE-based implementation generic enough to cover wide range of problems with various condition numbers and/ or dynamic ranges. Thus, our main implementation is based on FPEs of size 8 with the early-exit technique in conjunction with two different FPEs for results and errors, which are merged at the end.

For the distributed memory DOT product, Algorithm 3 outlines the FPE-based solution: each MPI process invokes a local routine to conduct DOT products on their local copies of vectors of size N_k. This local DOT product routine is composed of the call to `twoprod` EFT (Algorithm 2) for the exact multiplication of two floating-point numbers; and, then, accumulation of the output result and the error to the process local FPEs with the help of Algorithm 4, which relies upon the `twosum` EFT (Algorithm 1). Then, the FPEs with the result and the error are combined into one by invoking Algorithm 4 in a loop over the FPE with errors: ExpansionAccumulate($fpe, fperr[i]$) as in Algorithm 5. Finally, to complete the distributed memory DOT product, we perform the global reduction on FPEs and round the FPE-result to the target precision using either `Add3` [24] or `NearSum` [12] algorithms; both steps are explained in details in Section 4.3.

Algorithm 3: Distributed DOT product of vectors a and b with FPEs.

```
Function dot($N, a, b, fpe, fperr$)
    local DOT product with subvectors of size $N_k$
    for $i = 0 \to N_k - 1$ do
        $res$ = twoprod($a[i], b[i], err$)
        ExpansionAccumulate($fpe, res$);
        ExpansionAccumulate($fperr, err$);
    end
    Merge FPEs with ExpansionAccumulate($a, x$)
    MPI reduction of FPEs
    Rounding to the target format
```

Algorithm 4: Adding a floating-point number x to a floating-point expansion a of size p.

```
Function ExpansionAccumulate($a, x$)
    Input: $x$ is a floating-point number.
    Output: $a$ is a FPE containing the result.
    for $i = 0 \to p - 1$ do
        $(a[i], x)$ := twosum($a[i], x$)
    end
```

4.3. Re-installing Reproducibility of PCG

Let us now inspect the sources of non-deterministic computations in the PCG solver as well as present our mitigation strategies for them. To remind, we deal with the MPI implementation of the solver where each process conducts computations on its own local slices of the matrix as well as the vectors (see Section 3.2 and Figure 2).

DOT *products (S2, S6, S8):* The main issue of non-determinism emerges from DOT products and, thus, parallel reductions such as `MPI_Allreduce()` that is used in computing the tolerance τ and both β and ρ. Hence, we 1) exploit the ExBLAS approach to build reproducible and correctly-rounded DOT product; 2) construct DOT product

[1] We are working on developing a concept of weak reproducibility – reproducibility under a certain accuracy guarantee, e.g. defined as the input tolerance, that is not necessarily correct rounding.

solely based on FPEs; 3) extend the ExBLAS- and FPE-based DOT products to distributed memory in order to make them suitable for the PCG algorithm in Figure 2. Listings 1 and 2 provide pseudo codes for our implementation of the distributed DOT product using the ExBLAS and lightweight strategies, respectively. After carrying out local DOT products using either ExBLAS- or FPE-based implementations, we perform global reduction that is split into three stages:

- `MPI_Reduce()` acting on either long accumulators or FPEs. For the ExBLAS approach, since the long accumulator is an array of long integers, we apply regular reduction. Note that we may need to carry an extra intermediate normalization after the reduction of 2^{K-1} long accumulators, where $K = 64 - 52 = 12$ is the number of carry-safe bits per each digit of long accumulator. For the FPE approach, we renormalize FPEs optionally using the Priest's renormalization method [23, 25] and define the MPI operation that is based on the `twosum` EFT, see Algorithm 5;
- Rounding to double: for long accumulators, we use the ExBLAS-native `Round()` routine. To guarantee correctly rounded results of the FPE-based computations, we employ the `Add3` algorithm from [24] for FPEs of size three and the `NearSum` algorithm from [12] for FPEs of size eight; both may require renormalization before. Listing 3 presents our implementation of the `Add3` algorithm;
- `MPI_Bcast()` to distribute the result of DOT product to the other processes as only master performs rounding.

Listing 1: Reproducible Allreduce with ExBLAS.

```
std::vector<int64_t> h_superacc(BIN_COUNT);
exblas::exdot (..., &h_superacc[0]);
exblas::Normalize (&h_superacc[0]);
MPI_Reduce (&h_superacc[0],..., BIN_COUNT,
    MPI_LONG, MPI_SUM);
if (myId == 0) {
  beta = exblas::Round (&h_superacc[0]);
}
MPI_Bcast (&beta, 1, MPI_DOUBLE, ...);
```

Listing 2: Reproducible Allreduce with FPEs only.

```
std::vector<double> fpe(N);
dot (..., &fpe[0]);
renormalize(&fpe[0]);     // optional
MPI_Op Op;    // user-defined reduction operation
MPI_Op_create (fpesum, 1, &Op);
MPI_Reduce (&fpe[0], ..., N, MPI_DOUBLE, Op);
if (myId == 0) {
  beta = Round (&fpe[0]);    // Add3
}
MPI_Bcast (&beta, 1, MPI_DOUBLE, ...);
```

Algorithm 5: Aggregation of two FPEs of size p.

```
Function fpesum($a, b$)
    Input: $b$ is a FPE.
    Output: $a$ is a FPE containing the result.
    for $i = 0 \to p - 1$ do
        ExpansionAccumulate(a,b[i])
    end
```

Listing 3: Correct rounding of a FPE to double with `Add3` [24].

```
inline static T Round( const T *fpe ) {
    union {
        T d;
        int64_t l;
    } thdb;
    T tl;
    T th = twosum(fpe[1], fpe[2], tl);
    if (tl != 0.0) {
        thdb.d = th;
        // if the mantissa of th is odd, we are done
        if (!(thdb.l & 1)) {
            // choose the rounding direction
            // depending of the signs of th and tl
            if ((tl > 0.0) ^ (th < 0.0))
                thdb.l++;
            else
                thdb.l--;
            th = thdb.d;
        }
    }
    // final addition rounded to nearest
    return fpe[0] + th;
}
```

Sparse matrix-vector product (S1): The other reproducibility issue hides in the computation of the sparse matrix-vector product. With the current distributed implementation of this operation, each MPI process computes its dedicated part w_k of the vector w by multiplying a block of rows A_k by the vector e. Since the computations are carried locally and sequentially, they are deterministic and, thus, reproducible. However, some parts of the code like $a * b + c * d * e$ and $a+ = b * c$ – present in the original implementation of PCG – may not always provide with the same result.

Let us consider this test case: $y = a * x + b * y$, which is presented in more details in [26]. The problem with this is that for performance reasons, the C++ language standard allows compilers to change the execution order of this type of operation. It even allows merging multiplications and summations with fused multiply-add (`fma`) instructions. This computes $a * x + b$ in a single instruction with only a single rounding operation at the end. Hence, a compiler might translate $a * b + c * d$ to two multiplications $t1 = a * x$ and $t2 = b * y$, and a subsequent summation $y = t1 + t2$; it might generate a single multiplication $t = b * y$ with a subsequent fma $y = \texttt{fma}(a, x, t)$, which gives a slightly different result; or it may even compute $t = a * x$ first and then use the fma $y = \texttt{fma}(b, y, t)$.

Our approach to solve this issue is to explicitly instruct compilers to use fma. Note that the underlying architecture should support fma; otherwise, this may lead to the runtime error. This is possible through the std::fma instruction added to the C++ 11 language standard. With this option, we avoid non-determinism in the order of operations, reduce the number of rounding errors from three to two, and, therefore, achieve reproducibility for this type of operations. Consequently, we accomplish reproducibility for the sparse matrix-vector multiplication.

AXPY(-type) vector updates (S3, S4, S7): For computing the AXPY(-type) vector updates, we rely upon the sequential MKL implementation of AXPY(-type). However, we consider replacing this call to MKL's AXPY(-type) by our implementation using fma as the algorithm is strictly memory-bound and is not performance crucial.

Application of the preconditioner (S5): The application of the Jacobi preconditioner is rather simple: first, the inverse of the diagonals are computed and then the application of the preconditioner only involves element-wise multiplication of two vectors. Thus, this part is both correctly rounded and reproducible.

Reproducibility and accuracy of both approaches: It is evident that the results provided by ExBLAS DOT are both correctly-rounded and reproducible. With the lightweight DOT, we search for minimal size of FPE such that we still preserve every bit of both the result and the error. For the studied 3D Poisson's equation, the sweet spot is the FPE of size 3 that ensures identical results to ExBLAS and the reference highly accurate solution. FPE of size 4 leaves the last digit of the FPE not utilized, e.g. for β the final FPE is $[4.93844e - 20, 1.02e - 32, -1.7499e - 49, 0]$. However, we aim also to be generic and, hence, we provide the implementation that relies on FPEs of size eight with the early-exit technique, which is our main FPE-based implementation. Additionally, we add a check for both FPE-based implementations for the case when the condition number and/ or the dynamic range are too large and we cannot keep every bit of information. Then, the warning is thrown, containing also a suggestion to switch to the ExBLAS-based implementation. But, note that these lightweight implementations are designed for moderately conditioned problems or with moderate dynamic range in order be accurate, reproducible, but also high performing since the ExBLAS version can be very resource demanding, specially on the small core count. To conclude, if the information about the problem is know in advance, it is worth pursuing the lightweigh approach.

5. Experimental Results

5.1. Setup

The experiments in this section employed IEEE754 double-precision arithmetic and were carried out in two different clusters:

- The *MareNostrum4* (MN4) supercomputer at *Barcelona Supercomputing Center (BSC)*: This platform consists of SD530 Compute Racks with an Intel Omni-Path high performance network interconnect. Each node comprises two 24-core Intel Xeon Platinum 8160 processors (2.10 GHz) and 96 Gbytes of DDR4 RAM. The platform runs the SuSE Linux Enterprise Server operating system. The codes in this platform were compiled using GCC v7.2.0, Intel MPI v2018.1, and MKL v2017.4.
- The *Tintorrum* cluster at UJI: This is a 8-node cluster, where each node is equipped with two 8-core Intel Xeon(R) E5-2630v3 processors (Haswell-EP) (for a total of 128 cores), running at 2.4 GHz, with 20 MBytes of L3 on-chip cache (LLC or last level of cache), and with 64 GBytes of DDR3 RAM. The operating system running in the cluster is Linux version 2.6.32-642.4.2.el6.centos.plus.x86_64. The codes were compiled with GCC v5.3.0, OpenMPI v1.10.2, and Intel MKL v2017.1.

For the experimental analysis, we leveraged a sparse (symmetric positive definite) coefficient matrix arising from the finite-difference method of a 3D Poisson's equation with 27 stencil points.

The right-hand side vector b in the iterative solvers was always initialized to the product $A(1, 1, \ldots, 1)^T$, and the PCG iteration was started with the initial guess $x_0 = 0$. The parameter that controls the convergence of the iterative process was set to 10^{-8}.

5.2. Performance Evaluation

In the following, we analyze the performance of two reproducible versions of the PCG algorithm parallelized with MPI: one that relies on the ExBLAS approach, and an alternative variant that is based on floating-point expansions (FPEs) of size eight with the early-exit technique. Hereafter, we will refer to them as *Exblas* and *Opt*, accordingly.

Our experiments evaluate the strong scaling of these reproducible implementations compared against the regular (non-reproducible) version of PCG using MPI.

To begin with, we compare our regular MPI implementation of PCG with the *High Performance Conjugate Gradients (HPCG) benchmark* [27, 28]. HPCG has been introduced as an alternative to the *LINPACK benchmark* [29] with the specific purpose of exercising the computational units and producing data access patterns that mimic those present in a variety of real HPC applications. In the experiments, we have used the version 3.1 of this library. Compared with the HPCG benchmark, our regular MPI implementation is composed of the same type of numerical kernels and, therefore, exhibits analogous data access patterns and the arithmetic-to-memory operations ratios. The only difference resides in the implementation of the communications. Concretely, in order to realize the SpMV kernel, we invoke the `MPI_Allgatherv` routine to merge the partial results for all the subvectors. In contrast, HPCG implements an ad-hoc pattern which only sends/ receives the required elements, reducing the communication time. The main drawback of the HPCG alternative is that this communication pattern is optimized for a specific problem, but it may not always provide the best solution for a more general case.

We analyze the performance of the three implementations in the aforementioned clusters. Specifically, in order to assess the strong scalability, we fix the matrix size to n=16,003,008 (252^3) and increase the number of cores. Table 2 reports the total execution time (averaged for 5 different executions) of the different MPI PCG solvers on both platforms, varying the number of cores (from 48 to 768 in MareNostrum4 and from 16 to 128 in Tintorrum) as we maintain the problem size. The behaviour of the strong scaling experiment could be expected for a parallel algorithm dealing with a sparse linear algebra operation. This experiment in particular reports an important increase of the overhead when the number of nodes becomes large as the communication cost then dominates the execution time. Unfortunately, we cannot evaluate a larger problem to increase the weight of the computational cost, as the problem dimension is restricted by the capacity of the memory.

Execution time in seconds of the implementations on MareNostrum4				
Number of cores	*Regular*	*Exblas*	*Opt (FPE8EE)*	*FPE3*
48	1.5758E+01	5.55393E+01	2.1573E+01	2.1496E+01
96	9.3028E+00	2.9915E+01	1.2607E+01	1.2451E+01
192	6.6396E+00	1.7293E+01	8.2237E+00	8.2163E+00
384	6.2954E+00	1.1382E+01	7.0970E+00	7.7251E+00
768	1.0517E+01	1.3573E+01	1.0951E+01	1.0967E+01

Execution time in seconds of the implementations on Tintorrum				
Number of cores	*Regular*	*Exblas*	*Opt (FPE8EE)*	*FPE3*
16	3.7159E+01	1.2599E+02	5.0196E+01	5.0148E+01
32	5.1876E+01	1.1436E+02	6.8336E+01	6.4005E+01
64	6.8282E+01	9.3668E+01	6.8315E+01	7.0045E+01
128	6.4809E+01	7.8815E+01	6.6984E+01	6.7101E+01

Table 2: Strong scalability of different MPI implementations of the Preconditioned Conjugate Gradient method on MareNostrum4 and Tintorrum, where one process is pinned to one core.

Figure 3 reports the total execution time (averaged for 5 different executions) of the reproducible MPI PCG solvers for the two clusters normalized with respect to the execution time of the regular MPI version, when we vary the number of cores. To note that plots present the strong scaling evaluation. In these graphs, we can observe that the difference of both versions with respect to the regular one is higher on a small number of cores, and it decreases when the number of cores increases. Moreover, we can see that the overhead of the *Opt* implementation compared with the regular version is smooth and small, while the overhead of the *Exblas* version is significant, especially on the MareNostrum4 cluster.

5.3. Accuracy and Reproducibility Evaluation

In addition to the performance results, we report also the results of the accuracy and reproducibility evaluation. For that, we develop a generator of ill-conditioned matrices. This generator scales the first row and the first column of the matrix so that dot product attains specified condition number and, hence, the matrix. Additionally, we derive a sequential version of the code that relies on the GNU Multiple Precision Floating-Point Reliably (MPFR) library [30]

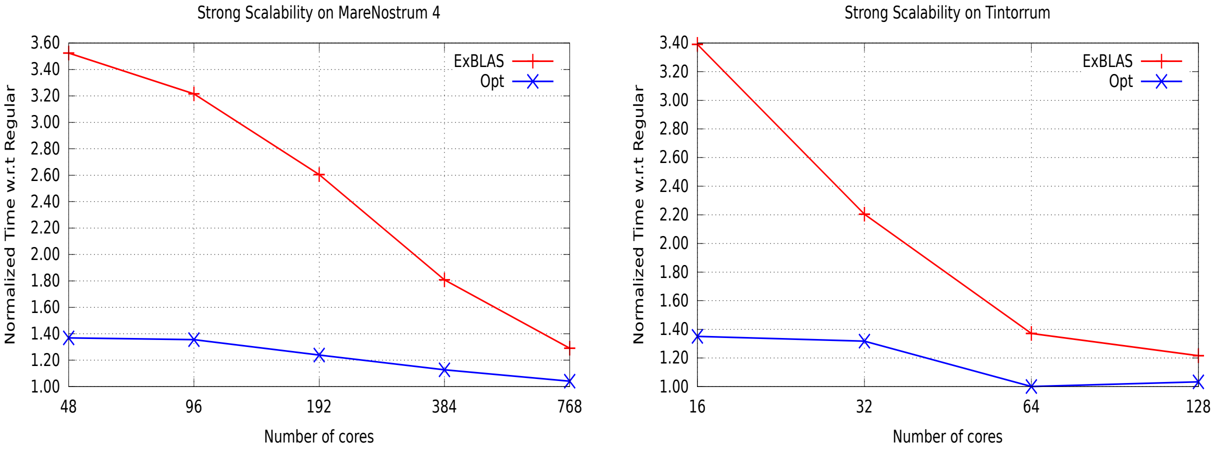

Figure 3: Analysis of the strong scalability of the different reproducible versions of the MPI PCG, when the time is normalized with respect to the regular non-deterministic MPI version.

Iteration	Residual			
	MPFR	*Original* 1 proc	*Original* 48 procs	*Exblas & Opt*
0	0x1.19f179eb7f032p+49	0x1.19f179eb7f03**3**p+49	0x1.19f179eb7f03**3**p+49	0x1.19f179eb7f032p+49
2	0x1.f86089ece9f75p+38	0x1.f86089**f08810d**p+38	0x1.f86089e**d07a76**p+38	0x1.f86089ece9f75p+38
9	0x1.fc59a29d329ffp+28	0x1.fc59a29d**1b6a**p+28	0x1.fc59a29d**2e989**p+28	0x1.fc59a29d329ffp+28
10	0x1.74f5ccc211471p+22	0x1.74f5cc**b8203ad**p+22	0x1.74f5ccc**1fafef**p+22	0x1.74f5ccc211471p+22
...	...	...	...	...
40	0x1.7031058eb2e3ep-19	0x1.703105**aea0e8a**p-19	0x1.7031058e**8ff5a**p-19	0x1.7031058eb2e3ep-19
42	0x1.4828f76bd68afp-23	0x1.4828f**6fabbf2a**p-23	0x1.4828f76b**b9038**p-23	0x1.4828f76bd68afp-23
45	0x1.8646260a70678p-26	0x1.8646260**1300d2**p-26	0x1.8646260a7**1301**p-26	0x1.8646260a70678p-26
47	0x1.13fa97e2419c7p-33	0x1.13fa9**8038c44e**p-33	0x1.13fa97e**54e903**p-33	0x1.13fa97e2419c7p-33

Table 3: Accuracy and reproducibility comparison on the intermediate and final residual against MPFR for a matrix with condition number of 10^{12}. The matrix is generated following the procedure from Section 5.1 with n=4,019,679 (159^3).

Iteration	Direct error			
	MPFR	*Original* 1 proc	*Original* 48 procs	*Exblas & Opt*
0	0x1p+0	0x1p+0	0x1p+0	0x1p+0
2	0x1.2644a05cb**37e8**p+14	0x1.2644a05cb**2f58**p+14	0x1.2644a05cb369ap+14	0x1.2644a05cb37e8p+14
9	0x1.d7346214bcb1ap+1	0x1.d7346214b**a579**p+1	0x1.d7346214b**c6dc**p+1	0x1.d7346214bcb1ap+1
10	0x1.3be1faca16608p+0	0x1.3be1faca1**934f**p+0	0x1.3be1faca1**6236**p+0	0x1.3be1faca16608p+0
...	...	...	...	...
40	0x1.e21b4p-31	0x1.**dead18**p-31	0x1.**dae6c**p-31	0x1.e21b4p-31
42	0x1.e21c38p-31	0x1.**deae1**p-31	0x1.**dae7b8**p-31	0x1.e21c38p-31
45	0x1.e21c28p-31	0x1.**deae**p-31	0x1.**dae7a8**p-31	0x1.e21c28p-31

Table 4: Direct error comparison against MPFR for a matrix with condition number of 10^{12}. The matrix is generated following the procedure from Section 5.1 with n=4,019,679 (159^3).

– a C library for multiple (arbitrary) precision floating-point computations on CPUs – as a highly accurate reference implementation. This implementation uses 2,048 bits of accuracy for computing dot product (192 bits for internal product of two floating-point numbers) and performs correct rounding of the computed result to double precision.

Table 3 reports the intermediate and final residual on each iteration of the PCG solver for the matrix with the number of rows/ columns equal to n=4,019,679 (159^3) and the condition number of 10^{12}. The results are exposed with all digits using hexadecimal representation. For this test, the tolerance was 10^{-8} and it took 47 iterations for all four implementations to converge under this accuracy requirement. We used one node on the MareNostrum4 cluster with 48 processes each pinned to one core. We present only few iterations, however the difference is present on all iterations. The *ExBLAS* and *Opt* (FPE of size eight) implementations deliver both accurate and reproducible results that are identical with the MPFR library. The original code shows the difference from one digit on the initial iteration and up to six digits on the last iteration on 48 cores/ processes. We also add the results of the original code on one core/ process to highlight the reproducibility issue. To show these results, we merge two columns of the ExBLAS and Opt results as they are identical. Furthermore, we have also computed the direct error (see Table 4) where the difference is even larger: the error is 0x1p+0 on the initial iteration for both MPFR and orig, however the error is completely different (except of the exponent) from the 31st iteration with 0x1.f42fe8p-31 for MPFR and 0x1.ecfb68p-31 for the original version. Note that the results on ExBLAS and Opt are identical to MPFR.

We assume that this discrepancy in accuracy and reproducibility becomes larger at scale (more nodes) due to the stronger impact of the topology and reduction trees.

6. Related Work

To enhance reproducibility, Intel proposed the "Conditional Numerical Reproducibility" (CNR) option in its Math Kernel Library (MKL). Although CNR guarantees reproducibility, it does not ensure correct rounding, meaning the accuracy is arguable. Additionally, the cost of obtaining reproducible results with CNR is high. For instance, for large arrays the MKL's summation with CNR was almost 2x slower than the regular MKL's summation on the Mesu cluster hosted at the Sorbonne University [6].

Demmel and Nguyen implemented a family of algorithms – that originate from the works by Rump, Ogita, and Oishi [31, 12] – for reproducible summation in floating-point arithmetic [32, 7]. These algorithms always return the same answer. They first compute an absolute bound of the sum and then round all numbers to a fraction of this bound. In consequence, the addition of the rounded quantities is exact, however the computed sum using their implementations with two or three bins is not correctly rounded. Their results yielded roughly 20 % overhead on 1024 processors (CPUs only) compared to the Intel MKL `dasum()`, but it shows 3.4 times slowdown on 32 processors (one node). Ahrens, Nguyen, and Demmel extended their concept to few other reproducible BLAS routines, distributed as the ReproBLAS library (`http://bebop.cs.berkeley.edu/reproblas/`), but only with parallel reproducible reduction. Furthermore, the ReproBLAS effort was extended to reproducible tall-skinny QR [33].

The other approach to ensure reproducibility is called ExBLAS, which is initially proposed by Collange, Defour, Graillat, and Iakymchuk in [6]. ExBLAS is based on combining long accumulators and floating-point expansions in conjuction with error-free transformations. This approach is presented in Section 2.1. Collange et al. showed [6] that their algorithms for reproducible and accurate summation have 8 % overhead on 512 cores (32 nodes) and less than 2 % overhead on 16 cores (one node). While ExSUM covers wide range of architectures as well as distributed-memory clusters, the other routines primarily target GPUs. Exploiting the modular and hierarchical structure of linear algebra algorithms, the ExBLAS approach was applied to construct reproducible LU factorizations with partial pivoting [8].

Recently, Mukunoki and Ogita presented their approach to implement reproducible BLAS, called OzBLAS [18], with tunable accuracy. This approach is different from both ReproBLAS and ExBLAS as it does not require to implement every BLAS routine from scratch but relies on high-performance (vendor) implementations. Hence, OzBLAS implements the Ozaki scheme [34] that follows the fork-join approach: the matrix and vector are split (each element is sliced) into sub-matrices and sub-vectors for secure products without overflows; then, the high-performance BLAS is called on each of these splits; finally, the results are merged back using, for instance, the NearSum algorithm. Currently, the OzBLAS library includes dot product, matrix-vector product (gemv), and matrix-matrix multiplication (gemm). These algorithmic variants and their implementations on GPUs and CPUs (only dot) reassure reproducibility of the BLAS kernels as well as make the accuracy tunable up-to correctly rounded results.

7. Conclusions and Future Work

In this work, we addressed the issue of reproducibility of iterative solvers using a representative instance of the Preconditioned Conjugate Gradient method. We at first analyzed the MPI implementation of the PCG method and identified two major sources of non-deterministic behavior, namely dot products and compiler optimization. The latter may change the order of operations or replace some of them in favor of the fused multiply-add (`fma`) operation. For reproducible and distributed dot product, we leveraged the ExBLAS-approach as well as proposed its lightweight strategy solely based on FPEs. Both strategies offer to split the `MPI_Allreduce` routine into the combination of `MPI_Reduce` and `MPI_Bcast`. To tackle compiler interference in computations, we reconstruct computations as well as explicitly invoke fma instructions. Both strategies deliver identical results to ensure reproducibility of PCG such as in the number of iterations, the intermediate and final residuals, the direct errors, as well as the output vector-solution. The lightweight reproducible version of PCG outperforms the ExBLAS-based one by 2.57x on a single node, showing the overhead of roughly 36 %, due to the lower demand for FLOPs and memory. When the communication starts to have stronger impact, both versions show very low overhead compared to the original non-deterministic implementation: 29 % for ExBLAS and 4 % for lightweight on 768 cores of MareNostrum4; 21 % for ExBLAS and 3.4 % for lightweight on 128 cores of Tintorrum. This is a solid argument to use reproducible PCG at scale. The code is available at https://github.com/riakymch/ReproCG.

With this study we want to promote reproducibility by design through the proper choice of the underlying libraries as well as the careful programmability effort. For instance, a brief guidance would be 1) for fundamental numerical computations use reproducible underlying libraries such as ExBLAS, ReproBLAS, or OzBLAS [18]; 2) analyze the algorithm and make it reproducible through eliminating any uncertainties that may violate associativity such as reductions and use/ non-use of fmas. Additionally, we try to argue the need for the bit-wise reproducible and correctly-rounded results for iterative solvers as they will anyway be enhanced on next iterations as we do not reach the desired tolerance and, thus, do not exploit at full the obtained bit-wise results.

Our future work aims to develop a hybrid implementation of the PCG code, combining MPI processes with dynamic task-based approaches like OpenMP or OmpSs. At one hand, the hybridization will reduce the communication burden being more focused on inner node computations and work balancing, especially on nodes with large core counts such as at MareNostrum4. At another hand, it will introduce a new challenge in the form of a double-level reduction: an initial reduction among tasks inside a process/ node, followed by one among processes. Moreover, we plan to conduct deeper analysis of the lightweight approach to support our experimental results. One idea is to bind the length of FPEs to the condition number of the input problem and/ or its dynamic range similarly to the work by Carson and Nigham [35] for the mixed-precision direct linear solver.

Acknowledgment

To begin with, we would like to thank the reviewers for their thorough reading of the article as well as their valuable comments and suggestions. This research was partially supported by the European Union's Horizon 2020 research, innovation programme under the Marie Skłodowska-Curie grant agreement via the Robust project No. 842528 as well as the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the H2020 EC RIA Programme; in particular, the author gratefully acknowledges the support of Vicenç Beltran and the computer resources and technical support provided by BSC. The researchers from Universitat Jaume I (UJI) and Universidad Politécnica de Valencia (UPV) were supported by MINECO project TIN2017-82972-R. Maria Barreda was also supported by the POSDOC-A/2017/11 project from the Universitat Jaume I.

References

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. V. der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM, 1994.

[2] Y. Saad, Iterative methods for sparse linear systems, 3rd ed., SIAM, Philadelphia, PA, USA, 2003.

[3] W. Gropp, T. Hoefler, R. Thakur, E. Lusk, Using advanced MPI: Modern features of the message-passing interface, MIT Press, 2014.

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh, Basic linear algebra subprograms for Fortran usage, ACM TOMS 5 (1979) 308–323.

[5] J. J. Dongarra, J. D. Croz, S. Hammarling, I. Duff, A set of level 3 basic linear algebra subprograms, ACM TOMS 16 (1990) 1–17.

[6] S. Collange, D. Defour, S. Graillat, R. Iakymchuk, Numerical reproducibility for the parallel reduction on multi- and many-core architectures, ParCo 49 (2015) 83–97.

[7] J. Demmel, H. D. Nguyen, Parallel Reproducible Summation, IEEE Transactions on Computers 64 (2015) 2060–2070.

[8] R. Iakymchuk, S. Graillat, D. Defour, E. S. Quintana-Orti, Hierarchical Approach for Deriving a Reproducible LU factorization, IJHPCA (2019) 1–13. To appear. HAL preprint: hal-01419813.

[9] R. Iakymchuk, D. Defour, S. Collange, S. Graillat, Reproducible and Accurate Matrix Multiplication, Springer LNCS 9553 (2016) 126–137.

[10] N. J. Higham, Accuracy and stability of numerical algorithms, 2nd ed., SIAM, Philadelphia, PA, 2002. doi:10.1137/1.9780898718027.

[11] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, S. Torres, Handbook of Floating-Point Arithmetic, Birkhäuser, 2010. doi:10.1007/978-0-8176-4705-6.

[12] S. M. Rump, T. Ogita, S. Oishi, Accurate floating-point summation part ii: Sign, k-fold faithful and rounding to nearest, SIAM J. Sci. Comput. 31 (2008) 1269–1302.

[13] S. M. Rump, Computer-assisted proofs and self-validating methods, in: B. Einarsson (Ed.), Handbook on Accuracy and Reliability in Scientific Computation, SIAM, 2005, pp. 195–240.

[14] D. H. Bailey, High-precision computation: applications and challenges, in: Proceedings of ARITH-21, IEEE, 2013, p. 1. Keynote talk.

[15] D. R. Lutz, C. N. Hinds, High-precision anchored accumulators for reproducible floating-point summation, in: Proceedings of ARITH-24, IEEE, London, UK, 2017, pp. 98–105.

[16] N. Burgess, C. Goodyer, D. R. Lutz, C. N. Hinds, High-precision anchored accumulators for reproducible floating-point summation, IEEE Transactions on Computers (2019). Submitted.

[17] IEEE Computer Society, IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008, 2008.

[18] D. Mukunoki, T. Ogita, K. Ozaki, Accurate and reproducible blas routines with ozaki scheme for many-core architectures, in: Proc. International Conference on Parallel Processing and Applied Mathematics (PPAM2019), 2019. To appear.

[19] D. E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, volume 2, Addison-Wesley, 1969.

[20] T. Ogita, S. M. Rump, S. Oishi, Accurate sum and dot product, SIAM J. Sci. Comput 26 (2005) 1955–1988.

[21] U. Kulisch, V. Snyder, The Exact Dot Product As Basic Tool for Long Interval Arithmetic, Computing 91 (2011) 307–313.

[22] R. Iakymchuk, S. Collange, D. Defour, S. Graillat, ExBLAS: Reproducible and accurate BLAS library, in: Proceedings of the NRE2015 workshop held as part of SC15. Austin, TX, USA, November 15-20, 2015, 2015.

[23] Y. Hida, X. S. Li, D. H. Bailey, Algorithms for quad-double precision floating point arithmetic, in: Proceedings of ARITH-15, 2001, pp. 155–162. doi:10.1109/ARITH.2001.930115.

[24] S. Boldo, G. Melquiond, Emulation of a FMA and Correctly Rounded Sums: Proved Algorithms Using Rounding to Odd, IEEE Transactions on Computers 57 (2008) 462–471. doi:10.1109/TC.2007.70819.

[25] D. M. Priest, Algorithms for arbitrary precision floating point arithmetic, in: 10th IEEE Symposium on Computer Arithmetic, IEEE, 1991, pp. 132–143.

[26] M. Wiesenberger, L. Einkemmer, M. Held, A. Gutierrez-Milla, X. Xavier Saez, R. Iakymchuk, Reproducibility, accuracy and performance of the feltor code and library on parallel computer architectures, Computer Physics Communications 238 (2019) 145–156.

[27] HPCG - high performance Conjugate Gradients, 2015. https://software.sandia.gov/hpcg.

[28] J. Dongarra, M. A. Heroux, Toward a new metric for ranking high performance computing systems, Sandia Report SAND2013-4744, Sandia National Lab., 2013.

[29] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary, HPL - a portable implementation of the high-performance Linpack benchmark for distributed-memory computers, 2008. http://www.netlib.org/benchmark/hpl.

[30] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding, ACM TOMS 33 (2007) 13. doi:10.1145/1236463.1236468.

[31] S. M. Rump, T. Ogita, S. Oishi, Fast high precision summation, Nonlinear Theory and Its Applications, IEICE 1 (2010) 2–24.

[32] J. Demmel, H. D. Nguyen, Fast reproducible floating-point summation, in: Proceedings of ARITH-21, 2013, pp. 163–172.

[33] H. D. Nguyen, J. Demmel, Reproducible tall-skinny QR, in: Proceedings of ARITH-22, 2015, pp. 152–159. doi:10.1109/ARITH.2015.28.

[34] K. Ozaki, T. Ogita, S. Oishi, S. M. Rump, Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications, Numerical Algorithms 59 (2012) 95–118.

[35] E. Carson, N. J. Higham, Accelerating the solution of linear systems by iterative refinement in three precisions, SIAM J. Sci. Comput. 40 (2018) A817–A847. doi:10.1137/17M1140819.