International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

Special Issue Paper

General framework for re-assuring numerical reliability in parallel Krylov solvers: A case of bi-conjugate gradient stabilized methods

The International Journal of High Performance Computing Applications
2024, Vol. 38(1) 17–33
© The Author(s) 2023
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/10943420231207642
journals.sagepub.com/home/hpc
Sage

Roman Iakymchuk[1,2], Stef Graillat[3] and José I. Aliaga[4]

Abstract
Parallel implementations of Krylov subspace methods often help to accelerate the procedure of finding an approximate solution of a linear system. However, such parallelization coupled with asynchronous and out-of-order execution often makes more visible the non-associativity impact in floating-point operations. These problems are even amplified when communication-hiding pipelined algorithms are used to improve the parallelization of Krylov subspace methods. Introducing reproducibility in the implementations avoids these problems by getting more robust and correct solutions. This paper proposes a general framework for deriving reproducible and accurate variants of Krylov subspace methods. The proposed algorithmic strategies are reinforced by programmability suggestions to assure deterministic and accurate executions. The framework is illustrated on the preconditioned BiCGStab method and its pipelined modification, which in fact is a distinctive method from the Krylov subspace family, for the solution of non-symmetric linear systems with message-passing. Finally, we verify the numerical behavior of the two reproducible variants of BiCGStab on a set of matrices from the SuiteSparse Matrix Collection and a 3D Poisson's equation.

Keywords
Numerical reliability, reproducibility, accuracy, ExBLAS, PBiCGStab, pipelined PBiCGStab, HPC

1. Introduction

Solving large and sparse linear systems of equations appears in many scientific applications spanning from circuit and device simulation, quantum physics, large-scale eigenvalue computations, and up to all sorts of applications that include the discretization of partial differential equations (PDEs) as described by Barrett and et al. (1994). In this case, Krylov subspace methods fulfill the roles of standard linear algebra solvers (Saad, 2003). The Conjugate Gradient (CG) method can be considered as a pioneer of such iterative solvers operating on symmetric and positive definite (SPD) systems. Other Krylov subspace methods have been proposed to find the solution of more general classes of non-symmetric and indefinite linear systems. These include the Generalized Minimal Residual method (GMRES) by Saad and Schultz (1986), the Bi-Conjugate Gradient (BiCG) method by Fletcher (1976), the Conjugate Gradient Squared (CGS) method by Sonneveld (1989), and the widely used BiCG stabilized (BiCGStab) method by Van der Vorst (1992) as a smoother converging version of the above two. Moreover, preconditioning is usually incorporated in real implementations of these methods in order to accelerate the convergence of the methods and improve their numerical features.

One would expect that the results of the sequential and parallel implementations of Krylov subspace methods to be identical, for instance, in the number of iterations, the intermediate and final residuals, as well as the sought-after solution vector. However, in practice, this is not often the case due to different reduction trees – the Message Passing Interface (MPI) libraries 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,

[1]Umeå University, Umeå, Sweden
[2]Uppsala University, Uppsala, Sweden
[3]Sorbonne Université, CNRS, LIP6, Paris, France
[4]Universitat Jaime I, Castellón de la Plana, Spain

Corresponding author:
Roman Iakymchuk, Umeå University, MIT-Huset, 901 87 Umeå, Sweden.
Email: riakymch@cs.umu.se

and, therefore, violates reproducibility. We aim to ensure identical and accurate outputs of computations, including the residuals/errors, as in our view this is a way to ensure *robustness* and *correctness* of iterative methods. In this case, the robustness and correctness have a threefold goal: *reproducibility*[1] of the results with the *accuracy guarantee* as well as *sustainable (energy-efficient)* algorithmic solutions.

The implementation of Krylov subspace methods on massively parallel systems reveals their scalability problems. Mainly, because the synchronization of global communications, especially the reductions, delays parallel executions. The most common solution has been the developments of communication-avoiding and communication-hiding methods and, also, the use of new MPI functions to hide the communications, overlapping their execution with the computation of iterative methods. In Cools and Vanroose (2017), the authors propose a general framework for deriving *pipelined Krylov subspace methods*, in which the recurrences are reformulated to make the parallelization easier. Again, these changes impact on the robustness and correctness of the iterative methods.

In general, Krylov subsbpace methods are built from three components: sparse-matrix vector multiplication Ax (SPMV), DOT product between two vectors (x, y), and scaling a vector by a scalar with the following addition of two vectors $y := \alpha x + y$ (AXPY). If a block data distribution is used, only AXPY is performed locally, while SPMV needs to get some elements from the other processes, using point-to-point or the `MPI_Alltoallw()` collective MPI operations, before completing the computation, and DOT products requires communication and computation, for example, via the `MPI_Allreduce()` collective, among MPI processes. Although SpMV has the highest amount of floating-point operations (flops), at large scale DOT products become the most time-consuming component of Krylov subspace methods due to the required global communication. This justifies the use of pipelined versions of Krylov subspace methods.

In this paper, we aim to re-ensure reproducibility of Krylov subspace methods in parallel environments. Our contributions are the following:

- We propose a *general framework for deriving reproducible Krylov subspace methods*. We follow the bottom-up approach and ensure reproducibility of Krylov subspace methods via reproducibility of their components, including the global communication. We build our reproducible solutions on the ExBLAS (Collange and et al., 2015) approach and its lighter version.
- Even when applying our reproducible solutions, we particularly stress the importance of arranging computations carefully to be executed deterministically, for example, avoid possibly replacements by compilers of $a*b + c$ in the favor of fused multiply-add (`fma`) operation or postponing divisions in case of data initialization (i.e. divide before use). For instance, we provide customized AXPY(-like) operations using `fma`, which reduces round-offs to one or two per AXPY(-like) operation. Furthermore, we refer to the 30-year-old but still up-to-date guide 'What every computer scientist should know about floating-point arithmetic' by Goldberg (1991).
- We optimize the SPMV implementation by reducing the number of elements received in each process, changing the use of `MPI_Allgatherv()` to the combination of `MPI_Alltoallw()`. In the reproducible versions of dot product, we rely upon only one collective operation, namely `MPI_Allreduce()`, instead of `MPI_Reduce()` plus `MPI_Bcast()` using ExBLAS data.
- We verify the applicability and performance of the proposed methodology on the preconditioned BiCGStab (PBiCGStab) and the pipelined PBiCGStab method. We derive two reproducible variants of each method and test them on a set of large SuiteSparse matrices and a 3D Poisson's equation.

This journal article extends our previous conference paper (Iakymchuk et al., 2022). In particular, we include the pipelined preconditioned BiCGStab as another test case, optimize SpMV and reduce the number of global collectives in reproducible versions, as well as validate the implementations on larger matrices. Other information is also added to make the paper self-contained.

This paper is structured as follows. Section 2 reviews several aspects of computer arithmetic as well as the ExBLAS approach. Section 3 proposes a general framework for constructing reproducible Krylov subspace methods. Section 4 introduces the PBiCGStab and the pipelined PBiCGStab methods, describing their MPI implementation in detail. Later, we evaluate the two reproducible implementations of PBiCGStab and pipelined PBiCGStab 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 are composed of a significand, an exponent, and a sign:

$$x = \pm\underbrace{x_0.x_1\ldots x_{M-1}}_{mantissa} \times b^e, 0 \le x_i \le b-1, x_0 \ne 0,$$

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 (IEEE Computer Society (2008)), created in 1985 and then revised in 2008 and in 2019, has led to a considerable enhancement in the reliability of numerical computations by rigorously specifying

Table 1. Parameters for three IEEE arithmetic precisions.

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

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}.

The IEEE 754 standard requires correctly rounded results for the basic arithmetic operations ($+, -, \times, /, \sqrt{}$, fma). It means that they 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; but correct rounding alone is not necessary to achieve reproducibility. 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 754 standard in 2019. These mechanisms, once implemented in hardware, will simplify our reproducible algorithms – like the ones used in the ExBLAS by Iakymchuk et al. (2015), ReproBLAS by Demmel and Nguyen (2015), OzBLAS by Mukunoki et al. (2019) 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 error during the course of computations. The first approach uses EFT to compute both the result and the rounding error, storing them in a floating-point expansion (FPE). This 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` (Knuth, 1969) and for multiplication that is `twoprod` (Ogita et al., 2005). The code of these two operations are, respectively, shown in Algorithm 1 and 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 and Snyder (2011) 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.

```
FnFunction Input: $a, b$ are two floating-point numbers.
Output: $r, s$ are the result and the error, resp.
$[r, s]$ = 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.

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

The ExBLAS project (Iakymchuk et al., 2015) is an attempt to derive a 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 Collange et al., 2015). 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 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 paper is the DOT product of two vectors, which is a crucial fundamental BLAS operation. The EXDOT algorithm is based on the reproducible parallel reduction and the `twoprod` EFT: the algorithm accumulates the result and the error of `twoprod` EFT to same FPEs and then follows the reduction scheme. 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. These and the other routines – such as matrix-vector product, triangular solve and matrix–matrix multiplication – are distributed in the ExBLAS library[2].

3. General framework for reproducible Krylov solvers

This section provides the outline of a general framework for deriving a reproducible version of any traditional Krylov subspace method. The framework is based on two main concepts: 1) identifying the issues caused by parallelization and, hence, the non-associativity of floating-point computations and 2) carefully mitigating these issues primarily with the help of computer arithmetic techniques as well as programming guidelines. The framework was implicitly used for the derivation of the reproducible variants of the Preconditioned Conjugate Gradient (PCG) method in Iakymchuk et al. (2020a, 2020b).

The framework considers the parallel platform to consist of K processes (or MPI ranks), denoted as $P_1, P_2, \ldots, P_K$. In this framework, the coefficient matrix A is partitioned into K blocks of rows $(A_1, A_2, \ldots, A_k)$, where each P_k stores one row-block with the k-th *distribution block* $A_k \in \mathbb{R}^{p_k \times n}$, and $n = \sum_{k=1}^{K} p_k$. Additionally, vectors are partitioned and distributed in the same way as 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. Besides, scalars are replicated on all K processes.

3.1. Identifying sources of non-reproducibility

The first step is to identify sources of non-associativity and, thus, non-reproducibility of the Krylov subspace methods in parallel environments. As it can be verified in Figure 1, there are four common operations as well as message-passing communication patterns associated with them: sparse matrix-vector product (SPMV) which requires some communications, via Alltoallw collective, so that each process has the needed elements to compute the computation, DOT product with the Allreduce collective, scaling a vector with the following addition of two vectors (AXPY and AXPY-like), and the application of the preconditioner. Hence, we investigate each of them.

In general, associativity and reproducibility are not guaranteed when there is perturbation of floating-point operations in parallel execution. For instance, invoking the `MPI_Allreduce()` collective operation cannot ensure the same result (its execution path) as it depends on the data, the network topology, and the underlying algorithmic implementation. Under these assumptions, AXPY(-like) and SPMV are associativity-safe as they are performed locally on local slices of data. The application of preconditioner can also be considered safe, for example, the Jacobi preconditioner, until all operations are reduction-free; more complex preconditioners will certainly raise an issue. Thus, the main issue of non-determinism emerges from parallel reductions (steps $S2$, $S6$, and $S7$ in Figure 1).

3.2. Re-assuring reproducibility

We construct our approach for reassuring reproducibility by primarily targeting DOT products and parallel reductions. Note that the non-deterministic implementation of the Krylov subspace method utilizes the DOT routine from a BLAS library like Intel MKL followed by `MPI_Allreduce()`. Thus, we propose to refine this procedure into three steps:

- exploit the ExBLAS and its lighter FPE-based versions to build reproducible and correctly rounded DOT products;
- extend the ExBLAS- and FPE-based DOT products to distributed memory by employing `MPI_Allreduce()`. This collective acts on either long accumulators or FPEs. For the ExBLAS approach, we apply regular reduction, since the long accumulator is an array of long integers. Note that we may need to carry an extra intermediate normalization after the reduction of $2{*}2^{K-1}$ long accumulators, where $K = 64 - 52 = 12$ is the number of carry-safe bits per each digit of the long accumulator. For the FPE approach, we define the MPI operation that is based on the `twosum` EFT. Thus, at this point, the choice of the reduction algorithm underneath `MPI_Allreduce()` does not have an impact on the computations as every bit of information is stored;
- 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 `NearSum` algorithm from Rump et al. (2008). It is worth mentioning that the rounding operation is performed locally and does not require any communication. In the previous versions of the code as in Iakymchuk et al. (2022), we split the reduction into three steps: `MPI_Reduce()`, rounding, and `MPI_Bcast()`. However, this is negligible as we re-assure control of the reduction

while ($\tau > \tau_{\max}$)

Step	Operation	Kernel	Communication
$S1:$	$d := Ap$	SpMV	Alltoallw
$S2:$	$\rho := \beta / \langle p, d \rangle$	DOT product	Allreduce
$S3:$	$x := x + \rho p$	AXPY	–
$S4:$	$r := r - \rho d$	AXPY	–
$S5:$	$y := M^{-1} r$	Apply preconditioner	depends
$S6:$	$\tau := \|\|r\|\|_2$	DOT product + sqrt	Allreduce
$S7:$	$\alpha := \beta, \beta := \langle y, r \rangle$	DOT product	Allreduce
$S8:$	$\alpha := \beta / \alpha$	scalar operation	–
$S9:$	$p := y + \alpha p$	AXPY-like	–

end while

Figure 1. Preconditioned conjugate gradient method with annotated BLAS kernels and message-passing communication.

operation and, hence, eliminate the performance penalty of using two collectives with one extra synchronization.

It is evident that the results provided by ExBLAS DOT are both correctly rounded and reproducible. With the lightweight DOT, we aim also to be generic and, hence, we provide the implementation that relies on FPEs of size eight with the early-exit technique. This way the working precision of the computations using FPEs is increased up to 8*52 bits as mentioned in Hida et al. (2001) for the double-double arithmetic. Additionally, we add a check for the FPE-based implementations to cover a 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, especially on the small core count. To sum up, if the information about the problem is known in advance, it is worth pursuing the lightweight approach.

3.3. Programmability effort

It is important to note that compiler optimization and especially the usage of the fused-multiply-and-add (`fma`) instruction, which performs $a*b + c$ with the extended precision and the single rounding at the end, may lead to some non-deterministic results. For instance, in the SPMV computation, each MPI rank computes its dedicated part d_k of the vector d by multiplying a block of rows A_k by the vector p. 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 PBiCGStab – may not always yield to the same result (Wiesenberger et al., 2019). This is due to the fact that, for performance reasons, the C++ language standard allows compilers to change the execution order of this type of operation. It also allows merging multiplications and summations with fused multiply-add (`fma`) instructions. Hence, a compiler might translate $a*b + c*d$ to two multiplications $t1 = a*b$ and $t2 = c*d$, and a subsequent summation $t1 + t2$; it might generate a single multiplication $t = c*d$ with a subsequent `fma`(a, b, t), which gives a slightly different result; or it may even compute $t = a*b$ first and then use the `fma`(c, d, t). Thus, we advise to instruct compilers to use `fma` explicitly via `std::fma` in C++ 11, assuming the underlying architecture supports `fma`.

Another important observation is to carefully perform divisions and initialization of data. For instance, the choice of b in the Krylov solvers is the value $b = Ad$, with $d = 1/\sqrt{N}(1, \ldots, 1)^T$. In this case, we suggest to compute $b = Ad$ for $d = (1,\ldots,1)^T$ first and then scale b by $1/\sqrt{N}$, as we observed a slightly faster convergence (up to 7%) for the Krylov solver.

4. Reproducible BiCGStab

The classic Biconjugate Gradient Stabilized method (BiCGStab) by Van der Vorst (1992) was proposed as a fast and smoothly converging variant of the BiCG (Fletcher, 1976) and CGS (Sonneveld, 1989) methods. We present here the preconditioned BiCGStab (PBiCGStab) and the pipelined preconditioned BiCGStab (p-PBiCGStab), their design and implementation with Message Passing Interface (MPI).

For both methods, we consider a linear system $Ax = b$, where the coefficient matrix $A \in \mathbb{R}^{n \times n}$ is sparse 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.

```
Compute preconditioner for $A \to M$
Set starting guess $x^0$
Initiate $r^0 := b - Ax^0, p^0 := r^0, \tau^0 := \| r^0 \|_2, j := 0$

while $(\tau^j > \tau_{\max})$
```

Step	Operation	Kernel	Comm
$S1:$	$\hat{p}^j := M^{-1}p^j$	Apply precond.	–
$S2:$	$s^j := A\hat{p}^j$	SpMV	Alltoallw
$S3:$	$\alpha^j := \langle r^0, r^j \rangle / \langle r^0, s^j \rangle$	DOT product	Allreduce
$S4:$	$q^j := r^j - \alpha^j s^j$	AXPY-like	–
$S5:$	$\hat{q}^j := M^{-1}q^j$	Apply precond.	–
$S6:$	$y^j := A\hat{q}^j$	SpMV	Alltoallw
$S7:$	$\omega^j := \langle q^j, y^j \rangle / \langle y^j, y^j \rangle$	Two DOT products	Allreduce
$S8:$	$x^{j+1} := x^j + \alpha^j \hat{p}^j + \omega^j \hat{q}^j$	Two AXPY	–
$S9:$	$r^{j+1} := q^j - \omega^j y^j$	AXPY-like	–
$S10:$	$\beta^j := \frac{\langle r^0, r^{j+1} \rangle}{\langle r^0, r^j \rangle} * \frac{\alpha^j}{\omega^j}$	DOT product	Allreduce
$S11:$	$\tau^{j+1} := \| r^{j+1} \|_2$	DOT product + sqrt	Allreduce
$S12:$	$p^{j+1} := r^{j+1} + \beta^j (p^j - \omega^j s^j)$	Two AXPY-like	–

```
end while
```

Figure 2. Formulation of the PBiCGStab solver annotated with computational kernels and communication. The threshold τ_{max} is an upper bound on the relative residual for the computed approximation to the solution. In the notation, $\langle \cdot, \cdot \rangle$ computes the DOT (inner) product of its vector arguments.

Additionally, and for simplicity, we integrate the Jacobi preconditioner (Saad (2003)) in our implementations, which is composed of the diagonal elements of the matrix ($M = \text{diag}(A)$). In consequence, the application of the preconditioner is conducted on a vector and requires an element-wise multiplication of two vectors.

4.1. Message-passing parallel PBiCGStab implementation

The algorithmic description of the classical iterative PBiCGStab is presented in Figure 2. The loop body consists of two SPMV ($S2$ and $S6$), two preconditioner applications ($S1$ and $S5$), five DOT products ($S3$, $S7$, $S10$, and $S11$), six AXPY(-like) operations ($S4$, $S8$, $S9$, and $S12$), and a few scalar operations (Barrett and et al. (1994)).

As described in Section 3, the framework includes a reproducible implementation of the most common operations in a parallel implementation of a Krylov subspace method. Therefore, we next perform a communication and computation analysis of a message-passing implementation of the PBiCGStab solver. From there, we derive the reproducible version by following the guide from Section 3.

For clarity, hereafter we will drop the superindices that denote the iteration count in the variable names. Thus, for example, x^j 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. Taking into account these previous considerations, we analyze the different computational kernels ($S1$–$S12$) that compose the loop body of a single PBiCGStab iteration in Figure 2.

4.1.1. Sparse matrix-vector product (S2, S6). This kernel needs as input operands: the coefficient matrix A, which is distributed by blocks of rows, and the corresponding vector ($\widehat{p}$ or $\widehat{q}$), which is partitioned and distributed using the same partitioning as A. For simplicity, we just explain below how $S2$ is computed.

In theory, prior to computing this kernel, we would need to obtain a replicated copy of the distributed vector $\widehat{p}$ in all processes using `MPI_Allgatherv()`, denoted as $\widehat{p} \to e$, so that vector e would be the only array that is replicated in all processes. But not all elements of e are required in all processes to compute the local SPMV, only those column indexes which are in the A_k and are not in $\widehat{p}_k$, denoted as e_k. Therefore, the communication pattern is defined by the matrix pattern and the matrix distribution, whereas the gathering of e_k can be done using `MPI_Alltoallw()`. As the matrix pattern and distribution are not changed within the loop, the communication structures can be defined before the loop, simplifying the communication step.

The computation can then proceed in parallel, yielding the vector result s in the expected distributed state with no further communication involved. At the end, each MPI process owns the corresponding piece of the computed vector. To ensure the reproducibility of this computation, the local DOT products between the sparse rows of A_k and e_k are based on `fma` as outlined in 3.3.

4.1.2. DOT *products (S3, S7, S10, S11).* The next kernel in the loop body is the DOT product in the step $S3$ between the distributed vectors r^0 and s. Here, each process can compute concurrently a partial result $P_k : \rho_k = \langle r^0_k, s_k \rangle$ and when all processes have finished this partial computation, these intermediate values have to be reduced into a globally-replicated scalar $\alpha := \sigma/(\rho_1 + \rho_2 + \cdots + \rho_K)$. We can apply the same idea to the DOT products in the steps $S7$, $S10$, and $S11$, yielding a total of five process synchronizations (in MPI, via `MPI_Allreduce()`) since all scalars are globally-replicated. But, the number of synchronization can be reduced to four, considering that communications in S10 and S11 can be merged in a single `MPI_Allreduce()`.

The easiest solution to compute ρ_k is to call to the DOT routine from the Intel MKL or similar libraries, however this will not guarantee reproducibility even when `fma` are used internally. Thus, we enforce reproducibility by applying our two ExBLAS-based strategies, following the guideline as in Section 3.2.

AXPY(-like) vector updates ($S4$, $S8$, $S9$, $S12$): The next kernel is the AXPY-like kernel in the step $S4$, which involves the distributed vectors q, r, s and the globally-replicated scalar α. The operations in the steps $S8$, $S9$, and $S12$ follow the same idea because all scalars are globally-replicated. In this type of kernels, all processes can perform their local parts of the computation to obtain the result without any communication: P_k: $q_k = r_k - \alpha\ s_k$.

While AXPY($y = \alpha x + y$) can directly rely on the MKL library routine, AXPY-like ($z = \alpha y + x$) requires, at least, two routines in order to be implemented (SCAL/COPY + AXPY). Looking for a robust and correct solution, the use of MKL routines is a bad alternative since each one introduces a rounding error. Additionally, this alternative is more expensive because some vector must be traversed more than once. Instead, we propose to rely on `fma` that computes each element of the solution of both axpy and axpy-like with a single rounding and only one pass through the vectors. Therefore, in the reproducible versions, we provide our own implementations for SPMV, AXPY, and AXPY-like (do not rely on any external BLAS libraries) and, hence, have the overall control of computations, assuring their correct rounding and reproducibility.

4.1.3. Application of the preconditioner (S1, S5). The kernel in the step S1 consists of applying the Jacobi preconditioner M, scaling the vector p 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 elements: $P_k : \widehat{p}_k = M_k^{-1} p_k$. The same procedure can be applied on the step $S5$ to scale the vector q, resulting in $\widehat{q}$.

There is no routine in the MKL library to implement the element-wise product of two vector, therefore, an ad hoc implementation has to be done. Reproducibility is ensured if this code is based on `fma` and the order of operations is deterministic as mentioned in Section 3.3.

4.2. Message-passing parallel p-PBiCGStab implementation

Cools and Vanroose (2017) propose two main steps for deriving the pipelined version of a Krylov subspace method:

- *Communication-avoiding:* In which the number of global communications is reduced, rearranging the original recurrences. Usually more terms appear in the new recurrences and, therefore, there are more vector operations.
- *Hiding communications:* Since global communications are the most time-consuming component of Krylov subspace methods at large scale, the alternative to reduce their impact on the performance of parallel implementations is their simultaneous execution (overlapping) with SPMV. This technique is implemented using non-blocking collectives, such as `MPI_Iallreduce()`, which require the use of `MPI_Wait()` to check when the communication is complete.

The algorithmic description of the pipelined preconditioned BiCGStab (p-PBiCGStab) is presented in Figure 3. The loop body consists of two SPMV ($S10$ and $S18$), two preconditioner applications ($S9$ and $S17$), six DOT products ($S8 \cup S11$ and $S16 \cup S19$), 18 AXPY/AXPY-like operations ($S1$-$S7$ and $S12$-$S15$), and a few scalar operations (Cools and Vanroose (2017)). It is worth mentioning that the pipelined PBiCGStab may show different convergence behavior compared to the standard PBiCGStab due to the different way floating-point operations are performed and, thus, the round off errors are propagated and accumulated differently.

The analysis of the computational kernels of the algorithm is very similar to the described above for the parallelization of PBiCGStab in Section 4.1. The only difference is how the DOT products are implemented.

4.2.1. DOT *products ($S8 \cup S11$, $S16 \cup S19$).* Although, there are six DOT in Figure 3, only two global synchronizations are required because more than one reduction is complete in each step. Therefore, before the synchronization is initiated, the partial result related to the corresponding reductions has to be computed locally in each process. Then, obtained values are stored in local vectors which are used to compute the global values using collectives. The overlapping requires the use of non-blocking collectives which decompose each reduction in two steps: the first step ($S8$ and $S16$) properly executes `MPI_Iallreduce()` starting the

Compute preconditioner for $A \to M$
Set starting guess x^0
Initiate $r^0 := b - Ax^0, \hat{r}^0 := M^{-1}r^0, w^0 := A\hat{r}^0, \hat{w}^0 := M^{-1}w^0, t^0 := A\hat{w}^0, \alpha^0 := \langle r^0, r^0\rangle / \langle r^0, w^0\rangle, \beta^{-1} := 0, j := 0$

while $(\tau^j > \tau_{\max})$

Step	Operation	Kernel	Comm
$S1:$	$\hat{p}^j := \hat{r}^j + \beta^{j-1}(\hat{p}^{j-1} - \omega^{j-1}\hat{s}^{j-1})$	Two AXPY-like	–
$S2:$	$s^j := w^j + \beta^{j-1}(s^{j-1} - \omega^{j-1}z^{j-1})$	Two AXPY-like	–
$S3:$	$\hat{s}^j := \hat{w}^j + \beta^{j-1}(\hat{s}^{j-1} - \omega^{j-1}\hat{z}^{j-1})$	Two AXPY-like	–
$S4:$	$z^j := t^j + \beta^{j-1}(z^{j-1} - \omega^{j-1}v^{j-1})$	Two AXPY-like	–
$S5:$	$q^j := r^j - \alpha^j s^j$	AXPY-like	–
$S6:$	$\hat{q}^j := \hat{r}^j - \alpha^j \hat{s}^j$	AXPY-like	–
$S7:$	$y^j := w^j - \alpha^j z^j$	AXPY-like	–
$S8:$	$\langle q^j, y^j\rangle, \langle y^j, y^j\rangle$	Two DOT products	Iallreduce
$S9:$	$\hat{z}^j := M^{-1}z^j$	Apply precond.	–
$S10:$	$v^j := A\hat{z}^j$	SPMV	Alltoallw
$S11:$	$\omega^j := \langle q^j, y^j\rangle / \langle y^j, y^j\rangle$	Two DOT products	Wait for S8
$S12:$	$x^{j+1} := x^j + \alpha^j\hat{p}^j + \omega^j\hat{q}^j$	Two AXPY	–
$S13:$	$r^{j+1} := q^j - \omega^j y^j$	AXPY-like	–
$S14:$	$\hat{r}^{j+1} := \hat{q}^j - \omega^j(\hat{w}^j - \alpha^j\hat{z}^j)$	Two AXPY-like	–
$S15:$	$w^{j+1} := y^j - \omega^j(t^j - \alpha^j v^j)$	Two AXPY-like	–
$S16:$	$\langle r^0, r^{j+1}\rangle, \langle r^0, w^{j+1}\rangle$ $\langle r^0, s^j\rangle, \langle r^0, z^j\rangle$	Four DOT products	Iallreduce
$S17:$	$\hat{w}^{j+1} := M^{-1}w^{j+1}$	Apply precond.	–
$S18:$	$t^{j+1} := A\hat{w}^{j+1}$	SPMV	Alltoallw
$S19:$	$\beta^j := \frac{\langle r^0, r^{j+1}\rangle}{\langle r^0, r^j\rangle} * \frac{\alpha^j}{\omega^j}$ $\alpha^{j+1} := \frac{\langle r^0, r^{j+1}\rangle}{\langle r^0, w^{j+1}\rangle + \beta^j\langle r^0, s^j\rangle - \beta^j\omega^j\langle r^0, z^j\rangle}$	Four DOT products	Wait for S16

end while

Figure 3. Formulation of the **pipelined** PBiCGStab solver annotated with computational kernels and communication. The threshold $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution. In the notation, $\langle \cdot, \cdot \rangle$ computes the Dot (inner) product of its vector arguments.

global communication, which continues while other steps are performed, for example, $S9$ and $S10$. When the global values have to be used, the second step has to be done, calling `MPI_Wait()`, since execution can only continue if the global communication is completed. We follow here the 'golden rule' of the non-blocking communication – start as soon as the data are available and wait right before they are needed.

5. Experimental results

In this section, we report a variety of numerical experiments to examine the convergence, scalability, accuracy, and reproducibility of the original and two reproducible versions of PBiCGStab and p-PBiCGStab. In our experiments, we employed IEEE754 double-precision arithmetic and conducted them on the dual Intel Xeon Gold 6240R CPU @2.4 GHz nodes with 48 cores and 384 GB of memory each at Fraunhofer ITWM. Nodes are connected with the HDR Infiniband.

5.1. Evaluation on the suitesparse matrices

We carried out tests on a range of different linear systems from the SuiteSparse matrix collection on a single node using 2, 8, 16, 32, and 48 (full) cores. Table 2 lists a set of tested matrices with the number of rows/columns N and the number of nonzeros nnz. We aim to show the reproducibility, accuracy, and performance of our algorithmic implementations on matrices with various loads, that is, number of nonzeros, as well as complexities. The right-hand side vector b in the iterative solvers was always initialized to the product $Ad, d = 1/\sqrt{N}(1, \ldots, 1)^T$, where N is the number of rows/columns of A. However, in both ExBLAS- and FPE-based versions, marked as ReproPBiCGStab in the table, we computed $b = Ad$, $d = (1,\ldots,1)^T$ and then scaled b by $1/\sqrt{N}$. In all implementations, iterations were started with the initial guess $x_0 = 0$. The parameter that controls the convergence of the iterative process is $\|r^j\|_2 / \|r^0\|_2 \leq 10^{-6}$. We want to specify that $\|r^j\|_2 = \sqrt{|(r^j, r^j)|}$ since some works use $\|r^j\|_2 = \sqrt{|(r^0, r^j)|}$.

Table 2. Convergence of the **PBiCGStab and its reproducible versions** (ReproPBiCGStab, identical results reported for both) on a set of the SuiteSparse matrices. The initial guess is $x^0 = 0$. The number of iterations required to reach the tolerance of 10^{-6} on the scaled residual, i.e. $\|r^i\|_2/\|r^0\|_2$, is reported along with the corresponding true residual $\|b - Ax^i\|_2$. iterX stands for runs on X MPI processes. The last two columns show the overhead of the reproducible versions with 48 cores/MPI processes, for example, 1.09x for the add32 matrix.

					PBiCGStab			ReproPBiCGStab			
Matrix	Prec	N	nnz	$\|r^0\|_2$	iter1	iter32	$\|b - Ax^i\|_2$	iter	$\|b - Ax^i\|_2$	FPE	ExBLAS
af_shell10	Jac	1,508,065	52,259,885	1.48e+05	9	9	2.18e-02	9	2.18e-02	2.71	3.30
atmosmodd	Jac	1,270,432	8,814,880	3.75e+03	221	230	2.68e-03	222	9.55e-04	3.48	4.06
atmosmodj	Jac	1,270,432	8,814,880	3.75e+03	220	227	3.46e-03	229	3.25e-03	3.64	4.34
atmosmodl	Jac	1,489,752	10,319,760	1.85e+04	133	130	1.80e-02	132	1.68e-02	2.69	3.07
atmosmodm	Jac	1,489,752	10,319,760	3.50e+05	77	77	2.43e-01	75	2.41e-01	3.15	3.62
audikw_1	Jac	943,695	77,651,847	1.58e+07	11	11	8.14e+00	11	8.04e+00	1.63	1.89
bone010	Jac	986,703	47,851,783	8.55e+03	12	12	5.91e-03	12	5.91e-03	2.34	2.36
boneS10	Jac	914,898	40,878,708	7.17e+03	12	12	3.92e-03	12	3.92e-03	2.39	2.20
Bump_2911	Jac	2,911,419	127,729,899	1.91e+14	12	12	1.82e+08	12	1.82e+08	2.97	3.13
cage14	Jac	1,505,785	27,130,349	1.00e+00	5	5	1.55e-07	5	1.55e-07	1.88	2.01
cage15	Jac	5,154,859	99,199,551	1.00e+00	6	6	1.12e-07	6	1.12e-07	1.82	2.06
circuit5M_dc	Jac	3,523,317	14,865,409	1.02e+04	5	5	6.52e-03	5	6.52e-03	3.27	3.45
CurlCurl_3	Jac	1,219,574	13,544,618	2.42e+10	17	17	2.14e+04	17	2.14e+04	2.56	3.09
CurlCurl_4	Jac	2,380,515	26,515,867	2.10e+10	19	19	1.18e+04	19	1.18e+04	3.12	3.60
ecology1	Jac	1,000,000	4,996,000	1.96e+01	8	8	8.77e-06	8	9.66e-06	3.04	3.64
ecology2	Jac	999,999	4,995,991	1.96e+01	8	9	7.38e-06	8	9.67e-06	2.60	2.84
Hardesty1	Jac	938,905	12,143,314	9.99e+00	17	19	9.28e-06	19	4.60e-06	4.02	4.82
ML_Geer	Jac	1,504,002	110,686,677	4.89e+02	2886	2889	2.83e-04	3060	1.19e-04	2.71	2.69
orsreg_1	Jac	2,205	14,133	4.83e+00	225	231	4.26e-06	210	4.68e-06	1.01	0.82
Queen_4147	Jac	4,147,110	316,548,962	1.94e+14	52	51	6.81e+07	51	7.80e+07	2.12	2.43
rdb3200L	Jac	3,200	18,880	9.96e+00	641	610	9.90e-06	583	3.17e-06	0.92	0.83
s3dkq4m2	Jac	90,449	4,427,725	6.08e+02	23	23	7.27e-05	23	7.27e-05	1.66	1.84
tmt_Unsym	Jac	917,825	4,584,801	6.45e-06	6489	5969	9.34e-12	5388	1.20e-11	3.29	4.63
Transport	Jac	1,602,111	23,487,281	2.45e-02	561	592	2.35e-08	557	1.74e-08	2.65	3.08
vas_Stokes_2M	Jac	2,146,677	65,129,037	4.19e-01	6411	7352	3.34e-07	6664	3.49e-07	1.60	3.09

Table 2 reports the number of required iterations to reach the stopping criterion as well the final true residual for PBiCGStab and ReproPBiCGStab; the latter marks both ExBLAS- and FPE-based variants as they report identical results independently from the number of cores/MPI processes used. We also report the initial residual ($\|r_0\|_2$) which can serve as an indicator in combination with the final true residual of how the convergence unfolds. For the original version, we display the number of iterations on one (iter1) and 32 cores (iter32) as they differ. In fact, there is a variability of the results between the other core counts too. Notably, the two reproducible variants show a tendency to deliver more reliable accuracy of the approximate result (the final true residual) and/or converge faster. For instance, the reproducible variants require significantly less iterations for the vas_stokes_2M, orsreg_1, rdb3200L, Transport, tmt_unsym matrices. The reproducible variants are slightly slower for only two matrices, namely ML_Geer and atmosmodj. For the other matrices, mostly symmetric matrices, the results are comparable between reproducible and non-reproducible versions in terms of the number of iterations; however, there is a fluctuation in the final true residual for the original non-deterministic version.

The table also reports the overhead of the reproducible versions against the original non-deterministic version as the normalized mean time on 48 MPI processes. The two reproducible versions perform well with the overhead under 3x for the majority of the test matrices. The FPE version generally shows better performance than the ExBLAS version: one third of the test matrices show the overhead under 2x.

Table 3 shows similar results for the non-deterministic pipelined PBiCGStab and its reproducible variants. The tendency of reproducible variants to converge faster and to deliver more reliable accuracy is preserved. For instance, the reproducible variants require a lower number of iterations for five matrices: atmosmodl, atmosmodm, atmosmodd, orsreg_1, tmt_unsym, and ML_Geer. The reproducible variants require more iterations for four matrices. It is not unusual for rounding errors to cancel in stable

Table 3. Convergence of the **pipelined PBiCGStab and its reproducible versions** (p-ReproPBiCGStab, identical results reported for both) on a set of the SuiteSparse matrices. The initial guess is $x^0 = 0$. The number of iterations required to reach the tolerance of 10^{-6} on the scaled residual, that is, $\|r^i\|_2/\|r^0\|_2$ is reported along with the corresponding true residual $\|b - Ax^i\|_2$. iterX stands for runs on X MPI processes. The last two columns show the overhead of the reproducible versions with 48 cores/MPI processes, for example, 1.07x for the add32 matrix.

					p-PBiCGStab			p-ReproPBiCGStab			
Matrix	Prec	N	nnz	$\|r^0\|_2$	iter1	iter32	$\|b - Ax^i\|_2$	iter	$\|b - Ax^i\|_2$	FPE	ExBLAS
af_shell10	Jac	1,508,065	52,259,885	1.48e+05	9	9	2.18e-02	9	2.18e-02	2.38	2.81
atmosmodd	Jac	1,270,432	8,814,880	3.75e+03	222	223	2.95e-03	222	9.56e-04	3.22	3.88
atmosmodj	Jac	1,270,432	8,814,880	3.75e+03	220	227	3.36e-03	229	3.03e-03	3.22	3.96
atmosmodl	Jac	1,489,752	10,319,760	1.85e+04	140	138	1.76e-02	134	1.82e-02	3.15	3.73
atmosmodm	Jac	1,489,752	10,319,760	3.50e+05	77	78	2.08e-01	77	2.25e-01	2.85	3.38
audikw_1	Jac	943,695	77,651,847	1.58e+07	11	11	8.10e+00	11	8.05e+00	1.95	2.15
bone010	Jac	986,703	47,851,783	8.55e+03	12	12	5.91e-03	12	5.91e-03	2.23	2.51
boneS10	Jac	914,898	40,878,708	7.17e+03	12	12	3.92e-03	12	3.92e-03	2.70	2.47
Bump_2911	Jac	2,911,419	127,729,899	1.91e+14	12	12	1.82e+08	12	1.82e+08	3.09	3.38
cage14	Jac	1,505,785	27,130,349	1.00e+00	5	5	1.55e-07	5	1.55e-07	2.13	2.51
cage15	Jac	5,154,859	99,199,551	1.00e+00	6	6	1.12e-07	6	1.12e-07	1.82	2.09
circuit5M_dc	Jac	3,523,317	14,865,409	1.02e+04	5	5	6.52e-03	5	6.52e-03	2.77	3.28
CurlCurl_3	Jac	1,219,574	13,544,618	2.42e+10	17	17	2.14e+04	17	2.14e+04	2.65	3.06
CurlCurl_4	Jac	2,380,515	26,515,867	2.10e+10	19	19	1.18e+04	19	1.18e+04	3.07	3.63
ecology1	Jac	1,000,000	4,996,000	1.96e+01	9	8	9.90e-06	8	9.56e-06	3.08	3.52
ecology2	Jac	999,999	4,995,991	1.96e+01	8	8	1.08e-05	9	1.07e-05	3.75	4.60
Hardesty1	Jac	938,905	12,143,314	9.99e+00	17	18	5.13e-06	19	6.11e-06	2.72	3.38
ML_Geer	Jac	1,504,002	110,686,677	4.89e+02	2426	3707	5.64e-02	2903	5.91e-02	1.34	1.55
orsreg_1	Jac	2,205	14,133	4.83e+00	239	249	4.80e-06	176	4.17e-06	1.29	1.20
Queen_4147	Jac	4,147,110	316,548,962	1.94e+14	52	51	1.08e+08	50	1.38e+08	2.18	2.44
rdb3200L	Jac	3,200	18,880	9.96e+00	671	634	4.53e-06	660	8.81e-06	1.20	1.30
s3dkq4m2	Jac	90,449	4,427,725	6.08e+02	23	23	7.33e-05	23	7.27e-05	1.84	1.98
tmt_Unsym	Jac	917,825	4,584,801	6.45e-06	6641	6794	9.76e-06	5148	1.48e-09	3.40	4.12
Transport	Jac	1,602,111	23,487,281	2.45e-02	580	582	2.45e-08	587	2.31e-08	2.67	3.15
vas_Stokes_2M	Jac	2,146,677	65,129,037	4.19e-01	6880	6408	2.77e-07	6503	4.23e-07	2.03	2.21

algorithms (see Higham (2002), e.g. page 19) yielding faster convergence of the method. As a consequence, it may happen that a computation with more precision takes more time to converge than a one with less precision. With the pipelined PBiCGStab, we were able to converge to the approximate solution of vas_stokes_2M under the tolerance of 10^{-6}. When the required tolerance is increased to, for example, 10^{-8} or higher, the pipelined methods may not converge for vas_stokes_2M, ML_Geer, and tmt_unsym. We leave this as a future work and foresee to investigate this correlation between the requested accuracy and the abilities of the solvers, potentially employing some healing techniques like residual replacement as well as more advanced preconditioners.

In addition, the table exhibits the overhead of the reproducible pipelined BiCGStab variants against the original version on 48 MPI processes. The two reproducible versions show the overhead of 3x for most of the tested SuiteSparse matrices. As for the standard BiCGStab method, the FPE version generally shows better performance than the ExBLAS: three quarters of the cases exhibit the overhead under 3x; for the rest, the overhead never exceeds 4x.

Figure 4 presents the convergence history in terms of the residual computed at every iteration of both the standard and pipelined PBiCGStab methods. The depicted two matrices, namely orsreg_1 and tmt_unsym, represent the beneficial scenarios for the reproducible variants, when they reach the approximate solution in significantly less iterations than their non-deterministic variants. In fact, these results demonstrate a sort of desired scenario when the reproducible variants converge to the solution faster despite yielding more costly computations per each iteration. In the case of these two matrices, which may not be generic, the standard and pipelined PBiCGStab non-deterministic variants require more iterations on various MPI processes. Moreover, the number of iterations to reach the approximation of the solution fluctuates significantly among runs of

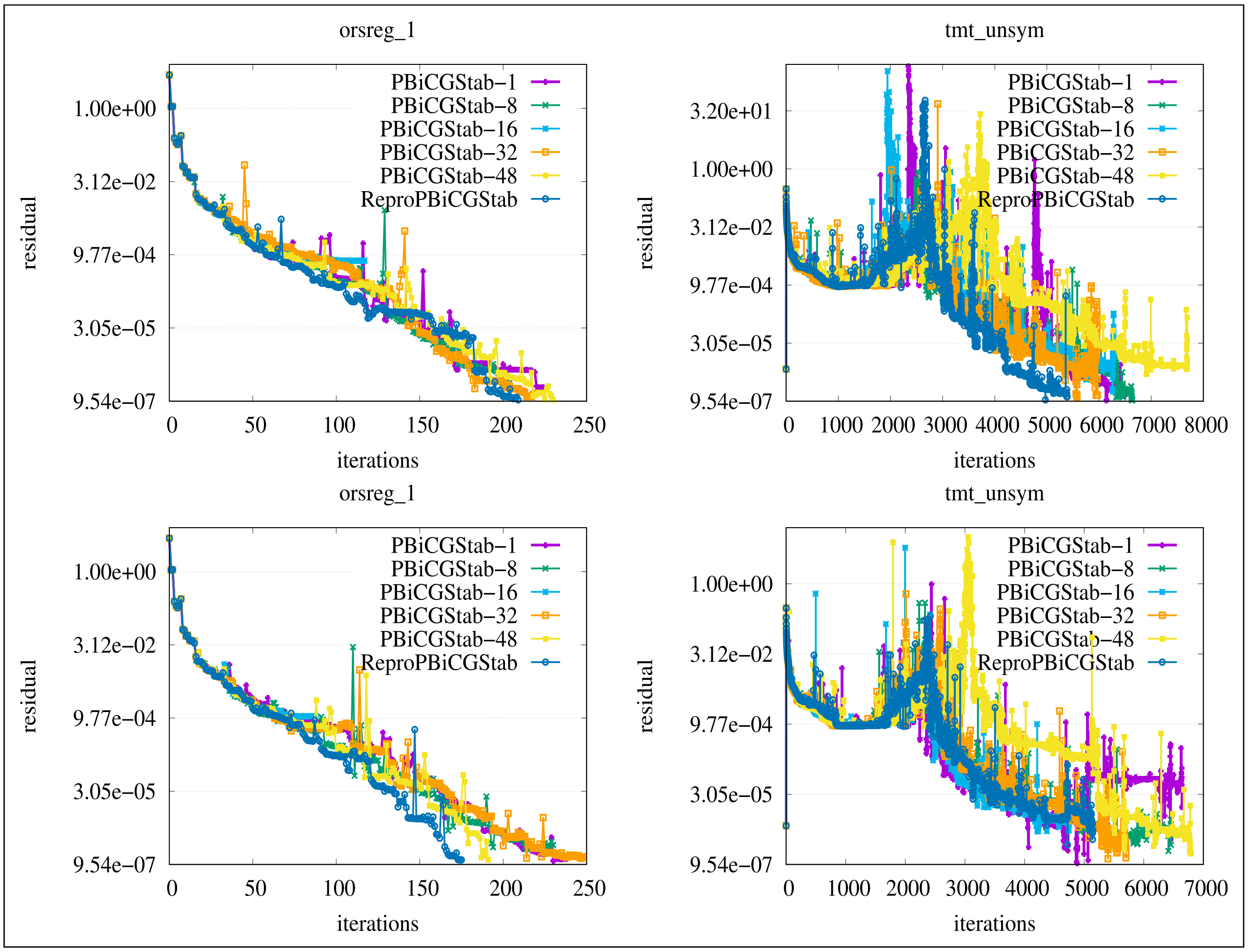

Figure 4. Residual history of the standard PBiCGStab and its reproducible variants (first row), and the pipelined PBiCGStab and its reproducible variants (second row); orsreg_1 results are shown in the first column, while tmt_unsym in the second column, see Table 2 for details on matrices. Note that the last iteration is not shown.

the same non-deterministic variant on a different process count.

Figure 5 demonstrates the strong scalability results – when the problem is fixed but the number of allocated resources varies – for the original and both ExBLAS- and FPE-based standard and pipelined PBiCGStab variants on the Queen_4147 matrix. The figures in the left column report the mean execution time for the entire loop of the solver among five samples, while the figures in the right column show the performance overhead of the reproducible versions. We select the Queen_4147 due to the large number of nonzero elements, 316 millions. As we observed, the smaller number of nonzeros leads to the worse scalability, especially on the large core count, and higher overhead for reproducible variants, but never more than 8x. For small matrices like orsreg_1, a lower number of cores is a preferable option to reach an approximation to the solution with the sustainable resource utilization. In these experiments, MPI communication is performed within a node, most likely being exposed to intra-node communication via shared memory. All variants show good scalability results for Queen_4147 with 28x (24x), 29x (29x), and 31x (31x) speed up on 48 MPI processes, when compared to the one process runs, for the original, FPE, and ExBLAS variants of the standard PBiCGStab (pipelined PBiCGStab), respectively. The reproducible variants demonstrate higher/better speedup due to extra floating-point operations. The overhead of the ExBLAS and FPE variants compared to the standard variant is reduced to nearly 2.5x and 2.3x, accordingly, on 48 MPI processes; the pipelined versions exhibit slightly higher overhead on the small core count. The scalability on the other matrices from Tables 2 and 3 shows variable patterns and overhead.

Note that the average execution time per loop for many matrices from Tables 2 and 3 is not sufficient for distributed memory computations. This is due to the fact that the potential performance gain from extra nodes is demolished by communication.

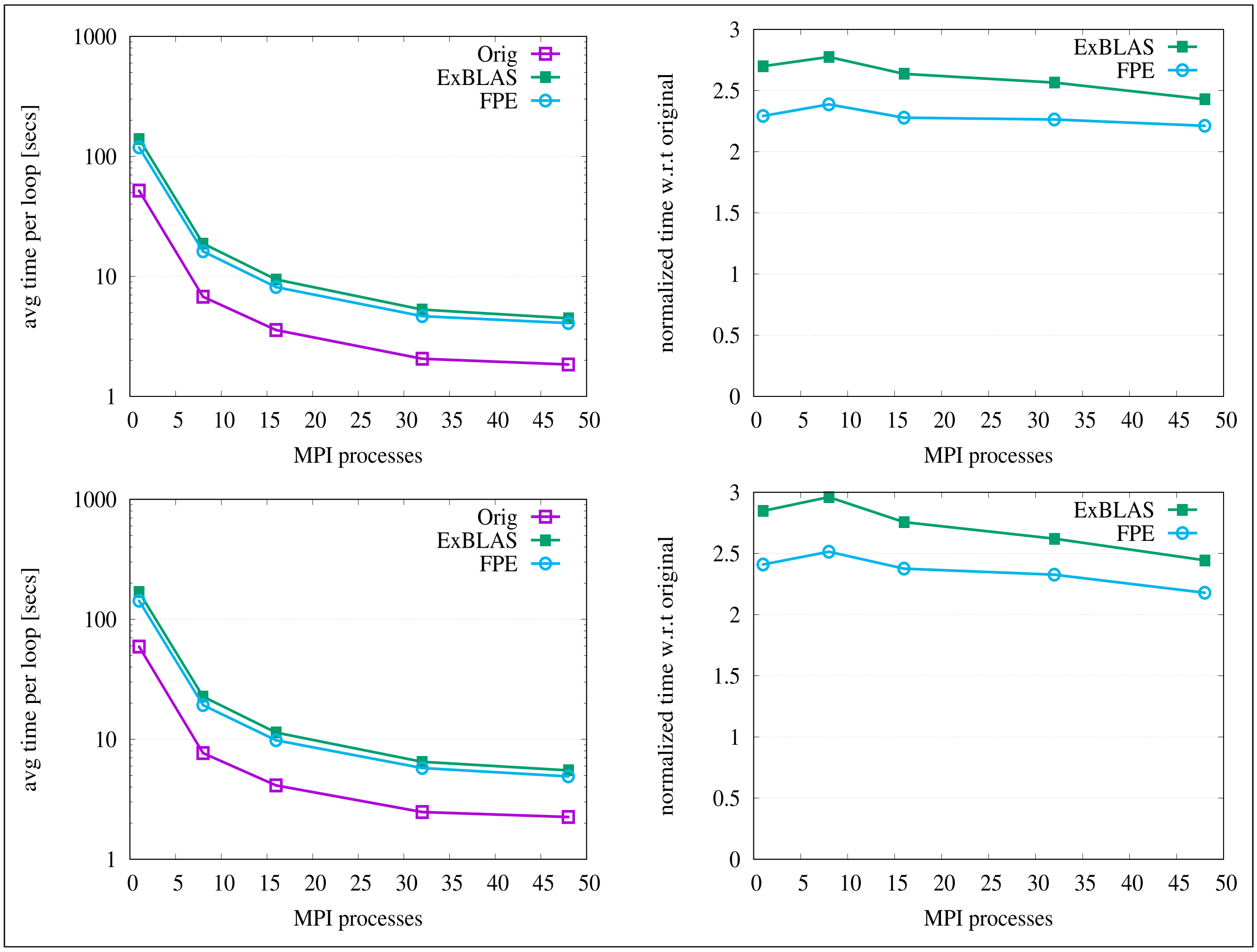

Figure 5. Strong scaling results of the standard PBiCGStab and its reproducible variants (first row), and the pipelined PBiCGStab and its reproducible variants (second row) for the Queen_4147 matrix, see Table 2; plots in the first column report the measure time, while plots in the second show the overhead.

5.2. Scalability

We leverage a sparse SPD coefficient matrix arising from the finite-difference method of a 3D Poisson's equation with 27 stencil points. We perturb the matrix with the values 1.0 − 0.0001 below the central point to create the unsymmetric 27-point stencil aka the e-type model (Cools and Vanroose, 2017). Given that the theoretical cost of PBiCGStab is $t_c \approx 4nnz + 26n$ floating-point arithmetic operations, where nnz denotes the number of nonzeros of the original matrix and its size n, the execution time of the method is usually dominated by that of the SPMV kernel. Therefore, in order to analyze the weak scalability of the method, we maintain the number of nonzero entries per node. For this purpose, we modified the original matrix, transforming it into a band matrix, where the lower and upper bandwidths ($bandL$ and $bandU$, respectively) depend on the number of nodes employed in the experiment as follows:

$$bandL = bandU = 100 \times \#nodes \quad \rightarrow$$

$$nnz = (bandL + bandU + 1) \times n.$$

With 32 nodes, the bandwidth ranges between 100 and 3200. With this approach we can then maintain the number of rows/columns of the matrix equal to n=4M (4,019,679), while increasing its bandwidth and, therefore, the computational workload proportionally to the hardware resources, as required in a weak scaling experiment.

The right-hand side vector b in the iterative solvers was always initialized to the product of A with a vector containing ones only; and the PBiCGStab iteration was started with the initial guess $x_0 = 0$. The parameter that controls the convergence of the iterative process was set to 10^{-6}.

Figure 6 reports the results of both strong and weak scaling for the reproducible variants against the original version. For the strong scaling, we fix the problem to 64M nonzeros and vary the number of nodes/cores used, while

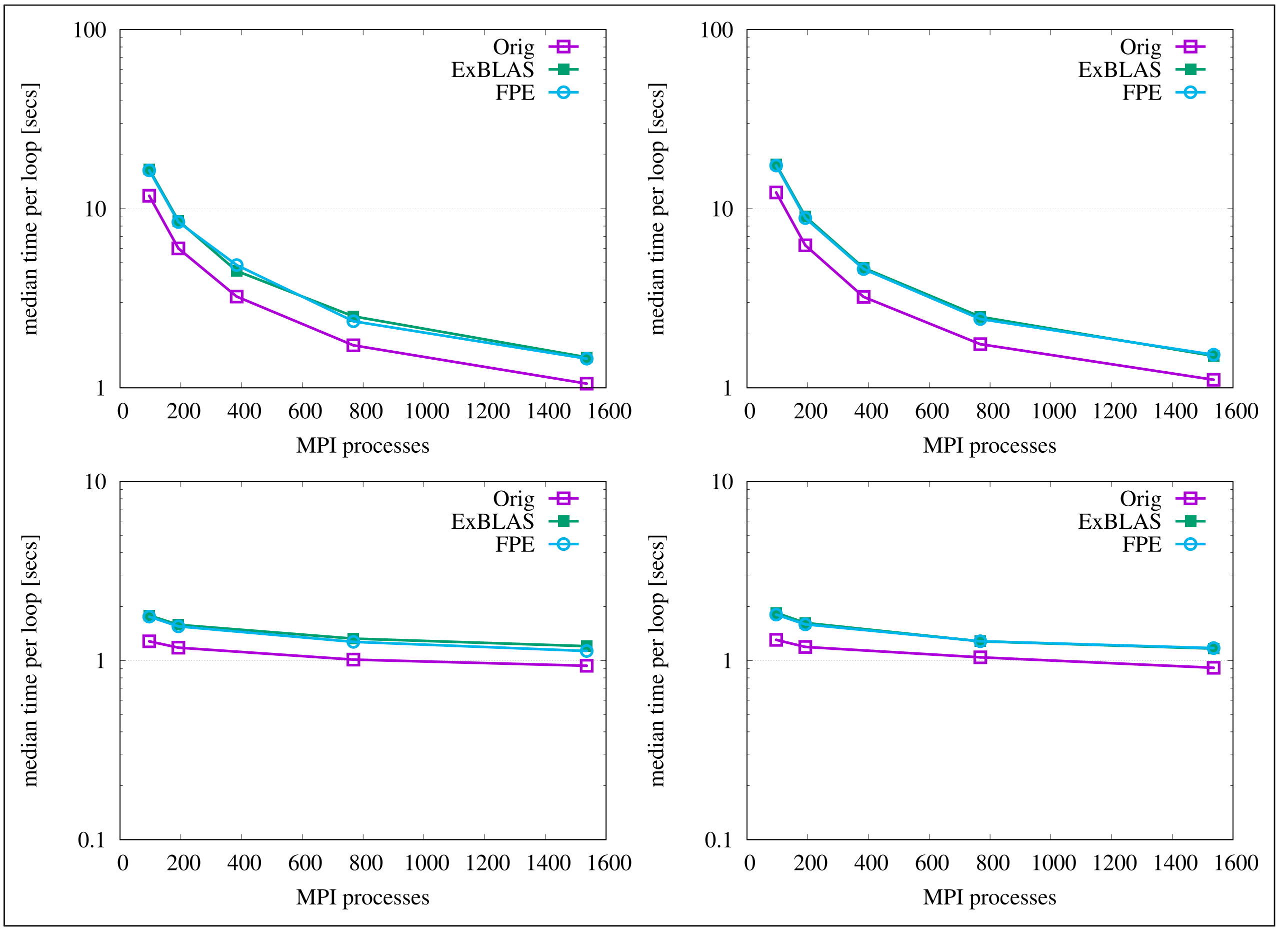

Figure 6. Strong (top row) and weak (bottom row) scalability of the reproducible PBiCGStab variants; the standard PBiCGStab results are shown in the left column of plots, while the pipelined PBiCGStab results in the right column.

for the weak scaling the work load per node is kept constant as 4M nonzeros by varying the bandwidth with respect to the number of nodes involved; presumably, there is enough load to hide the impact of communication. We select median time among five runs to limit the impact of the outliers. We run the tests within a single allocation for 32 nodes to make sure that there is no additional unnecessary perturbations to the measured time. For the strong scaling tests, the standard and pipelined PBiCGStab variants show a similar convergence behavior. However, for the standard variants the global reductions are not overlapped with computations and may show higher overhead in case of the FPE reproducible version due to a more complex reduction operation. For the standard reproducible versions, the overhead on 32 nodes is 37.8% and 40.2% for the FPE and ExBLAS versions, accordingly. For the pipelined reproducible versions, the performance penalties are similar with 38.0% for the FPE version and 35.9% for the ExBLAS. The weak scalability experiments show expected behavior with the slightly declining line of the execution time and the overheads around 35%.

5.3. Accuracy and reproducibility

In addition, we derive a sequential version of the PBiCGStab as in Figure 2 that relies on the GNU Multiple Precision Floating-Point Reliably (MPFR) library (Fousse and et al. (2007)) – a C library for multiple (arbitrary) precision floating-point computations on CPUs – as a highly accurate reference implementation. This implementation uses 2048 bits of accuracy for computing DOT products, 192 bits for internal element-wise product, and performs correct rounding of the computed result to double precision.

Table 4 reports the intermediate and final (except from the original version that takes longer) scaled residual on each iteration of the PBiCGStab solvers for the orsreg_1 matrix, as in Table 2, under the tolerance of 10^{-6} on eight MPI processes. We also add the results of the original code on one core/process to highlight the reproducibility issue. The results are presented with all digits using hexadecimal representation. We report only few iterations, however the difference is present on all iterations. The sequential MPFR version of PBiCGStab confirms the

Table 4. Accuracy and reproducibility of the intermediate and final residual against the Multiple Precision Floating-Point Reliably (MPFR) library for the orsreg_1 matrix, see Table 2.

	Residual			
Iteration	MPFR	Original 1 proc	Original 8 procs	*ExBLAS & FPE*
0	0x1.3566ea57eaf3fp+2	0x1.3566ea57ea**b49**p+2	0x1.3566ea57ea**b49**p+2	0x1.3566ea57eaf3fp+2
1	0x1.146d37f18fbd9p+0	0x1.146d37f18f**aaf**p+0	0x1.146d37f18f**ab**p+0	0x1.146d37f18fbd9p+0
…	…	…	…	…
99	0x1.cedf0ff322158p-13	**0x1.88008701ba87p-12**	**0x1.04e23203fa6fcp-12**	0x1.cedf0ff322158p-13
100	0x1.be3698f1968cdp-13	**0x1.55418acf1af27p-12**	0x1.**fbf5d3a5d1e49**p-13	0x1.be3698f1968cdp-13
…	…	…	…	…
208	0x1.355b0f18f5ac1p-20	**0x1.19edf2c932ab8p-18**	0x1.**b051edae310c7**p-20	0x1.355b0f18f5ac1p-20
209	0x1.114dc7c9b6d38p-20	**0x1.19b74e383f74ep-18**	0x1.**a18fc929018d4**p-20	0x1.114dc7c9b6d38p-20
210	0x1.03b1920a49a7ap-20	**0x1.19c846848f361p-18**	0x1.**c7eb5bbc198b1**p-20	0x1.03b1920a49a7ap-20

accuracy and reproducibility of parallel ExBLAS and FPE variants by reporting identical number of iterations, intermediate residuals, and both the final true and initial scaled residuals. However, the MPFR variant of PBiCGStab converges to the approximate solution in 4.04e-01 s, while the ExBLAS and FPE variants take 3.94e-02 and 3.33e-02 s (10.24x and 12.14x faster), accordingly, on eight MPI processes; the overhead of MPFR is 2.14x and 2.68x for ExBLAS and FPE using one MPI process. The original version of PBiCGStab shows the discrepancy from few digits on the initial iteration and up to almost the entire number on the final iterations; the count of required iterations also differs from the reproducible and MPFR variants.

We extend our study of accuracy and reproducibility to provide more details on the execution time of the MPFR version of PBiCGStab by comparing it against the two reproducible versions, namely, FPE and ExBLAS. Table 5 reports the execution time of the MPFR version and its overhead against the FPE and ExBLAS version on a set of SuiteSparse matrices. On a single process, the MPFR version generally requires 2x more time. This gap grows with the number of parallel resources used. For instance, on 16 cores/MPI processes, the MPFR overhead can be as large as 40x compared to the FPE version with the identical accuracy of both. However, the reproducible versions are not only the faster way for accurate and reproducible computations (e.g. for numerical verification), but also they are as accurate as the MPFR implementation of PBiCGStab.

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 (Collange and et al., 2015).

Demmel and Nguyen (2013, 2015) implemented a family of algorithms – that originate from the works by Rump et al. (2010, 2008) – for reproducible summation in floating-point arithmetic. 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 (Nguyen and Demmel (2015)).

The other approach to ensure reproducibility is called ExBLAS, which is initially proposed by Collange et al. (2015). ExBLAS is based on combining long accumulators and floating-point expansions in conjunction with error-free transformations. This approach is presented in Section 2. Collange et al. (2015) showed 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 (Iakymchuk et al., 2019).

Table 5. Execution time of the sequential MPFR version of PBiCGStab under the tolerance of 10^{-6} and its comparison against the FPE and ExBLAS reproducible versions on a set of the SuiteSparse matrices, see Table 2; iterX stands for runs on X MPI processes and the values in iterX columns show how many times FPE/ExBLAS is faster.

	MPFR	FPE gain			ExBLAS gain		
Matrix	(secs)	iter1	iter8	iter16	iter1	iter8	iter16
af_shell10	12.516	2.3	17.6	35.1	2.0	15.0	29.9
atmosmodd	236.325	2.7	20.1	39.7	2.2	16.5	32.6
atmosmodj	246.302	2.7	20.2	39.9	2.2	16.5	32.7
atmosmodl	159.122	2.6	19.4	38.0	2.1	15.6	30.9
atmosmodm	84.019	2.5	18.2	35.8	2.0	14.8	29.2
audikw_1	11.754	2.1	14.1	27.2	1.8	12.7	24.5
bone010	12.830	2.1	15.9	31.1	1.9	14.0	27.5
boneS10	11.671	2.3	16.7	32.9	2.0	14.7	28.9
Bump_2911	34.846	2.3	17.0	32.4	2.0	14.5	27.6
cage14	7.062	2.4	15.6	28.8	2.0	13.7	25.3
cage15	29.513	2.4	15.8	29.3	2.0	13.8	25.6
circuit5M_dc	13.443	2.2	15.1	28.8	1.7	12.1	23.3
CurlCurl_3	17.701	2.8	21.0	41.1	2.3	17.1	34.0
CurlCurl_4	38.168	2.8	20.7	40.5	2.2	17.0	33.3
ecology1	5.916	2.4	18.2	36.8	1.9	14.5	29.4
ecology2	6.255	2.5	19.4	38.8	2.0	15.3	31.0
Hardesty1	13.692	2.2	17.1	33.9	1.8	13.7	27.6
ML_Geer	6156.862	1.8	13.6	26.9	1.4	12.4	24.7
orsreg_1	0.404	2.7	12.2	13.0	2.1	10.0	11.7
Queen_4147	241.308	2.0	14.9	29.6	1.7	12.8	25.6
rdb3200L	1.764	2.9	17.9	23.3	2.3	14.4	19.9
s3dkq4m2	2.060	2.1	16.1	30.0	1.8	13.9	26.1
Transport	779.205	2.4	17.8	35.0	2.0	15.0	29.4
vas_Stokes_2M	14666.959	2.1	10.3	19.7	1.8	9.4	18.1

Mukunoki and Ogita presented their approach to implement reproducible BLAS, called OzBLAS (Mukunoki et al., 2019), 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 (Ozaki et al., 2012) 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 products, 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.

The proposed framework was implicitly used to derive the reproducible preconditioned Conjugate Gradient (PCG) variants with the flat MPI (Iakymchuk et al., 2020b) and hybrid MPI plus OpenMP tasks (Iakymchuk et al. 2020a). The reproducible PCG variants were primarily verified on the 3D Poisson's equation with 27 stencil points showing the good scalability and low performance overhead (under 30% for both the ExBLAS and lightweight variants) on up to 768 cores of the MareNostrum4 cluster.

7. Conclusions

Parallel Krylov subspace methods may exhibit the lack of reproducibility when implemented in parallel environments as the results in Tables 2–4 confirm. Such numerical reliability is needed for debugging and validation & verification. In this work, we proposed a general framework for re-constructing reproducibility and re-assuring accuracy in any Krylov subspace method. Our framework is based on two steps: analysis of the underlying algorithm for numerical abnormalities; addressing them via algorithmic solutions and programmability hints. The algorithmic solutions are build around the ExBLAS project, namely: ExBLAS that effectively combines long accumulator and FPEs; FPEs for the lightweight version. The programmability effort was focused on: explicitly invoking `fma` instructions to avoid replacements by compilers; customized and `fma`-based axpy and axpy-like operations instead of MKL or similar BLAS libraries; as well as to postpone the division to the moment where it is required.

As test cases, we used the preconditioned standard and pipelined BiCGStab methods and derived two reproducible algorithmic variants for each of them. It is worth mentioning that the two BiCGStab methods are in fact different algorithms with different set of operations yielding non-identical computation path and the divergent way rounding errors are propagate; this difference can be witnessed by the convergence history in Figure 4 even when using the reproducible variants. The reproducible variants deliver identical results of the standard and also pipelined PBiCGStab, which are confirmed by its MPFR version, to ensure reproducibility in the number of iterations, the intermediate and final residuals, as well as the sought-after solution vector. We verified our implementations on a set of the SuiteSparse matrices, showing the performance overhead of nearly 2.0x for the ExBLAS and FPE-based versions, with a noticeably lower overhead for the latter. Tests with the 27-point stencil on 32 nodes show a low performance overhead of 35 %–40%. The code is available at https://github.com/riakymch/ReproPBiCGStab.

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 and 2) analyze the algorithm and make it reproducible by eliminating any uncertainties and non-deterministic order of computations that may violate associativity such as reductions and use/non-use of `fma` and postponing divisions until actually needed. 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 the full obtained bit-wise results. This becomes more evident with the mixed-precision approaches, which we foresee to pursue.

Our future work is to investigate the residual replacement strategy in the pipelined Krylov subspace solvers such as the pipelined PBiCGStab (p-PBiCGStab) (Cools and Vanroose (2017)) and to study if such strategy can be mitigated by the higher precision provided by long accumulator and FPEs. We believe that there is a potential of using higher precision provided by long accumulator and FPEs in order to mitigate the different way rounding errors are propagate as well as to cope with the attainable precision loss in p-PBiCGStab.

Acknowledgments

We acknowledge the usage of computing resources kindly provided by Fraunhofer ITWM.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the Department of Computing Science at Umeå University; the EU H2020 MSCA-IF Robust project (No. 842528); the French ANR InterFLOP project (No. ANR-20-CE46-0009). The research from Universitat Jaume I was funded by the project PID2020-113656RB-C21 via MCIN/AEI/10.13039/501100011033 and project UJI-B2021-58.

ORCID iDs

Roman Iakymchuk https://orcid.org/0000-0003-2414-700X
Jose I. Aliaga https://orcid.org/0000-0001-8469-764X

Notes

1. Reproducibility is the ability to obtain a bit-wise identical and accurate result for multiple executions on the same data in various parallel environments.
2. ExBLAS repository: https://github.com/riakymch/exblas

References

Barrett R (1994) *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. 2nd edition. Philadelphia, PA: SIAM.

Collange C, Defour D, Graillat S, et al. (2015) Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Parallel Computing* 49: 83–97.

Cools S and Vanroose W (2017) The communication-hiding pipelined BiCGstab method for the parallel solution of large unsymmetric linear systems. *Parallel Computing* 65: 1–20. DOI: 10.1016/j.parco.2017.04.005.

Demmel J and Nguyen HD (2013) Fast reproducible floating-point summation. *Proceedings of ARITH-* 21: 163–172.

Demmel J and Nguyen HD (2015) Parallel reproducible summation. *IEEE Transactions on Computers* 64(7): 2060–2070.

Fletcher R (1976) Conjugate gradient methods for indefinite systems. In: Watson GA (ed). *Numerical Analysis*. Berlin, Heidelberg: Springer, 73–89.

Fousse L, Hanrot G, Lefèvre V, et al. (2007) MPFR: a multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software* 33(2): 13. DOI: 10.1145/1236463.1236468.

Goldberg D (1991) What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys* 23(1): 5–48. DOI: 10.1145/103162.103163.

Hida Y, Li XS and Bailey DH (2001) Algorithms for quad-double precision floating point arithmetic. *Proceedings 15th IEEE*

Symposium on Computer Arithmetic. ARITH-15 2001 15: 155–162. DOI: 10.1109/ARITH.2001.930115.

Higham NJ (2002) *Accuracy and Stability of Numerical Algorithms*. 2nd edition. Philadelphia, PA: SIAM. DOI: 10.1137/1.9780898718027.

Iakymchuk R, Collange S, Defour D, et al. (2015) ExBLAS: reproducible and accurate BLAS library. In: Proceedings of the NRE2015 Workshop Held as Part of SC15, Austin, TX, USA, 15–20 November 2015. pp. 1–4.

Iakymchuk R, Graillat S, Defour D, et al. (2019) Hierarchical approach for deriving a reproducible LU factorization. *IJHPCA* 1: 1–13. To Appear HAL preprint: hal-01419813.

Iakymchuk R, Graillat S and José A (2022) General framework for deriving reproducible krylov subspace algorithms: a bicgstab case study, In: Proc. Of PPAM 2022, Gdansk, Poland, September 11-14, 202216. Springer LNCS, pp. –29. DOI: 10.1007/978-3-031-30442-2_2.

Iakymchuk R, Vayá MB, Graillat S, et al. (2020a) Reproducibility of parallel preconditioned conjugate gradient in hybrid programming environments. *The International Journal of High Performance Computing Applications* 34(5): 502–518. DOI: 10.1177/1094342020932650.

Iakymchuk R, Barreda M, Wiesenberger M, et al. (2020b) Reproducibility strategies for parallel preconditioned conjugate gradient. *Journal of Computational and Applied Mathematics* 371: 112697. DOI: 10.1016/j.cam.2019.112697.

Knuth DE (1969) *The Art of Computer Programming: Seminumerical Algorithms*. Reading: Addison-Wesley, 2.

Kulisch U and Snyder V (2011) The exact dot product as basic tool for long interval arithmetic. *Computing* 91(3): 307–313.

Mukunoki D, Ogita T and Ozaki K (2019) *Accurate and reproducible blas routines with ozaki scheme for many-core architectures*. In: Proc. Of PPAM 2019, Bialystok, Poland, September 8-11, 2019. Springer LNCS, pp. 516–527. DOI: 10.1007/978-3-030-43229-4_44

Nguyen HD and Demmel J (2015) Reproducible tall-skinny QR. *2015 IEEE 22nd Symposium on Computer Arithmetic* 22: 152–159. DOI: 10.1109/ARITH.2015.28.

Ogita T, Rump SM and Oishi S (2005) Accurate sum and dot product. *SIAM Journal on Scientific Computing* 26: 1955–1988.

Ozaki K, Ogita T, Oishi S, et al. (2012) Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. *Numerical Algorithms* 59(1): 95–118.

Rump SM, Ogita T and Oishi S (2009) Accurate floating-point summation part II: sign, K-fold faithful and rounding to nearest. *SIAM Journal on Scientific Computing* 31(2): 1269–1302.

Rump SM, Ogita T and Oishi S (2010) Fast high precision summation. *Nonlinear Theory and Its Applications, IEICE* 1(1): 2–24.

Saad Y (2003) *Iterative Methods for Sparse Linear Systems*. 2nd edition. Philadelphia, PA, USA: SIAM.

Saad Y and Schultz MH (1986) GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing* 7: 856–869.

Sonneveld P (1989) CGS, A fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing* 10(1): 36–52.

van der Vorst HA (1992) Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing* 13(2): 631–644. DOI: 10.1137/0913035.

Wiesenberger M, Einkemmer L, Held M, et al. (2019) Reproducibility, accuracy and performance of the Feltor code and library on parallel computer architectures. *Computer Physics Communications* 238: 145–156.

IEEE *Computer Society (2008) IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008. pp 1-70, 2008. https://ieeexplore.ieee.org/document/4610935

Author biographies

Roman Iakymchuk is Associate Professor at the Division of Scientific Computing, Department of Information Technology, Uppsala University (UU), Sweden. He is also Associate Professor at the Department of Computing Science at Umeå University (UmU), Sweden. At UmU, he is a co-Principal Investigator of EuroHPC JU Center of Excellence in Exascale CFD (CEEC) and leads the work package on Exascale Algorithms. Roman develops the Exact BLAS (ExBLAS) library for fast, accurate, and numerically reproducible computations. He extended this idea to Krylov type solvers in hybrid parallel environments. He conducts his research on numerical linear algebra, accuracy and precision of computations, parallel programming models as well as enabling sustainable computations.

Stef Graillat received his PhD degree in 2005 from Université de Perpignan, France. He is Professor in Computer Science at Sorbonne Université and a deputy director of LIP6 laboratory. His research interests are in computer arithmetic, floating-point arithmetic, and validated computing.

José I. Aliaga is Professor in the department of Computer Science and Engineering in the University Jaume I (UJI), Castellón. His main research interests include the application of high-performance computing on sparse numerical linear algebra and Krylov subspace methods, improving both the performance and the energy efficiency of the parallel implementations in hardware accelerators, shared-memory multiprocessors and clusters. Nowadays, all these techniques are being applied in the field of machine learning, particularly in medical image processing. José has published more than 75 papers in journals and conferences, and has also participated in over 40 research projects funded by both national and private organizations (in Spain or within the EU), leading 10 of these projects. He was also involved in five technology transfer contracts (three in Spain and two within the EU), leading four of them.