General Framework for Deriving Reproducible Krylov Subspace Algorithms: BiCGStab Case

Roman Iakymchuk[1,2](✉), Stef Graillat[2], and José I. Aliaga[3]

[1] Umeå University, Umeå, Sweden
riakymch@cs.umu.se
[2] Sorbonne Université, CNRS, LIP6, Paris, France
{roman.iakymchuk,stef.graillat}@lip6.fr
[3] Universitat Jaume I, Castellón de la Plana, Spain
aliaga@uji.es

Abstract. Parallel implementations of Krylov subspace algorithms often help to accelerate the procedure to find the solution of a linear system. However, from the other side, such parallelization coupled with asynchronous and out-of-order execution often enlarge the non-associativity of floating-point operations. This results in non-reproducibility on the same or different settings. This paper proposes a general framework for deriving reproducible and accurate variants of a Krylov subspace algorithm. The proposed algorithmic strategies are reinforced by programmability suggestions to assure deterministic and accurate executions. The framework is illustrated on the preconditioned BiCGStab method for the solution of non-symmetric linear systems with message-passing. Finally, we verify the two reproducible variants of PBiCGStab on a set matrices from the SuiteSparse Matrix Collection and a 3D Poisson's equation.

Keywords: Reproducibility · accuracy · floating-point expansion · long accumulator · fused multiply-add · preconditioned BiCGStab

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) [3]. In this case, Krylov subspace methods fulfill the roles of standard linear algebra solvers [15]. 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) [16], the Bi-Conjugate Gradient (BiCG) method [7], the Conjugate Gradient Squared (CGS) method [17], and the widely used BiCG stabilized (BiCGStab) method by Van der Vorst [18] as a smoother converging version of the above two. 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, 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. The robustness and correctness in this case have a three-fold goal: *reproducibility*[1] of the results with the *accuracy guarantee* as well as *sustainable (energy-efficient)* algorithmic solutions.

In general, Krylov subspace algorithms 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 $x := \alpha x + y$ (AXPY). If a block data distribution is used, only AXPY is perfomed locally, while SpMV needs to gather the full operand vector, e.g. via the `MPI_Allgatherv()` collective, and DOT product requires communication and computation, e.g. via the `MPI_Allreduce()` collective, among MPI processes.

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

– we propose a *general framework for deriving reproducible Krylov subspace algorithms*. We follow the bottom-up approach and ensure reproducibility of Krylov subspace algorithms via reproduciblity of their components, including the global communication. We build our reproducible solutions on the ExBLAS [4] approach and its lighter version.
– even when applying our reproducible solutions, we particularly stress the importance of arranging computations carefully, e.g. 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). 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 [9].
– we verify the applicability of the proposed method on the preconditioned BiCGStab algorithm. We derive two reproducible variants and test them on a set of SuiteSparse matrices and a 3D Poisson's equation.

This article is structured as follows. Section 2 reviews several aspects of computer arithmetic as well as the ExBLAS approach. Section 3 proposes a general

[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.

framework for constructing reproducible Krylov subspace methods. Section 4 introduces the preconditioned BiCGStab algorithms and describes in details its MPI implementation. We evaluate the two reproducible implementations of PBiCGStab in Sect. 5. Finally, Sect. 6 draws conclusions.

2 Background

At first, we will use a floating-point arithmetic that consists in approximating real numbers by numbers that have a finite, fixed-precision representation adhering to the IEEE 754 standard. 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; 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 [4], ReproBLAS [6], OzBLAS [12] 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 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` [10] and for multiplication that is `twoprod` EFT [13]. 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 [11] 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.

The ExBLAS project[2] 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

[2] ExBLAS repository: https://github.com/riakymch/exblas.

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 [4]). Thus, ExBLAS assures reproducibility through assuring correct-rounding.

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

Step	Operation	Kernel	Communication
$S1$:	$d := Ap$	SpMV	Allgatherv
$S2$:	$\rho := \beta / < p, d >$	DOT product	Allreduce
$S3$:	$r := r - \rho d$	AXPY	–
$S4$:	$y := M^{-1} r$	Apply preconditioner	depends
$S5$:	$p := y + \alpha p$	AXPY(-type)	–
$S6$:	$\tau := \sqrt{< r, r >}$	DOT product + sqrt	Allreduce

end while

Fig. 1. Standard preconditioned Krylov subspace method with annotated BLAS kernels and message-passing communication.

Our interest in this article 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` 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.

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; 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 [1,2].

The framework considers the parallel platform to consist of K processes (or MPI ranks), denoted as P_1, P_2, ..., P_K. In this, the coefficient matrix A is partitioned into K blocks of rows (A_1, A_2, ..., 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, ..., r_K and r_k is stored in P_k. Besides, scalars are replicated on all K processes.

Identifying Sources of Non-reproducibility. The first step is to identify sources of non-associativy and, thus, non-reproducibility of the Krylov subspace methods in parallel environments. As it can verify in Fig. 1, there are four common operations as well as message-passing communication patterns associated with them: sparse matrix-vector product (SpMV) and Allgatherv for gathering the vector[3], DOT product with the Allreduce collective, scaling a vector with the following addition of two vectors (AXPY(-type)), 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, while 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 and SpMV are associativity-safe as they are performed locally on local slices of data. The application of preconditioner can also be considered safe, e.g. the Jacobi preconditioner, until all operations are reduction-free; more complex preconditioners will certain raise an issue. Thus, the main issue of non-determinism emerges from parallel reductions (steps S3 and S6 in Fig. 1).

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 four steps:

- exploit the ExBLAS and its lighter FPE-based versions to build reproducible and correctly-rounded DOT product;
- extend the ExBLAS- and FPE-based DOT products to distributed memory by employing `MPI_Reduce()`. This collective acts 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 define the MPI operation that is based on the `twosum` EFT;
- 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 [14] for FPEs;
- distribute the result of DOT product to the other processes by `MPI_Bcast()` as only master performs rounding.

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

[3] Certainly, there are better alternatives for banded or similar sparse matrices, but using `MPI_Allgatherv` is the simplified solution for nonstructured sparse matrices.

with the early-exit technique. 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 sum up, if the information about the problem is know in advance, it is worth pursuing the lightweight approach.

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 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 provide with the same result [19]. This is due to the fact that for

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

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

Step	Operation	Kernel	Comm
$S1:$	$\tilde{s}^j := M^{-1}p^j$	Apply precond.	–
$S2:$	$s^j := A\tilde{s}^j$	SpMV	Allgatherv
$S3:$	$\alpha^j := < r^0, r^j > / < r^0, s^j >$	DOT product	Allreduce
$S4:$	$q^j := r^j - \alpha^j s^j$	AXPY-like	–
$S5:$	$\tilde{y}^j := M^{-1}q^j$	Apply precond.	–
$S6:$	$y^j := A\tilde{y}^j$	SpMV	Allgatherv
$S7:$	$\omega^j := < q^j, y^j > / < y^j, y^j >$	Two DOT products	Allreduce
$S8:$	$x^{j+1} := x^j + \alpha^j p^j + \omega^j q^j$	Two AXPY	–
$S9:$	$r^{j+1} := q^j - \omega^j y^j$	AXPY-like	–
$S10:$	$\beta^j := \frac{<r^0, r^{j+1}>}{<r^0, r^j>} * \frac{\alpha^j}{\omega^j}$	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
```

Fig. 2. Formulation of the 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, $<\cdot, \cdot>$ computes the DOT (inner) product of its vector arguments.

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` ($\texttt{fma}(a, b, t)$), which gives a slightly different result; or it may even compute $t = a*b$ first and then use the `fma` ($\texttt{fma}(c, d, t)$). Thus, we advise to instruct compilers to use `fma` explicitly via `std::fma` in C++ 11, assuming the underlying architecture supports `fma`.

4 BiCGStab

The classic Biconjugate Gradient Stabilized method (BiCGStab) [18] was proposed as a fast and smoothly converging variant of the BiCG [7] and CGS [17] methods. We consider the 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. The algorithmic description of the classical iterative PBiCGStab is presented in Fig. 2. For simplicity, we integrate the Jacobi preconditioner [15] in our implementation, which is composed of the diagonal elements of the matrix ($M = diag(A)$), whereas its application is conducted on a vector and requires an element-wise multiplication of two vectors.

As described in Sect. 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 BiCGStab solver. From there, we derive the reproducible version by following the guide from Sect. 3.

Message-Passing Parallel BiCGStab Implementation. 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 Fig. 2.

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 ($\tilde{s}$ or $\tilde{y}$), which is partitioned and distributed using the same partitioning as A. For simplicity, we just explain below how S2 is computed.

Prior to computing this kernel, we need to obtain a replicated copy of the distributed vector $\tilde{s}$ in all processes, denoted as $\tilde{s} \rightarrow e$; 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 v concurrently: $P_k : s_k = A_k\, e$. This kernel thus requires assembling the distributed pieces of the vector $\tilde{s}$ into a single vector e that is replicated in all processes (in MPI, for example via `MPI_Allgatherv()`). 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.

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 $P_k : \rho_k = <r^0_k, s_k>$ 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, and communications in S10 and S11 can be merged in a single `MPI_Allreduce()`.

AXPY(-type) *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 these types of kernels, all processes can perform their local parts of the computation to obtain the result without any communication: $P_k : q_k = r_k - \alpha\, s_k$.

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: $P_k : \tilde{s}_k = M_k^{-1} p_k$. The same procedure can be applied on the step S5 to scale the vector q, resulting in $\tilde{y}$.

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. In our experiments, we employed IEEE754 double-precision arithmetic and conducted them on the SkyLake partition at Fraunhofer with a dual Intel Xeon Gold 6132 CPU @2.6 GHz, 28 cores, and 192 GB of memory. Nodes are connected with the 54 Gbit/s FDR Infiniband.

Evaluation on the SuiteSparse Matrices. We carried out tests on a range of different linear systems from the SuiteSparse matrix collection on a single SkyLake node using 1, 2, 4, 8, 16, and 28 (full) cores. Table 1 lists a set of tested matrices with the number of rows/columns N and the number of nonzeros nnz. The right-hand side vector b in the iterative solvers was always initialized to the product $Ad, d = \frac{1}{\sqrt{N}}(1, \ldots, 1)^T$, 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 $\frac{1}{\sqrt{N}}$. The PBiCGStab iterations were started with the initial guess $x_0 = 0$. The parameter that controls the convergence of the iterative process is $\|r^j\|_2/\|r^0\|_2 \leq 10^{-6}$.

Table 1 also 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. For the original version, we display the number of iterations on single and eight cores as they differ. Notably, the two reproducible variants show the tendency to deliver better accuracy of the approximate result (the final true residual) or converge faster, for example for orsreg_1, rdb3200l, and tmt_unsym matrices.

Figure 3 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 preconditioned BiCGStab variants on the s3dkq4m2 and af_shell10 matrices. The figure reports the mean execution time for the entire loop of the solver among five samples. We select these matrices due to their large number of nonzero elements, i.e. enough work to show scalability. Note that MPI communication is performed within a node, most likely being exposed to intra-node communication via shared memory. All three variants show good scalability

Table 1. Convergence of the PBiCGStab and ReproPBiCGStab 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. $\|r^j\|_2/\|r^0\|_2$, is reported along with the corresponding true residual $\|b - Ax^j\|_2$.

Matrix	Prec	N	nnz	$\|r^0\|_2$	BiCGStab			ReproBiCGStab	
					iter1	iter8	$\|b - Ax^j\|_2$	iter	$\|b - Ax^j\|_2$
add32	Jac	4,960	19,848	$6.38e-05$	36	36	$4.97e-09$	35	$7.12e-09$
bcsstk18	Jac	11,948	149,090	$5.29e+18$	7	7	$7.51e+02$	7	$7.51e+02$
bcsstk26	Jac	1,922	30,336	$3.80e+19$	11	11	$5.62e+03$	11	$5.62e+03$
orsreg_1	Jac	2,205	14,133	$2.34e+01$	225	228	$4.18e-06$	210	$4.68e-06$
pde2961	Jac	2,961	14,585	$9.24e-02$	128	123	$5.28e-08$	125	$2.67e-07$
rdb3200l	Jac	3,200	18,880	$9.92e+01$	641	605	$4.09e-06$	583	$3.17e-06$
saylr4	Jac	3,564	22,316	$9.44e+06$	10	10	$1.95e-03$	10	$7.26e-05$
s3dkq4m2	Jac	90,449	4,427,725	$3.70e+05$	23	23	$7.26e-05$	23	$7.27e-05$
af_shell10	Jac	1,508,065	52,259,885	$1.48e+05$	12	12	$3.44e-04$	12	$3.44e-04$
atmosmodd	Jac	1,270,432	8,814,880	$3.75e+03$	255	272	$3.41e-05$	257	$2.33e-05$
atmosmodm	Jac	1,489,752	10,319,760	$3.50e+05$	117	110	$3.47e-03$	109	$2.73e-03$
cage15	Jac	5,154,859	99,199,551	$1.00e+00$	8	8	$4.56e-09$	8	$4.56e-09$
tmt_unsym	Jac	917,825	4,584,801	$6.45e-06$	6957	7458	$7.44e-12$	5969	$1.02e-11$
Hardesty1	Jac	938,905	12,143,314	$9.99e+00$	24	24	$8.45e-08$	25	$8.61e-08$
ecology1	Jac	1,000,000	4,996,000	$1.96e+01$	11	12	$1.30e-07$	12	$9.08e-08$
ecology2	Jac	999,999	4,995,991	$1.96e+01$	14	13	$1.79e-08$	13	$5.39e-08$
CurlCurl_3	Jac	1,219,574	13,544,618	$2.42e+10$	24	24	$2.00e+02$	24	$2.00e+02$

results for s3dkq4m2 with 10.4×, 12.8×, and 13.3× speed up on 16 MPI processes for the original, FPE, and ExBLAS variants, respectively; the corresponding speed up of 8.8×, 12.2×, and 12.8× for af_shell10. The reproducible variants demonstrate higher speedup due to extra floating-point operations. The overhead of the ExBLAS and FPE variants compared to the original variant is reduced to 2.4× and 2× for s3dkq4m2 as well as to 1.9× and 2.2× for af_shell10, accordingly, on 28 MPI processes. The scalability on the other matrices from Table 1 shows the similar pattern and overhead. However, the smaller number of nonzeros leads to the worse scalability. For instance, for the orsreg_1 matrix, the original and ExBLAS/FPE variants are only 4× and 8×, respectively, faster on 16 MPI processes.

Note that the average execution time per loop for many matrices 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.

Scalability. We leverage a sparse s.p.d. 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 [5]. The fact that the vector involved in the SpMV kernel has to be replicated in all MPI ranks constrains the size of the largest problem that can be solved. 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 non-zero entries per node. For this purpose, we modified the

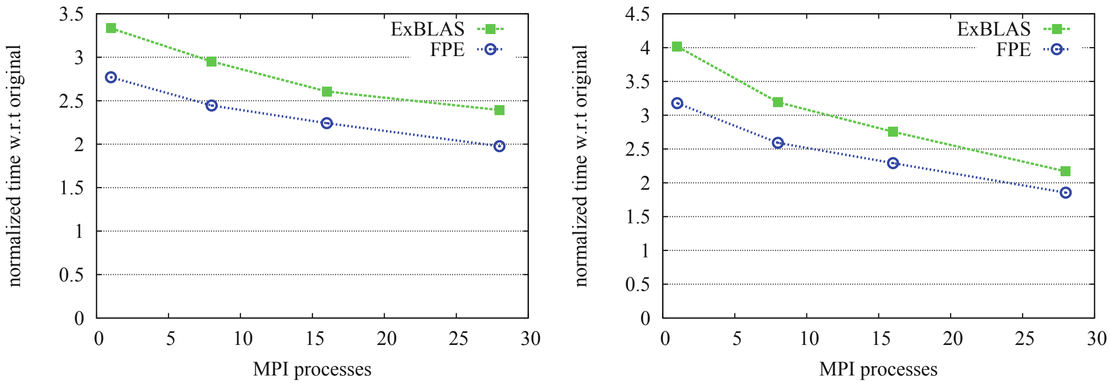

Fig. 3. Strong scaling results of the original and reproducible PBiCGStab variants with the Jacobi preconditioner on one SkyLake node for the s3dkq4m2 (left) and af_shell10 (right) matrices, see Table 1 for details.

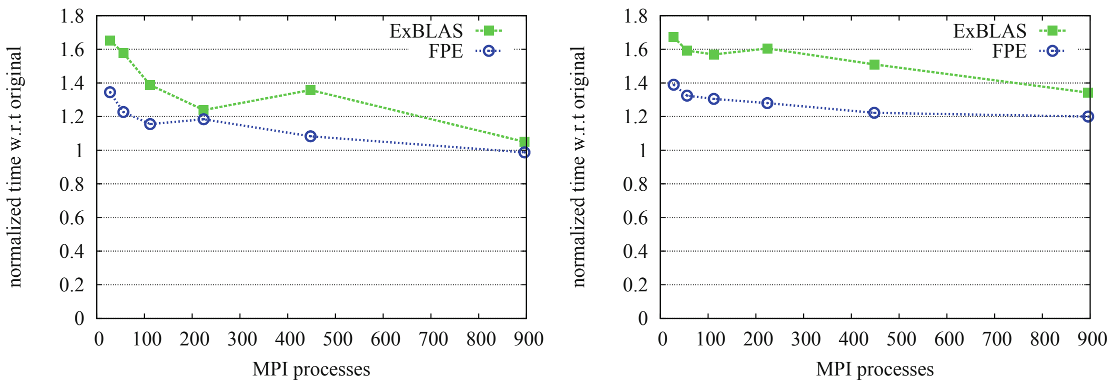

Fig. 4. Strong (left) and weak (right) scalability of the reproducible PBiCGStab variants with the normalized time against the non-deterministic MPI variant.

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:

$$bandL = bandU = 100 \times \#nodes \quad \rightarrow \quad nnz = (bandL + bandU + 1) \times n.$$

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^{-8}.

Figure 4 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 16M non-zeros and varied the number of nodes/cores used, while for the weak scaling the work load per node was fixed to 4M non-zeros and the bandwidth was increased with respect to the number of nodes involved. For both scalability cases, the initial overhead is the same, namely 67% for the version with ExBLAS and 38–40% for FPE. With the strong scaling, the overhead reduces to 8.2% for ExBLAS and 3.0% for FPE as the communication starts to take over and the overhead between the two versions narrows. For the weak scaling, the matrix size is kept constant per node so that there is enough load to hide the impact of communication.

Accuracy and Reproducibility. In addition, we derive a sequential version of the preconditioned BiCGStab as in Fig. 2 that relies on the GNU Multiple Precision Floating-Point Reliably (MPFR) library [8] – a C library for multiple (arbitrary) precision floating-point computations on CPUs – as a highly accurate

Table 2. Accuracy and reproducibility of the intermediate and final residual against MPFR for the orsreg_1 matrix, see Table 1.

Iteration	Residual			
	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

reference implementation. This implementation uses 2,048 bits of accuracy for computing dot product, 192 bits for internal element-wise product, and performs correct rounding of the computed result to double precision.

Table 2 reports the intermediate and final (except from original that takes longer) scaled residual on each iteration of the PBiCGStab solvers for the orsreg_1 matrix, as in Table 1, 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 confirms the 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 3.39e−01 s, while the ExBLAS and FPE variants take 3.95e−02 and 2.75e−02 s (8.57× and 12.32× faster), accordingly, on eight MPI processes. The original code 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.

6 Conclusions

Parallel Krylov subspace algorithms may exhibit the lack of reproducibility when implemented in parallel environments as the results in Table 2 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 algorithm. Our framework is based on two steps: analysis of the underlying algorithm for the arithmetic 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 only for the leightweight version. The programmability effort was focused on: explicitly invoking `fma` instructions to avoid replacements by compilers as well as to postpone the division to the moment where it is required. As a test case, we used the preconditioned BiCGStab algorithm and derived two reproducible algorithmic variants of it. Both reproducible variants deliver identical results of 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 the SuiteSparse matrices, showing the performance overhead of 2.5× and 2× for the ExBLAS and FPE-based versions, accordingly; tests with the 27-point stencil on 32 nodes show almost negligible overhead of 8% and 3%, respectively.

Acknowledgment. This research was partially supported by 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.

References

1. Iakymchuk, R., et al.: Reproducibility of parallel preconditioned conjugate gradient in hybrid programming environments. IJHPCA **34**(5), 502–518 (2020). https://doi.org/10.1177/1094342020932650
2. Iakymchuk, R., et al.: Reproducibility strategies for parallel preconditioned conjugate gradient. JCAM **371**, 112697 (2020). https://doi.org/10.1016/j.cam.2019.112697
3. Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd edn. SIAM (1994)
4. Collange, S., et al.: Numerical reproducibility for the parallel reduction on multi- and many-core architectures. Parallel Comput. **49**, 83–97 (2015). https://doi.org/10.1016/j.parco.2015.09.001
5. Cools, S., Vanroose, W.: The communication-hiding pipelined BiCGstab method for the parallel solution of large unsymmetric linear systems. Parallel Comput. **65**, 1–20 (2017). https://doi.org/10.1016/j.parco.2017.04.005
6. Demmel, J., Nguyen, H.D.: Parallel reproducible summation. IEEE Trans. Comput. **64**(7), 2060–2070 (2015). https://doi.org/10.1109/TC.2014.2345391
7. Fletcher, R.: Conjugate gradient methods for indefinite systems. In: Watson, G.A. (ed.) Numerical Analysis. LNM, vol. 506, pp. 73–89. Springer, Heidelberg (1976). https://doi.org/10.1007/BFb0080116
8. Fousse, L., et al.: MPFR: a multiple-precision binary floating-point library with correct rounding. ACM TOMS **33**(2), 13 (2007). https://doi.org/10.1145/1236463.1236468
9. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. **23**(1), 5–48 (1991). https://doi.org/10.1145/103162.103163
10. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms, vol. 2. Addison-Wesley (1969)
11. Kulisch, U., Snyder, V.: The exact dot product as basic tool for long interval arithmetic. Computing **91**(3), 307–313 (2011). https://doi.org/10.1007/s00607-010-0127-7

12. Mukunoki, D., Ogita, T., Ozaki, K.: Reproducible BLAS routines with tunable accuracy using Ozaki scheme for many-core architectures. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K. (eds.) PPAM 2019. LNCS, vol. 12043, pp. 516–527. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43229-4_44
13. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. **26**, 1955–1988 (2005). https://doi.org/10.1137/030601818
14. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation part II: sign, K-fold faithful and rounding to nearest. SIAM J. Sci. Comput. **31**(2), 1269–1302 (2008). https://doi.org/10.1137/07068816X
15. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM, Philadelphia (2003). https://doi.org/10.1137/1.9780898718003
16. Saad, Y., Schultz, M.H.: GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. **7**, 856–869 (1986). https://doi.org/10.1137/0907058
17. Sonneveld, P.: CGS, a fast Lanczos-type solver for nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. **10**(1), 36–52 (1989). https://doi.org/10.1137/0910004
18. van der Vorst, H.A.: Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. **13**(2), 631–644 (1992). https://doi.org/10.1137/0913035
19. Wiesenberger, M., et al.: Reproducibility, accuracy and performance of the Feltor code and library on parallel computer architectures. CPC **238**, 145–156 (2019). https://doi.org/10.1016/j.cpc.2018.12.006