General framework for deriving reproducible Krylov subspace algorithms: A BiCGStab case study

Roman Iakymchuk, Stef Graillat, José I Aliaga

▶ **To cite this version:**

Roman Iakymchuk, Stef Graillat, José I Aliaga. General framework for deriving reproducible Krylov subspace algorithms: A BiCGStab case study. 2021. hal-03382119v1

HAL Id: hal-03382119

https://hal.sorbonne-universite.fr/hal-03382119v1

Preprint submitted on 18 Oct 2021 (v1), last revised 3 Nov 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

General framework for deriving reproducible Krylov subspace algorithms: A BiCGStab case study

Roman Iakymchuk[a,b], Stef Graillat[a], José I. Aliaga[c]

[a] *Sorbonne University, France*
[b] *Fraunhofer ITWM, Germany*
[c] *Universitat Jaime I, Spain*

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 in parallel environments with message-passing. Finally, we verify the two reproducible variants of PBiCGStab on a set matrices from the SuiteSparse Matrix Collection.

Keywords: Reproducibility, accuracy, floating-point expansion, long accumulator, fused multiply-add, preconditioned BiCGStab, High-Performance Computing.

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) [1]. In this case, Krylov subspace methods fulfill the roles of standard linear algebra solvers [2]. 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) [3], the Bi-Conjugate Gradient (BiCG) method [4], the Conjugate Gradient Squared (CGS) method [5], and the widely used BiCG stabilized (BiCGStab) method by H.A. Van der Vosrt [6] 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 BiCGStab 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 [7] 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 threefold goal: *reproducibility* of the results with the *accuracy guarantee* as well as *sustainable (energy-efficient)* algorithmic solutions.

In general, Krylov subspace algorithms are build 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 the addition of

Email addresses: roman.iakymchuk@sorbonne-universite.fr (Roman Iakymchuk), stef.graillat@sorbonne-universite.fr (Stef Graillat), aliaga@icc.uji.es (José I. Aliaga)

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 [8] approach and its lightweight version using floating-point expansions only.
- 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.

This article is structured as follows. Section 2 reviews several aspects of computer arithmetic, in particular floating-point expansion and long accumulator, as well as the ExBLAS approach for accurate and reproducible computations. Section 3 proposes a general 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 Section 5. Finally, Section 6 reviews related work, while Section 7 draws conclusions and outlines future directions.

2. Background

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

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

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

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

The IEEE 754 standard requires correctly rounded results for the basic arithmetic operations ($+, -, \times, /, \sqrt{}$, `fma`). It means that the operations are performed as if the result was first computed with an infinite precision and then rounded to the floating-point format. The correct rounding criterion guarantees a unique, well-defined answer, ensuring bit-wise reproducibility for a single operation. Several rounding modes are provided. The standard also contains the reproducibility clause that forwards the reproducibility issue to language standards. Emerging attention to reproducibility strives to draw more careful attention to the problem by the computer arithmetic community. It has led to the inclusion of error-free transformations (EFTs) for addition and multiplication – to return the exact outcome as the

result and the error – to assure numerical reproducibility of floating-point operations, into the revised version of the standard. These mechanisms, once implemented in hardware, will simplify our reproducible algorithms – like the ones used in the ExBLAS [8], ReproBLAS [11], 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 errors during the course of computations. The first approach uses EFT to compute both the result and the rounding error and stores them in a floating-point expansion (FPE), which is an unevaluated sum of p floating-point numbers, whose components are ordered in magnitude with minimal overlap to cover the whole range of exponents. Typically, FPE relies upon the use of the traditional EFT for addition that is `twosum` [13] (Algorithm 1) and for multiplication that is `twoprod` EFT [14] (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 [15] proposed to use a 4288-bit long accumulator for the exact dot product of two vectors composed of `binary64` numbers; such a large long accumulator is designed to cover all the severe cases without overflows in its highest digit.

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

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

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

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

2.1. ExBLAS – Exact BLAS

The ExBLAS project [16] is an attempt to derive fast, accurate, and reproducible BLAS library by constructing a multi-level approach for these operations that are tailored for various modern architectures with their complex multi-level memory structures. On one side, this approach is aimed to be fast to ensure similar performance compared to the non-deterministic parallel versions. On the other side, the approach is aimed to preserve every bit of information before the final rounding to the desired format to assure correct-rounding and, therefore, reproducibility. Hence, ExBLAS combines together long accumulator and FPE into algorithmic solutions as well as efficiently tunes and implements them on various architectures, including conventional CPUs, Nvidia and AMD GPUs, and Intel Xeon Phi co-processors (for details we refer to [8]). 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 [13] and long accumulators, so it is correctly rounded and reproducible. In practice, the latter is invoked only once per overall summation that results in the little overhead (less than 8 %) on accumulating large vectors. Our interest in this article is the dot product of two vectors, which is another crucial fundamental BLAS operation. The EXDOT algorithm is based on the previous EXSUM algorithm and the `twoprod` EFT [14] (see Algorithm 2): the algorithm accumulates the result and the error of `twoprod` to same FPEs and then follows the EXSUM 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 (`https://github.com/riakymch/exblas`).

3. General framework for deriving a reproducible Krylov subspace algorithm

The use of Krylov subspace methods to solve sparse linear systems ($Ax = b$) allows that the nonzero pattern of the original matrix is never changed, such that memory requirement is maintained during all the execution. These methods are always defined means of an iteration in which the matrix A is only involved in sparse matrix-vector products, and some additional vector operations help to find the sought-after solution vector. For symmetric problems, the corresponding methods (CG [2], MR [2] and its variants) only manages one Krylov subspace, but for the general case one or two Krylov subspaces could be generated. In the first case (GMRES [3], QMR [17] and its variant), the main problem is that the number of computations grows in each iteration, and therefore, the practical implementations limits the number of iterations, incorporating a restarting technique to complete the solution of the linear system. For the second case, the first defined methods (BiCG [4], QMR [17] and its variants) need to manage both A and A^T, and therefore, the optimized implementations double the memory requirements. The alternative is the definition of

Transpose-Free variants of the methods, such as BiCGStab [6] or TFQMR [18], in which the recurrences are changed to avoid the use of A^T.

Additionally, the use of preconditioning improves the convergence of the method. Basically, the problem is modified by a matrix $M^{-1}Ax = M^{-1}b$, such that the condition number of new matrix, $M^{-1}A$, was better than the condition number of A. There are many alternatives to compute matrix M. We consider to use the Jacobi preconditioning, which is $M = diag(A) \iff a_{jj} \neq 0$, produces sufficient results despite being very simple to obtain and also to compute the matrix inverse.

This section provides the outline of a general framework for deriving the 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 [19, 20].

3.1. Identifying sources of non-reproducibility

Our first step is to identify sources of non-associativy and, thus, non-reproducibility of the Krylov subspace methods in parallel environments. We consider a general scheme for the Krylov subspace method presented in Figure 1. This scheme outlines four common operations as well as message-passing communication patterns associated with them: sparse matrix-vector product (SpMV) and Allgatherv for gathering the vector, DOT product with the Allreduce collective, scaling a vector with the following addition of two vectors (AXPY), and the application of the preconditioner. Hence, we investigate each of these four operations. We assume that we deal with the MPI implementation of the solver where each process conducts computations on its own local slices of the matrix as well as the vectors.

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

Step	Operation	Kernel	Communication
S1 :	$w := Ad$	SpMV	Allgatherv
S2 :	$\rho := \beta / < d, w >$	DOT product	Allreduce
	...		
S3 :	$x := x + \rho d$	AXPY	none
	...		
S4 :	$z := M^{-1} r$	Apply preconditioner	depends
	...		
S5 :	$\tau := < r, r >$	DOT product	Allreduce
	...		

end while

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

In general, associativity and reproducibility are not guaranteed when there is perturbation of data in parallel execution. For instance, while invoking the `MPI_Allreduce()` collective one 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. Thus, the main issue of non-determinism emerges from DOT products and, thus, parallel reductions with `MPI_Allreduce()` in steps S1 and S5 in Figure 1.

3.2. Re-assuring reproducibility: strategies and programmability effort

We construct our approach for reassuring reproducibility by primarily targetting 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 that follows by `MPI_Allreduce()`. Thus, we propose to refine this procedure into four steps as follows

- 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 emploting `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, see Algorithm 1;

- 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 [21] for FPEs.
- Distribute the result of DOT product to the other processes by `MPI_Bcast()` as only master performs rounding.

Listings 1 and 2 provide pseudo codes for our implementation of the distributed DOT product using the ExBLAS and lightweight strategies, respectively.

Listing 1: Reproducible Allreduce with ExBLAS.

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

Listing 2: Reproducible Allreduce with FPEs only.

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

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. 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.

Regarding AXPY(-type) and application of the preconditioner, we rely upon the sequential MKL implementation of AXPY and the usage of the Jacobi preconditioner. The later is rather simple and only involves element-wise multiplication of two vectors.

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-determenistic results. For instance, in the distributed implementation of PBiCGStab, each MPI process computes its dedicated part w_k of the vector w by multiplying a block of rows A_k by the vector e. Since the computations are carried locally and sequentially, they are deterministic and, thus, reproducible. However, some parts of the code like $a * b + c * d * e$ and $a+ = b * c$ – present in the original implementation of PBiCGStab – may not always provide with the same result [22]. 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 even allows merging multiplications and summations with fused multiply-add (`fma`) instructions. Hence, a compiler might translate $a * b + c * d$ to two multiplications $t1 = a * x$ and $t2 = b * y$, and a subsequent summation $y = t1 + t2$; it might generate a single multiplication $t = b * y$ with a subsequent `fma` $y =$ `fma`(a, x, t), which gives a slightly different result; or it may even compute $t = a * x$ first and then use the `fma` $y =$ `fma`(b, y, t). Thus, we advise to instruct compilers to use `fma` explicitly via `std::fma` in C++ 11 (in case the underlying architecture supports `fma`).

Another important observation is to carefully perform divisions and initialization of data. For instance, at the initialization in the Krylov solver $d = \frac{1}{\sqrt{N}}(1, \ldots, 1)^T$, N is the number of rows/ columns of A, and $b = Ad$; d is used only once to initialize b. In this case, we suggest to compute $b = Ad$ for $d = (1, \ldots, 1)^T$ first and then scale b by $\frac{1}{\sqrt{N}}$ as we observed a slightly faster convergence (up to 7 %) for the Krylov solver.

4. BiCGStab

The classic Biconjugate Gradient Stabilized method (BiCGStab) [6] was proposed as a fast and smoothly converging variant of the BiCG [4] and CGS [5] methods. We present here the preconditioned BiCGStab algorithm: its design and implementation with Message Passing Interface (MPI).

4.1. Preconditioned Biconjugate Gradient Stabilized Solver

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 Figure 2. The loop body consists of two sparse matrix-vector product (SpMV) (S1 and S7), six DOT products (S2, S4, S8, S11, and S14), six AXPY (-like) operations (S3, S5, S9, S10, and S12), and two preconditioner applications (S6 and S13), and a few scalar operations [1]. For simplicity, in our implementation of the BiCGStab method, we integrate the Jacobi preconditioner [2], which is composed of the elements in the diagonal of the matrix ($M := D = diag(A)$). In consequence, the application of the preconditioner is conducted on a vector and requires an element-wise multiplication of two vectors.

Compute preconditioner for $A \rightarrow M$
Set starting guess x^0
Initialize $r^0 := b - Ax^0$, $p^0 := r^0$, τ^0, $j := 0$ (iteration count)

$r^0 := b - Ax^0$
$\tau^0 :=\| r^0 \|_2$
while $(\tau^j > \tau_{\max})$

Step	Operation	Kernel
S1 :	$v^j := Ap^j$	SpMV
S2 :	$\alpha^j := < r^0, r^j > / < r^0, v^j >$	DOT product
S3 :	$\tilde{s}^j := r^j - \alpha^j v^j$	AXPY-like
S4 :	$\tilde{\tau}^j := \| \tilde{s}^j \|_2$	DOT product + sqrt
S5 :	$x^{j+1} := x^j + \alpha^j p^j$	AXPY
	if$(\tilde{\tau}^j < \tilde{\tau}_{max})$**then break**	
S6 :	$s^j := M^{-1}\tilde{s}^j$	Apply precond.
S7 :	$\tilde{v}^j := As^j$	SpMV
S8 :	$\omega^j := < \tilde{v}^j, s^j > / < \tilde{v}^j, \tilde{v}^j >$	Two DOT products
S9 :	$x^{j+1} := x^j + \omega^j s^j$	AXPY
S10 :	$r^{j+1} := s^j - \omega^j \tilde{v}^j$	AXPY-like
S11 :	$\beta^j := \frac{<r^0, r^{j+1}>}{<r^0, r^j>} * \frac{\alpha^j}{\omega^j}$	DOT product
S12 :	$\tilde{p}^{j+1} := r^{j+1} + \beta^j(p^j - \omega^j v^j)$	Two AXPY-like
S13 :	$p^{j+1} := M^{-1}\tilde{p}^{j+1}$	Apply precond.
S14 :	$\tau^{j+1} := \| r^{j+1} \|_2$	DOT product + sqrt
	$j := j + 1$	

end while

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

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

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

Step	Operation	Communicat.
	$\sigma' := \sigma$	–
S1 :		
S1.1 :	$p \rightarrow e$	Allgatherv
S1.2 :	$v := Ae$	–
S2 :	$\alpha := \sigma / < \hat{r}, v >$	Allreduce
S3 :	$\tilde{s} := r - \alpha v$	–
S5 :	$x := x + \alpha p$	–
S6 :	$s := M^{-1}\tilde{s}$	–
S4 :	$\tau := sqrt(< s, s >)$	Allreduce
	if$(\tilde{\tau}^j < \tilde{\tau}_{max})$**then break**	–
S7 :		
S7.1 :	$s \rightarrow e$	Allgatherv
S7.2 :	$\tilde{v} := Ae$	–
S8 :	$\omega := < \tilde{v}, s > / < \tilde{v}, \tilde{v} >$	Allreduce(2)
S9 :	$x := x + \omega s$	–
S10 :	$r := s - \omega v$	–
S11 :	$\sigma := < \hat{r}, r >$	Allreduce(2)
S14 :	$\tau := sqrt(< r, r >)$	
S12 :	$\tilde{p} := r + \frac{\sigma}{\sigma'} * \frac{\alpha}{\omega}(p - \omega v)$	–
S13 :	$p := M^{-1}\tilde{p} + z$	–
	$j := j + 1$	

end while

Figure 3: Message-passing formulation of the PBiCGStab solver annotated with communication.

4.2. Message-passing Parallel BiCGStab Implementation

In this subsection we perform a communication analysis of a message-passing implementation of the BiCGStab solver. 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. We consider the following aspects in the analysis:

- The parallel platform consists of K processes (or MPI ranks), denoted as $P_1, P_2, \ldots, P_K$.
- The coefficient matrix A is partitioned into K blocks of rows ($A_1, A_2, \ldots, A_k$), where each process stores one row-block with the k-th *distribution block* $A_k \in \mathbb{R}^{p_k \times n}$ is stored in P_k, and $n = \sum_{k=1}^{K} p_k$.
- Vectors are partitioned and distributed in the same way as with the block-row distribution of A. For example, the residual vector r is partitioned as $r_1, r_2, \ldots, r_K$ and r_k is stored in P_k.
- The scalars $\alpha, \beta, \omega, \sigma, \tau, \tilde{\tau}$ are replicated on all K processes.

Taking into account these previous considerations, we analyze the different computational kernels (*S1–S14*) that compose the loop body of a single PBiCGStab iteration in Figure 2.

Sparse matrix-vector product (S1, S7): This kernel needs as input operands: the coefficient matrix A, which is distributed by blocks of rows, and the corresponding vector (p or s), which is partitioned and distributed using the same partitioning as A. For simplicity, we just explain below how *S1* is computed.

Prior to computing this kernel, we need to obtain a replicated copy of the distributed vector p in all processes, denoted as $p \rightarrow e$; 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: $P_k : v_k := A_k\, e$. This kernel thus requires assembling the distributed pieces of the vector p into a single vector e that is replicated in all processes (in MPI, for example via an `MPI_Allgatherv()`). The computation can then proceed in parallel, yielding the vector result v 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 (S2, S4, S8, S11, S14):* The next kernel in the loop body is the DOT product in the step *S2* between the distributed vectors $\hat{r}$ and r. Here, each process can compute concurrently a partial result $P_k : \rho_k :=< \hat{r}_k, r_k >$ 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 *S4*, *S8*, *S11* and *S14*, yielding a total of six process synchronizations (in MPI, via `MPI_Allreduce()`) since $\alpha, \beta, \omega, \sigma, \tau, \tilde{\tau}$, are globally-replicated.

AXPY*(-type) vector updates (S3, S5, S9, S10, S12):* The next kernel is the AXPY-like kernel in the step *S3*, which involves the distributed vectors $\tilde{s}, r, v$ and the globally-replicated scalar α. The operations in the steps *S5*, *S9*, *S10* 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: $P_k : \tilde{s}_k := r_k - \alpha\, v_k$.

Application of the preconditioner (S6, S13): The kernel in the step *S6* consists of applying the Jacobi preconditioner M, scaling the vector $\tilde{s}$ 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 $\tilde{s}$ elements: $P_k : s_k := M_k^{-1}\tilde{s}_k$. The same procedure can be applied on the step *S13* to scale the vector $\tilde{p}$, resulting in p.

We can easily reduce the number of synchronizations in the loop body of the PBiCGStab solver if we re-arrange the operations so that the step *S14* is pushed up next to the step *S11*. Thus, the simultaneous execution of these two reductions, and also the two reductions in step *S8*, decrease the number of process synchronizations from six to four per iteration. The reorganized algorithm and communications are summarized in Figure 3.

5. Experimental Results

In this section, we report a variety of numerical experiments to examine the convergence, 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.

5.1. Evaluation on the SuiteSparse matrices

We carried out the tests on a wide 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 2 lists a set of tested matrices with the their condition numbers $k(A)$, number of rows/ columns N, and 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 ReproBiCGStab in the table, we computed $b = Ad, d = (1, \ldots, 1)^T$ and then scaled b by $\frac{1}{\sqrt{N}}$. The (P)BiCGStab 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 2 also reports the number of required iterations to reach the stopping criterion as well the final true residual for BiCGStab and ReproBiCGStab; the later 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) as well as converge faster, for example for fs_760_3, orsreg_1, and rdb3200l matrices.

We select the fs_760_3 and rdb3200l matrices and illustrate their convergence history in Figure 4; the former is computed without the preconditioner. This figure reveals that: for fs_760_3 (left plot) the methods convergence smoothly at the initial iterations but then struggles to attain the last bits to satisfy the stopping criterion; for rdb3200l (right plot) the situation is inverse with the long 'warm up' phase until the 500th iteration that follows with the steep descend.

Matrix	Prec	$k(A)$	N	nnz	$\|r^0\|_2$	BiCGStab			ReproBiCGStab	
						iter1	iter8	$\|b - Ax^j\|_2$	iter	$\|b - Ax^j\|_2$
1138_bus	Jac	$8.6e+06$	1,138	4,064	$3.50e+06$	5	5	$1.72e-04$	5	$1.72e-04$
add32	Jac	$1.4e+02$	4,960	19,848	$6.38e-05$	36	36	$4.97e-09$	35	$7.12e-09$
bcsstk14	Jac	$1.3e+10$	1,806	63,454	$3.17e+18$	8	8	$1.11e+03$	8	$1.11e+03$
bcsstk18	Jac	$6.5e+01$	11,948	149,090	$5.29e+18$	7	7	$7.51e+02$	7	$7.51e+02$
bcsstk26	Jac	$1.7e+08$	1,922	30,336	$3.80e+19$	11	11	$5.62e+03$	11	$5.62e+03$
bcsstm25	-	$6.1e+09$	15,439	15,439	$4.76e+15$	956	828	$6.89e+01$	863	$6.81e+01$
bfw782a	Jac	$1.7e+03$	782	7,514	$1.09e-01$	267	-	$1.26e-07$	179	$1.70e-07$
cdde6	Jac	$1.8e+02$	961	4,681	$3.31e-01$	126	122	$2.26e-07$	120	$5.54e-07$
fs_760_3	-	$1.0e+20$	760	5,816	$2.71e+14$	741	790	$1.49e+01$	724	$1.53e+01$
jpwh_991	Jac	$1.4e+03$	991	6,027	$1.46e-01$	22	22	$3.62e-07$	2	$3.62e-07$
orsreg_1	Jac	$6.7e+03$	2,205	14,133	$2.34e+01$	225	228	$4.18e-06$	210	$4.68e-06$
pde2961	Jac	$6.4e+02$	2,961	14,585	$9.24e-02$	128	123	$5.28e-08$	125	$2.67e-07$
rdb3200l	Jac	$1.1e+03$	3,200	18,880	$9.92e+01$	641	605	$4.09e-06$	583	$3.17e-06$
s3dkq4m2	Jac	$1.9e+11$	90,449	4,427,725	$3.70e+05$	23	23	$7.26e-05$	23	$7.27e-05$
saylr4	Jac	$6.9e+06$	3,564	22,316	$9.44e+06$	10	10	$1.95e-03$	10	$7.26e-05$

Table 2: Convergence of the BiCGStab and ReproBiCGStab on a set of the SuiteSparse matrices with various condition numbers $k(A)$, number of rows/ columns N, and number of nonzeros nnz. The initial guess is all-zero $x^0 = 0$. The Jacobi preconditioner (Jac) is applied where applicable. 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 $\|b - Ax^j\|_2$.

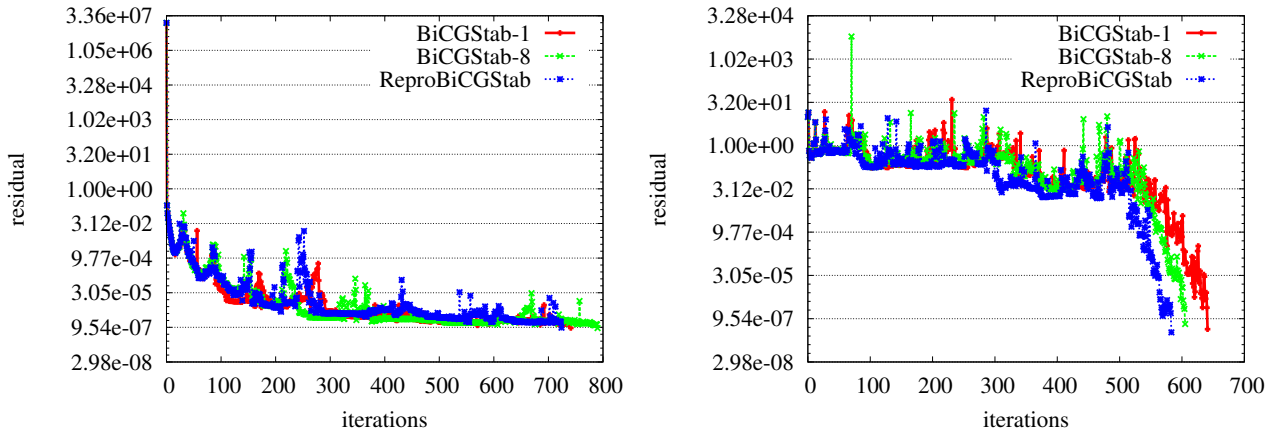

Figure 4: Residual history for the fs_760_3 (left, without preconditioner) and rdb3200l (right) SuiteSparse matrices, see Table 2 for details. The lines represent the norms of recursive residuals $\|r^j\|_2$ for BiCGStab on one and eight MPI processes, as well as ReproBiCGStab that stands for both ExBLAS and FPE versions.

5.2. Performance

Figures 5 and 6 demonstrate 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 orsreg_1 matrices. The figures report average execution time for the entire loop of the solver among up to five samples. We select the s3dkq4m2 matrix due to its 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 results with 10.4x, 12.8x, and 13.3x speed up on 16 MPI processes for the original, FPE, and ExBLAS variants, respectively. The reproducible variants demonstrate higher speedup due to extra floating-point operations. The right plot shows the overhead of the ExBLAS and FPE variants compared to the original variant is reduced to 2x and 2.4x, accordingly, on 28 MPI processes. The scalability results on the orsreg_1 matrix and others from Table 2 show 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 4x and 8x, respectively, faster on 16 MPI processes.

Note that the average execution time per loop for all studied 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 among these nodes.

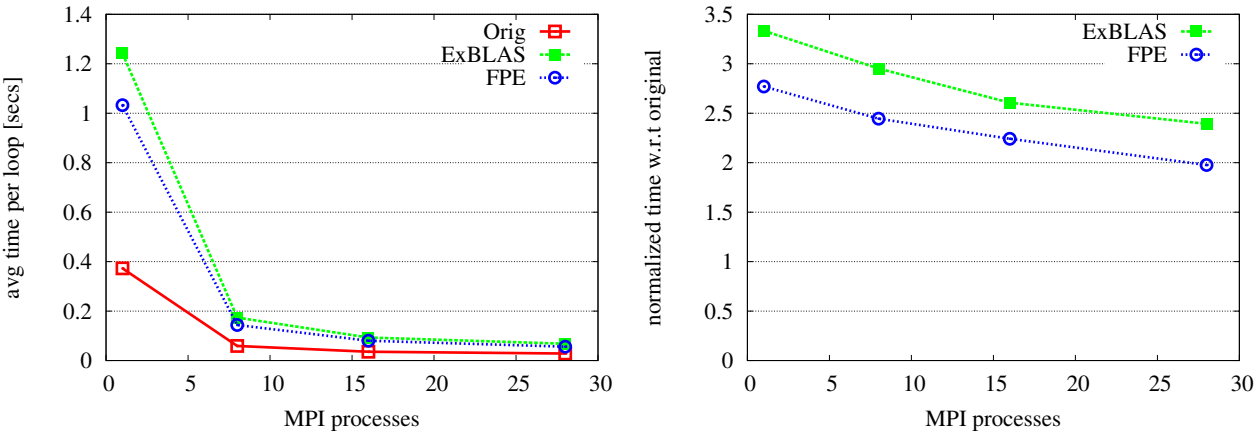

Figure 5: Strong scaling results of the original and reproducible PBiCGStab variants with the Jacobi preconditioner on one SkyLake node for the s3dkq4m2 matrix, see Table 2 for details. All methods were set to converge to the tolerance 10^{-6}.

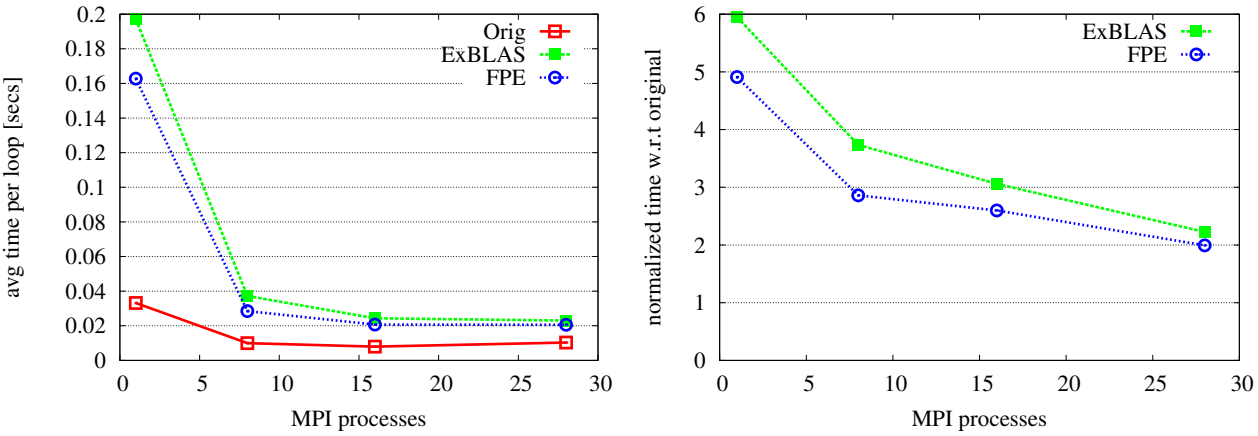

Figure 6: Strong scaling results of the original and reproducible PBiCGStab variants with the Jacobi preconditioner on one SkyLake node for the orsreg_1 matrix, see Table 2 for details. All methods were set to converge to the tolerance 10^{-6}.

5.3. Accuracy and Reproducibility

In addition, we derive a sequential version of the preconditioned BiCGStab as in Figure 2 that relies on the GNU Multiple Precision Floating-Point Reliably (MPFR) library [23] – a C library for multiple (arbitrary) precision floating-point computations on CPUs – as a highly accurate reference implementation. This implementation uses 2,048 bits of

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

Table 3 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 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 accuracy and reproducibility of both the ExBLAS and FPE variants of BiCGStab 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 seconds, while the ExBLAS and FPE variants take 3.95e-02 and 2.75e-02 seconds (8.57x and 12.32x faster), accordingly, on eight MPI processes. The original code shows the discrepancy from few digits on the initial iteration and up to the entire number as the count of required iterations differs from the reproducible and MPFR variants.

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.fbf5d3a5d1e49p-13**	0x1.be3698f1968cdp-13
...	...	...	...	...
208	0x1.355b0f18f5ac1p-20	**0x1.19edf2c932ab8p-18**	**0x1.b051edae310c7p-20**	0x1.355b0f18f5ac1p-20
209	0x1.114dc7c9b6d38p-20	**0x1.19b74e383f74ep-18**	**0x1.a18fc929018d4p-20**	0x1.114dc7c9b6d38p-20
210	0x1.03b1920a49a7ap-20	**0x1.19c846848f361p-18**	**0x1.c7eb5bbc198b1p-20**	0x1.03b1920a49a7ap-20

Table 3: Accuracy and reproducibility of the intermediate and final residual against MPFR for the orsreg_1 matrix, see Table 2.

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 [8].

Demmel and Nguyen implemented a family of algorithms – that originate from the works by Rump, Ogita, and Oishi [24, 21] – for reproducible summation in floating-point arithmetic [25, 11]. 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 [26].

The other approach to ensure reproducibility is called ExBLAS, which is initially proposed by Collange, Defour, Graillat, and Iakymchuk in [8]. ExBLAS is based on combining long accumulators and floating-point expansions in conjuction with error-free transformations. This approach is presented in Section 2.1. Collange et al. showed [8] 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 [27].

Mukunoki and Ogita presented their approach to implement reproducible BLAS, called OzBLAS [12], 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 [28] that follows the fork-join approach: the matrix and vector are split (each element is sliced) into sub-matrices and sub-vectors for secure products without overflows; then, the high-performance BLAS is called on each of these splits; finally, the results are merged back using, for instance, the NearSum algorithm. Currently, the OzBLAS library includes dot product, matrix-vector product (gemv), and matrix-matrix multiplication (gemm). These algorithmic variants and their implementations on GPUs and CPUs (only dot) reassure reproducibility of the BLAS kernels as well as make the accuracy tunable up-to correctly rounded results.

The proposed framework was implicitly used to derive the reproducible preconditioned Conjugate Gradient (PCG) variants with the flat MPI [19] and hybrid MPI plus OpenMP tasks [20]. 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 leightweight variants) on up to 768 cores of the MareNostrum4 cluster.

7. Conclusions and Future Work

Parallel Krylov subspace algorithms may exhibit the lack of reproducibility when implemented in parallel environments as the results in Table 3 confirm. In this work we proposed a general framework for the re-construction of reproducibility and re-assurance of accuracy in any Krylov subspace algorithm. Our framework is based on two steps: analysis of the underflying algorithm for the non-associativity and, hence, non-reproducibility issues; addressing these issues via algorithmic solutions and programmability hints. The algorithmic solutions are build arround the ExBLAS project, namely: ExBLAS that effectively combines long accumulator and FPEs as well as FPEs only for the leightweight version. The later can cope with the problems for up to 10^{50} of their dynamic range. The programmability effort was focused on: explicitly invoking `fma` instructions to avoid replacements by compilers as well as to postpone divisions to the moment they are actually needed. 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 the MPFR library, 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 matrices from the SuiteSparse Matrix Collection: the results show 2.5x and 2x performance overhead compared to the original version for the ExBLAS and lightweight variants, respectively. The code is available at `https://github.com/riakymch/ReproBiCGStab`.

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 [12]; 2) analyze the algorithm and make it reproducible by eliminating any uncertainties that may violate associativity such as reductions and use/ non-use of `fma`s and postponing divisions until actually needed.

Our future work aims to investigate a possibility of employing the reproducibility stategies as an alternative to the residual replacement strategy in the pipelined Krylov subspace solvers such as pipelined preconditioned BiCGStab (p-PBiCGStab) [29]. 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 couple with the attainable precision lose in p-PBiCGStab.

Acknowledgment

This research was partially supported by the European Union's Horizon 2020 research, innovation programme under the Marie Skłodowska-Curie grant agreement via the Robust project (No. 842528) as well as by the French National Agency for Research (ANR) via the InterFLOP project (No. ANR-20-CE46-0009). The research from Universitat Jaume I was funded by the project PID2020-113656RB-C21, supported by MCIN/AEI/10.13039/501100011033.

References

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

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

[3] Y. Saad, M. H. Schultz, Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems, Siam Journal on Scientific and Statistical Computing 7 (1986) 856–869.

[4] R. Fletcher, Conjugate gradient methods for indefinite systems, in: G. A. Watson (Ed.), Numerical Analysis, Springer Berlin Heidelberg, Berlin, Heidelberg, 1976, pp. 73–89.

[5] P. Sonneveld, CGS, A Fast Lanczos-Type Solver for Nonsymmetric Linear systems, SIAM Journal on Scientific and Statistical Computing 10 (1989) 36–52. doi:10.1137/0910004.
[6] H. A. van der Vorst, Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems, SIAM J. Sci. Stat. Comput. 13 (1992) 631–644. doi:10.1137/0913035.
[7] W. Gropp, T. Hoefler, R. Thakur, E. Lusk, Using advanced MPI: Modern features of the message-passing interface, MIT Press, 2014.
[8] S. Collange, D. Defour, S. Graillat, R. Iakymchuk, Numerical reproducibility for the parallel reduction on multi- and many-core architectures, ParCo 49 (2015) 83–97.
[9] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Comput. Surv. 23 (1991) 5–48. doi:10.1145/103162.103163.
[10] IEEE Computer Society, IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008, 2008.
[11] J. Demmel, H. D. Nguyen, Parallel Reproducible Summation, IEEE Transactions on Computers 64 (2015) 2060–2070.
[12] D. Mukunoki, T. Ogita, K. Ozaki, Accurate and reproducible blas routines with ozaki scheme for many-core architectures, in: Proc. International Conference on Parallel Processing and Applied Mathematics (PPAM2019), 2019. To appear.
[13] D. E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, volume 2, Addison-Wesley, 1969.
[14] T. Ogita, S. M. Rump, S. Oishi, Accurate sum and dot product, SIAM J. Sci. Comput 26 (2005) 1955–1988.
[15] U. Kulisch, V. Snyder, The Exact Dot Product As Basic Tool for Long Interval Arithmetic, Computing 91 (2011) 307–313.
[16] R. Iakymchuk, S. Collange, D. Defour, S. Graillat, ExBLAS: Reproducible and accurate BLAS library, in: Proceedings of the NRE2015 workshop held as part of SC15. Austin, TX, USA, November 15-20, 2015, 2015.
[17] R. W. Freund, N. M. Nachtigal, Qmr: A quasi-minimal residual method for non-hermitian linear systems, Numerische Mathematik 60 (1991) 315–339.
[18] R. W. Freund, Transpose-free quasi-minimal residual methods for non-hermitian linear systems, in: G. Golub, M. Luskin, A. Greenbaum (Eds.), Recent Advances in Iterative Methods, Springer New York, New York, NY, 1994, pp. 69–94.
[19] R. Iakymchuk, M. Barreda, M. Wiesenberger, J. I. Aliaga, E. S. Quintana-Orti, Reproducibility Strategies for Parallel Preconditioned Conjugate Gradient, JCAM 371 (2020) 112697. Available online 2 January 2020. DOI: 10.1016/j.cam.2019.112697.
[20] R. Iakymchuk, M. Barreda, S. Graillat, J. I. Aliaga, E. S. Quintana-Ortí, Reproducibility of Parallel Preconditioned Conjugate Gradient in Hybrid Programming Environments, IJHPCA 34 (2020) 502–518. Available OnlineFirst 17 June 2020. DOI: 10.1177/1094342020932650.
[21] S. M. Rump, T. Ogita, S. Oishi, Accurate floating-point summation part ii: Sign, k-fold faithful and rounding to nearest, SIAM J. Sci. Comput. 31 (2008) 1269–1302.
[22] M. Wiesenberger, L. Einkemmer, M. Held, A. Gutierrez-Milla, X. Xavier Saez, R. Iakymchuk, Reproducibility, accuracy and performance of the feltor code and library on parallel computer architectures, Computer Physics Communications 238 (2019) 145–156.
[23] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding, ACM TOMS 33 (2007) 13. doi:10.1145/1236463.1236468.
[24] S. M. Rump, T. Ogita, S. Oishi, Fast high precision summation, Nonlinear Theory and Its Applications, IEICE 1 (2010) 2–24.
[25] J. Demmel, H. D. Nguyen, Fast reproducible floating-point summation, in: Proceedings of ARITH-21, 2013, pp. 163–172.
[26] H. D. Nguyen, J. Demmel, Reproducible tall-skinny QR, in: Proceedings of ARITH-22, 2015, pp. 152–159. doi:10.1109/ARITH.2015.28.
[27] R. Iakymchuk, S. Graillat, D. Defour, E. S. Quintana-Orti, Hierarchical Approach for Deriving a Reproducible LU factorization, IJHPCA (2019) 1–13. To appear. HAL preprint: hal-01419813.
[28] K. Ozaki, T. Ogita, S. Oishi, S. M. Rump, Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications, Numerical Algorithms 59 (2012) 95–118.
[29] S. Cools, W. Vanroose, The communication-hiding pipelined BiCGstab method for the parallel solution of large unsymmetric linear systems, Parallel Computing 65 (2017) 1–20. doi:10.1016/j.parco.2017.04.005.