Comparison of Reproducible Parallel Preconditioned BiCGSTAB Algorithm Based on ExBLAS and ReproBLAS

Xiaojun Lei
leixiaojun19@gscaep.ac.cn
Graduate School of Chinese Academy of Engineering Physics
Haidian Qu, Beijing Shi, China

Tongxiang Gu
txgu@iapcm.ac.cn
Institute of Applied Physics and Computational Mathematics
Haidian Qu, Beijing Shi, China

Stef Graillat*
stef.graillat@lip6.fr
Sorbonne Université, CNRS, LIP6
Paris, F-75005, France

Xiaowen Xu
xwxu@iapcm.ac.cn
Institute of Applied Physics and Computational Mathematics
Haidian Qu, Beijing Shi, China

Jing Meng
mengpersist56@163.com
Taishan University
Taian, Shandong Province, China

ABSTRACT

Krylov subspace algorithms are important methods for solving linear systems. In order to efficiently solve large-scale linear systems, parallelism techniques are often applied. However, parallelism often enlarge the non-associativity of floating-point operations, which can lead to non-reproducibility of the computations. This paper compares the performance of the parallel preconditioned BiCGSTAB algorithm implemented with two different libraries (ExBLAS and ReproBLAS) that can ensure the reproducibility of computations. To address the effect of the compiler, we explicitly utilize the FMA instructions. Finally, numerical experiments show that based on two BLAS implementations, the BiCGSTAB algorithms are reproducible. By contrast, the BiCGSTAB algorithm based on ExBLAS is more accurate but more time-consuming than the one based on ReproBLAS.

CCS CONCEPTS

• **Mathematics of computing** → **Solvers**.

KEYWORDS

floating-point arithmetic, reproducibility, ExBLAS, ReproBLAS, Parallel Preconditioned BiCGSTAB

ACM Reference Format:
Xiaojun Lei, Tongxiang Gu, Stef Graillat, Xiaowen Xu, and Jing Meng. 2023. Comparison of Reproducible Parallel Preconditioned BiCGSTAB Algorithm Based on ExBLAS and ReproBLAS. In *International Conference on High Performance Computing in Asia-Pacific Region (HPC ASIA 2023), February 27-March 2, 2023, Singapore, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3578178.3578234

*Corresponding author

HPC ASIA 2023, February 27-March 2, 2023, Singapore, Singapore

ACM ISBN 978-1-4503-9805-3/23/02…$15.00
https://doi.org/10.1145/3578178.3578234

1 INTRODUCTION

In many scientific and engineering calculation fields, such as nuclear reactor simulation, radiation (magneto) hydrodynamics, radiation diffusion problems, oil and gas resource exploration, numerical weather prediction, etc. [10, 18, 32], differential equations are often used as mathematical models to describe problems. In order to simulate on a computer, it is necessary to discretize the differential equations to obtain a set of linear equations. Therefore, efficiently solving linear equations is the key to numerical simulation. For large-scale sparse matrices generated in actual problems, iterative methods based on Krylov subspace [25] are more advantageous because of the high computational complexity of direct solvers. However, the direct use of iterative methods may not converge or convergence can be very slow. The usual remedy is to use preconditioning techniques.

When the matrix is symmetric and positive definite, the Conjugate Gradient (CG) method [11] is one of the most effective Krylov subspace methods. When the matrix is unsymmetric, the BiConjugate Gradient Stabilized (BiCGSTAB) [28] and Generalized Minimal Residual (GMRES) methods [26] are the preferred Krylov subspace methods. GMRES is based on long recurrences, the amount of computation and storage increases linearly with the number of iterations. However, BiCGSTAB is based on short recurrences, the amount of computation remains the same for each iteration. Therefore, we take BiCGSTAB as an example to study.

When solving linear equations, numerical reproducibility failures may be risen in parallel computation. That is to say that the computing result for the same input data may vary from one run to another or even from one parallel machine to another. The non-reproducible behavior causes validation and debugging issues, and may even lead to deadlocks [22]. Moreover, this behavior will get worse on heterogeneous architectures, which combine together different programming environments that may follow different floating-point models and offer different intermediate precision or different operators [6, 7, 31]. The essence of the problem is that massively data instructions used in parallel environments do not guarantee the same execution order of floating-point operations. In order to guarantee numerical reproducibility, recent works [6, 7, 22, 29, 31] have tried to tackle the problem. For the above reasons, we may expect that the results of the sequential

and parallel implementations of Preconditioned BiCGSTAB (abbreviated as PBiCGSTAB) are identical, for instance, in the number of iterations, the intermediate and final residuals, as well as the output vector. However, this is often not the case in practice, due to different reduction trees - the Message Passing Interface (MPI) [13] provides 14 different implementations, data alignment, instructions used, order of the input data, etc. Each of these factors impacts the order of floating-point operations especially additions that are commutative but not associative. This can lead to non-reproducible results. Therefore, our aim is to implement reproducible parallel Preconditioned BiCGSTAB algorithm that can achieve the same result using the different number of processes on the same platform. For that, we implemented the reproducible parallel precondition BiCGSTAB solver using the two main BLAS libraries that can ensure reproducibility for basic operations; these are the ExBLAS and the ReproBLAS libraries, respectively.

The rest of the paper is organized as follows. Section 2 introduces the ExBLAS method and the ReproBLAS method. Section 3 introduces the PBiCGSTAB algorithms and describes in detail its MPI implementation. We present strategies for ensuring reproducibility of PBiCGSTAB in Section 4 and evaluate corresponding implementations in Section 5. Section 6 introduces some related work. Finally, Section 7 draws conclusions and proposes future work.

2 METHODOLOGY

2.1 ExBLAS

The ExBLAS project [15] 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 multilevel memory structures. ExBLAS combines together long accumulator and floating-point expansion (FPE) into algorithmic solutions. It has two features. On one side, this method aims to save each bit of information before finally rounding to the desired format. It is accurate and correctly rounded , so it is reproducible. On the other side, it is effectively tuned and implemented on various architectures, including conventional CPUs, Nvidia and AMD GPUs, and Intel Xeon Phi co-processors.

The corner stone of ExBLAS is reproducible parallel reduction. The ExBLAS parallel reduction relies upon FPEs with the `TwoSum` error-free transformation (EFT) [19] and long accumulators. In practice, the latter is invoked only once per overall summation that results in the little overhead (less than 8%) on accumulating large vectors. For details of the algorithm, please refer to [5]. In this article, we are concerned with the distributed dot product of two vectors. The dot product problem can be transformed into a summation problem using the `TwoProd` EFT [24]. The ExBLAS library also contains other routines, such as matrix vector multiplication, triangular solver and matrix matrix multiplication. The library is available at https://github.com/riakymch/exblas.

2.2 ReproBLAS

ReproBLAS aims at providing users with a set of (Parallel and Sequential) Basic Linear Algebra Subprograms that guarantee reproducibility regardless of the number of processors, of the data partitioning, of the way reductions are scheduled, and more generally of the order in which the sums are computed. ReproBLAS has three assumptions: (1) Floating-point numbers are binary and correspond to the IEEE 754-2008 Floating-Point Standard. (2) Floating-point operations are performed in round-to-nearest mode (ties may be broken arbitrarily). (3) Underflow occurs gradually (subnormal numbers must be supported) [22].

The corner stone of ReproBLAS [1] is reproducible summation, which is independent of the summation order. The reproducible summation algorithm can be found in [8, 9]. It is communication-optimal, in the sense that it does just one pass over the data in the sequential case, or one reduction operation in the parallel case, requiring an "accumulator" represented by just 6 floating-point words (more can be used if higher precision is desired). The arithmetic cost with a 6-word accumulator is $7n$ floating-point additions to sum n words, and (in IEEE double precision) the final error bound can be up to 10^{-8} times smaller than the error bound for conventional summation. Let us denote $T = \sum_{j=0}^{n-1} x_j$, and $\overline{T}$ the floating-point approximation to the summation obtained with ReproBLAS. The absolute error of the reproducible summation algorithm is $|T-\overline{T}| \leq n2^{W(1-K)} \max|x_j| + 7u|\sum_{j=0}^{n-1} x_j|$, where W is the width of the bins, K is the number of accumulators, u is the unit roundoff, the default configuration under double precision is $W = 40, K = 3$ [1]. The library is available at https://bebop.cs.berkeley.edu/reproblas/.

3 ALGORITHM(S)

3.1 Preconditioned BiCGSTAB

The BiCGSTAB algorithm was developed to solve nonsymmetric linear systems. The algorithmic description of the classical iterative PBiCGSTAB is presented in Algorithm 1 [2]. The loop body consists of sparse matrix-vector products (SpMV), dot products, AXPY (-like) operations, the preconditioner application, and a few scalar operations. For simplicity, in our implementation of the PBiCGSTAB method, we integrate the Jacobi preconditioner, which is composed of the diagonal elements of the matrix. As a consequence, the use of Jacobi preconditioner requires two vectors to be multiplied element by element.

3.2 Message-Passing Parallel PBiCGSTAB

In our parallel Preconditioned BiCGSTAB algorithm, we use L processes. The matrix A is partitioned into L blocks of rows $(A_1, A_2, \ldots, A_i, \ldots, A_L)$, where each process stores one row-block. Vectors are partitioned and distributed in the same way as with the block-row distribution of A. In lines 13 and 18 of Algorithm 1, it needs a SpMV operation, where we call *MPI_Allgatherv*() to synthesize the local vector of each process into a full operand vector. At lines 4, 14, 16, 19, and 22 of Algorithm 1, the distributed dot product needs to be computed, for non-reproducible parallel Preconditioned BiCGSTAB, we call *MPI_Allreduce*. We have implemented a reproducible distributed dot product, see Listing 1, which is different from the ordinary dot product. This is also the main difference between the reproducible parallel Preconditioned BiCGSTAB and the non-reproducible parallel Preconditioned BiCGSTAB. For details on reproducible distributed dot products, please see Section 4.

Algorithm 1: The Preconditioned BiConjugate Gradient Stabilized Method (PBiCGSTAB) [2].

Input: Coefficient matrix A, right-hand side vector b, initial guess $x^{(0)}$, convergence threshold ϵ
Output: Approximate solution $x^{(i)}$

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
Choose $\tilde{r}$ (for example, $\tilde{r} = r^{(0)}$)
for $i = 1, 2, \ldots$
  $\rho_{i-1} = \tilde{r}^T r^{(i-1)}$
  if $\rho_{i-1} = 0$ method fails
  if $i = 1$
    $p^{(i)} = r^{(i-1)}$
  else
    $\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
    $p^{(i)} = r^{(i-1)} + \beta_{i-1}(p^{(i-1)} - \omega_{i-1} v^{(i-1)})$
  endif
  solve $M\hat{p} = p^{(i)}$
  $v^{(i)} = A\hat{p}$
  $\alpha_i = \rho_{i-1}/\tilde{r}^T v^{(i)}$
  $s = r^{(i-1)} - \alpha_i v^{(i)}$
  if $\|s\|_2 \leq \epsilon$: set $x^{(i)} = v^{(i-1)} + \alpha_i \hat{p}$ and stop
  solve $M\hat{s} = s$
  $t = A\hat{s}$
  $\omega^{(i)} = t^T s / t^T t$
  $x^{(i)} = x^{(i-1)} + \alpha_i \hat{p} + \omega^{(i)} \hat{s}$
  $r^{(i)} = s - \omega^{(i)} t$
  if $\|r^{(i)}\|_2 \leq \epsilon$: break
  for continuation it is necessary that $\omega^{(i)} \neq 0$
end

4 STRATEGIES FOR REPRODUCIBILITY

In [16], the article implements a reproducible parallel Preconditioned BiCGSTAB algorithm based on ExBLAS, also implemented with its lighter FPE-based version. In this paper, we implement a reproducible parallel Preconditioned BiCGSTAB algorithm based on ReproBLAS, and compare the performance of the solvers based on ExBLAS and ReproBLAS. In this section, we state our reproducible strategy for the parallel Preconditioned BiCGSTAB algorithm. We show the sources of non-deterministic computations in the PBiCGSTAB solver, and present our mitigation strategies.

Dot products: The main issue of non-determinism emerges from dot products and, thus, parallel reductions such as *MPI_Allreduce*(). We use the ExBLAS method to provide a reproducible and correctly rounded dot product, and use the ReproBLAS method to provide a reproducible dot product. In [14], the authors provided a pseudocode for implementing distributed dot product with ExBLAS. Listing 1 provides pseudocodes for our implementation of the distributed dot product using the ReproBLAS.

Sparse matrix-vector product: Each process has a local A_i, and the complete vector is obtained by using *MPI_Allgatherv*(), then each process computes a SpMV, since the computations are carried locally and sequentially, they are deterministic and, thus, reproducible.

```
double_binned *isum = NULL;
double_binned *local_isum = binned_dballoc(K
    );
binnedBLAS_dbddot(K, n, r, 1, r, 1,
    local_isum);
MPI_Reduce(local_isum, isum, 1,
    binnedMPI_DOUBLE_BINNED(K),
binnedMPI_DBDBADD(K), 0, MPI_COMM_WORLD);
if(myId == 0){
    tol = binned_ddbconv(K, isum);
}
MPI_Bcast(&tol, 1, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
```

Listing 1: Reproducible Allreduce with ReproBLAS.

AXPY(-type) vector updates: For computing the axpy(-type) vector updates, the local vector is updated independently in each processor, they are deterministic and, thus, reproducible.

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

5 NUMERICAL RESULTS

5.1 Setup

The following experiments are performed on Sugon HPC cluster with 172 compute nodes, consisting of two 12-core Intel E5-2680 v3 processors each (24 cores per node), 64 GB DDR4 ECC memory 2133MHz. The system uses Intel Omni-Path high-speed computing network. The OS used by the cluster is Redhat7.2. The compiler is intel-composer_xe_2017.0.098. The MPI library used for these experiments is mpich-3.2-gnu.

The matrix used for experiments is a 3D Possion matrix of dimension $n = 159^3$, which is a 27-point stencil(rows corresponding to interior nodes have 27 nonzeros). The mathematical expression of the matrix can be found in [27]. We use a scaling factor to scale the first row and first column of the matrix. We also tested the matrix of SuiteSparse (http://sparse.tamu.edu/), and the experimental results are shown in the appendix.

The right-hand side vector b in the iterative solvers was always initialized to the product $A(1, 1, \ldots, 1)^T$, and the PBiCGSTAB iteration was started with the initial guess $x_0 = 0$. The stopping criterion is $\|r^j\|_2 \leq 10^{-8}$, where j is the number of iterations.

To reproduce the numerical experiments of this study, the code is openly available through GitHub at https://github.com/programmer-lxj/Reproducible-BiCGSTAB-is-based-on-ExBLAS-and-ReproBLAS.

5.2 Accuracy and Reproducibility Evaluation

In this section, we report the results of the accuracy and reproducibility evaluation. Additionally, we derive a sequential version of the code that relies on the GNU Multiple Precision Floating-Point Reliably (MPFR) library [12] - a C library for multiple (arbitrary)

precision floating-point computations on CPUs - as a highly accurate reference implementation. This implementation uses 2048 bits of accuracy for computing dot product (192 bits for internal product of two floating-point numbers) and performs correct rounding of the computed result to double precision.

We analyzed the reproducibility of the reproducible parallel PBiCGSTAB. Hereafter, we will call them ExBLAS and ReproBLAS for short. In ReproBLAS, there is a parameter K controlling accuracy, when there is no special description, we take the default $K = 3$. With double as the working precision, when $W = 40, K = 3$, $|T - \overline{T}| \leq n2^{40\times(-2)} \max|x_j| + 7 \times 2^{-53}|\sum_{j=0}^{n-1} x_j|$. When $K \geq 3$, $|T - \overline{T}| \approx 7 \times 2^{-53}|\sum_{j=0}^{n-1} x_j|$, therefore choose the smallest K that can achieve the highest summation accuracy. In Table 1, the matrix size is $n = 4,019,679$ and the condition number is 10^{12}. Table 1 shows the 2-norm of the intermediate and final residual of the PBiCGSTAB solver in each iteration, i.e. $\|r^j\|_2$. Table 2 shows the 2-norm of the intermediate and final residual of the ReproBLAS in each iteration when $K = 2$ and $K = 3$. We used one node on the Sugon cluster with 24 processes each pinned to one core. We present only few iterations, however the difference is present on all iterations. The ExBLAS implementation delivers both accurate and reproducible results that are identical with the MPFR library. The ReproBLAS implementation delivers reproducible results. Furthermore, we have also computed the direct error (see Table 3). Table 3 shows the infinity norm of the approximate and exact solutions of the PBiCGSTAB solver in each iteration, i.e. $\|x^j - x^*\|_\infty$. Note that the result on ExBLAS is identical to MPFR. Table 4 shows the infinity norm of the approximate and exact solutions of the ReproBLAS in each iteration when $K = 2$ and $K = 3$. Figure 1 represent the norms of residuals $\|r^j\|_2$ for PBiCGSTAB on 24 MPI processes.

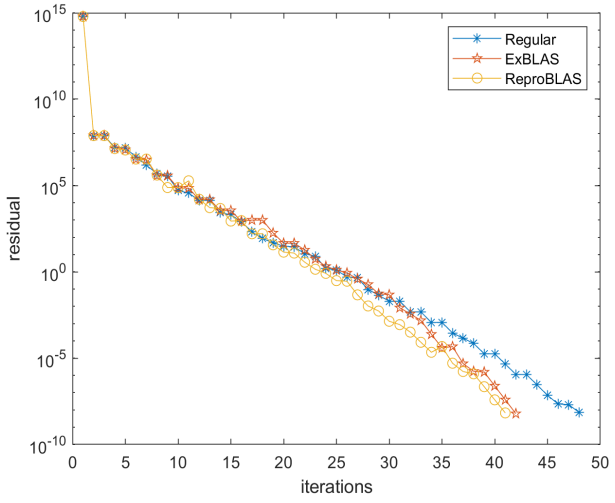

Figure 1: Residual history.

5.3 Performance Evaluation

Firstly, the performance of regular, ExBLAS-based and Reproblas-based dot product are analyzed. Hereafter, we will call them ExBLASdot and ReproBLASdot for short. We use the same data generation method as the ReproBLAS library [14], i.e. $a_i = \sin(2.0 * \pi * (i/n - 0.5))$, $b_i = \sin(2.0 * \pi * (i/n - 0.5))$, $n = 16200000$. Table 5 reports the total execution time of different dot products (averaged for 5 different executions). Figure 2 reports the total execution time (averaged for 5 different executions) of the reproducible dot product for this cluster normalized with respect to the execution time of the regular MPI version, when we vary the number of cores. From table 5 and figure 2, we can see that ReproBLASdot is faster than ExBLASdot product.

Then, we analyzed the performance of the reproducible parallel PBiCGSTAB. Our experiments evaluate the strong scaling of this reproducible implementation compared against the regular (non-reproducible) version of PBiCGSTAB using MPI. We also analyze the performance of the three implementations in the aforementioned cluster. Specifically, in order to assess the strong scalability, we fix the matrix size to $n = 16,003,008 (252^3)$ and increase the number of cores. Table 6 reports the total execution time (averaged for 5 different executions) of the different MPI PBiCGSTAB solvers on this platform, varying the number of cores (from 48 to 144 in Sugon) as we maintain the problem size, the number of iterations in parentheses. Figure 3 reports the total execution time (averaged for 5 different executions) of the reproducible MPI PBiCGSTAB solvers for this cluster normalized with respect to the execution time of the regular MPI version, when we vary the number of cores. From table 6 and figure 3, we can see that ReproBLAS is faster than ExBLAS. We also found that ReproBLAS is faster than regular MPI version when using 72, 96, and 144 processes, because ReproBLAS has fewer iterations, so the total time is shorter when the same accuracy conditions are met. It can be seen from this example that although the dot product uses a reproducible dot product, each iteration pays a certain amount of overhead, but it may reduce the total number of iterations and reduce the total overhead. Table 7 reports the average time of each iteration (averaged for 5 different executions) of the different MPI PBiCGSTAB solvers on this platform, varying the number of cores (from 48 to 144 in Sugon) as we maintain the problem size. Figure 4 reports the average time of each iteration (averaged for 5 different executions) of the reproducible MPI PBiCGSTAB solvers for this cluster normalized with respect to the execution time of the regular MPI version, when we vary the number of cores. We fixed the total number of iterations to 60, figure 5 reports the total execution time (averaged for 5 different executions) of the reproducible MPI PBiCGSTAB solvers for this cluster normalized with respect to the execution time of the regular MPI version.

From our experiments, we can observe the following facts:

- The dot product based on ReproBLAS is faster than that based on ExBLAS.
- Reproducible parallel PBiCGSTAB based on ExBLAS library can get the same results as MPFR. It is reproducible and accurate.
- Reproducible parallel PBiCGSTAB based on ReproBLAS library has the same result no matter how many processes are used, but it cannot get the same result as MPFR.
- When $K = 2$, the reproducible parallel PBiCGSTAB based on the ReproBLAS library is also reproducible, but the result is different from K greater than or equal to 3.

Table 1: Accuracy and reproducibility comparison on the intermediate and final residual against MPFR for a matrix with condition number of 10^{12}. The matrix is generated following the procedure from Section 5.1 with $n = 4,019,679(159^3)$.

Residual					
Iteration	MPFR	Original 1 proc	Original 48 proc	ExBLAS 48 proc	ReproBLAS 48 proc($K = 3$)
0	6.199998000000062e+14	6.199998000000064e+14	6.199998000000064e+14	6.199998000000062e+14	6.199998000000062e+14
2	7.773390285775344e+07	7.773396235537358e+07	7.773396235219534e+07	7.773390285775344e+07	7.773396235223217e+07
18	1.800391538018418e+02	**9.920664169207433**e+02	**7.848983492248201**e+02	1.800391538018418e+02	**3.647699220519532e+01**
19	4.527368042985061e+01	**9.920511721210302e+02**	**1.708520638885489e+02**	4.527368042985061e+01	**1.377536701580542**e+01
...	...	...	...	...	...
38	1.572043303241533e-06	**3.155149206702169e-03**	**2.941521670086655e-04**	1.572043303241533e-06	**2.215388966485101e-07**
39	2.453500079721513e-07	**1.269826897510669e-03**	**2.783864941272381e-04**	2.453500079721513e-07	**3.834134369067694e-08**
40	3.854451111920641e-08	**1.237807074881981e-03**	**4.132770031308131e-05**	3.854451111920641e-08	**6.701333941850470e-09**
41	6.068369560766913e-09	**4.530740507879870e-05**	**4.060713358832429e-05**	6.068369560766913e-09	
49		**7.217500962630385e-08**	**2.373922675763727e-08**		
50		**1.275390456398120e-08**	**3.914596612172984e-09**		
51		**9.575558932042771e-09**			

Table 2: Comparison of the reproducibility of the intermediate and final residuals of the matrix with a condition number of 10^{12}. The matrix is generated following the procedure from Section 5.1 with $n = 4,019,679(159^3)$.

Residual				
Iteration	ReproBLAS 24 proc($K = 2$)	ReproBLAS 48 proc($K = 2$)	ReproBLAS 24 proc($K = 3$)	ReproBLAS 48 proc($K = 3$)
0	6.199998000000062e+14	6.199998000000062e+14	6.199998000000062e+14	6.199998000000062e+14
2	7.773396235223238e+07	7.773396235223238e+07	7.773396235223217e+07	7.773396235223217e+07
18	5.598082147961621e+02	5.598082147961621e+02	3.647699220519532e+01	3.647699220519532e+01
19	1.401936884527657e+02	1.401936884527657e+02	1.377536701580542e+01	1.377536701580542e+01
...	...	...	...	...
38	6.514290840710756e-05	6.514290840710756e-05	2.215388966485101e-07	2.215388966485101e-07
39	1.434720487921011e-05	1.434720487921011e-05	3.834134369067694e-08	3.834134369067694e-08
40	1.187160733888966e-05	1.187160733888966e-05	6.701333941850470e-09	6.701333941850470e-09
41	2.536931803781759e-06	2.536931803781759e-06		
42	2.193618822360498e-06	2.193618822360498e-06		
43	2.815475613230721e-07	2.815475613230721e-07		
44	3.861719136834829e-08	3.861719136834829e-08		
45	7.197952430468738e-09	7.197952430468738e-09		

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

Direct error					
Iteration	MPFR	Original 1 proc	Origina 48 proc	ExBLAS 48 proc	ReproBLAS 48 proc($K = 3$)
0	1.000000000000000e+00	1.000000000000000e+00	1.000000000000000e+00	1.000000000000000e+00	1.000000000000000e+00
2	9.468739489168004e+04	9.468739489167897e+04	9.468739489167897e+04	9.468739489168004e+04	9.468739489167897e+04
18	3.759048477414416e-01	**2.473965872164404e+00**	**4.032131048428883**e-01	3.759048477414416e-01	**5.105001804315423e-02**
19	5.816123121107641e-02	**3.747532604204130e-01**	**4.049929311272970e-01**	5.816123121107641e-02	**9.454486314514732e-03**
...	...	...	...	...	...
38	8.071978641055466e-10	**4.484255959891215e-07**	1.**59412679834325**4e-07	8.071978641055466e-10	**1.228213974968639e-09**
39	8.072296164840509e-10	**4.481986183302311e-07**	**3.417345129097527e-08**	8.072296164840509e-10	**1.228311674594806e-09**
40	8.076090907138678e-10	**1.625782982683788e-07**	**3.417765470636880e-08**	8.076090907138678e-10	**1.228331658609250e-09**
41	8.076523894118282e-10	**1.630189810919447e-07**	**3.606841825209983e-09**	8.076523894118282e-10	
49		**7.744984653612619e-10**	**8.554250641168437e-10**		
50		**7.744849206403615e-10**	**8.554250641168437e-10**		
51		**7.745686314564182e-10**			

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

Direct error				
Iteration	ReproBLAS 24 proc($K = 2$)	ReproBLAS 48 proc($K = 2$)	ReproBLAS 24 proc($K = 3$)	ReproBLAS 48 proc($K = 3$)
0	1.000000000000000e+00	1.000000000000000e+00	1.000000000000000e+00	1.000000000000000e+00
2	9.468739489167897e+04	9.468739489167897e+04	9.468739489167897e+04	9.468739489167897e+04
18	2.044747069809166e-01	2.044747069809166e-01	5.105001804315423e-02	5.105001804315423e-02
19	2.044624037738916e-01	2.044624037738916e-01	9.454486314514732e-03	9.454486314514732e-03
...	...	...	...	...
38	3.604734521989172e-08	3.604734521989172e-08	1.228213974968639e-09	1.228213974968639e-09
39	6.525545215296802e-09	6.525545215296802e-09	1.228311674594806e-09	1.228311674594806e-09
40	1.458647203023133e-09	1.458647203023133e-09	1.228331658609250e-09	1.228331658609250e-09
41	1.229433221894283e-09	1.229433221894283e-09		
42	1.230030521881531e-09	1.230030521881531e-09		
43	1.230010315822483e-09	1.230010315822483e-09		
44	1.230061830170825e-09	1.230061830170825e-09		
45	1.230070045821208e-09	1.230070045821208e-09		

- For ill-conditioned matrices, using a high-precision library to calculate the dot product may reduce the number of iterations.
- The reproducible parallel PBiCGSTAB based on the ReproBLAS library is faster than the reproducible parallel PBiCGSTAB based on the ExBLAS library.

6 RELATED WORK

Accuracy and reproducibility issues have different motivations and natures [21], but they are due to the same reason, which is caused by rounding errors. Although achieving reproducibility will not improve accuracy, increasing accuracy will help reduce the severity of reproducible problems. Here, we briefly introduce the

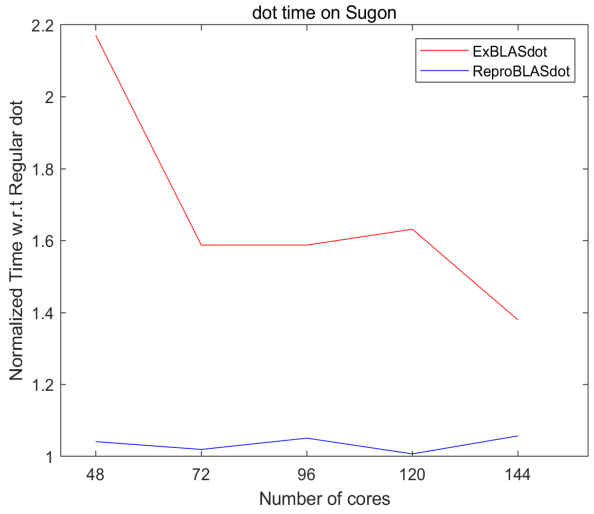

Figure 2: Analysis of the reproducible version of the MPI dot product, when the time is normalized with respect to the regular non-deterministic MPI version.

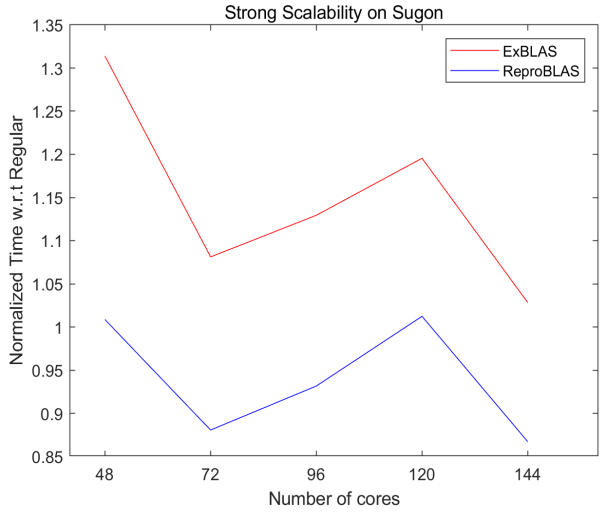

Figure 3: Analysis of the strong scalability of the reproducible version of the MPI PBiCGSTAB, when the time is normalized with respect to the regular non-deterministic MPI version.

Table 5: Different MPI implementations of dot product on Sugon, where one process is pinned to one core.

Execution time in seconds			
Number of cores	Regular dot	ExBLASdot	ReproBLASdot
48	0.1846646	0.4009640	0.1922632
72	0.2446498	0.3884392	0.2494264
96	0.1876278	0.2979166	0.1971776
120	0.1635804	0.2669234	0.1647500
144	0.1847008	0.2548258	0.1952600

Table 6: Strong scalability of different MPI implementations of the PBiCGSTAB method on Sugon, where one process is pinned to one core.

Execution time in seconds			
Number of cores	Regular	ExBLAS	ReproBLAS
48	$1.9791063e+02(43)$	$2.6004507e+02(43)$	$1.9961982e+02(43)$
72	$1.9511409e+02(50)$	$2.1095089e+02(43)$	$1.7181676e+02(43)$
96	$1.6530460e+02(47)$	$1.8669468e+02(43)$	$1.5399638e+02(43)$
120	$1.4927750e+02(43)$	$1.7845440e+02(43)$	$1.5111953e+02(43)$
144	$1.6825363e+02(50)$	$1.7301399e+02(43)$	$1.4589754e+02(43)$

Table 7: Time for one iteration of different MPI implementations of the PBiCGSTAB method on Sugon, where one process is pinned to one core.

Execution time in seconds			
Number of cores	Regular	ExBLAS	ReproBLAS
48	4.6025729	6.0475598	4.6423215
72	3.9022818	4.9058347	3.9957387
96	3.5171191	4.3417369	3.5813112
120	3.4715697	4.1501025	3.5144077
144	3.3624477	4.0235813	3.3929661

reproducible work of hardware manufacturers, several BLAS, and Krylov subspace methods.

Intel has introduced a version of their Math Kernel Library (MKL) that supports reproducibility under certain restrictive conditions

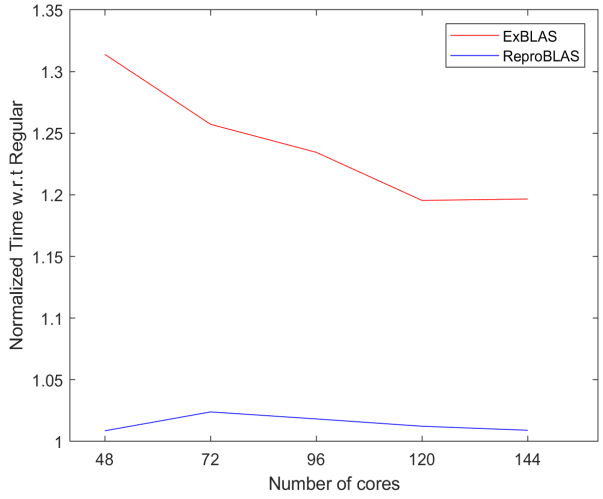

Figure 4: The time for one iteration of the reproducible version of the MPI PBiCGSTAB, which is normalized to the regular non-deterministic MPI version.

[30]. NVIDIA's cuBLAS routines are, by default, reproducible under the same conditions [23].

Collange, Defour, Graillat and Iakymchuk proposed ExBLAS [15]. ExBLAS is based on combining long accumulators and floating-point expansions in conjunction with error-free transformations. Demmel and Nguyen proposed a series of reproducible summation algorithms [8, 9]. Ahrens, Nguyen, and Demmel extended their concept to few other reproducible BLAS routines, distributed as the ReproBLAS library [1]. Mukunoki and Ogita presented their approach to implement reproducible BLAS, called OzBLAS [20], with

Table 8: Accuracy and reproducibility comparison on the intermediate and final residual against MPFR for the orsreg_1 matrix.

Residual					
Iteration	MPFR	Original 1 proc	Original 12 proc	ExBLAS 12 proc	ReproBLAS 12 proc($K = 3$)
0	7.206090395833573e+05	7.206090395833547e+05	7.206090395833575e+05	7.206090395833573e+05	7.206090395833574e+05
2	2.692606300836220e+04	2.692606300836336e+04	2.692606300836246e+04	2.692606300836220e+04	2.692606300836220e+04
10	1.411246260813212e+01	1.411246260496860e+01	1.411246261755428e+01	1.411246260813212e+01	1.411246260785536e+01
11	3.876226316868169e+00	3.876226319233361e+00	3.876226309633867e+00	3.876226316868169e+00	3.876226317074641e+00
...	...	...	...	...	...
35	2.314863739651161e-08	4.860845660911387e-08	6.527227051058278e-07	2.314863739651161e-08	6.133768171231175e-07
36	1.953972392467954e-08	2.497853962282123e-08	1.022778424195564e-07	1.953972392467954e-08	3.838906711474632e-07
37	1.936298845265811e-08	1.315490770381772e-08	4.408541846856823e-08	1.936298845265811e-08	6.455780118417509e-08
38	5.248376788902260e-09	5.852804371830325e-09	3.093522687975534e-08	5.248376788902260e-09	2.972253801317432e-08
39			1.054132837636201e-08		2.588257736412002e-08
40			1.989920059665807e-08		1.726414308163342e-08
41			7.518172991635087e-09		1.839832879922502e-08
42					4.63749298904597 4e-09

Table 9: Comparison of the reproducibility of the intermediate and final residuals for the orsreg_1 matrix.

Residual				
Iteration	ReproBLAS 1 proc($K = 2$)	ReproBLAS 12 proc($K = 2$)	ReproBLAS 1 proc($K = 3$)	ReproBLAS 12 proc($K = 3$)
0	7.206090395833567e+05	7.206090395833567e+05	7.206090395833574e+05	7.206090395833574e+05
2	2.692606300836070e+04	2.692606300836070e+04	2.692606300836220e+04	2.692606300836220e+04
10	1.411246192435790e+01	1.411246192435790e+01	1.411246260785536e+01	1.411246260785536e+01
11	3.876226844033272e+00	3.876226844033272e+00	3.876226317074641e+00	3.876226317074641e+00
...	...	...	...	...
35	1.136396751498361e-06	1.136396751498361e-06	6.133768171231175e-07	6.133768171231175e-07
36	3.021784694970428e-07	3.021784694970428e-07	3.838906711474632e-07	3.838906711474632e-07
37	3.063590261531213e-08	3.063590261531213e-08	6.455780118417509e-08	6.455780118417509e-08
38	1.426609689040662e-08	1.426609689040662e-08	2.972253801317432e-08	2.972253801317432e-08
39	1.058915049056299e-08	1.058915049056299e-08	2.588257736412002e-08	2.588257736412002e-08
40	7.559375007802276e-09	7.559375007802276e-09	1.726414308163342e-08	1.726414308163342e-08
41			1.839832879922502e-08	1.839832879922502e-08
42			4.637492989045974e-09	4.637492989045974e-09

Table 10: Direct error comparison against MPFR for the orsreg_1 matrix.

Direct error					
Iteration	MPFR	Original 1 proc	Origina 12 proc	ExBLAS 12 proc	ReproBLAS 12 proc($K = 3$)
0	1.000000000000000e+00	1.000000000000000e+00	1.000000000000000e+00	1.000000000000000e+00	1.000000000000000e+00
2	1.471714605212971e+00	1.471714605213005e+00	1.471714605212973e+00	1.471714605212971e+00	1.471714605212971e+00
10	6.978257384979702e-04	6.978257386137665e-04	6.978257381344832e-04	6.978257384979702e-04	6.978257385079623e-04
11	7.101831311118190e-05	7.101831310007967e-05	7.101831314271223e-05	7.101831311118190e-05	7.101831311029372e-05
...	...	...	...	...	...
35	4.555467114641942e-12	3.179123631014136e-12	1.423583473325607e-11	4.555467114641942e-12	7.357559006493375e-12
36	2.924327446862662e-13	3.659295089164516e-13	4.185429780534378e-12	2.924327446862662e-13	5.562661442581884e-12
37	2.622346784164620e-13	1.835198659705384e-13	9.154899061059041e-13	2.622346784164620e-13	4.801270492293952e-12
38	2.333688797762079e-13	1.508793090465588e-13	4.084510507595951e-13	2.333688797762079e-13	3.195221864871201e-13
39			2.862154957483654e-13		2.675637489346627e-13
40			1.083577672034153e-13		2.515765373800605e-13
41			2.562394740834861e-13		2.566835632933362e-13
42					1.917355163527645e-13

Table 11: Direct error comparison for the orsreg_1 matrix.

Direct error				
Iteration	ReproBLAS 1 proc($K = 2$)	ReproBLAS 12 proc($K = 2$)	ReproBLAS 1 proc($K = 3$)	ReproBLAS 12 proc($K = 3$)
0	1.000000000000000e+00	1.000000000000000e+00	1.000000000000000e+00	1.000000000000000e+00
2	1.471714605212970e+00	1.471714605212970e+00	1.471714605212971e+00	1.471714605212971e+00
10	6.978257649087327e-04	6.978257649087327e-04	6.978257385079623e-04	6.978257385079623e-04
11	7.101831080902343e-05	7.101831080902343e-05	7.101831311029372e-05	7.101831311029372e-05
...	...	...	...	...
35	4.534927988686377e-12	4.534927988686377e-12	7.357559006493375e-12	7.357559006493375e-12
36	7.233880161550132e-12	7.233880161550132e-12	5.562661442581884e-12	5.562661442581884e-12
37	1.978417429882029e-12	1.978417429882029e-12	4.801270492293952e-12	4.801270492293952e-12
38	3.190780972772700e-13	3.190780972772700e-13	3.195221864871201e-13	3.195221864871201e-13
39	1.894040480010517e-13	1.894040480010517e-13	2.675637489346627e-13	2.675637489346627e-13
40	1.552091788425969e-13	1.552091788425969e-13	2.515765373800605e-13	2.515765373800605e-13
41			2.566835632933362e-13	2.566835632933362e-13
42			1.917355163527645e-13	1.917355163527645e-13

tunable accuracy. Chohra introduced RARE-BLAS (Reproducible, Accurately Rounded and Efficient BLAS) that benefits from recent accurate and efficient summation algorithms [3, 4].

Regarding the reproducible Krylov subspace method, Iakymchuk et al. realized the reproducibility of pure MPI parallel Preconditioned CG on the CPU based on ExBLAS, use Jacobi preconditioner [14]. Further, they realized the reproducibility of the pure MPI parallel Preconditioned BiCGSTAB algorithm on the CPU based on ExBLAS, use Jacobi preconditioner [16]. Furthermore, they have also achieved reproducibility in the MPI+OpenMP environment [17]. Mukunoki et al. realizes the reproducibility of the CG solver on the CPU and GPU [21].

7 CONCLUSIONS AND FUTURE WORK

We emphasized the reproducibility problem of the parallel Preconditioned BiCGSTAB algorithm. We at first analyzed the MPI implementation of the PBiCGSTAB method and identified two major sources of non-deterministic behavior, namely dot products and

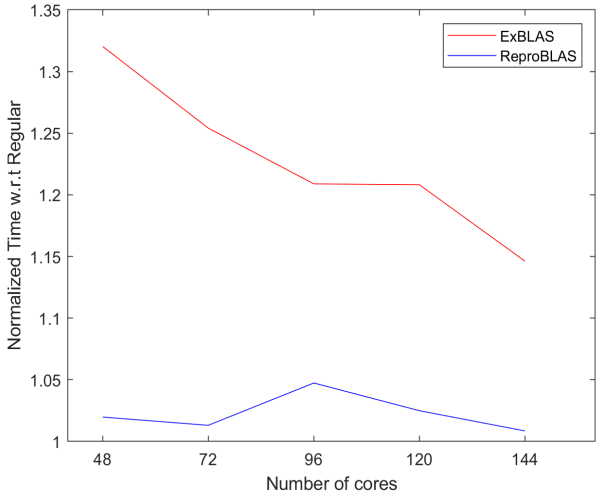

Figure 5: Total time of the reproducible version of the MPI PBiCGSTAB, which is normalized to the regular non-deterministic MPI version.

compiler optimization. The latter may change the order of operations or replace some of them in favor of the fused multiply-add (fma) operation. To tackle compiler interference in computations, we reconstruct computations as well as explicitly invoke fma instructions. For reproducible and distributed dot product, we use two methods, which are based on ExBLAS and based on ReproBLAS. Both the reproducible parallel preconditioned BiCGSTAB method realize the reproducibility of the number of iterations, intermediate and final residuals and the final output vector. The ExBLAS method is more accurate than the ReproBLAS method, but the ReproBLAS method is faster. Since the dot product accounts for a relatively low percentage of computation time in the parallel preconditioned BiCGSTAB, when we fix the total number of iterations to 60, the average time cost of reproducible parallel preconditioned BiCGSTAB method based on ExBLAS is 1.23 times that of the non-reproducible parallel preconditioned BiCGSTAB, the average time cost of reproducible parallel preconditioned BiCGSTAB method based on ReproBLAS is only 1.02 times that of the non-reproducible parallel preconditioned BiCGSTAB.

In the future, we will add a comparison with OzBLAS and RARE-BLAS. We also intend to compare reproducible parallel preconditioned BiCGSTAB method based on four BLAS implementations in a mixed MPI+OpenMP environment.

ACKNOWLEDGMENTS

The second author was supported in part by Science Challenge Project, China (TZ2016002), NSF of China (61472462, 11671049, 11601033, 62032023) and the foundation of key laboratory of computational physics, China. The third author was supported by the InterFLOP (ANR-20-CE46-0009) project of the French National Agency for Research (ANR). The fourth author was supported by National Natural Science Foundation of China (No. 62032023). The fifth author was supported by the Project of Natural Science Foundation of Shandong Province (no.ZR2021MA092).

8 APPENDICES

We use the same parameter settings as in Section 5. Table 8 shows the 2-norm of the intermediate and final residual of the PBiCGSTAB solver in each iteration for the orsreg_1 matrix (http://sparse.tamu.edu/HB/orsreg_1). Table 9 shows the 2-norm of the intermediate and final residual of the ReproBLAS in each iteration when $K = 2$ and $K = 3$. Table 10 shows the infinity norm of the approximate and exact solutions of the PBiCGSTAB solver in each iteration for the orsreg_1 matrix. Table 11 shows the infinity norm of the approximate and exact solutions of the ReproBLAS in each iteration when $K = 2$ and $K = 3$.

REFERENCES

[1] Peter Ahrens, James Demmel, and Hong Diep Nguyen. 2020. Algorithms for efficient reproducible floating point summation. *ACM Transactions on Mathematical Software (TOMS)* 46, 3 (2020), 1–49.
[2] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. 1994. *Templates for the solution of linear systems: building blocks for iterative methods.* SIAM.
[3] Chemseddine Chohra, Philippe Langlois, and David Parello. 2015. Efficiency of reproducible level 1 BLAS. In *International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics*. Springer, 99–108.
[4] Chemseddine Chohra, Philippe Langlois, and David Parello. 2016. Reproducible, accurately rounded and efficient BLAS. In *European Conference on Parallel Processing*. Springer, 609–620.
[5] Caroline Collange, David Defour, Stef Graillat, and Roman Iakymchuk. 2015. Numerical reproducibility for the parallel reduction on multi-and many-core architectures. *Parallel Comput.* 49 (2015), 83–97.
[6] Defour D. Graillat S. & Iakymchuk R. Collange, C. 2015. Numerical reproducibility for the parallel reduction on multi-and many-core architectures. *Parallel Comput.* 49 (2015), 83–97.
[7] M. Corden. 2013. Differences in floating-point arithmetic between Intel R Xeon R processors and the Intel R Xeon PhiTM coprocessor. *Technical report* (2013).
[8] James Demmel and Hong Diep Nguyen. 2013. Fast reproducible floating-point summation. In *2013 IEEE 21st Symposium on Computer Arithmetic*. IEEE, 163–172.
[9] James Demmel and Hong Diep Nguyen. 2014. Parallel reproducible summation. *IEEE Trans. Comput.* 64, 7 (2014), 2060–2070.
[10] Ali H Dogru, Larry S Fung, Usuf Middya, Tareq M Al-Shaalan, Tom Byer, Hahn Hoy, Werner Artur Hahn, Nabil Al-Zamel, J Pita, Kesavalu Hemanthkumar, et al. 2011. New frontiers in large scale reservoir simulation. In *SPE Reservoir Simulation Symposium*. OnePetro.
[11] Roger Fletcher. 1976. Conjugate gradient methods for indefinite systems. In *Numerical analysis*. Springer, 73–89.
[12] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)* 33, 2 (2007), 13–es.
[13] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. 2014. *Using advanced MPI: Modern features of the message-passing interface*. MIT Press.
[14] Roman Iakymchuk, Maria Barreda, Matthias Wiesenberger, José I Aliaga, and Enrique S Quintana-Ortí. 2020. Reproducibility strategies for parallel preconditioned conjugate gradient. *J. Comput. Appl. Math.* 371 (2020), 112697.
[15] Roman Iakymchuk, Sylvain Collange, David Defour, and Stef Graillat. 2015. ExBLAS: Reproducible and accurate BLAS library. In *NRE: Numerical Reproducibility at Exascale*.
[16] Roman Iakymchuk, Stef Graillat, and José Aliaga. 2021. General framework for deriving reproducible Krylov subspace algorithms: A BiCGStab case study. (2021).
[17] Roman Iakymchuk, Maria Barreda Vayá, Stef Graillat, José I Aliaga, and Enrique S Quintana-Orti. 2020. Reproducibility of parallel preconditioned conjugate gradient in hybrid programming environments. *The International Journal of High Performance Computing Applications* 34, 5 (2020), 502–518.
[18] Ryuji Kimura. 2002. Numerical weather prediction. *Journal of Wind Engineering and Industrial Aerodynamics* 90, 12-15 (2002), 1403–1414.
[19] Donald Ervin Knuth. 1997. Seminumerical algorithms. *The art of computer programming* 2 (1997).

[20] Daichi Mukunoki, Takeshi Ogita, and Katsuhisa Ozaki. 2019. Reproducible BLAS routines with tunable accuracy using ozaki scheme for many-core architectures. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 516–527.

[21] Daichi Mukunoki, Katsuhisa Ozaki, Takeshi Ogita, and Roman Iakymchuk. 2021. Conjugate Gradient Solvers with High Accuracy and Bit-wise Reproducibility between CPU and GPU using Ozaki scheme. In *The International Conference on High Performance Computing in Asia-Pacific Region*. 100–109.

[22] Hong Diep Nguyen, James Demmel, and Peter Ahrens. 2018. ReproBLAS: Reproducible BLAS.

[23] C Nvidia. 2013. Cublas library user guide.

[24] Takeshi Ogita, Siegfried M Rump, and Shin'ichi Oishi. 2005. Accurate sum and dot product. *SIAM Journal on Scientific Computing* 26, 6 (2005), 1955–1988.

[25] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[26] Youcef Saad and Martin H Schultz. 1986. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing* 7, 3 (1986), 856–869.

[27] Alemayehu Shiferaw, Ramesh Chand Mittal, et al. 2011. An efficient direct method to solve the three dimensional Poisson's equation. *American Journal of Computational Mathematics* 1, 04 (2011), 285.

[28] Henk A Van der Vorst. 1992. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing* 13, 2 (1992), 631–644.

[29] Oreste Villa, Daniel Chavarria-Miranda, Vidhya Gurumoorthi, Andrés Márquez, and Sriram Krishnamoorthy. 2009. Effects of floating-point non-associativity on numerical computations on massively multithreaded systems. In *Proceedings of Cray User Group Meeting (CUG)*, Vol. 3.

[30] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[31] Nathan Whitehead and Alex Fit-Florea. 2011. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. *Tech. rep.* 21, 01 (2011), 18749–19424.

[32] Xu Xiaowen, Mo Zeyao, and An Hengbin. 2009. Algebraic two-level iterative method for 2-D 3-T radiation diffusion equations. *Chinese Journal of Computational Physics* 26, 1 (2009), 1.