Usability of Markov Chain Monte Carlo Preconditioners in Practical Problems

1st Emre Sahin
Hartree Centre, STFC
Warrington, United Kingdom
emre.sahin@stfc.ac.uk

2nd Anton Lebedev
Hartree Centre, STFC
Warrington, United Kingdom
anton.lebedev@stfc.ac.uk

3rd Maksims Abaļenkovs
Hartree Centre, STFC
Warrington, United Kingdom
maksims.abalenkovs@stfc.ac.uk

4th Vassil Alexandrov
Hartree Centre, STFC
Warrington, United Kingdom
vassil.alexandrov@stfc.ac.uk

***Abstract*—In this paper, we present the results of our exploration of the applicability of preconditioners computed using the Markov Chain Monte Carlo Matrix Inversion ((MC)^{2}MI) method to a variety of linear systems from the domain of quantum chromodynamics, plasma physics and engineering. The latter two are represented by matrices extracted from BOUT++ and Nektar++ codes with specific problem statements. Additionally, we present the scaling behaviour of our implementation for GPUs and CPUs.**

***Keywords*-Monte Carlo Matrix Inversion, Hybrid Algorithms for Linear Algebra, accelerators**

I. Introduction

Solving linear systems of equations in the form of $Ax = b$, where $A \in \{\mathbb{R}, \mathbb{C}\}^{n\times n}$ or inverting a matrix A is of unquestionable importance in many scientific fields. Iterative solvers such as generalized minimal residuals (GMRES) or bi-conjugate gradient (BiCG/BiCGstab) solvers are used widely to compute the solutions of these systems and such approaches are often the method of choice due to their predictability and reliability when considering accuracy and speed. They, however, may become prohibitive for large-scale problems as they can be very time consuming to compute. The complexity of these methods, in the serial case, is $O(kn^2)$ for dense matrices in the iterative methods case and $O(n^3)$ for direct methods with dense matrices while solving linear systems if common elimination or annihilation schemes (e.g. Gaussian elimination, Gauss-Jordan methods) are employed [1]. Iterative methods hence often rely on preconditioners to speed up the computations and/or to ensure faster convergence.

Monte Carlo (MC) method's complexity is linear in matrix size [2], [3] and can quickly yield a rough estimate of the solution by sampling a random variable whose mathematical expectation is the desired solution. For some practical problems, an estimate is sufficient or even favourable, due to the accuracy of the underlying data. Therefore, it should be pointed out, that Monte Carlo methods may be efficiently used as preconditioners.

Depending on the method used to compute the preconditioner, the savings and results vary. A very sparse preconditioner may be computed quickly, but it is unlikely to greatly reduce the run time to solution. On the other hand, computing a rather dense preconditioner is computationally expensive and might be time or cost-prohibitive. Therefore, finding a good preconditioner that is computationally efficient, while still providing substantial improvement to the iterative solution process, is a worthwhile research topic.

We have improved on the developments presented in [4] and [5] by expanding the application to complex matrices and incorporating symmetry conditions. Here we present the results of our exploration of the applicability of these improvements and the overall method to a set of linear systems encountered in physics and engineering - specifically in lattice quantum chromodynamics (LQCD) and plasma physics simulations. The latter is performed using BOUT++ [6] and Nektar++ [7] and hence required extraction of system matrices from these frameworks since ((MC)^{2}MI) currently still requires a matrix to be explicitly present. Additionally, we illustrate the scalability of this method across multiple computing devices.

The next section provides a basic overview of the method, delineating introduced changes. Section III shows the approach and methodology applied in the enhancement of the parallel implementations. Section IV presents the considered cases and the corresponding results and analysis. Section V summarises the results and outlines the future work.

II. Monte Carlo Approach

Monte Carlo methods are probabilistic methods that use random numbers to either simulate a stochastic behaviour or to estimate the solution of a problem via sampling of a (appropriately chosen) random variable. They are good candidates for parallelisation because many independent samples are used to estimate the solution. These samples can be calculated in parallel, thereby speeding up the solution-finding process. Since Monte Carlo methods possess the following main generic properties [2], [3]: efficient distribution of the compute data, minimum communication during the computation and increased precision being achieved by adding extra refinement computations, it is not surprising that they are a mainstay in the domain of High-Performance Computing. A weak point of Monte Carlo methods, especially in parallel environments, is their dependency upon the availability of independent (pseudo) random numbers to produce acceptable results.

A. Algorithm

The core algorithm remains unchanged and has been presented in [4]. Differing from the previous version here we do

978-1-6654-1128-8/21/$31.00 ©2021 IEEE
DOI 10.1109/ScalA54577.2021.00011

not eliminate entries that would result in negligible transition probabilities for a Markov Chain simulation. Although fruitful if done correctly, this approach is ultimately a preprocessing technique and not the subject of the analysis presented here. Additionally, it does not generalize to arbitrary inputs, since the truncation thresholds of matrix entries are dependent on the system the matrix represents.

Algorithm 1: Monte Carlo Algorithm for Inverting General Matrices

```
1) Read in matrix $B$
    a) Input matrix $B$, parameters $\varepsilon$ and $\delta$
2) Determine if $B^\dagger = B$. If true set isSymm to true.
3) Calculate intermediate matrices ($\hat{B}$, $B_1$)
    a) Split $B = \hat{B} - (\hat{B} - B)$, where $\hat{B}$ is a diagonally
       dominant matrix: $\hat{b}_{ij} = b_{ij} + \delta_{ij}\alpha_j ||B||$
    b) $B_1 := \text{diag}(\hat{B})$.
4) Perform row-scaling on $\hat{B}$ to enforce unit diagonal: $\hat{B} :=
   B_1^{-1}\hat{B}$.
5) Subtract the identity matrix from the scaled $\hat{B}$ to obtain
   a matrix $C$ s.t. $\text{diag}(C) = \{0, \dots, 0\}$
6) Compute the transition probability matrix $P$ s.t. $p_{i,j} =
   \frac{|c_{i,j}|}{\sum_j |c_{i,j}|}$.
7) Perform a random walk on the matrix $C$ using $P$ as
   transition probabilities and accumulate results into the
   $\hat{B}^{-1}$ matrix.
8) Recovery of $B^{-1}$ from $\hat{B}^{-1}$
    a) Compute $\hat{B}^{-1} := \hat{B}^{-1}B_1^{-1}$
    b) Compute $S = \hat{B} - B$
        i) Define $S_i \in \mathbb{C}^{n\times n}$, where $i = 1, 2, \dots, M =
           nnz(S)$ where $M$ is the number of non-zero
           elements of $S$, in a way that each $S_i$ has just
           one of the non-zero elements of matrix $S$.
        ii) Set $B_n^{-1} := \hat{B}^{-1}$
        iii) Apply $B_{i-1}^{-1} = B_i^{-1} + \frac{B_i^{-1} S_i B_i^{-1}}{1 - \text{Tr}\left(B_i^{-1} S_i\right)}$ for $i =
             n, n-1, \dots, 1$
    c) Then $B^{-1} = B_0^{-1}$
9) If isSymm = true $B^{-1} := (B^{-1} + (B^{-1})^\dagger)/2$
```

The definition of the $\hat{B}$ matrix in step 3 includes contributions of a norm of B, with different weights α_i for each row, to its diagonal. This ensures that the resulting matrix is (strictly) diagonally dominant and that the resulting random walks will terminate. The last step is, in general, necessary in cases where the input matrix is symmetric, as the preconditioner, being an approximate inverse, should be symmetric, too. Unfortunately, the MC iteration - in the formulation presented in [4] - will yield an asymmetric preconditioner and hence the result needs to be symmetrized by the above procedure for symmetric matrices.

We call attention to *step 6* of the algorithm, where the transition probability matrix is computed. The absolute values are required to ensure meaningful transition probabilities of the Markov Chain. Simultaneously real and complex inputs are handled. Our previous implementation of the algorithm relied on a simplified computation of signs of matrix entries, which were used to compute $\hat{B}$. In this paper we present results of tests with complex input matrices provided to us by our collaborators in the Exascale Lattice-QCD project (Exalat).

Several enhancements of the algorithm, as well as modifications concerning GPU implementation, are listed in the next section and were able to substantially improve its performance in generating rough inverses of the input matrices. The result can then be used directly as a preconditioner for solving a system of linear algebraic equations or further improved. We propose the use of an iterative refinement process to further enhance the quality of the preconditioner. The decision of whether those additional steps are taken is based upon the required accuracy and can be freely selected, depending on user requirements.

III. Parallelization details and issues

The steps in the algorithm presented above can be relabelled in a more legible fashion:

3) Transformation into a diagonally-dominant matrix
4),5) Transformation of the result to suit the *Neumann series expansion*
6), 7) Use Monte Carlo method to compute a sparse approximation of the inverse matrix $\hat{B}^{-1}$.
8) Recover an approximate inverse of B from $\hat{B}^{-1}$.
9) Symmetrize the result in case of a (complex) symmetric input matrix.

It must be noted that phase 4 (recovery) requires in general $\mathcal{O}(n^3)$ operations and hence is generally neglected. Prior numerical experiments have demonstrated that it is not compulsory to obtain an effective preconditioner and is in general an impediment to an *efficient* preconditioner. The first step is the notable exception to this rule as it is always applied.

A. GPU implementation

Due to the irregular data access and computation kernels of short duration the use of the method on GPUs is challenging. Nonetheless, it is possible to accelerate the computation of the preconditioner using (MC)^{2}MI if care is taken to keep the GPU sufficiently busy.

Since the number of different entries visited by a chain is not known *a priori* the entire set of Markov Chains is simulated at once and used to fill a contiguous array corresponding to one row of the approximate inverse. The results are sorted by magnitude in descending order and only a prescribed number of the largest entries are retained in the final sparse approximate inverse. An extension to multiple GPUs can be achieved trivially, by splitting the matrix into blocks of rows, but produces additional overhead when combining the results of the separate GPUs.

Compared to the host machine the GPU has a very limited amount of memory and requires a more elaborate approach to memory handling. Due to memory constraints storage of a dense block of an inverse on the device is not feasible, and neither is on-the-fly transfer of computed entries to the host – due to latency constraints. We have opted to allocate and fill a block of the sparse inverse on the device and transfer it to the

TABLE I
MATRIX SET

Matrix	Dimension	Non-zeros	Sparsity	Symmetry
circuit5M_dc	3,523,317	19,194,193	$1.5 \cdot 10^{-6}\%$	symmetric
nonsym_r3_a11	20,930	638,733	0.15%	asymmetric
sym_r6_a11	1,314,306	36,951,316	0.02%	symmetric
H2hat	6,144	11,501,568	30.47%	hermitian
H2	12,288	4,202,496	2.78%	hermitian
NLD	8,192	24,576	$4.66 \cdot 10^{-2}\%$	asymmetric

host matrix at the end of the computation. This differs from our MPI/OpenMP implementation in so far as the computation of each row requires additional memory management overhead but the final reduction of the separate blocks of the inverse is cheaper since the necessary storage and data layout is known beforehand. The downside being that for some matrices entries of the inverse may be lost for some rows, whilst others contain unused entries ($= 0$). The problem of superfluous 0's is taken care of by pruning the entries of the final approximate inverse after it has been assembled on the host.

IV. NUMERICAL EXPERIMENTS

A. Test cases

The set of matrices chosen for the assessment of the usability of the method in practical problems is listed in Table I. The set contains symmetric and non-symmetric matrices of varying sizes and filling fractions. Matrices *nonsym_r3_a11* and *sym_r6_a11* have been provided by our collaborators and are representative of systems occurring in climate simulations. The matrix *circuit5M_dc* has been taken from Florida University's matrix collection. Matrices *H2*, *H2hat* represent operators in lattice quantum chromodynamics (LQCD) computations and the *NLD* matrix resulted from a finite element discretization of a non-linear diffusion equation in 1D using BOUT++ and is intended to represent typical applications in plasma physics. The latter matrices have been used to evaluate whether ($(\text{MC})^2$MI) would be suitable as a preconditioner for problems in the fields of LQCD and plasma physics.

We have included the very large, extremely sparse circuit5M_dc matrix with the intent to have a sufficient workload to utilize the computing resources to their maximum for scalability analyses of the matrix-based implementation.

B. Execution Environment

The software were compiled using GCC 7.3.0 and OpenMPI 4.0.4 and the numerical experiments on the LQCD and BOUT++/Nektar++ matrices were performed on the CPUs of the ScafellPike system hosted at the Hartree Centre. A single node of this system features two Intel Xeon Gold Skylake processors (E5-6142 v5, 2.6 GHz, turbo boost up to 3.7 GHz), sporting 16 cores each. This node is comprised of two NUMA entities with 94 GB and 96 GB of memory. One Skylake processor has an L3 cache of 22 MB shared amongst 16 cores, and cores have individual L2, L1d and L1i caches of 1 MB, 32 KB and 32 KB in size.

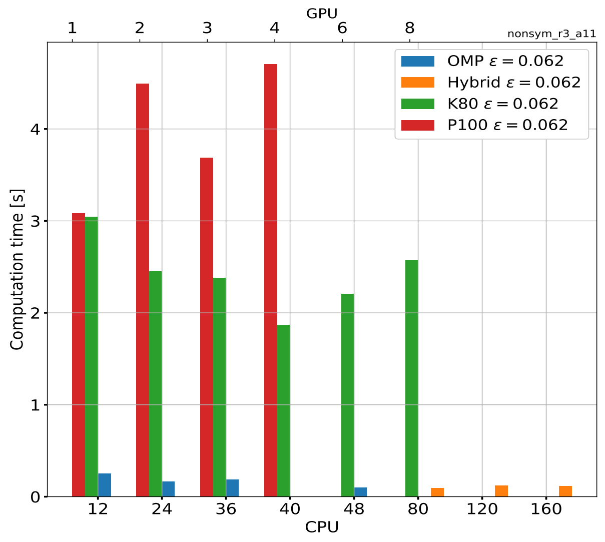

Fig. 1. Execution time of the preconditioner computation for a small matrix *nonsym_r3_a11*. The small size emphasises the management overhead on the GPUs.

The computed preconditioners were validated using the GMRES, CG and BiCGstab implementations provided by NumPy (1.19.3) run with Python 3.6.2.

Experiments were performed with precisions of $\epsilon, \delta = 2^{-3}, 2^{-4}$ and the scaling of the diagonal was performed using $\alpha = 5$. The choice of the above values for ϵ, δ implies that the GPUs are likely to be underutilized since earlier results indicate that $\epsilon, \delta < 0.01$ are to be preferred.

C. Scalability

In Fig. 1 the execution time of the preconditioner computation using up to 4 P100 GPUs and up to 8 K80 GPUs is shown for the *nonsym_r3_a11* matrix. It is evident that the small size of the matrix, coupled with the moderate precision requirements results in a severe underutilization of the computing units of P100s and therefore an over-emphasis of the preprocessing and memory management overhead. Indeed the runtimes decrease when scaling up to 4 K80's, increasing afterwards due to the management overhead becoming noticeable. For small matrices, the CPU is a sufficiently powerful device to perform the computation. This will not hold with increasing precision of the computation, due to the increase in the amount and length of Markov chains that will require evaluation.

Considering Fig. 1 one can immediately conclude, that speed-up of the GPU is going to be non-existent for small matrices. Furthermore one can see, that for small matrices the memory management overhead limits the scalability of the OpenMP/Hybrid version of the code, too. In the case of hybrid ($(\text{MC})^2$MI) an additional overhead is introduced by MPI communication.

In the case of the very large matrix *circuit5M_dc*, the performance gain using GPUs is even more pronounced (c.f. Fig. 2) and the scalability shows the typical behaviour of the

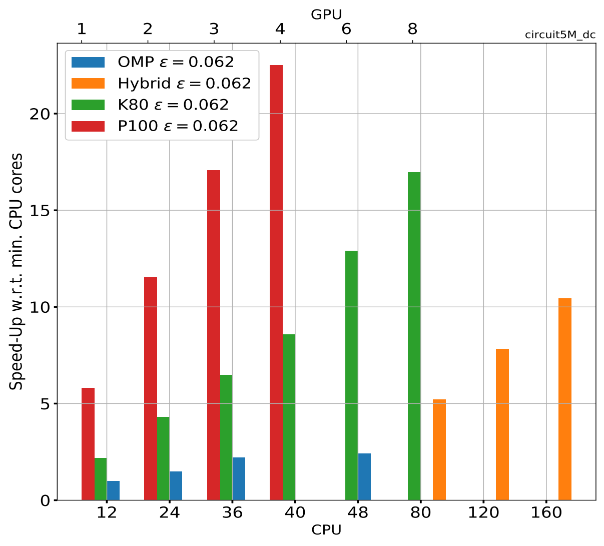

Fig. 2. Speed-up of the calculation performed on GPUs in comparison to CPU cores for the *circuit5M_dc* matrix and a precision of $\epsilon = 0.0625$.

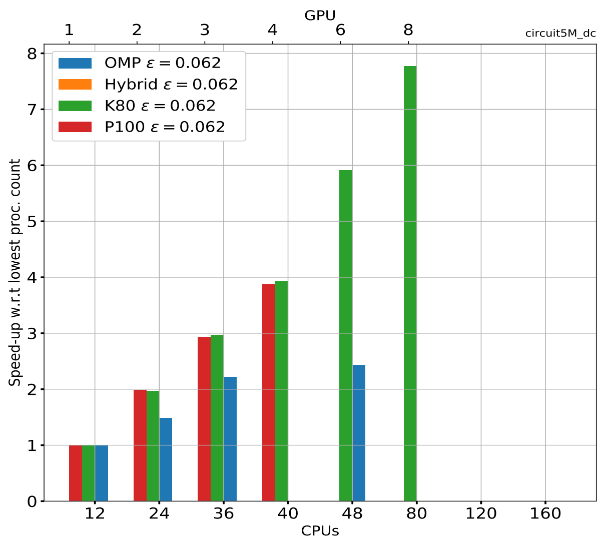

Fig. 4. Speed-up of the calculation performed on different architectures in comparison to the lowest number of processing units chosen for the architecture for the *circuit5M_dc* matrix.

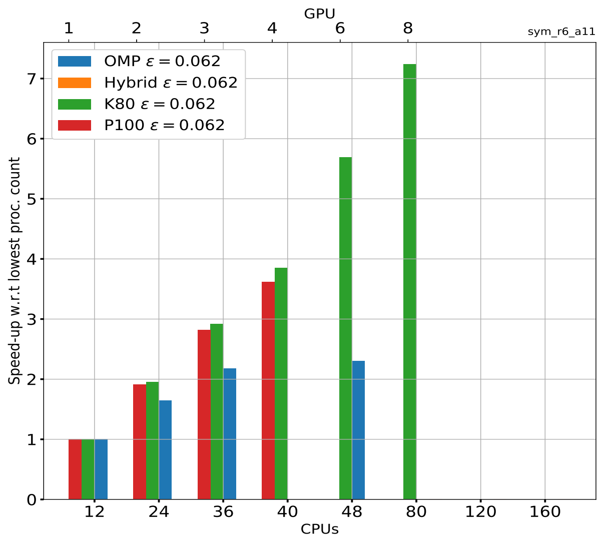

Fig. 3. Speed-up of the calculation performed on different architectures in comparison to the lowest number of processing units chosen for the architecture for the *sym_r6_a11* matrix.

method when it is not dominated by memory management overhead.

The speed-up decreases when using 3 or more GPUs, which is to be attributed to the overhead introduced by the memory management. This fact is better demonstrated when one compares Fig. 3 to Fig. 4, where the speed-up is almost ideal.

The scalability of the method across multiple devices of the same architecture is shown in Figs. 3 and 4 and requires a sufficient amount of work to hide the current memory management overhead. Here the speed-up is computed w.r.t. the lowest number of processing units of the architecture.

For the large matrices of the set, the scaling is, as expected, sub-linear. Furthermore, it can be seen, that the larger matrix results in better scaling behaviour. This can be attributed to the increase in the time the GPUs spend performing the MCMC simulation, thereby reducing the influence of preprocessing and memory management overhead.

A factor limiting the performance, which has not yet been eliminated, is the necessity to compact the pre-processed matrix on the host before the MC iteration may be performed. As a byline, we would like to remark, that this method has been implemented to run on an FPGA with the help of our collaborators at Maxeler. First tests on the *circuit5M_dc* matrix indicate that a single FPGA could accelerate the computation of the preconditioner by a factor of $\sim 12.7\times$, while also demonstrating that the usage of FPGAs will require careful analysis of the required precision of the calculations.

D. Practical Usage

In a first trial, we considered the extension of our implementation to complex input matrices and have applied the method to compute a preconditioner for LQCD matrices *H2*, *H2hat*. The results depicted in Fig. 5 and 6 for the *H2* matrix and CG and GMRES solvers demonstrate that the method can be effective for this type of problem. Since the matrix is hermitian (i.e., complex symmetric) the method of conjugate gradients (CG) is a natural choice. This intuition is borne out by the number of steps required by CG and GMRES solvers to achieve the same relative error of 10^{-10}. It is however surprising, that as soon as a preconditioner is introduced GMRES becomes the better choice.

Since CG's convergence behaviour is strongly dependent on the system matrix being symmetric (positive definite) it is not surprising that the number of iterations required when

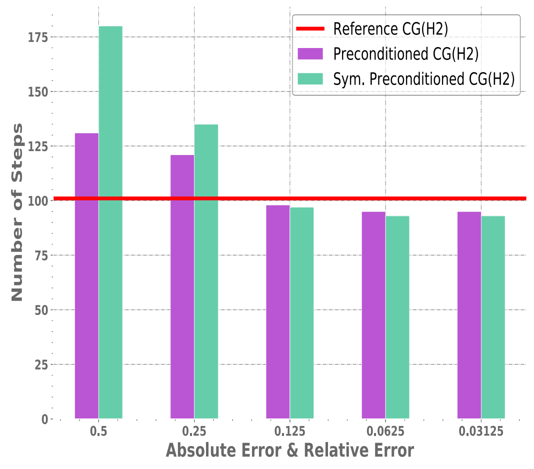

Fig. 5. The number of steps required to solve a complex linear system with the CG solver and H2 as system matrix. Note that the non-symmetric preconditioner levels out just after dropping below the non-preconditioned step count.

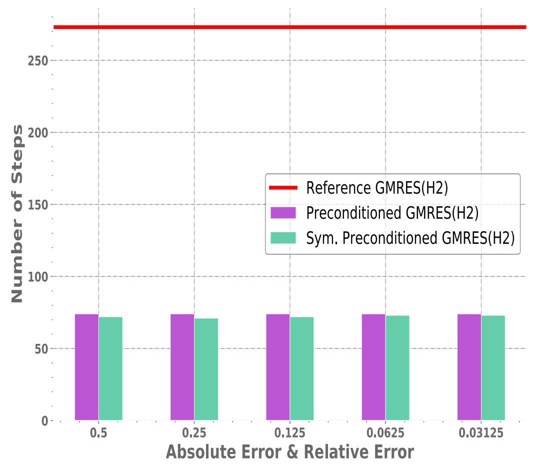

Fig. 6. The number of steps required for the *H2* matrix using GMRES solver

using an asymmetric preconditioner does not decrease, whereas when the preconditioner is symmetric the number of iterations steadily decreases. This is intuitively pleasing, as a higher precision (smaller ϵ, δ) of ((MC)^{2}MI) will result in a more asymmetric matrix. We currently have no explanation for the increase in the number of steps required for PCG to converge if the preconditioner has been computed to a low precision ($\frac{1}{2}, \frac{1}{4}$). This anomaly is currently under investigation.

In Fig. 7 we present the evolution of the number of iterations required by the (non-)preconditioned GMRES and BiCGstab solvers for a solution of a non-linear diffusion equation in 1D (*NLD* matrix). Note that the values given in Table I are

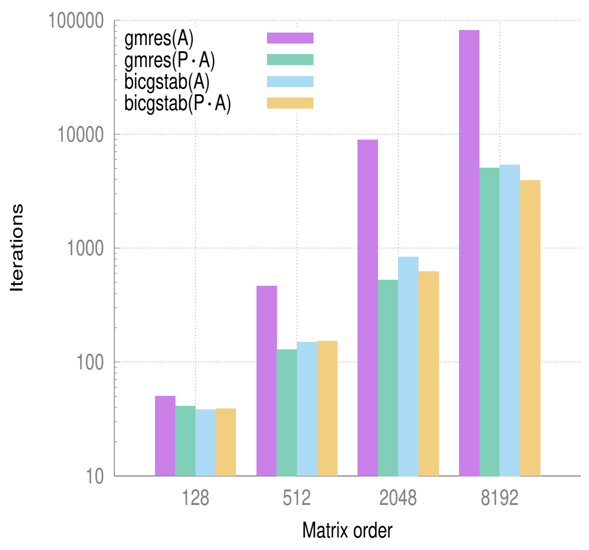

Fig. 7. The number of steps taken by GMRES and BiCGstab methods for a non-linear diffusion problem with and without the asymmetric ((MC)^{2}MI) preconditioner. $P \cdot A$ signifies a preconditioned solver.

provided only as an example for the largest matrix considered. As can be observed usage of the ((MC)^{2}MI) preconditioner will consistently reduce the number of steps required by GMRES. For BiCGstab usage of this preconditioner appears to increase the number of iterations for very small matrices $n < 1000$. Currently, we attribute this to the random nature of the preconditioner computation. Further analysis of this behaviour is warranted, but not of primary concern since small matrices are not the primary application area for iterative solvers and constitute rather academic exercises. With the growing dimension of the matrix usage of the preconditioner becomes increasingly beneficial, especially if GMRES is used as a solver. This indicates that it will be worthwhile to implement ((MC)^{2}MI) as a matrix-free preconditioner in BOUT++, resp. PETSc [8].

In Fig. 8 we observe a varying behaviour for a variety of equations: Helmholtz equation (H), steady advection-diffusion equation (SAD) and unsteady advection-diffusion (UAD) equation with varying order of finite elements. The last number signifies the dimension of the matrix. Here the use of the ((MC)^{2}MI) preconditioner leads to a consistently lower number of steps for BiCGstab, yet for the unsteady advection-diffusion problem, it consistently yields a higher number of steps required. This seems to suggest that the method is not suitable for such problems, but further analysis is required to pinpoint the exact reason for this behaviour as it still eludes us.

V. Conclusions and Future Work

A basic understanding of modern architectures suggests that random access to data may be a performance-limiting factor. Profiling results obtained for the OpenMP implementation support this intuition since a large proportion of the simulation time is spent waiting for data and on memory management.

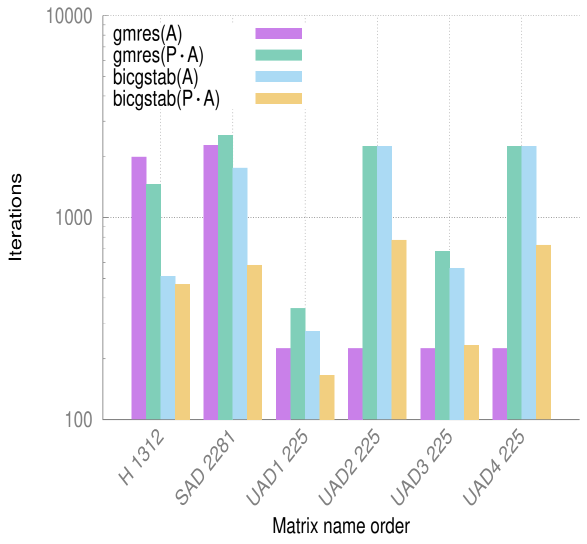

Fig. 8. The number of steps taken by GMRES and BiCGstab methods for varying problems obtained with Nektar++ with and without the asymmetric ((MC)^{2}MI) preconditioner. $P \cdot A$ signifies a preconditioned solver and the last number in the block label signifies the dimension of the matrix.

Orchestration of the GPU threads in a way as to reduce divergences appears possible but will require further work as will the reduction of the load-imbalance. Implementation of the method on FPGAs suggests that said architecture can be used as accelerators, much like GPUs can be and the preliminary results suggest that this architecture is worth further consideration, especially in a modular cluster environment.

Numerical experiments presented above have demonstrated that ((MC)^{2}MI) can be applied to problems currently under consideration. The next development step will be to implement the ((MC)^{2}MI) preconditioner as a matrix-free method to enable in-situ testing with frameworks such as Nektar++ and BOUT++, as well as with our academic partners from the University of Portsmouth (LQCD). We will give preference to implementing a matrix-free version for BOUT++ since the latter relies on PETSc and would hence enable us to investigate the methods' suitability to a much more diverse set of problems, i.e., topological optimization of mechanical and optical structures.

Hence we envision a development of the matrix-free implementation for the CPU in the near future. A multi-architecture implementation that will allow us to run the preconditioner computation on multiple device types is planned thereafter.

VI. Acknowledgements

We would like to thank the United Kingdom Atomic Energy Authority (UKAEA) for the inclusion into the NEPTUNE project and especially to Sue Thorne (Rutherford Appleton Laboratory, STFC), Benjamin Dudson (The University of York) and Chris Cantwell (Imperial College London) for the active support in acquiring the necessary matrices. Additionally, we are very grateful to Tobias Becker, Gary Robinson and Bastiaan Willem Kwaadgras (Maxeler) for their help in porting the ((MC)^{2}MI) code to the FPGA hardware. We are also grateful for the partial support by EPSRC through Lattice Field Theory at the Exascale frontier project (EP/V001248/1) and Prof. Antonio Rago, University of Plymouth, UK, for providing LQCD related matrices.

References

[1] G. Golub and C. Loan, *Matrix computations*, ser. Johns Hopkins studies in the mathematical sciences. Johns Hopkins University Press, 1996. [Online]. Available: http://books.google.es/books?id=mlOa7wPX6OYC

[2] J. Straßburg and V. N. Alexandrov, "Enhancing monte carlo preconditioning methods for matrix computations," in *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*, 2014, pp. 1580–1589. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2014.05.143

[3] V. N. Alexandrov and O. A. Esquivel-Flores, "Towards monte carlo preconditioning approach and hybrid monte carlo algorithms for matrix computations," *Computers & Mathematics with Applications*, vol. 70, no. 11, pp. 2709–2718, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.camwa.2015.08.035

[4] A. Lebedev and V. Alexandrov, "On advanced monte carlo methods for linear algebra on advanced accelerator architectures." in *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, 2018.

[5] M. E. Şahin, A. Lebedev, and V. Alexandrov, "Empirical analysis of stochastic methods of linear algebra," *Lecture Notes in Computational Science*, vol. 7, June 2020.

[6] B. Dudson, P. Hill, and J. Parker, "Bout++," online repository, 2020. [Online]. Available: http://boutproject.github.io

[7] S. Sherwin, M. Kirby, C. Cantwell, and D. Moxey, "Nektar++," online, 2021. [Online]. Available: https://www.nektar.info

[8] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Web page," 2019. [Online]. Available: https://www.mcs.anl.gov/petsc