Empirical Analysis of Stochastic Methods of Linear Algebra

Mustafa Emre Şahin[1], Anton Lebedev[2], and Vassil Alexandrov[1]

[1] The Hartree Centre, Keckwick Ln, Warrington, UK
`vassil.alexandrov@stfc.ac.uk`
[2] Helmholtz Zentrum Dresden-Rossendorf, Bautzner Landstr. 400, Dresden, Germany `a.lebedev@hzdr.de`

Abstract. In this paper we present the results of an empirical study of stochastic projection and stochastic gradient descent methods as means of obtaining approximate inverses and preconditioners for iterative methods. Results of numerical experiments are used to analyse scalability and overall suitability of the selected methods as practical tools for treatment of large linear systems of equations. The results are preliminary due to the code being not yet fully optimized.

Keywords: Linear Algebra · Stochastic Methods · High-Performance Computing.

1 Introduction

In this paper we present an empirical evaluation of three numerical methods derived for the stochastic solution of linear algebraic systems. All methods are evaluated regarding their suitability as tools for computing an approximate inverse matrix. Furthermore, we consider their scalability and the usefulness of the approximate inverses as preconditioners for iterative solvers for linear systems of the form $A\boldsymbol{x} = \boldsymbol{b}$, where $A \in \{\mathbb{R}, \mathbb{C}\}^{n\times n}$ is henceforth designated system matrix.

Solution of linear algebraic systems is of paramount importance in almost every domain of scientific computing. In case of table-top numerical experiments such systems may very well be solved using well-known methods like LU decomposition, but for a wide variety of cases the size of the matrices would prevent them from being storable even given the abundant storage space available nowadays. In such cases the matrices have, however, generally a very sparse structure, reducing their memory footprint enormously. The downside being, that an inverse of a sparse matrix is generally dense and a method like LU decomposition hence becomes unfeasible. To obtain a solution in such cases iterative methods such as generalized minimal residues (GMRES) iteration or bi-conjugate gradient stabilized (BiCGstab) iteration are employed, which solve the linear system by merit of a fixed-point iteration. Since there's no silver bullet for complexity of the problem iterative methods may suffer of slow convergence towards the solution. This problem one attempts to ameliorate using preconditioners - matrices which modify the problem and ideally reduce the number of iterations and, ideally also the runtime to solution.

Here we consider the suitability of two methods, stochastic gradient descent with missing values (mSGD) and stochastic projection (SP) as ways to compute said preconditioners.

2 Algorithm Description

Our main focus in this work is on stochastic projection (SP), or randomized Kazmarz method, as described in [6] and on the stochastic gradient descent with missing values (mSGD) - as given in [5]. Both methods are assessed also regarding their sensitivity to the initial conditions. The latter is achieved by comparing convergence behaviour with a random initial guess at the solution to an initial guess provided by the Markov Chain Monte Carlo Matrix Inversion (MCMCMI) method described in [4]. In the following a brief description of each method is provided.

2.1 Stochastic Projection

The basic idea is fairly simple, and the implementation follows roughly eq (2.13) of [6]. The main idea is to project the solution vector successively and orthogonally onto arbitrarily chosen subspaces until the accumulated effect leads the iteration into the subspace of the true solution. An obvious extension to block-projections has been mentioned in [6] and implemented here. The iteration step is given as follows:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + A_i^t \left(A_i A_i^t\right)^{-1} \left(\boldsymbol{b}_i - A_i \boldsymbol{x}_k\right) \ . \tag{1}$$

Here A_i is a randomly selected block of rows of the matrix and $\boldsymbol{b}_i$ the corresponding subset of entries of the right-hand-side vector of

$$A\boldsymbol{x} = \boldsymbol{b} \ . \tag{2}$$

The intuitive simplicity of this approach is paid for by its performance. Furthermore the computation of a matrix inverse in each step is required. Depending on the block size the computation of said inverse, or a solution of a dense system, may incur a significant cost.

2.2 Stochastic Gradient Descent

In the present modification, as proposed by Ma and Needell in [5] a gradient descent takes place not over the entirety of the column space of the matrix, but rather over a randomly selected subspace (of dimension 1). Ma and Needell derive the method under the assumption that an entry a_{ij} of the system matrix itself is missing with a probability p. This results in the following iteration rule:

$$x_{k+1} = \boldsymbol{x}_k - \alpha_k \left[\frac{1}{p^2}\left(\boldsymbol{a}_i\left(\boldsymbol{a}_i^t x_k - p b_i\right)\right) - \frac{1-p}{p^2} diag\left(\boldsymbol{a}_i \boldsymbol{a}_i^t\right) \boldsymbol{x}_k\right] \ . \tag{3}$$

Here $\boldsymbol{a}_i$ is the i-th column-vector of the matrix and the last part of the step utilises a diagonal matrix derived from the outer product of $\boldsymbol{a}_i$ with itself. Convergence of the gradient descent method depends on its step-size. The latter can be either fixed or variable and an expression for a variable step size is provided as a function of an arbitrary parameters $0 < c, r < 1$, where c is called "learning rate", the number of iterations k and the smallest eigenvalue λ_{min} of the system matrix:

$$\alpha_k = \frac{c}{|\lambda_{min}|} r^{\frac{k}{T}} \ . \tag{4}$$

Here T is the number of steps after which the step size will be shrunk by a factor $r < 1$. Again, the relative simplicity of the method is compensated by the existence of two parameters in (3): α_k, p. Both are, as a rule, unknown *a-priori* and have to either be guessed or, in case of the step size, computed using (4). An educated guess as to the parameters may very well result in sub-optimal choices, whereas the computation of α_k using the above equation introduces yet another pair of parameters.

2.3 Markov Chain Monte Carlo Matrix Inversion

The third solution method presented here is the approximation of an inverse using Markov Chain Monte Carlo for Matrix Inversion (MCMCMI) as presented in [4]. The fundamental idea of the method is to employ the Neumann series to compute an inverse of a diagonally-dominant matrix. To reduce the cost the Neumann series is evaluated stochastically using Markov Chains. Since the series is infinite estimates as to the number and length of chains are required for a practical application. Such an estimate was provided in [3] and the method has been implemented both for GPUs and CPUs. In the current case it stands as a stand-alone method of providing preconditioners for iterative methods, as well as a source of initial conditions for the mSGD and SP iterations.

The algorithm can be split into the following 5 phases (Notice that phases 1 and 5 are only necessary when the initial matrix is not a *diagonally dominant matrix (ddm)*): 1) Initial matrix is transformed into a ddm, 2) Transformation of ddm for suitable *Neumann series expansion*, 3) Monte Carlo method is applied to calculate sparse approximation of the inverse matrix, 4) Given 2, calculate the inverse of the ddm from 3, 5) Recovery process is applied to calculate the inverse of the original matrix due to the transformation in 1. It must be noted that the last phase 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 method requires two tolerance values ϵ - an error bound on the stochastic error, which determines the maximum amount of chains required - and δ, a truncation error bound which affects the length of the chains. Both have to be provided by the user and together dictate the precision of the approximation.

The major caveat of this method is that it is strictly valid only for diagonally dominant matrices if the recovery procedure is not being applied. Nevertheless, as has been demonstrated in [4], the obtained approximation of the unrecovered inverse is often sufficient as a preconditioner for iterative systems.

3 Implementation and Experiments

The effectiveness of the chosen methods has been assessed using a set of sparse matrices of varying size and structure, intended to represent multiple domains of scientific computing. It is provided in table 1. The set contains matrices from climate simulations (`nonsym_r3_a11, sym_r6_a11`), semiconductor electronics (`circuit5M_dc`, from [2]), computational fluid dynamics (`rdb2048`, [2]) and simple mathematics (`2DFDLaplace_20x20`).

Table 1: Matrix set.

Matrix	Dimension	Non-zeros	Sparsity	Symmetry
circuit5M_dc	$3,523,317 \times 3,523,317$	19,194,193	$1.5 \cdot 10^{-6}$%	symmetric
nonsym_r3_a11	$20,930 \times 20,930$	638,733	0.15%	non-symmetric
sym_r6_a11	$1,314,306 \times 1,314,306$	36,951,316	0.02%	symmetric
rdb2048	2048×2048	12032	0.28%	non-symmetric
2DFDLaplace_20x20	361×361	1729	1.32%	symmetric

3.1 Implementation Details and Caveats

The methods described in the previous section have been implemented in C++. Parallelization of all of the methods for use on CPUs has been achieved by combining MPI and OpenMP (OMP) parallelisation (so-called hybrid parallelisation). A GPU implementation was available only for MCMCMI at the time of writing. This has been done using NVIDIA's CUDA programming model. To utilise multiple GPUs the implementation utilises OpenMP threads s.t. each GPU is controlled by one thread.

Parallelisation is similar for all methods and architectures. The main process reads the matrix A and the run-time parameters. It then broadcasts this data to the workers. Each worker computes the fraction of rows/columns it has to process based on its rank. At the end of the computation the resultant local block of A^{-1} is purged of entries smaller than the user-prescribed tolerance (10^{-9} resp. 10^{-14} for `circuit5M_dc`). Then it is collected onto the master process for storage. It is important to note that, with increasing matrix size broadcast operations introduce a non-negligible communication overhead.

In the case of the GPU implementation of MCMCMI the main process is the master thread of OpenMP and workers are the GPUs. For the hybrid implementation (MPI+OpenMP) the master process is MPI rank 0 and worker processes have rank> 0. The distribution of the matrix blocks is uniform among the processes, with the last block of rows being assigned to the master due to it being potentially smaller than the other blocks. This preferential treatment of the master is done to mask potential load imbalance by letting the master prepare the output arrays prior to collection. Within each node (MPI rank) the computation of the approximate inverse is parallelised over the rows of the block using OpenMP `dynamic` distribution to ameliorate load imbalance among the rows.

The mSGD and SP implementations follow the same structure, but acceleration on the worker processes is achieved by utilising Eigens' built-in parallelisation using OpenMP threads instead of manually distributing the columns of the inverse.

The main drawback of the CPU-oriented implementation of MCMCMI is the lack of resources which could be used to compact and sort the row of the approximate inverse, before the entries that will be retained are extracted. The current implementation of stream compaction introduces significant overhead but is still preferable to sorting an entire row of the approximate inverse if it contains only a handful of non-vanishing values.

Numerical experiments for SP and mSGD were run on the Scafell Pike system of the Hartree centre, which consists of nodes fitted with 2x XEON gold E5-6142 v5 processors resulting in 32 cores per node and due to HyperThreading in 64 threads per node.

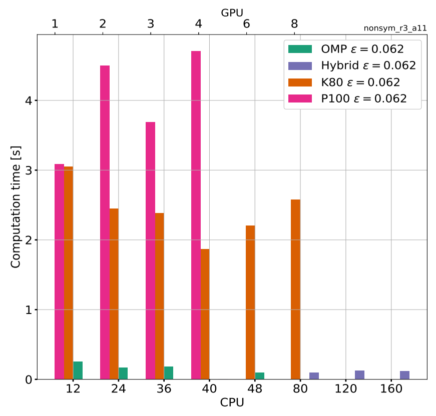
The experiments for the assessment of MCMCMI and to generate optimized initial conditions for mSGD and SP were executed on the `hemera` system of the Helmholtz Centre Dresden-Rossendorf by A.L. The systems running the K80 experiment set consisted of 8x NVIDIA K80 GPUs with 32GB VRAM on a system with 2 Xeon E5-2630v3 CPUs with HyperThreading enabled. For hybrid MCMCMI experiments the nodes used contained 2 Xeon Gold 6148 CPUs with 20cores/40 threads each. The P100 experiments were performed on systems containing 4 P100 GPUs with 16GB VRAM each connected via the NVLink interface to a node of 2 Xeon Gold 6136 CPUs with 12 cores/24 threads each. All systems are connected via Infiniband with 56 Gb/s.

3.2 Numerical Experiments and Analysis of Results

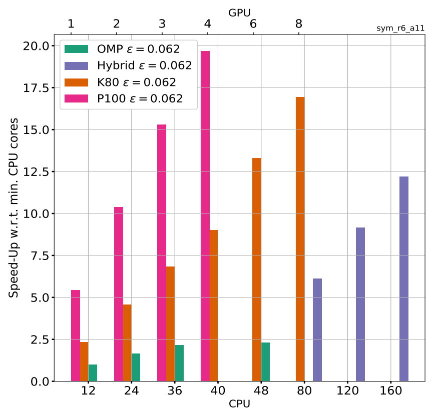
MCMCMI For the sake of simplicity we chose $\epsilon = \delta$ for the simulations, although this is by no means necessary. This choice ensures that neither stochastic (ϵ) nor truncation errors (δ) dominate.

From fig. 1a one can immediately see (note that the upper horizontal axis enumerates GPUs), that the usage of the GPU many-core architecture is of little meaning if the workload is small. This is the case when the matrix to be inverted is small. In this case the GPU will spend most of the time in memory-management overhead. Hence, for the `nonsym_r3_a11` matrix an optimal number of GPUs, dependent on their architecture, appears to exist. The latter is understandable since different architectures generally feature vastly different numbers of processing units.

However, requiring a precise approximate inverse (lowering the relative error ϵ) and working with large matrices leads to a full utilization of the GPU's com-



(a) Execution time of the preconditioner computation for a small matrix. Management overhead on GPUs is emphasized by the small size.

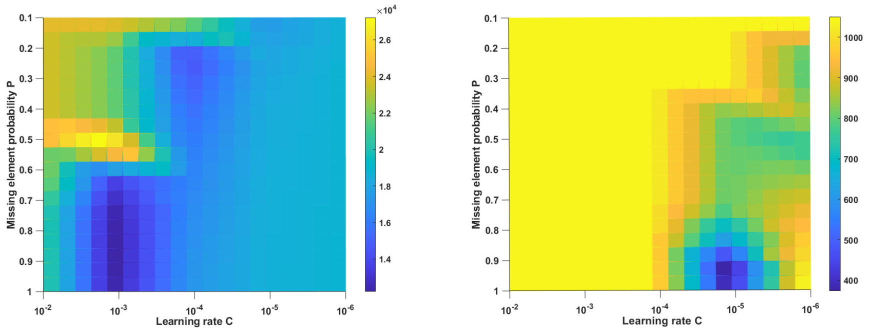


(b) Speed-up in comparison to 12 CPU cores for the sym_r6_a11 matrix and a precision of $\epsilon = 0.0625$.

Fig. 1: Execution time and speed-up of MCMCMI depending on the underlying architecture and matrix size. Note that the lower axis enumerates CPUs of the OpenMP or MPI+OpenMP (hybrid) implementation and the upper axis GPUs of a GPU-only implementation.

puting resources, resulting in a good scalability across multiple GPUs. Similarly for CPUs. Fig. 1b serves to illustrate these assertions. In the case of the hybrid implementation, utilizing MPI and OpenMP parallelization, one can see that the overhead of distributing the system matrix A and the final collection of the preconditioner onto the main process inhibits scaling of the method. Increase of the tolerances beyond 0.125 is seldom useful since the execution time becomes dominated by communication and memory management overhead even for small matrices and few processes/GPUs.

mSGD Parameters The missing element probability and learning rate parameters of mSGD are unknown and hard to estimate *a-priori*. To elucidate the sensitivity of the method to the choice of these parameters a parameter search has been performed. To this end $r = 0.5$ and $T = N_{total}/100$, with $N_{total} = 10^5$ have been chosen and c, p varied. The results of the parameter search are shown in fig. 2. They indicate the existence of an optimal choice of parameters with a bias towards higher values for the missing probability p. The choice of an optimal parameter set will not, however, improve the convergence and errors dramatically. As can be seen in the figure the best reduction of the error is achieved for the `rdb2048` matrix and is $\propto 3\times$. It is also interesting to observe, that in contrast to the non-symmetric matrices of fig. 2 the optimal parameter set for the symmetric discrete Laplacian is located towards higher learning rates. This



(a) `nonsym_r3_a11` matrix. (b) `rdb2048` matrix.

Fig. 2: Dependency of the error of mSGD on the chosen parameters.

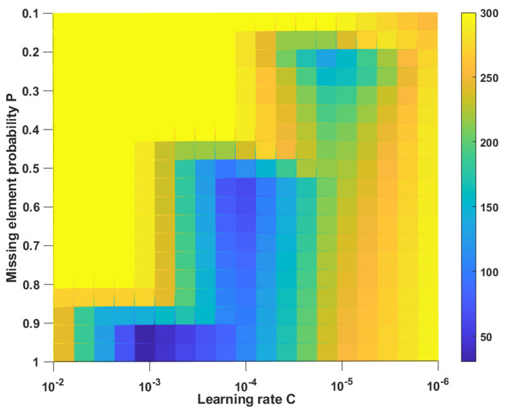
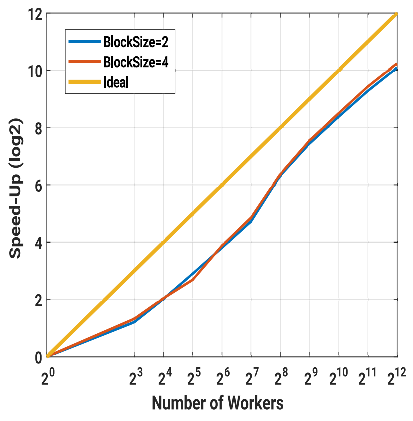


Fig. 3: Dependency of the error of mSGD on the chosen parameters for `2DFDLaplace_20x20` matrix.

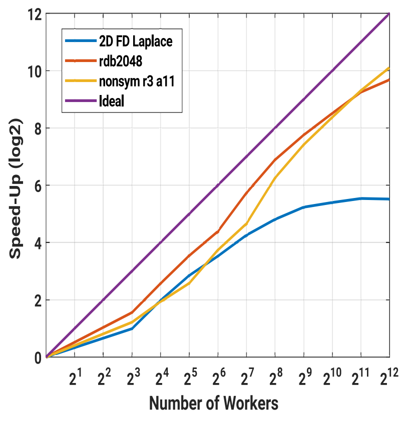
holds, too, for the large `sym_r6_a11` matrix, too, and suggests the possibility that the learning rate may be influenced by the symmetry of the matrix.

Scalability For mSGD and SP the scaling behaviour across multiple nodes using hybrid parallelization is illustrated in fig. 4. The recline of the speed-up for the smallest matrix of the set is to be expected since each thread/worker cannot receive less than one column of the approximate inverse to process. Processes that do not receive any columns remain idle but participate in the initial matrix broadcast, hence slowing the process down.

Increasing the size of the matrix leads to a better speed-up, as can be seen for the case of the `nonsym_r3_a11` matrix. Comparing the speed-up for `nonsym_r3_a11` to that of `rdb2048` we can see, that the latter reclines markedly



(a) SP for `nonsym_r3_a11` matrix.



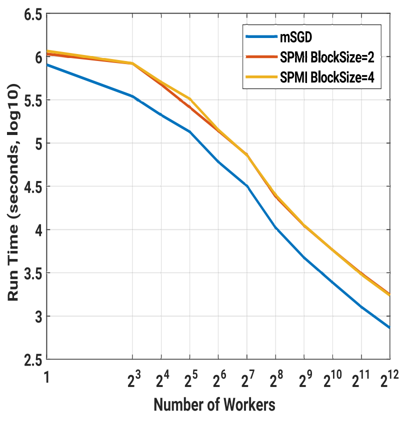
(b) mSGD

Fig. 4: Speed-Up achieved by both methods for different matrices of the chosen matrix set. For the small Laplace matrix one can clearly observe a saturation. Note the logarithmic scale of the axes.

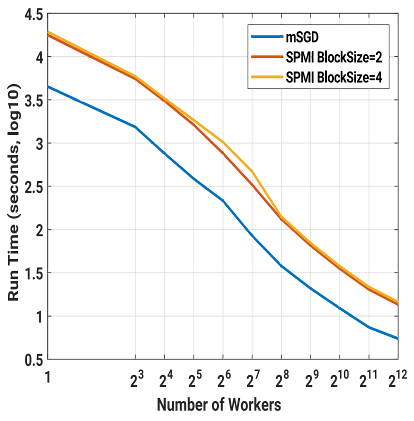
around 2^{11} workers, which corresponds to the point where each worker has to process one column only. The non-vanishing speed-up for the case where the number of workers is larger than the number of columns of the matrix is startling, but could possibly be attributed to a wider distribution of the MPI processes on the machine, thereby relieving the pressure on compute resources such as caches and memory bandwidth. The tendency of the speed-up curve to be convex at around 2^7 workers (threads) we attribute to caching effects.

It is instructive to compare the achieved speed-up and execution times of these methods to the speed-up of MCMCMI which can be easily inferred from fig. 1a. For a relatively small matrix of dimension ~ 20000 the MCMCMI implementation does not scale well. The reason for this is the memory management imposed by the use of sparse matrices and the fact that even at a relatively high precision of $\epsilon = 0.0625$ the Markov chains are still relatively short and hence the overall computational cost is still modest. In fact, if one compares the execution times provided in figs. 1a, 5 one can immediately see, that although MCMCMI does not scale well it clearly has the advantage with regards to the practical requirement of short run times.

Error Convergence As can be observed in fig. 6 usage of the rough approximate inverse provided by MCMCMI reduces the error of both SP and mSGD significantly. Unexpectedly it appears to have no effect on the behaviour of the error for SP. This suggests a certain agnosticism of the method with regards to the starting vector/matrix for the iteration. The similarity of the behaviour of mSGD and SP is misleading in the case of the `nonsym_r3_a11` matrix, since for mSGD the step-size was determined using (4) with the quantities provided in



(a) `nonsym_r3_a11` matrix.



(b) `rdb2048` matrix.

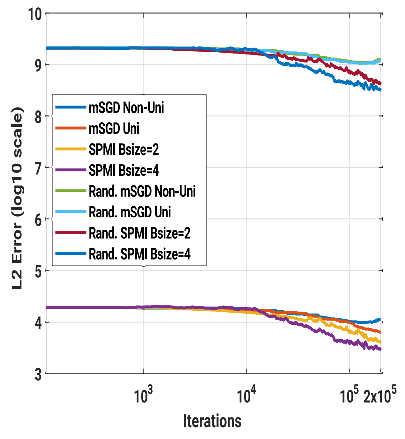
Fig. 5: Total execution times of the different methods.

sec. 3.2. One can see that the mSGD iteration starts to converge roughly after 10^3 steps, which corresponds the point at which $\frac{k}{T} = \frac{k}{10^3}$ becomes larger than one and by merit of $0 < r < 1$ the step-size shrinks. Indeed, if one considers the error behaviour for `rdb2048` it becomes apparent that it begins to converge only after the exponential becomes > 1.

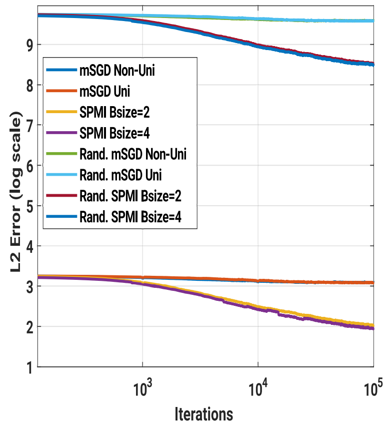
The case of `nonsym_r3_a11` matrix presents also the rare case where SP with a block size of 2×2 provides a better convergence in the early stages of the iteration, than a larger block size. Furthermore it can be seen, that in this case using a better $\boldsymbol{x}_0$ in conjunction with a non-uniform selection probability for the rows of the matrix in (3) leads to an unexpected worsening of the computed error.

Although SP outperforms mSGD with regards to the error convergence it is slower than the latter. To a large part this is due to the fact that in the current implementation of SP the timings include the error computations, which are not strictly necessary. Furthermore, SP requires the computation of the inverse of a small $m \times m,\ m \in \{1, 2, 3, 4\}$ matrix at each step. This cost, however, can be neglected currently, since it is dwarfed by the error computations. To reduce the impact of this inversion the block sizes were chosen such that the usage of an explicit expression for the calculation of the inverse of the block by the used library[3]. The dominant factor at the current development stage will be the memory management, though.

[3] The Eigen 3 template library.



(a) `nonsym_r3_a11` matrix



(b) `rdb2048` matrix

Fig. 6: Errors achieved by SP and mSGD for different matrices using a random matrix as initial condition, or an approximate inverse provided by MCMCMI.

4 Conclusions and Future Work

In summary we observe that in the present state of development neither of these methods is capable of competing against, for instance, MCMCMI as a method to compute an approximate inverse to be used as a preconditioner efficiently. This is chiefly due to the excessive execution times and slow error convergence. One has to note that the parameters of SP and mSGD have not been explored in full, and the analysis currently under way suggests that some sizeable improvements can be made.

The error behaviour of SP observed for all but the `nonsym_r3_a11` matrix suggests that a sequence of diminishing block sizes might be effective in accelerating the convergence of this method. This will be equivalent to successive refinement of projection spaces down to the one-dimensional solution space.

For mSGD the obvious point of optimization will be the choice of the parameters for (4), especially of T - as elaborated upon in the previous section. Especially the connection of the optimal step-size to the symmetry properties of the matrix are deserving of a particular focus, since this simple to check property would provide a simple criterion for the selection of the learning rate.

All three methods described above currently suffer the same deficiency - the need to broadcast the system matrix prior to the execution of the method. The necessary broadcast is the dominant factor determining the execution time of the MCMCMI process but is sub-dominant for mSGD and SP, since both are more expensive in their application. In principle none of the methods strictly require a matrix to be available and all should work in conjunction with matrix-free problems.

Multiple such tests are planned for the near future and will serve to assess the usefulness of the methods as preconditioners in the domain of electrodynamics and plasma physics. Furthermore, based on the observation that the lengths of empirical Markov Chains are much shorter than predicted by theory [1], we expect the need for further formal analysis of the MCMCMI method to provide bounds that are more precise.

References

1. Alexandrov, V.N.: Efficient parallel Monte Carlo methods for matrix computations. Math. Comput. Simul (47), 113–122 (1998)
2. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software (TOMS) **38**(1), 1 (2011), https://sparse.tamu.edu/
3. Dimov, I., Alexandrov, V.: A New Highly Convergent Monte Carlo Method for Matrix Computations. Mathematics and Computers in Simulation **47**(2-5), 165–181 (Aug 1998)
4. Lebedev, A., Alexandrov, V.: 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. Ma, A., Needell, D.: Stochastic Gradient Descent for Linear Systems with Missing Data. online (Feb 2017), http://arxiv.org/pdf/1702.07098v4
6. Sabelfeld, K., Loshchina, N.: Stochastic iterative projection methods for large linear systems. Monte Carlo Methods and Applications (2010). https://doi.org/10.1515/mcma.2010.020