Two-stage Asynchronous Iterative Solvers for multi-GPU Clusters

Pratik Nayak
Karlsruhe Institute of Technology
Germany
pratik.nayak@kit.edu

Terry Cojean
Karlsruhe Institute of Technology
Germany
terry.cojean@kit.edu

Hartwig Anzt
Karlsruhe Institute of Technology
Germany
hartwig.anzt@kit.edu

Abstract—**Given the trend of supercomputers accumulating much of their compute power in GPU accelerators composed of thousands of cores and operating in streaming mode, global synchronization points become a bottleneck, severely confining the performance of applications. In consequence, asynchronous methods breaking up the bulk-synchronous programming model are becoming increasingly attractive. In this paper, we study a GPU-focused asynchronous version of the Restricted Additive Schwarz (RAS) method that employs preconditioned Krylov subspace methods as subdomain solvers. We analyze the method for various parameters such as local solver tolerance and iteration counts. Leveraging the multi-GPU architecture on Summit, we show that these two-stage methods are more memory and time efficient than asynchronous RAS using direct solvers. We also demonstrate the superiority over synchronous counterparts, and present results using one-sided CUDA-aware MPI on up to 36 NVIDIA V100 GPUs.**

Index Terms—**Asynchronous iterative methods, Schwarz methods, GPUs, Krylov subspace solvers.**

I. Introduction

One of the main challenges in the exascale era is the need to move beyond the Bulk-Synchronous Programming (BSP) model towards asynchronous models to mitigate the effects of global synchronization points in heterogeneous clusters. This is reflected in the US Department of Energy listing the removal of synchronization bottlenecks as one of the programming challenges for the exascale era [1]. Many of the latest leadership HPC systems are not only heterogeneous but also heavily imbalanced towards accelerators. For example, each node of Oak Ridge's Summit supercomputer composes 2 POWER9 22-core CPUs and 6 Nvidia Tesla V100 GPUs, and the system's 27,648 GPUs (in 4,608 nodes) contribute to 98% of Summit's compute power.

In consequence, new algorithms should be designed keeping asynchronicity and fault tolerance in mind, while making sure that the highly-parallel nature of the accelerators is harnessed efficiently. From a linear solver perspective, there are two possible approaches. The first approach involves improving the load balancing of the algorithms and moving to more asynchronous programming techniques such as task-based approaches. This involves breaking up the algorithm into granular tasks and intelligently scheduling them. The second approach is to use asynchronous iterative methods. While the first approach provides some performance improvements, it may fail for some algorithms such as the traditional iterative methods that compute in a lock-step fashion and have synchronization points boiled into the algorithm. Asynchronous iterative methods, on the other hand, do not operate in a lock-step fashion, but instead allow for using the available data without explicitly synchronizing all processes at every iteration. This removes the bulk-synchronous character of the algorithm and allows us to move to a more asynchronous model where each process trades the global synchronization against the price of possibly computing on old data. Effectively, asynchronous iterative methods allow to trade computation cost for synchronization cost.

Previous works [2] (and references therein) have explored the benefits asynchronous iterative meth-

978-1-6654-2270-3/20/$31.00 ©2020 IEEE
DOI 10.1109/ScalA51936.2020.00007

ods can provide in terms of runtime performance, and have proved their convergence from a theoretical perspective. Schwarz methods are domain decomposition methods and their asynchronous variants have been studied quite extensively [3](and references therein). Yamazaki et al. [4] study an asynchronous variant of the Schwarz method, the Optimized Restricted Additive Schwarz solver for a well-balanced Laplace 2D problem on multicore CPU systems. Nonetheless, few production-ready implementations for HPC architectures and comprehensive investigations of their performance exist, in particular for machines featuring accelerators such as the Summit HPC system. Previous works [4] use direct solvers to solve the local subdomain problems. With generic problems, this is no longer efficient as the matrices no longer have a banded structure, which renders the factorization cost-prohibitive. Moreover, it is unnecessary to solve the local problems very accurately, particularly in the early domain updates. Hence, in this paper, we extend our Restricted Additive Schwarz (RAS) framework [5], which uses local direct solvers, to use iterative solvers for the local subdomain problems and show that these multi-stage iterative methods have significant advantages over their single stage counterparts. Up to our knowledge, this is novel work, and we hence highlight the following contributions:

- We extend our previous open-source framework, adding iterative solvers with preconditioners for the local problems on GPUs.
- We show that multi-stage asynchronous iterative methods can provide significant performance improvements over synchronous methods and over the local direct solvers on modern GPU-centric HPC systems. In particular, we show that iterative methods adapt well to general problems where the direct solvers become too cost-prohibitive.
- We study the effects of different local iterative solvers, preconditioners, local subdomain solver tolerances, and iteration counts on two different problems that are representative of actual simulations. We show that also when using iterative local solvers, it is cost-prohibitive to solve the subdomain problems to high accuracy.

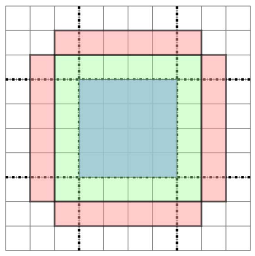


Figure 1: Decomposed 2-D grid with partitioning and overlap. Inspired by Yamazaki et al. [4]

In Section II, we first provide some background on the traditional Restricted Additive Schwarz method, asynchronous iterative methods, and the implications of the non-reproducible execution in asynchronous Additive Schwarz. In Section III, we describe the new algorithm framework with its capabilities, and the settings we use in the experimental evaluation in Section IV. We conclude in Section V.

II. Background

A. *Schwarz methods.*

Schwarz methods are a class of domain decomposition methods that were initially used as a theoretical tool to show the existence of solutions for the Laplace equation through the alternating method. Since then, many variants of the method have found their use, as iterative methods, e.g., [3] (and references therein), and as preconditioners, e.g., [6].

The general idea of domain decomposition (DD) solvers is to decompose the domain into distinct subsets so that the local solution for each of those subdomains can be computed in parallel. To that end, it is necessary to exchange data from each other to converge towards a global solution in an iterative process.

Consider a two-dimensional grid as shown in Figure 1. The dotted grid lines show the partitioning of the mesh into a 3x3 grid. The interior of one domain is shown in blue. If all cells are part of only one subdomain (i.e., no cells are part of two adjacent subdomains), the DD method is called a non-overlapping decomposition. If, for example, the subdomain owning the blue cells includes also the green cells as variables in the solution of the local problem, then the DD method is called an overlapping domain decomposition solver, in this case with overlap of $\gamma = 1$. In this setting, the cells marked red are the external interface points through which the information is exchanged, also known as the halo points.

In an overlapping Schwarz method, the values for the cells in the overlap are computed by several domains, and to form a unique global solution, the different locally-computed values need to be weighted [7]. With a modification proposed by Cai and Sarkis [8], the cells in the overlap are still considered as variables when computing the local solution, but discarded afterward. This modification – called Restricted Additive Schwarz (RAS) – converges faster [8] with the advantage of removing any need for weighting overlap solutions, therefore allowing a collision-free implementation on parallel computing architectures.

In a synchronous Schwarz solver, the iterations proceed in a lock-step fashion: the domains are decomposed as presented above, local solves are performed in parallel, each subdomain sends the required data to its "neighbors" and all the subdomains wait until data has been received from their respective neighbors. Global convergence is achieved when the global solution – computed after all domains solved the local problems – satisfies a pre-defined convergence criteria.

An asynchronous Schwarz solver removes the explicit synchronizations separating the iterations [3]. Instead, each subdomain proceeds by solving the local problem using the latest data it received from neighboring domains. This has several important implications: 1) there is no longer a concept of a single "global iteration count" as the distinct subdomains can differ in how many local updates they have completed; 2) local updates can potentially re-compute the same solution in case no new data was received; 3) the asynchronous Schwarz convergence and performance is not reproducible as – theoretically – each algorithm execution results in a different subdomain update order and 4) the lack of global synchronization points require alternative strategies for detecting convergence of the algorithm. Convergence for the asynchronous Schwarz methods and their variants have been shown, for example in [3] (and references therein).

B. *Local solvers*

As local solvers, we can either use a direct solver after computing a Cholesky/LU factorization of the matrix, or use iterative solvers such as Krylov subspace methods, for example CG/GMRES with or without preconditioners.

Direct solvers: For a symmetric positive definite (SPD) matrix, a direct solver with Cholesky factorization is the general choice. With a constant matrix, direct methods are attractive as we need to compute the expensive factorization only once and can use the relatively cheap triangular solves to solve the systems in the successive domain updates.

Iterative solvers: For a symmetric positive definite (SPD) matrix, the most widely used iterative solver is the preconditioned Conjugate Gradient (PCG) method. For non-SPD systems, GMRES/BiCGSTAB methods are popular.

To minimize storage requirements, when using a direct solver, it is imperative to reorder the matrix using reordering methods such as METIS fill-reduce, Approximate Minimum degree or others as elaborated within CHOLMOD [9]. With unbalanced matrices and matrices with large bandwidths, the resulting fill-in can be prohibitive. This is not the case for iterative solvers. Additionally, iterative subdomain solves come with the possibility to control the local solution accuracy in favor of reduced subdomain solver cost. This is particularly attractive as the local solution usually changes only marginally between consecutive global updates, and a few local solver iterations are often sufficient to update the local solutions.

III. Implementation

The testbed framework we developed for asynchronous iterative methods and published previously [5] is written in C++ and features a modern C++ design. The framework is designed for customization and extension, and features multiple options which can be tweaked or easily added to explore the various techniques and concepts in the context of asynchronous iterative methods.

At the core of the framework is the algorithm detailed in Section III-A. This algorithm is generic enough to be reused for other asynchronous iterative methods since it only calls different subcomponents, which are the `Initialize` interface detailed in Section III-B and the `Solve` interface detailed in Section III-C. The `Communicate` interface, used in both the `Initialize` and the `Solve` step, supports MPI one-sided and two-sided communication and is detailed in Section III-D.

The framework we developed interacts with external tools to provide specific functionalities, among which are `metis` for partitioning, DEAL.II [10] for problem generation, and GINKGO [11] for subdomain solvers and memory management features as presented in Section III-E.

A. Core Algorithm

Algorithm 1 shows the Schwarz framework. The initialization step handles the setup and initialization of the solver, while the solve step handles the local solves, communication, the convergence detection, and the termination of the solver. The performance data we report in Section IV account only for the solve step as the initialization and setup is a constant, non-repeating part of the RAS solver.

Algorithm 1 Schwarz Iterative solver

```
1:  procedure ITERATIVE_SOLUTION($A, x, b$)
2:      procedure INITIALIZATION_AND_SETUP
3:          Partition matrix
4:          Distribute data
5:          Initialize data
6:      procedure SOLVE
7:          while $iter < max\_iter$ or until convergence do
8:              Locally solve the matrix
9:              Exchange boundary information
10:             Update boundary information
11:             Check for Convergence
12:         Gather the final solution vector
```

B. Initialization and setup

The initialization and setup step consists of three main parts: 1. the generation and partitioning of the global system matrix; 2. the distribution/generation of the local subdomain matrices and right hand sides; and 3. the setup of the communication – which is detailed in Section III-D as it is a separate component impacting both the initialization and the solve characteristics.

1) Local subdomain matrices: The partitioning scheme assigns each grid point to one ore more subdomains (depending on the overlap setting). We use the `metis` partitioning as it is the most generic type of partitioning and performs well for different matrices. Using this information, the local subdomain and interface matrices are assembled. The latter is used to communicate between the different subdomains using an SpMV formulation.

We store all the matrices in the CSR format, which is a popular matrix format for handling sparse matrices, and which has been shown to perform well for a given generic matrix [12].

C. Solving

1) Local solution: Each subdomain owns its local matrix and right-hand side, and can thus independently and without synchronization with the other subdomains solve the local problem. In this paper, we focus on preconditioned Krylov solvers to compute the local subdomain solution. For SPD matrices, we use the Conjugate Gradient method and for non-SPD matrices we use the GMRES method. All solvers are taken from the GINKGO library and have backends for GPUs. For the iterative local solves, we use two preconditioner variants, a block-Jacobi preconditioner based on (block-) diagonal scaling, and an ILU preconditioner with the ILU factors being generated through the PARILU algorithm [13].

We compare the setting based on iterative subdomain solves with a version using direct subdomain solves. For the latter, we use factorizations from the well-known CHOLMOD library [9] for the Cholesky factorization (SPD problems) and the UMFPACK library for the LU factorizations. As we have constant factors, we pre-compute our factors on the CPU in the setup stage and transfer the factors to the GPU. For the triangular solves, we use the cuSPARSE level-scheduled triangular solver which has been shown to be very efficient [14].

The local subdomain update count refers to the traditional "global iteration" count of the RAS solver. Local iteration counts refer to the number of iterations performed by the local iterative solver in the subdomain.

2) Convergence detection: The framework we develop supports two convergence detection strategies: a centralized (tree-based) convergence detection mechanism and a decentralized strategy. We have studied the effects of convergence detection in a previous paper [5]. In this paper, we use the decentralized convergence algorithm as it has been shown to be scalable and better performing than the centralized algorithms [5].

D. Communication

A subdomain refers to the computational unit that performs a local solve and communicates the required data to its "neighbors". We use the CUDA-aware MPI to directly transfer the buffers between GPUs of different subdomains instead of intermediately staging them on the CPU. This allows for faster communication as the latencies from the CPU side are hidden (particularly for GPUs attached to the same socket and communicating via the CUDA NVLINK technology). We associate each subdomain to one MPI rank.

Synchronous Schwarz setup: In the synchronous version, each step in the SOLVE procedure in Algorithm 1 is performed in a lock-step fashion.

Asynchronous Schwarz setup: In the asynchronous version, each step in the SOLVE procedure in Algorithm 1 is executed without synchronizing with neighbors. This asynchronous communication is enabled through the RMA functions of the MPI-2 standard. More details regarding the implementation can be obtained from our previous paper [5] and from our Open source implementation[1].

[1]https://github.com/pratikvn/schwarz-lib

E. The GINKGO *framework*

GINKGO is a node-level high performance sparse linear algebra library featuring kernels for backends such as multicore machines (OpenMP), AMD GPUs (HIP) and NVIDIA GPUs (CUDA). GINKGO provides the user with simple and powerful linear operator abstraction to interface various solvers, preconditioners, and matrix operations. In the experimental analysis, we use GINKGO objects for all arrays, matrices, and vectors. This enables us to leverage GINKGO's executor concept to easily move data from one hardware architecture to another without code other than using another execution space. As we are focusing on the architecture of the Summit supercomputer, we run on the V100 GPUs using GINKGO's CUDA executor and the multicore executor for copies involving the host CPUs.

IV. EXPERIMENTAL ASSESSMENT

A. Hardware and software setup

In the experimental evaluation, we focus on two problems. The first problem we focus on is a Laplacian problem (2nd order FE) where we use a preconditioned CG method as the local solver. The second problem we focus on is an advection problem (5th order FE) where we employ a preconditioned GMRES method as the local solver. For all cases, we iterate on the RAS level until the global relative resnorm has dropped below a value of $1e-12$ and the overlap is set to 8. The local subdomain solver criterion in this paper refers to the relative norm *reduction*, where we aim to reduce the current resnorm of the current local solution and not the absolute resnorm.

All experimental evaluation is realized on the Summit supercomputer at the Oak Ridge National Lab. Each node composes of two IBM Power 9 CPUs and 6 NVIDIA Tesla V100 GPUs. The 3 GPUs connected to the same socket are directly connected by NVLINK bricks with a 50GB/s bidirectional bandwidth. Each V100 features 16GB of High Bandwidth Memory (HBM2) which is the on-device memory. The nodes are connected via a dual-rail EDR InfiniBand network (non-blocking fat-tree) providing a bandwidth of 23GB/s.

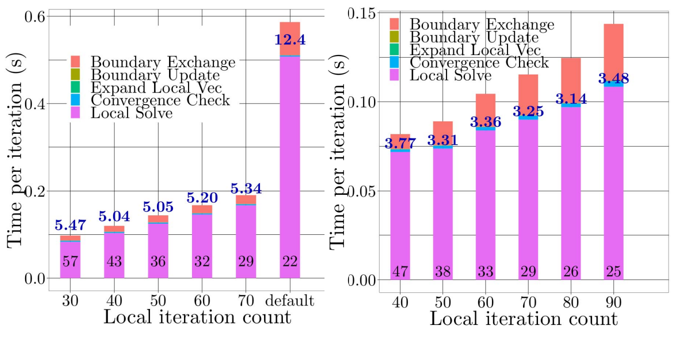
This paper's codebase uses GINKGO as a central building block for the Schwarz solver. Overall, local on-node (on-GPU) operations use GINKGO functionality, while the inter-node and inter-GPU communication is handled via MPI. As we perform our experiments on the Summit system, we use the IBM Spectrum MPI library (a flavor of Open-MPI). Related work by Yamazaki et al [4] analyzes the impact of using different MPI implementations including the performance of the one-sided RMA functions.

Investigating asynchronous iterative methods can be challenging for several reasons. First, by the nature of asynchronicity, the method is non-deterministic, which means that we are generally unable to predict or reproduce results and effects. Therefore, we can only report statistical data and draw weak assumptions. We acknowledge this aspect by averaging all data presented in this paper over ten runs. Second, a global iteration count does not bear any meaning. Removing explicit synchronization points means that some solver parts (i.e. subdomains) may have already completed a high number of local updates while other parts did not yet complete a single local update. To reflect this challenge, we base all performance data we report on the asynchronous Schwarz on averaged global iterations or the time-to-solution metric. This is the total time until global convergence is detected – which brings us to a third challenge: without global synchronization points, it is difficult to detect global convergence. The framework we develop supports both centralized and decentralized convergence detection, but due to better performance and scalability of the decentralized version, we use it for all our experiments.

B. The Laplacian problem

Figure 2(a) visualizes the runtime needed for one synchronous Schwarz iteration using two-sided MPI communication and a block-Jacobi preconditioned CG local solver averaged over all the subdomains (problem with 2 million unknowns divided into 6 subdomains.). The local solve dominates the overall runtime. As we increase the local solver iteration limit (max-iter count), the global RAS solver needs more domain updates to converge (as shown in the inset domain update counts at the bottom of the bars). At the same time, the local solver performs fewer iterations, and hence the overall local solver cost decreases. We see that for this configuration, the overall time-to-solution is minimized for a local solver iteration limit of 40.

Figure 3(a) shows the time to solution for the different preconditioners (problem size of 2 million unknowns divided over 6 subdomains). We observe that independent of the preconditioner choice, the asynchronous version completes faster than the synchronous version. The default setting uses a local stopping criterion of a relative residual threshold of $1e-6$. We observe that the block-Jacobi preconditioner is the most effective choice for this problem, likely due to its low application cost. Using a block-Jacobi preconditioned CG as local solver, the asynchronous version outperforms the synchronous version by about $1.25\times$.



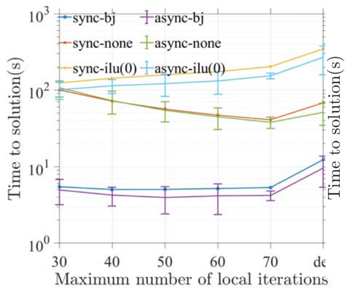
(a) (b)

Figure 2: Studying the problem with a preconditioned Krylov local solver. The runtime breakdown in the different functions for different local solver maximum iteration count. (Inset: Bottom of bars: no. of subdomain updates, above the bars: total time to solution in secs.) (a) Laplacian problem, (b) Advection problem.

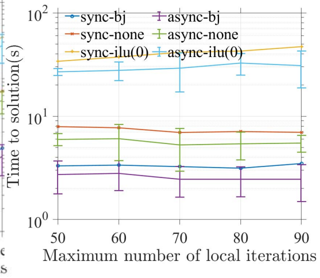
C. The Advection problem

Similar to Figure 2(a), Figure 2(b) visualizes the runtime needed for one synchronous RAS iteration using two-sided MPI communication with a block-Jacobi preconditioned GMRES as local solver. The runtimes are again averaged over all 6 subdomains, each handling a subset of the 1.7 million unknowns. The default case (with a tolerance of 1e-3) needs about 729s to converge and is omitted in the graph for readability. Furthermore, the system matrix is much denser than the Laplacian case, and hence the global solver spends more time in the boundary exchange routine.

Figure 3(b) compares the performance of different preconditioners. Though we expect the ILU preconditioner to perform better for the non-symmetric system matrix, we observe that the high preconditioner quality does not compensate for the expensive sparse triangular solves in the ILU preconditioner application. In consequence, the block-Jacobi is again the most efficient preconditioner. We also observe that the asynchronous version outperforms the synchronous counterpart in all settings, attaining a maximum speedup of 1.4×.



(a)

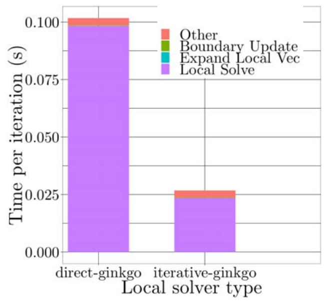


(b)

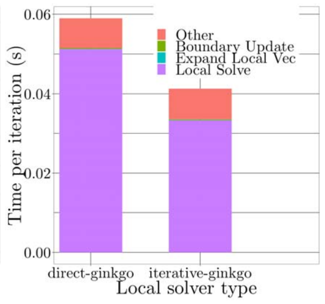
Figure 3: The time to solution for the different preconditioners over different local solver maximum iteration counts (a) Laplacian problem, (b) Advection problem.

D. Comparison with a local direct solver

In Figure 4(a), we assess the performance of the asynchronous RAS solver using either a direct local solver or an iterative local solver. For the direct local solver, we initially compute the Cholesky factorization of the local system matrix and apply triangular solves in every subdomain update. For the iterative local solver, we choose a Conjugate Gradient solver (CG) preconditioned with block-Jacobi. The global problem is composed of 2e5 unknowns partitioned into 6 subdomains. The asynchronous RAS with a direct subdomain solver needs on average 21 subdomain updates and completes after 2.05s. Using 60 iteration of the preconditioned CG, the local solves are less accurate, and the average number of subdomain updates increases to 27. However, while the communication cost of every subdomain update remains unaffected of the solver choice, the iterative local solver completes (on average) 4× faster than the direct local solver. In the end, the asynchronous RAS using the iterative local solver completes in 0.63s, which is about 3.25× faster than the asynchronous RAS using the direct local solver. Similar behavior is observed for the advection problem in Figure 4(b). The preconditioned GMRES is more expensive than the PCG, but still about 1.54× faster per subdomain update than the direct solve and overall the asynchronous solver completes in 0.73s while the synchronous version takes 1.05s which is about 1.4× slower.



(a)



(b)

Figure 4: Runtime breakdown of a single subdomain update (averaged) for the asynchronous RAS solver (direct v/s iterative local solvers) (a) Laplacian problem (b) Advection problem.

E. Analysis of the residual reduction.

Figure 5 shows the reduction of the residual norm for the 36 subdomains and the number of local iterations the local solver performs in each of the global subdomain updates. Subdomains that are "slow" (subd-31, bottom figure) have relatively more expensive local solves than the ones that are faster (subd-0, top figure). An asynchronous solver allows the faster subdomains to optimize

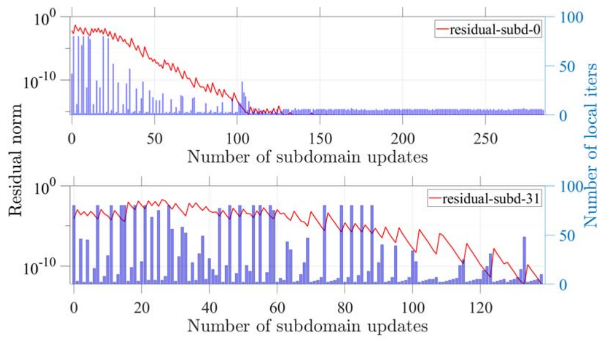


Figure 5: Reduction of the residual norm in the subdomains and the number of local iterations per subdomain with a max-iter count of 80 for the Advection problem.

their local solves by reducing the iteration limit if they have received less data from their neighbors and are computing on old data. This is reflected in the jumps of the residual norm. From a runtime perspective, when subdomain 0 is performing its 85th global update, subdomain 31 is on its 50th update. This local solver optimization in combination with the asynchronous communication allows the asynchronous RAS solver to outperform the synchronous version.

V. Conclusion and Future work

In this paper, we employ preconditioned Krylov methods as iterative subdomain solvers in an asynchronous Restricted Additive Schwarz (RAS) solver designed for multi-GPU cluster. For SPD and non-SPD problems, we analyze the effect of the subdomain solver iteration limit, and preconditioner type, and compare convergence and performance against a synchronous RAS method and an asynchronous RAS using a direct subdomain solver. In the performance study using up to 36 NVIDIA V100 GPUs, we demonstrate that the asynchronous RAS solver can for optimal parameter choices complete up to 1.4× faster than its synchronous counterpart and more than 3.25× faster than the asynchronous RAS solver using local direct solvers.

References

[1] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balanji *et al.*, "Exascale programming challenges," in *Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA. US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR)*, 2011.

[2] A. Frommer and D. B. Szyld, "On asynchronous iterations," *Journal of Computational and Applied Mathematics*, 2000.

[3] F. Magoulès, D. B. Szyld, and C. Venet, "Asynchronous optimized schwarz methods with and without overlap," *Numerische Mathematik*, vol. 137, no. 1, pp. 199–227, Sep 2017. [Online]. Available: https://doi.org/10.1007/s00211-017-0872-z

[4] I. Yamazaki, E. Chow, A. Bouteiller, and J. Dongarra, "Performance of asynchronous optimized Schwarz with one-sided communication," *Parallel Computing*, vol. 86, pp. 66–81, 2019. [Online]. Available: https://doi.org/10.1016/j.parco.2019.05.004

[5] P. Nayak, T. Cojean, and H. Anzt, "Evaluating asynchronous schwarz solvers on GPUs," *The International Journal of High Performance Computing Applications*, vol. 0, no. 0, p. 1094342020946814, 0. [Online]. Available: https://doi.org/10.1177/1094342020946814

[6] Z. Liu and Y. He, "Restricted additive schwarz preconditioner for elliptic equations with jump coefficients," *Advances in Applied Mathematics and Mechanics*, vol. 8, no. 6, pp. 1072–1083, 2016.

[7] A. Frommer and D. B. Szyld, "Weighted max norms, splittings, and overlapping additive Schwarz iterations," *Numerische Mathematik*, vol. 83, pp. 259–278, 1999.

[8] X. C. Cai and M. Sarkis, "Restricted additive Schwarz preconditioner for general sparse linear systems," *SIAM Journal of Scientific Computing*, 1999.

[9] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate," *ACM Trans. Math. Softw.*, vol. 35, no. 3, Oct. 2008. [Online]. Available: https://doi.org/10.1145/1391989.1391995

[10] G. Alzetta, D. Arndt, W. Bangerth, V. Boddu, B. Brands, D. Davydov, R. Gassmöller, T. Heister, L. Heltai, K. Kormann, M. Kronbichler, M. Maier, J. P. Pelteret, B. Turcksin, and D. Wells, "The deal.II library, Version 9.0," *Journal of Numerical Mathematics*, 2018.

[11] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing," *arXiv e-prints*, p. arXiv:2006.16852, Jun. 2020.

[12] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang, "Load-balancing sparse matrix vector product kernels on gpus," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3380930

[13] E. Chow and A. Patel, "Fine-grained parallel incomplete lu factorization," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015. [Online]. Available: https://doi.org/10.1137/140968896

[14] M. Naumov, "Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU," NVIDIA Technical Report NVR-2011-001, Tech. Rep., 2011.

Artifact Description

In order to ensure reproducibility of results, we provide the code used to obtain the results in this paper and elaborate on the settings and parameters used to produce these results.

Building instructions

The source code is open-source and available on github (https://github.com/pratikvn/schwarz-lib). The code is documented through doxygen and available online (https://pratikvn.github.io/schwarz-lib/doc/develop/index.html)

To build `schwarz-lib`, the following components are necessary:

1) An MPI implementation (OpenMPI, MPICH, Spectrum-MPI etc.)
2) GINKGO: Linear operator algebra library for the general framework.

Additionally, these components are necessary to reproduce the results in this paper:

1) DEAL.II For running the laplacian and advection benchmarks.
2) SUITESPARSE: For the direct factorizations for the local solvers.
3) METIS: For the matrix partitioning and domain decomposition.

Dependencies

The following describes the dependencies and the instructions to install them.

1) GINKGO*:* Detailed instructions on how to build Ginkgo are provided in the GINKGO documentation in the README.md and the installation page (INSTALL.md). The `expt-develop` branch needs to checked out as schwarz-lib depends on that branch of GINKGO.

To run the code on multiple GPU's, GINKGO needs to be compiled with the CMake option `GINKGO_BUILD_CUDA=on`.

2) DEAL.II*:* To run the laplacian and advection examples, you need to compile DEAL.II from the fork https://github.com/pratikvn/dealii using the branch `gko-expt-develop`.

Instructions to compile DEAL.II are available on the DEAL.II website, https://www.dealii.org/. As we do not use any CUDA implementations from dealii, the CMake option `DEAL_II_WITH_CUDA` can safely be switch off. To make sure there are no duplicate MPI definitions from DEAL.II, DEAL.II needs to be compiled with the CMake option `DEAL_II_WITH_MPI=ON`. Additionally, to make sure everything with Ginkgo works smoothly, DEAL.II needs to be built with GINKGO support using the CMake option `DEALII_WITH_GINKGO=on` and the installation path of GINKGO provided to DEAL.II with the CMake option `GINKGO_DIR`.

3) Suitesparse - CHOLMOD and UMFPACK: To run the direct solver comparisons, CHOLMOD and UMFPACK of the Suitesparse collections need to be installed. Installation instructions can be found in the README of the github repository, https://github.com/DrTimothyAldenDavis/SuiteSparse.

4) METIS*:* To partition the global matrix, it is necessary to install a partitioner. We use METIS http://glaros.dtc.umn.edu/gkhome/metis/metis/overview. We also provide two other naive partitioning for experimental purposes, the `regular-1d` and the `regular-2d`, if the user cannot install METIS for some reason.

Schwarz library

Finally, `schwarz-lib` can be compiled with the previous dependencies as necessary. To reproduce the results in this paper, it is necessary to use the CMake options `SCHWARZ_BUILD_DEALII=on`, `SCHWARZ_BUILD_CUDA=on`, `SCHWARZ_BUILD_CHOLMOD=on` `SCHWARZ_BUILD_UMFPACK=on` and `SCHWARZ_BUILD_METIS=on`. To compile the benchmarks, it is necessary to also use the CMake option `SCHWARZ_BUILD_BENCHMARKING=on`. The installation paths of the respective libraries need to be provided in the respective CMake options `LIBNAME_DIR` as elaborated in the installation documentation https://pratikvn.github.io/schwarz-lib/doc/develop/install_schwarz.html.

Benchmarking

The different flags available are explained on the benchmarking documentation page:

https://pratikvn.github.io/schwarz-lib/doc/develop/benchmarking_schwarz.html.To reproduce the results in this paper, the following flags have to be set:

1) Generic settings:
   a) Compile GINKGO, DEAL.II and SCHWARZ-LIB in Release mode.
   b) Set GINKGO executor to CUDA:
      i) `--executor=cuda`
   c) Set overlap to 8:
      i) `--overlap=8`
   d) Set partition to METIS:
      i) `--partition=metis`
   e) Set matrix type to non-symmetric for advection problem. [PARAMETER]:
      i) `--non_symmetric_matrix`
2) Solver settings:
   a) Generic settings:
      i) Set the maximum iteration count to a high value:
         A) `--num_iters=2000`
      ii) Set global solver tolerance to 1e-12.
         A) `--set_tol=1e-12`
      iii) Set local solver tolerance to [PARAMETER]:
         A) `--local_tol=1e-3`
   b) Communication settings:
      i) Enable MPI onesided or twosided as required [PARAMETER]:
         A) `--enable_onesided=true/false`
      ii) Set the MPI onesided remote communication type to get [PLATFORM-DEP]:
         A) `--remote_comm_type=get`
      iii) Set the lock type [PLATFORM-DEP]:
         A) `--lock_type=lock-all`
      iv) Set the flush type [PLATFORM-DEP]:
         A) `--flush_type=flush-local`
   c) Convergence settings:
      i) Enable convergence checking for twosided:
         A) `--enable_global_check`
      ii) Set onesided convergence check type:
         A) `--global_convergence_type=decentralized`
   d) Local subdomain solver settings:
      i) Set the local solver [PARAMETER]:
         A) `--local_solver=iterative-ginkgo/direct-ginkgo`
      ii) Set the local solver max iteration count [PARAMETER]:
         A) `--local_max_iters=100`
      iii) Set the local preconditioner [PARAMETER]:
         A) `--local_precond=block-jacobi/ilu/isai`

The flags marked with [PARAMETER] are parameters that can be experimented with. The specific parameter values are available in the paper. The flags marked with [PLATFORM-DEP] are flags that are platform dependent and hence the optimal values may differ on different platforms. We have provided the optimal values for the Summit platform.

Our setup

All our experiments were run on the Summit supercomputer at Oak-Ridge National Laboratory, US. Each node of Summit consists of 6 NVIDIA V100 GPU's connected to each other and the CPU sockets with NVLINK bridges. We used the IBM Spectrum MPI (v10.3.1.2) which is CUDA Aware. gcc-7.4.0 was used as the host compiler and the CUDA Toolkit 10.1.243 was used as the device compiler.

At the time of our experiments Summit had some problems with the transfer of large CUDA buffers, particularly off-node for one-sided communication with `MPI_Get`. An environment flag, `PAMI_CUDA_AWARE_THRESH` needed to be set to a high value to circumvent this at the cost of performance. Additionally, `MPI_Put` was found to have a bug and hence we had to resort to using `MPI_Get` instead.