Implementing Asynchronous Jacobi Iteration on GPUs

Yu-Hsiang Mike Tsai[0000-0001-5229-3739]*, Pratik Nayak[0000-0002-7961-1159]*,
Edmond Chow[0000-0003-0474-3752]†, and Hartwig Anzt[0000-0003-2177-952X]‡*
*Steinbuch Centre for Computing, Karlsruhe Institute of Technology
†School of Computational Science and Engineering, Georgia Institute of Technology
‡Innovative Computing Laboratory, University of Tennessee

***Abstract*—Computation on architectures that feature fine-grained parallelism requires algorithms that overcome load imbalance, inefficient memory access, serialization, and excessive synchronization. In this paper, we explore an algorithm for iteratively solving systems of linear equations that allows for asynchronous updates by different execution units and completely removes the need for synchronization. Methods of this type have been identified as potentially competitive for computations on Exascale machines, but practical implementations for GPU platforms have scarcely been studied. We present an asynchronous Jacobi iteration optimized for high-end GPUs, demonstrate the superiority of the algorithm over a highly tuned synchronous Jacobi iteration, and deploy the algorithm as production-ready implementation in the Ginkgo open source library. The ideas presented here on the algorithm design, implementation and performance can help guide the design of other asynchronous iterative methods on GPUs.**

***Index Terms*—asynchronous, Jacobi, GPUs, iterative methods**

I. Introduction

Alleviating synchronization bottlenecks has been one of the core objectives in preparing algorithms for exascale [1]. With hardware becoming more and more hierarchical, it has become crucial to avoid paying the cost of synchronization between the different levels of parallelization and between execution units on the same level, as these synchronizations can significantly hamper the scalability of algorithms [2], [3]. For example, GPUs, which form the workhorse of a node in many of the current and the upcoming exascale systems, have multiple levels of memory (L3, L2 and L1 caches) and execution parallelism (thread blocks, (sub)warps, and threads). The efficient use of these memory and compute hierarchies requires algorithms that minimize both the data transfers between the different levels of the memory hierarchies and the synchronizations between the parallel execution instances.

The performance of iterative solvers for solving sparse linear systems is typically limited by the main memory bandwidth. Thus, minimizing the data movement is crucial to the execution time. Various techniques such as merging kernels [4], incorporating lower precision [5] etc., have shown promise in reducing the data access volume. Tackling the synchronization problem on the other hand is more challenging. It is well known that traditional algorithms for solving linear systems such as Krylov methods require global synchronizations for orthonormalizing the Kyrlov basis vectors [6], [7]. With hundreds of thousands of parallel computing units, these global synchronizations become detrimental to runtime performance. While task-based approaches can help hide global synchronizations behind other operations [8], the potential of task-based execution is limited by the data dependencies within a single iteration: Krylov solvers do not allow for thread-asynchronous execution of the complete iteration process.

Although as a stand-alone solver it is less popular than Krylov methods, Jacobi and other stationary iterations are often used as a building block for complex algorithms such as Multigrid methods [9]. In this work, we present an asynchronous version of the Jacobi iteration on GPUs that can be used as plug-in replacement for synchronous Jacobi, while outperforming it in the time-to-solution metric.

1) After discussing related work in Section II and mathematical background in Section III, we present two strategies in Section IV for realizing asynchronous Jacobi on GPUs.
2) In Section IV, we also provide a few different strategies to record the behaviour of the asynchronous updates without affecting the overall performance.
3) We deploy the asynchronous Jacobi iteration as a production-ready algorithm in the open-source Ginkgo math library [10].
4) In Section V, we provide an in-depth analysis of the convergence and performance characteristics of asynchronous Jacobi when executing on the Summit supercomputer.
5) We discuss the findings that translate to the design of other asynchronous algorithms in Section VI.

II. Related work

The seminal work of Chazan and Miranker [11] introduced the idea of chaotic relaxation for solving systems of linear equations, where each parallel computation unit is completely asynchronous, with the drawback of computing on possibly stale data. For these types of methods, they proved necessary and sufficient conditions for convergence. Baudet [12] generalized these methods slightly, and for general (nonlinear) contracting operators, proved sufficient conditions for convergence.

In the context of block and domain decomposition methods, Frommer et al. [13] considered the Schwarz methods and evaluated the algebraic Schwarz and some multi-splitting methods. The benefits of the Schwarz methods from a practical standpoint were shown here with the asynchronous version being up to 50% faster in the presence of load imbalance. Other multi-stage and multi-level methods have also been explored [6], [14], [15].

There has also been some effort in modeling the asynchronous iteration. In particular, Wolfson-Pou and Chow [16] modeled the asynchronous version of the Jacobi method aiming to study the transient behavior of the method where they showed that the asynchronous method continues to converge in cases of process imbalance and even in cases where the synchronous version does not converge. These models were also extended to other methods such as the Multigrid methods [17], where sampling the delays from a probabilistic distribution, they studied the convergence behavior of asynchronous versions of additive multigrid methods.

Chow et al. [18] considered the asynchronous version of first and second order Richardson iterations. For both these methods, they studied the effects of the scalar parameter on asynchronous convergence. They showed that the convergence of the first order asynchronous Richardson can be better than that of the synchronous version, while convergence for the second order asynchronous version was very sensitive to the amount of asynchrony.

Anzt et al. [19] developed a block asynchronous relaxation method, where each block relaxation was a separate kernel launch and the values were updated asynchronously. They compared the performance of this block relaxation method on the GPU to Gauss-Seidel on the CPU and a GPU based Jacobi relaxation. They also observed significant speedups compared to a GPU based CG solver.

III. Background

We aim to solve the linear system

$$Ax = b, \quad A \in \mathbb{R}^{n\times n}, \quad b \in \mathbb{R}^n. \tag{1}$$

Let a nonsingular matrix M define the splitting $A = M - N$, and taking the iteration matrix $T = M^{-1}N$ and $c = M^{-1}b$, we can write the stationary iteration

$$x^{k+1} = Tx^k + c \tag{2}$$

where $k = 0, 1, 2, \ldots$, and x^0 is the initial guess [20].

In general, stationary iterations have the form $x_{n+1} = G(x_n)$. Asynchronous iterative methods for solving systems of the form $x = G(x)$, where $G : \mathbb{R}^n \to \mathbb{R}^n$, can be defined as the sequence of updates [11], [12], [18]

$$x_i^k = \begin{cases} x_i^{k-1} & \textit{if} \quad i \notin J_k \\ g_i(x_1^{s_1(k)}, x_2^{s_2(k)}, \cdots, x_n^{s_n(k)}) & \textit{if} \quad i \in J_k \end{cases} \tag{3}$$

where x_i^k denotes the ith component of x at time k. J_k is the set of indices which have been updated at k and $s_j(k) \le k-1$ is the last time the component j of x was updated before evaluating g_i at k.

For linear systems of the form $G(x) = Tx + c$, $T \in \mathbb{R}^{n\times n}$, $c \in \mathbb{R}^n$, any asynchronous iteration converges for any initial condition if and only if the spectral radius, ρ of the (componentwise) absolute value of iteration matrix is less than 1, i.e., $\rho(|T|) < 1$. Additionally, the set of delays, J_k and the delay sequences $s_j(k)$ need to satisfy the following conditions:

$$\lim_{k\to\infty} s_j(k) = \infty \text{ for } j = 1, ..., n \tag{4}$$

$$J_k \text{ is infinitely filled with each of } i \in 1, \cdots, n. \tag{5}$$

An additional aspect to note is that the condition for convergence of the asynchronous iterations, $\rho(|T|)$ is stricter than that for the synchronous version, $\rho(T)$. As observed from a practical standpoint by others, for example in [18], [21], this gives a pessimistic view of asynchronous iterative methods, although in practice the asynchronous method can converge even when the synchronous version does not converge.

IV. Implementation

The hierarchical parallelism available on the GPUs through thread blocks, (sub-)warps, and threads allows for asynchronous execution on multiple hierarchy levels. This suggests different strategies for realizing asynchronous algorithms. We evaluate the effect of the different strategies for the asynchronous Jacobi iteration.

A. The Laplacian problem

The target problem in all experimental evaluations is the Laplace problem in 2D discretized with the 5-point stencil, $[-1, -1, 4, -1, -1]$ and the Laplacian problem in 3D with the 7-pt stencil, $[-1, -1, -1, 6, -1, -1, -1]$. We use a symmetrically scaled version of the matrix such that its diagonal is all ones. This problem has been well analyzed and it has been shown that the optimal value of α for this problem is equal to 1 [18], which represents the inverse of the constant diagonal elements. Therefore, we use $\alpha = 1$ for our experiments.

B. Synchronous Jacobi iteration

The classical synchronous Jacobi iteration first computes the residual(error correction) and then updates the solution vector with the scaled residual. This process is repeated for the prescribed number of iterations. The computation of the residual involves a matrix vector product and is computationally the most expensive part of the algorithm. The computational cost is of the order $\mathcal{O}(nnz)$, where nnz is the number of nonzeros in the matrix. We store our matrix in a Compressed Sparse row (CSR) format, which is a versatile storage format for sparse matrices [22]. We note that steps 3 and 5 involve an explicit device-wide synchronization as each iteration requires all the elements of the solution vector to be updated before proceeding with the next iteration.

Algorithm 1 Synchronous Jacobi iteration

```
1: for i = 0 .. max_iterations-1 do
2:     Compute Residual: r = b - Ax
3:     Synchronize
4:     Update: x += $\alpha$ * r
5:     Synchronize
6: end for
```

C. Implementation 1: Static subwarp-to-rows

Subwarps are a concept featured in the cooperative group functionality of CUDA that allows splitting CUDA warps into smaller execution entities [23]. A subwarp can be as small as a single thread and as large as a complete warp (32 threads). In this implementation, every row of the iteration vector is handled by one subwarp, and the number of subwarps launched matches the number of rows in the solution vector. For a subwarp size of 1, this implementation (shown in Algorithm 2) assigns one thread to one row of the linear system, **x[i] +=** α***(b[i] - A(i, :)*x)**. The scheduling of the subwarps on the compute units is handled by the CUDA runtime. To ensure the consistency of the memory reads and writes across different threads, we use `__threadfence()` and demonstrate the need for this in Section V. `__threadfence()` only guarantees that the memory is up-to-date when other threads update the memory not a synchronous barrier between threads like `__syncwarp()` or `__syncthreads()`. Using the static subwarp-to-row strategy, only one subwarp updates an entry of the iteration vector, so all the updates are incremental.

Algorithm 2 Static: assign subwarp to the same row

```
1: for all row in [0, #rows) in parallel do
2:     Accumulate: temp = A(row, :) * x
3:     Update: x[row] += $\alpha$ * (b[row]-temp)
4:     Enforce memory updated: __threadfence();
5: end for
```

D. Implementation 2: Dynamic subwarp-to-row assignment

An alternative strategy is to reduce the number of launched thread blocks and require the subwarps to handle multiple rows in each iteration, see Algorithm 3. This implies that the subwarp-to-row assignment after the first cycle is handled by the runtime, which potentially allows for better load balancing and allows different subwarps to update different rows. The NVIDIA V100 GPU contains 80 Streaming Multiprocessors (SMs) and each SM has 4 warp schedulers. One SM can concurrently execute 4 warps (128 threads). Scheduling on each SM a single thread block of size 128 prevents the GPU from doing context switching, a concept that is fundamental to hiding latency in the absence of deep cache hierarchies. To allow for context switching, we oversubscribe the SMs by launching more thread blocks than multiprocessors available. Specifically, we launch **oscb**$\times$80 thread blocks of size 128 where **oscb** is the oversubscription factor ranging from 1 to 16. Using the dynamic subwarp-to-row strategy, multiple subwarps may update a row at a time.

Algorithm 3 Dynamic: Allow subwarps to handle the different rows

```
Require: num_subwarps $\leq$ num_rows
1: for all id in [0, #subwarps) parallel do
2:     Compute the number of available subwarps:
       num_subwarps
3:     Initialize: iter=0
4:     Initialize: row=id
5:     while iter < max_iterations do
6:         Accumulate: temp = A(row, :) * x
7:         Update: x[row] += $\alpha$ * (b[row] -
           temp)
8:         __threadfence();
9:         Get next working row: row +=
           num_subwarps
10:        if row $\geq$ num_rows then
11:            Modulo: row -= num_rows
12:            Increment: iter += 1
13:        end if
14:    end while
15: end for
```

Figure 1 visualizes the update process of the two implementations. The y-axis represents the distinct rows of the iteration vector, the columns represent the distinct iterations. The color code represents updates that happen in the same update cycle. The square represents the first subwarp, the circle represents the last subwarp of all the subwarps launched. The static subwarp-to-row implementation (left in Figure 1) launches enough subwarps to update all rows in the first update cycle. The dynamic subwarp-to-row assignment (right in Figure 1) usually uses fewer subwarps than rows present. Thus, more update cycles, $k > n$, are needed to complete n updates.

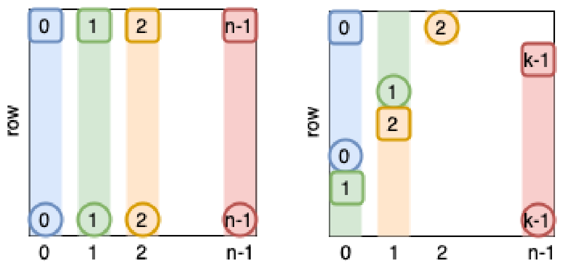

Fig. 1: Visualization of the update process of the static subwarp-to-row assignment (left) and the dynamic subwarp-to-row assignment (right).

E. Reproducibility of results and measurement strategies

In the experimental evaluation, we fix the relaxation factor $\alpha = 1$ ($(1 - \alpha) == 0$) for all implementations of the Jacobi iteration, both synchronous and asynchronous variants. This parameter selection allows for smooth convergence of the

Jacobi iteration for the 5-point stencil discretization of the Laplace problem, and prevents the compiler from employing unwanted memory optimizations.

Reproducibility of experimental results using asynchronous algorithms can be challenging or even impossible. For benchmarking purposes, it is thus necessary to rely on statistical data. As the CUDA runtime scheduling the thread blocks and orchestrating the execution is not reproducible, we report statistical information. To obtain any insight into the execution process, we need logging mechanisms for:

1) *Time measurement*: Records the start and completion time of a thread (all updates done by this thread).
2) *Final update value age*: Records the age of all values used for the last update of a vector value.
3) *Midway update value age*: Records the age of all values used for the vector value update midway through the iteration process.

It is important that the logging mechanism does not affect the CUDA scheduling or runtime performance. We achieve this by replacing the algorithm output in the iteration vector with the logging information (instead of accessing additional memory locations). This way, the logging functionality will not affect the performance of the memory-bound Jacobi iteration.[1] However, using the memory space of the iteration vector for logging mechanisms implies that when investigating the algorithm execution process with the logging mechanisms, no actual algorithm output is generated.

The iteration vector stores a 64-bit (IEEE754 double precision) value for each row. For the *time measurement*, every subwarp writes the timestamp of every row update into its row position.

For the value age analysis, we encode the age information in a customized 64-bit datatype that is then stored as the IEEE-754 double precision iteration vector, see Figure 2. The age of each of the row values itself and its four neighbors is encoded with 12 bits each. It can be decoded after applying the appropriate bit shift and the mask **`0xFFF`**. For example, as the row value age is always placed in the center, it can be decoded with **`(pos >> (2*12)) & 0xFFF`**.

The shifts that need to be applied for accessing the correct part of the customized data type are replacing the numeric values in the system matrix so that the 5-point stencil is instead stored as $\begin{bmatrix} & 0 & \\ 1 & 2 & 3 \\ & 4 & \end{bmatrix}$.

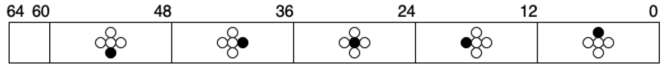

Fig. 2: Visualization of the customized 64-bit datatype that encodes the age of the row value and the neighboring elements.

[1]The time measurement has some overhead associated with accessing the system time.

V. Results

For all our experiments, we use the normalized version of the 5-pt stencil for the Laplacian problem. The right hand side is sampled from a uniform distribution from the interval $(-0.125, 0.125)$, and the initial guess is a zero vector. Each experiment consists of 10 warm-up runs and 100 runs for performance or logging measurements. We run these experiments for three grid sizes 100×100, 200×200, and 300×300, giving us matrices of sizes 10000×10000, 40000×40000, and 90000×90000 respectively. The overall time, backward error and some additional details are also computed a posteriori. All experiments use IEEE-754 double precision arithmetic and are run on the Summit supercomputer with a V100 GPU with CUDA-11.4.2 and gcc-9.3.0. The value of "update" in figure is the number of update times for each entry of solution.

Initially, we compare the synchronous Jacobi iteration with the static subwarp-per-row asynchronous Jacobi iteration (labeled "async"), which uses subwarp size of 1. Figure 3 visualizes how the median update time decreases for higher update counts. Each data point on the figure is a standalone experiment from 100 runs. We only collect the measurement after completion, so there is no overhead associated with the performance measurement. We observe that after a ramp-up phase, the cost of an update is constant for both synchronous and asynchronous Jacobi Iteration. An asynchronous update is 3 to 10$\times$ faster than a synchronous update. Figure 4 visualizes the median convergence of the synchronous and asynchronous Jacobi iteration over the update counts and indicates that the synchronous version has slightly superior convergence characteristics compared to the base asynchronous version. Figure 5 reveals that the higher update rate of the asynchronous Jacobi (more updates per second) results in superior convergence characteristics in the runtime metric.

Finally, Figure 5 shows the residual norm history over time with the synchronous and asynchronous versions both running for 1,000 updates. Here we see the clear advantage of the asynchronous version. However, we also see that the base asynchronous version stagnates for larger cases (grid size 300×300).

To further improve the GPU occupancy when running the asynchronous version, we increase the subwarp size. Figure 7 visualizes different versions of the async (= static(subwarp 1)) implementation with different subwarp sizes. Using larger subwarps reduces the update time[2], but the convergence degrades, see Figure 6. Figure 5 relating the convergence to the runtime reveals that choosing the subwarp size 1 is indeed the best choice. The analysis also reveals that `__threadfence()` is necessary for convergence of the Jacobi iteration: As mentioned in Section IV, `__threadfence()` is necessary to enforce consistency between reads and writes of the different threads for the asynchronous version. Without `__threadfence()`, the updates are much faster (see Figure 7), but the convergence can not be guaranteed because the updated values may no longer be immediately

[2]except for the case of subwarp size 32, that we investigate in detail later

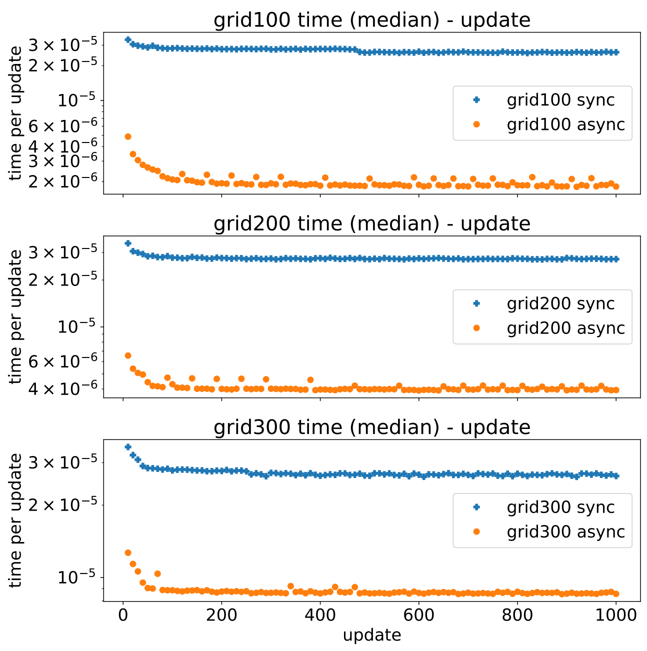

Fig. 3: The median time per update.

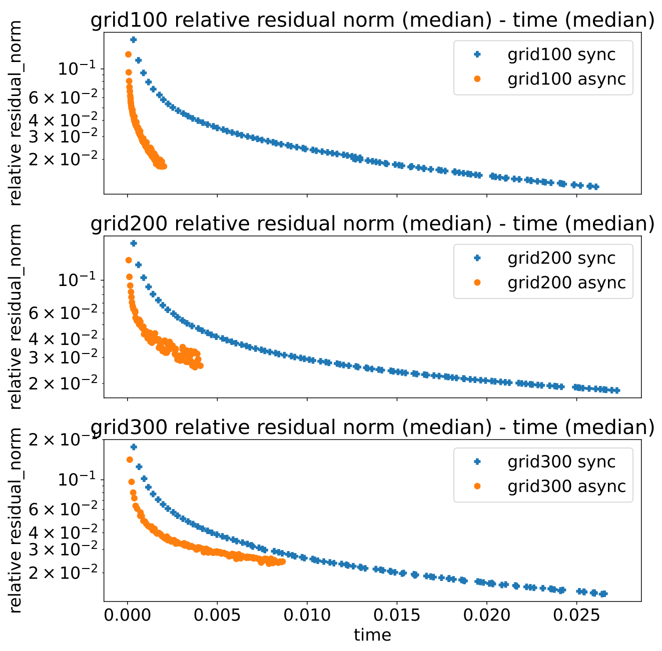

Fig. 5: the median relative residual norm with the median of time

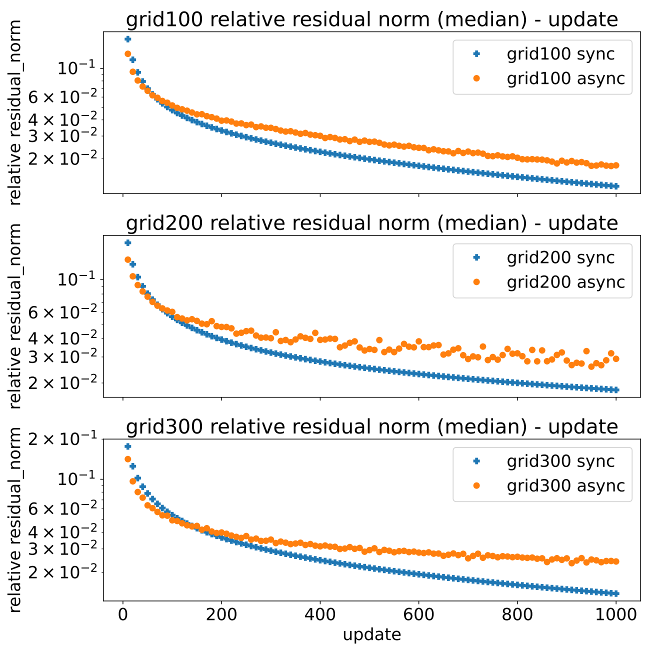

Fig. 4: The median relative residual norm related to the update count.

visible to other threads. As Figure 6 reveals that without `__threadfence()`, the asynchronous Jacobi iteration fails to converge for this test problem.

The analysis includes also the asynchronous Jacobi iteration using the dynamic subwarp-to-row strategy introduced in Algorithm 3. The dynamic assignment of subwarps to rows enables load balance and allows for faster execution. Furthermore, the oversubscription parameter **`oscb`** can be used for context switching to hide the latency. The analysis reveals that moderate oversubscription (**`oscb`** of 4, 8) gives the best results in both the convergence and performance metric, see Figure 8 and Figure 6. We note that for these oversubscription factors, the asynchronous Jacobi iteration exhibits convergence characteristics competitive with the synchronous variant, while processing the updates significantly faster.

We instantiate more threads than the maximum number of threads which the device can launch simultaneously. On a V100, the limit is equal to 80×2048 threads without accounting for other factors such as the number of registers or shared memory pressure. From Figure 9, we see that the second half of the result vector gets updated only after the first half of the vector has finished its updates. This demarcation (at $(80 * 2048)/32 = 5120$) in the updates can adversely affect the convergence rate.

For a deeper understanding of the execution of the asynchronous algorithm, we record the age (number of updates a value has seen) of the values used in an update. We classify the information propagation by the number of neighbors a node has and use the encoding {#connection of target}_{#connection of source} for a connection, see Figure 10. For example, **2_3** is the connection between a corner grid point, which has two connections, and the side grid point having three connections. When the age of each neighboring value is collected in the last update (final update value age measurement), no neighboring value can have seen more updates than the iteration limit. In the midway update value age collection, neighbors can have ages smaller or larger than

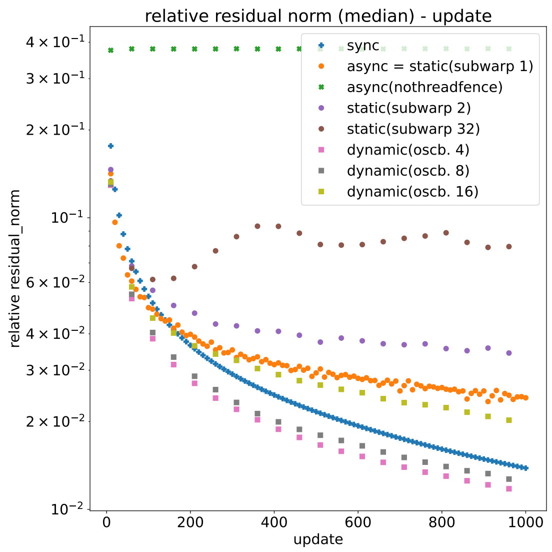

Fig. 6: The median of relative residual norm against update from different implementations with different configurations on the grid 300 x 300

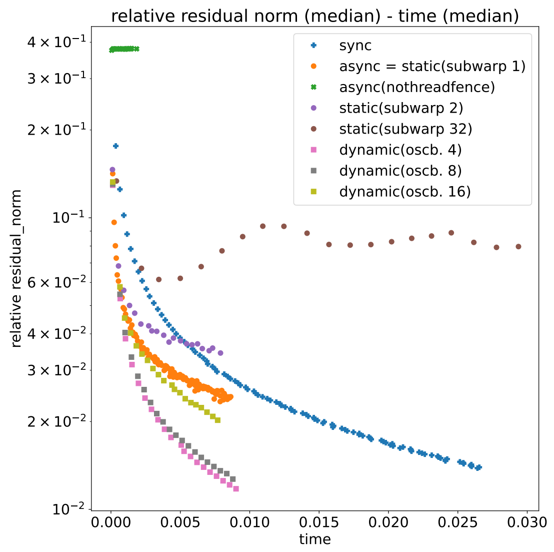

Fig. 8: The median of relative residual norm against time from different implementations with different configurations on the grid 300 x 300

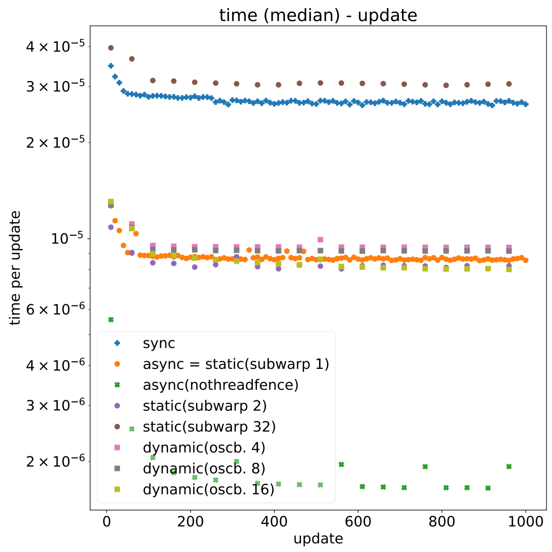

Fig. 7: The median of time per update against update from different implementations with different configurations on the grid 300 x 300

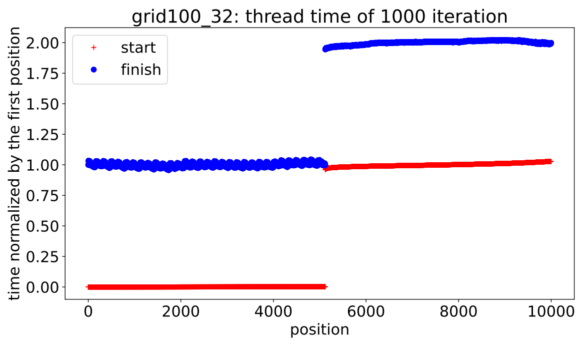

Fig. 9: Update schedule for each result index, for the static(subwarp 32) on grid size of 100×100

the current update.

To ensure logging the age does not incur significant overhead, we perform this analysis for a grid size 300×300. For this setting we note an overall execution overhead related to the age logging of 3.9% for the midway update value analysis and 6.4% for the final update value age measurement, respectively. In comparison, recording detailed runtime traces of the updates incurs about 20% overhead.

Figure 11 groups the age of the value communicated in the midway update according to the connection type. The blue boxplot represents the statistics for the async static (subwarp size = 1) implementation. The iteration limit is set to 1,000, the midway value age snapshot is taken for update 500. We observe that the median is close to 500, which is expected. For the async dynamic implementation with `oscb` = 4 (purple), we observe much less variation, which may explain the convergence characteristics similar to the synchronous version. On the other hand, for the async implementation without a threadfence (in green), we see that due to a lack of consistency in reads and writes between different threads, the updates are stale, leading to stagnation and the solver not converging.

Interesting variations in the value ages communicated across the connection types, in particular for static(subwarp 1). For connection_type **3_3** and **4_4**, updating the source or target grid points is equally expensive, and they are updated at the same rate (small variance in the boxplot). The connection_type

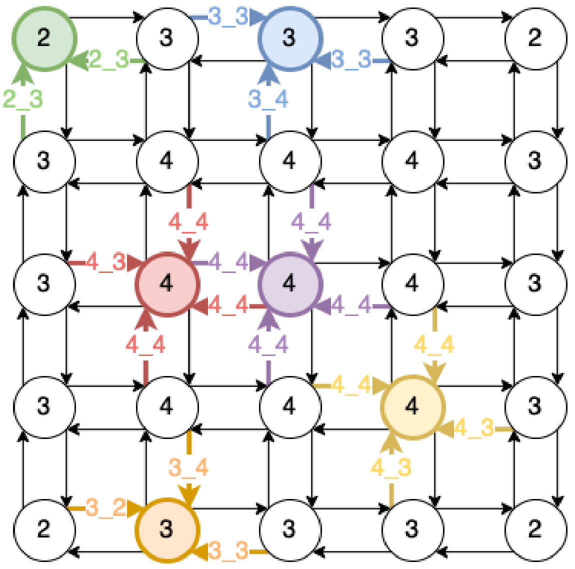

Fig. 10: Different types of the connections for each grid point. We classify them into 6 types. The number in the circle represents the number of connections. The arrow indicates the direction of the update. The name on the edge denotes {#connection of target}_{#connection of source} for the connection.

4_3 tends to communicate values from grid points that have seen more updates than the target grid point as updating the source grid point with 3 connections is faster than updating the target grid point with 4 connections. The connection_type **3_4** has a reversed trend against **4_3**. Interestingly, we do not observe the same trend for connection_types **3_2** and **2_3**. Likely, the reason is that there exist only 4 grid points with 2 connections, and the updates from and to those are processed in warps containing different workload, which is not optimal for GPU.

In Figure 12 we increase the grid size from 300x300 to 2800x2800 and observe that the overall time taken increases unlike for smaller grid sizes 100^2 300^2 in Fig. 8. For smaller problems, due to the low the GPU utilization, increasing problem size does not increase the time to solution. We also see a similar trend for the 3D 7-point stencil in Figure 13. For the 7 point stencil, we also implement two more different variants. A `syncwarp` and a `syncthreads` version, in which the synchronization is varied at different granularity levels: at the warp level and the thread-block level. Compared to the dynamic versions, these implementations allow for better cache re-usability. The syncthreads version takes overall less time due to better cache reusability as we see in Figure 13.

VI. Conclusion

We have presented different implementations of algorithms for the asynchronous Jacobi iteration. We investigated the convergence and performance of asynchronous execution and streaming multiprocessor oversubscription. We demonstrate

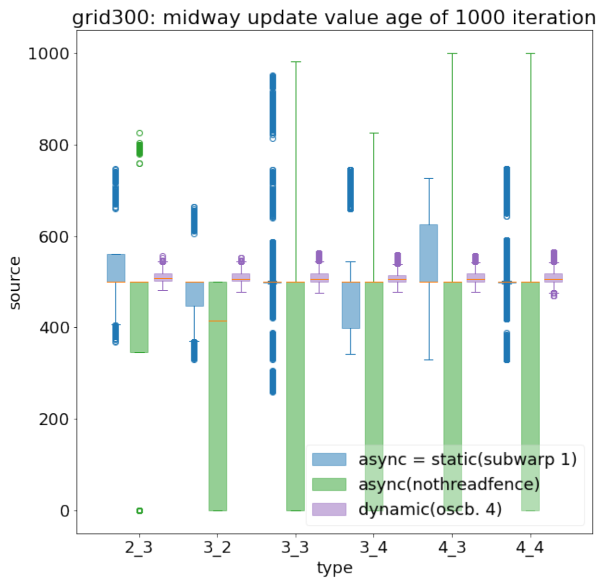

Fig. 11: The boxplot of the midway update value age for grid size 300×300 with 1000 iterations for the three Jacobi implementations: async(static), async(nothreadfence), and dynamic(oscb. 4).

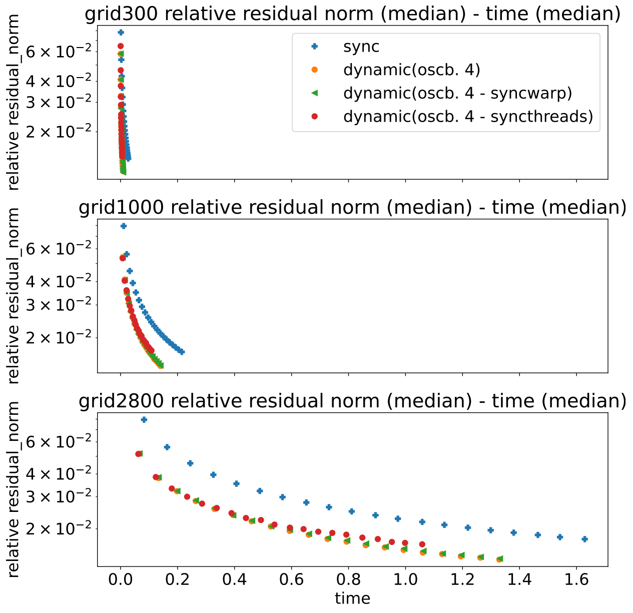

Fig. 12: Laplacian 2D problem $(300^2, 1000^2$ and $2800^2)$.

that by choosing suitable parameters, the asynchronous Jacobi iteration can outperform the synchronous Jacobi in both convergence and runtime metrics. When the problem is large enough such that the GPU is fully utilized, the asynchronous version gives the following benefit of reducing the launch overhead and the fusion of kernels. For small problem sizes, the asynchronous version can better utilize the GPU due to several steps running asynchronously. Additionally, we also

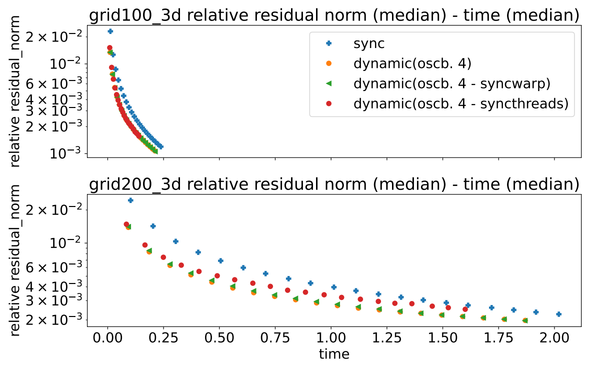

Fig. 13: Laplacian 3D problem (100^3 and 200^3).

provided a few methods to record the asynchronous updates, the runtime schedule and analyzed the observations to gain a better understanding of the different implementations. Finally, we note that the implementation is available to the community as free open-source software within the Ginkgo software framework.

Acknowledgments

This work was supported by the "Impuls und Vernetzungs-fond" of the Helmholtz Association under grant VH-NG-1241, and the US Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] S. Sachs and K. Yelick, "Exascale programming challenges. report of the 2011 workshop on exascale programming challenges, marina del rey, july 27–29, 2011," 2011.

[2] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap," *The International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, Feb. 2011, publisher: SAGE Publications Ltd STM. [Online]. Available: https://doi.org/10.1177/1094342010391989

[3] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, A. S. Z. Belloum, and R. V. Van Nieuwpoort, "The landscape of exascale research: a data-driven literature analysis," *ACM Computing Surveys*, vol. 53, no. 2, pp. 23:1–23:43, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3372390

[4] J. I. Aliaga, J. Pérez, and E. S. Quintana-Ortí, "Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers," in *Euro-Par 2015: Parallel Processing*, ser. Lecture Notes in Computer Science, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer, 2015, pp. 675–686.

[5] A. Abdelfattah, H. Anzt, A. Ayala, E. Boman, E. Carson, S. Cayrols, T. Cojean, J. Dongarra, R. Falgout, M. Gates, N. J. Higham, S. E. Kruger, S. Li, N. Lindquist, Y. Liu, J. Loe, P. Nayak, D. Osei-Kuffuor, S. Pranesh, S. Rajamanickam, T. Ribizel, B. Smith, K. Swirydowicz, S. Thomas, S. Tomov, I. Yamazaki, and U. M. Yang, "Advances in Mixed Precision Algorithms: 2021 Edition." Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), Tech. Rep. SAND2021-10227R, Aug. 2021. [Online]. Available: https://www.osti.gov/biblio/1814447-advances-mixed-precision-algorithms-edition

[6] P. Nayak, T. Cojean, and H. Anzt, "Two-stage Asynchronous Iterative Solvers for multi-GPU Clusters," in *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, Nov. 2020, pp. 9–18. [Online]. Available: 10/gphfs2

[7] E. Carson, "Communication-Avoiding Krylov Subspace Methods in Theory and Practice," Ph.D. dissertation, University of California, Berkeley, 2015. [Online]. Available: https://aspire.eecs.berkeley.edu/publication/communication-avoiding-krylov-subspace-methods-in-theory-and-practice/

[8] A. Roussel, "Parallelization of iterative methods to solve sparse linear systems using task based runtime systems on multi and many-core architectures: application to multi-level domain decomposition methods," Ph.D. dissertation, Université Grenoble Alpes, 2018.

[9] W. Hackbusch, "Multigrid Iterations," in *Iterative Solution of Large Sparse Systems of Equations*, ser. Applied Mathematical Sciences, W. Hackbusch, Ed. Cham: Springer International Publishing, 2016, pp. 265–324. [Online]. Available: https://doi.org/10.1007/978-3-319-28483-5_11

[10] 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," *ACM Transactions on Mathematical Software*, vol. 48, no. 1, pp. 2:1–2:33, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3480935

[11] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and Its Applications*, 1969.

[12] G. M. Baudet, "Asynchronous Iterative Methods for Multiprocessors," *Journal of the ACM (JACM)*, 1978. [Online]. Available: 10.1145/322063.322067

[13] A. Frommer, H. Schwandt, and D. B. Szyld, "Asynchronous weighted additive Schwarz methods," *Electronic Transactions on Numerical Analysis*, vol. 5, no. June, pp. 48–61, 1997.

[14] A. Frommer and D. B. Szyld, "Asynchronous two-stage iterative methods," *Numerische Mathematik*, pp. 1–18, 1994. [Online]. Available: 10.1007/s002110050085

[15] C. Glusa, E. G. Boman, E. Chow, S. Rajamanickam, and P. Ramanan, "Asynchronous one-level and two-level domain decomposition solvers," in *International Conference on Domain Decomposition Methods*. Springer, 2018, pp. 134–142.

[16] J. Wolfson-Pou and E. Chow, "Modeling the asynchronous Jacobi method without communication delays," *Journal of Parallel and Distributed Computing*, vol. 128, pp. 84–98, 2019, publisher: Elsevier Inc. [Online]. Available: https://doi.org/10.1016/j.jpdc.2019.02.002

[17] ——, "Asynchronous multigrid methods," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 101–110.

[18] E. Chow, A. Frommer, and D. B. Szyld, "Asynchronous Richardson iterations: theory and practice," *Numerical Algorithms*, vol. 87, no. 4, pp. 1635–1651, Aug. 2021. [Online]. Available: https://link.springer.com/10.1007/s11075-020-01023-3

[19] H. Anzt, "Asynchronous and multiprecision linear solvers-scalable and fault-tolerant numerics for energy efficient high performance computing," Ph.D. dissertation, 2012.

[20] Y. Saad, "Iterative methods for linear systems of equations: A brief historical journey," *Brenner, SC, Shparlinski, I., Shu, C.-W., Szyld, DB (eds.)*, vol. 75, pp. 197–216, 2020.

[21] E. Chow, E. Boman, J. J. Dongarra, and D. B. Szyld, "Asynchronous Iterative Solvers and Optimal Schwarz Domain Decomposition," 2017.

[22] 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 Transactions on Parallel Computing*, 2020, iSBN: 23294957 23294949. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-85083216197&partnerID=MN8TOARS

[23] Y. M. Tsai, T. Cojean, T. Ribizel, and H. Anzt, "Preparing Ginkgo for AMD GPUs – A Testimonial on Porting CUDA Code to HIP," in *Euro-Par 2020: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, B. Balis, D. B. Heras, L. Antonelli, A. Bracciali, T. Gruber, J. Hyun-Wook, M. Kuhn, S. L. Scott, D. Unat, and R. Wyrzykowski, Eds. Cham: Springer International Publishing, 2021, pp. 109–121.

Reproducibility Appendix

In order to ensure reproducibility of results, we provide the code as a Zenodo archive and elaborate on the settings and parameters used to produce these results.

Obtaining the source code

The source code is open-source and available on Zenodo (https://doi.org/10.5281/zenodo.7130225).

Building and installing Ginkgo

To build GINKGO, the following components are necessary:

1) The CMake ($\geq$ 3.13) build platform.
2) A C++-14 compiler, gcc-9.3.0 was used in this paper.
3) A CUDA ($\geq$ 9.2) installation, CUDA 11.4.2 was used in this paper.

The Ginkgo library and the source codes for this paper use the same canonical CMake setup as elaborated in the Ginkgo documentation (https://ginkgo-project.github.io/ginkgo/doc/develop/install_ginkgo.html).

Benchmarking

1) Set the desired kernel by macro in cuda/-solver/async_jacobi_kernels.cu of the source folder
   a) Set the parameters `USE_DYNAMIC`, `DYNAMIC_OSCB`, `SUBWARP_SIZE`, `APPLY_SYNC` to choose the configuration
      i) `USE_DYNAMIC`: 1 for dynamic implementation or 0 for static implmentation
      ii) `DYNAMIC_OSCB`: the number for oversubscription (only for dynamic)
      iii) `SUBWARP_SIZE`: the size of subwarp
      iv) `USE_THREADFENCE`: 1 for using `__threadfence();` or 0 for not using it.
      v) `APPLY_SYNC`: the sync methods. `NOSYNC` for no additional sync, `__syncwarp()` for syncing threads of the same warp, or `__syncthreads()` for syncing threads of the same block.
2) Build settings (changing the above parameter requires recompiling):
   a) Set the correct compiler, which may be necessary on Summit
      i) `-DCMAKE_CXX_COMPILER=g++ -DCMAKE_C_COMPILER=gcc`
   b) Compile GINKGO in release mode, (default with CMake)
      i) `-DCMAKE_BUILD_TYPE=Release`
   c) Enable Examples (should be `ON` by default):
      i) `-DGINKGO_BUILD_EXAMPLES=ON`
   d) Enable Cuda/OpenMP Module (should automatically detect):
      i) `-DGINKGO_BUILD_CUDA=ON -DGINKGO_BUILD_OMP=ON`
   e) Disable unused components (optional, may help in case of unexpected errors):
      i) `-DGINKGO_BUILD_MPI=OFF -DGINKGO_BUILD_HWLOC=OFF -DGINKGO_BUILD_TESTS=OFF`
   f) CMake command with above options, when in the source directory:
      i)
```
mkdir build && cd build &&
cmake -DCMAKE_CXX_COMPILER=g++
-DCMAKE_C_COMPILER=gcc
-DCMAKE_BUILD_TYPE=Release
-DGINKGO_BUILD_EXAMPLES=ON
-DGINKGO_BUILD_CUDA=ON
-DGINKGO_BUILD_OMP=ON
-DGINKGO_BUILD_MPI=OFF
-DGINKGO_BUILD_HWLOC=OFF
-DGINKGO_BUILD_TESTS=OFF ..
```
3) Go to the `async-jacobi` example when in the build directory:
   a) `cd examples/async-jacobi`
4) Benchmark settings controlled through command line variables:
   a) `./async-jacobi [executor] [type] [measurement type] [problem size] [iteration] [folder(optional)]`
      i) executor: `cuda` is the only one that is currently supported for both `sync` and `async`.
      ii) type: `sync` or `async`
      iii) measurement type: `normal`: the normal Jacobi iteration, `flow`: the final value update age measurement, `halfflow`: the midway value update age measurement, `time`: the time measurement, `normal_3d`: the normal Jacobi iteration but for 3d (7-pt) stencil problem
      iv) problem size: the grid size of 5-pt stencil problem except for the `normal_3d`. It is the grid size of 7-pt stencil problem when using `normal_3d`
      v) iteration: the number of updates to perform
      vi) folder: optional option. If it is provided and measurement type is not `normal` and `normal_3d`, it will write the #experiments(100) csv to the folder for the raw detail of the measurement type.

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. gcc-9.3.0 was used as the host compiler and the CUDA Toolkit 11.4.2 was used as the device compiler. The corresponding module command: `module load cuda/11.4.2 gcc/9.3.0 cmake` We only use one GPU of a node for experiments.