Karlsruhe Institute of Technology

Institute of Theoretical Informatics
Department of Computer Science
Karlsruhe Institute of Technology

Synchronization-free algorithms for exascale and beyond

A study of Asynchronous and Batched Iterative Methods

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenscaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Pratik Vivek Nayak
geboren in Ankola, Indien

Tag der mündlichen Prüfung:	23.11.2023
Erster Referent:	Prof. Dr. Hartwig Anzt
Zweiter Referent:	Prof. Dr. Achim Streit
Vorsitzender:	Prof. Dr. Hannes Hartenstein
Stellvertretender Vorsitzender:	Prof. Dr. Thorsten Strufe
Prüferin:	Prof. Dr. Anne Koziolek

This online-friendly version is archived with DOI:10.5281/zenodo.10311349

Abstract

Computing at scale has enabled scientific discovery in various fields, such as bioinformatics, energy, healthcare, and transportation. The algorithmic and computing landscapes must adapt as our mathematical models grow, aiming to provide a more accurate and realistic view of our physical world.

In the exascale[1] era, extracting maximum performance from a system requires efficient algorithms that can take advantage of the massive parallelism that these machines provide. The heterogeneous nature of these machines necessitates efficient implementations at single and multi-GPU levels, with GPUs providing most of the parallelism. Ensuring minimal synchronization bottlenecks is paramount for efficient computation across this whole hierarchy. Synchronization for information exchange is required for many state-of-the-art algorithms such as linear solvers, which form the workhorse of many scientific applications. Minimizing or removing these synchronizations can accelerate applications.

In this work, we look at two techniques that minimize synchronizations between parallel computing units. The first technique, batching, maximizes the available parallelism on a Graphics Processing unit (GPU) by utilizing the perfect parallelism available for the solution of independent but related linear systems. Mapping these independent linear system solutions at the appropriate compute hierarchy level enables almost perfect scaling for single and multi-GPU systems. With a careful design of the data structures and the linear solver kernels, we provide a high-performance implementation of the batched solvers that significantly outperform state-of-the-art implementations. We showcase the benefits of these batched solvers for problems from different origins, including integrations into two real-world applications.

The second technique removes existing synchronizations by following a data asynchronous approach. In this case, the parallel computing elements process the latest available local data and incorporate any required data from their neighbors asynchronously. Using a probabilistic model, we study and analyze these asynchronous iterative methods. We also showcase the benefits of using these asynchronous methods with an efficient GPU implementation that outperforms the synchronous variant. To enable scaling to multiple GPUs, we implement, evaluate, and analyze the asynchronous Schwarz methods and show that they can beat their synchronous counterparts for realistic test cases.

Synchronization-free techniques help accelerate scientific simulations, enabling them to maximize the available compute resources. This allows scientists to efficiently scale up their simulations, gain a deeper understanding of the physical phenomena, and perform ground-breaking science to further scientific inquiry.

[1]almost a million times faster compared to a generic laptop, 10^{18} operations/s

Zusammenfassung

Das wissenschaftliche Hochleistungsrechnen hat Fortschritte in verschiedenen großen Maß-stab hat wissenschaftliche Entdeckungen in verschiedenen Bereichen wie Bioinformatik, Energie, Gesundheitswesen und Allgemeiner Ingenieurwesen ermöglicht. Allerdings erfordert die sich verändernde Hardware, dass wir die verwendeten Algorithmen und Programmierparadigmen an anpassen um bessere Simulationen zu ermöglichen.

Um im Exascale-Zeitalter die maximale Leistung aus einem System herauszuholen, sind effiziente Algorithmen erforderlich, die die massive Parallelität dieser Maschinen nutzen können. Die heterogene Natur dieser Maschinen erfordert effiziente Implementierungen auf Einzel- und Multi-GPU-Ebene, wobei die GPUs den größten Teil der Parallelität liefern. Die Sicherstellung minimaler Synchronisierungsengpässe ist für eine effiziente Berechnung über die gesamte Hierarchie hinweg von größter Bedeutung. Die Synchronisierung zum Austausch von Informationen ist für viele moderne Algorithmen wie z. B. lineare Solver, die das Arbeitspferd vieler wissenschaftlicher Anwendungen bilden. Die Minimierung oder Beseitigung dieser Synchronisationen kann die verfügbare Parallelität maximieren und die Anwendungen beschleunigen.

In dieser Arbeit befassen wir uns mit zwei Techniken zur Minimierung von Synchronisationen zwischen parallelen Recheneinheiten minimieren. Die erste Technik, die Stapelverarbeitung, maximiert die verfügbare Parallelität auf einem Grafikprozessor, indem sie die perfekte Parallelität nutzt, die für die Lösung unabhängiger, aber verwandter linearer Systeme verfügbar ist. Die Abbildung dieser unabhängigen linearen Systemlösungen auf der entsprechenden Ebene der Rechenhierarchie ermöglicht eine nahezu perfekte Skalierung für Einzel- und Multi-GPU-Paradigmen. Durch ein sorgfältiges Design der Datenstrukturen und der linearen Solver-Kerne bieten wir eine hochleistungsfähige Implementierung der Batch-Solver, die den Stand der Technik deutlich übertrifft. Wir demonstrieren die Vorteile dieser Batched Solvers für einen vielfältigen Datensatz, einschließlich der Integration in zwei reale Anwendungen.

Bei der zweiten Technik werden bestehende Synchronisationen durch einen asynchronen Datenansatz entfernt. In diesem Fall verarbeiten die parallelen Rechenelemente die neuesten verfügbaren lokalen Daten und beziehen alle erforderlichen Daten von ihren Nachbarn asynchron ein. Mithilfe eines probabilistischen Modells untersuchen und analysieren wir diese asynchronen iterativen Methoden. Außerdem zeigen wir die Vorteile dieser asynchronen Methoden anhand einer effizienten GPU-Implementierung, die die synchrone Variante übertrifft. Um die Skalierung auf mehrere GPUs zu ermöglichen, implementieren, bewerten und analysieren wir die asynchronen Schwarz-Methoden und zeigen, dass sie ihre synchronen Gegenstücke für realistische Testfälle schlagen können.

Synchronisationsfreie Techniken helfen bei der Beschleunigung wissenschaftliche Simulationen zu beschleunigen und die verfügbaren Rechenressourcen Ressourcen zu maximieren. Dies ermöglicht es Wissenschaftlern, ihre Simulationen effizient zu skalieren Simulationen effizient zu skalieren, ein tieferes Verständnis der physikalischen Phänomene zu erlangen und bahnbrechende wissenschaftliche Forschung zu betreiben.

Acknowledgements

This work would not have been possible without the help of a few people, and I would like to express my gratitude to them. First and foremost, I would like to sincerely thank Hartwig Anzt, who has been an exemplary mentor. He encouraged me to explore and implement ideas and offered valuable advice and guidance when necessary. Since being the first member of his group, which now seems eons ago, the opportunities he has provided, from conferences and research visits to various collaborations, have helped me expand my horizons and become a better researcher.

I would also like to thank Terry Cojean for his support and help throughout my Ph.D. Much of my workflow has been some form of an adaptation of his recommendations, and I am grateful to him for that. I would like to thank my office mates, Yu-Hsiang Tsai (Mike/Jeff) and Thomas Grützmacher, for random discussions on various topics, technical and non-technical, making the office a fun place to be at.

Working with the whole FiNE team (Hartwig, Goran, Terry, Marcel, Yen-Chen, Gregor, Vasilis, Mike, Thomas, Tobias, Fritz, Claudius) has been an absolute pleasure. The daily meetings, post-COVID, have enabled us to work as a tightly-knit team, and I am very proud of the Ginkgo software we are developing. Discussions with the team members have always been constructive and productive. I hope we continue to build amazing software in the future. I would also like to thank Aditya and Isha, who have been an immense help in developing the batched algorithms. I think we were able to focus and efficiently work together to implement high-performing algorithms and develop tangible real-world application integrations.

I sincerely also thank the developers and contributors of the many open-source frameworks, packages, and tools, such as Emacs, Python, LaTeX, and many more, without whom this work would have been significantly more challenging to produce.

As an expat living far from where I was born, the group adventures, from running to mid-lunch summer swimming in the Baggersee to hiking in the Black Forest and the Alps, have been truly delightful. I am grateful to Hartwig and all the group members for making me feel at home in Karlsruhe.

Last, but not the least, I would like to thank all my friends and family for their unfailing support and encouragement.

Toll hier.

~~Nett hier.~~

Funding and resource acknowledgements

This work was supported by the “Impuls und Vernetzungsfond” 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.

Part of this work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

Part of this research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Part of this research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Contents

List of Figures

List of Tables

List of Definitions and Theorems

List of Algorithms

1 Introduction

Synchronization lies at the center of most state-of-the-art parallel algorithms, enabling information exchange between different parallel processing elements. Design and implementation of parallel algorithms require careful attention to data structures to enable efficient concurrent access and their memory layouts to maximize usage of all levels of caches and avoid synchronization overheads. This is particularly important for accelerators such as GPUs [DBM+11; AHL+11].

One of the central challenges of the exascale era has been to resolve the problem of the bulk-synchronous nature of our algorithms [AHL+11]. At exascale, with hundreds to thousands of processing elements, this bulk-synchronous nature hinders the scalability of algorithms, reducing their computational efficiency.

1.1 Parallelism at exascale and beyond

Parallel computing has gained popularity in recent decades due to the limits of heat dissipation, forcing chip manufacturers to package multiple cores into one chip rather than increase the operating frequency of a single core to maximize performance. Figure 1.1 shows the trends in the microprocessor data for the last 50 years. We observe that the single-core operating frequency and typical power has stagnated, whereas the number of logical cores packaged in a chip have increased significantly.

The explosion of core count has produced new compute architectures which, have led to an increased number of heterogeneous machines. The GPU is one such important architecture, which was initially developed for graphics processing but is now being used widely for general-purpose computing.

In contrast to the traditional CPU architectures, GPUs generally have a smaller, specialized instruction set [NVI22], which allows for faster overall operation of individual instructions due to faster decoding units. While CPUs are designed for Multiple Instruction Multiple Data (MIMD) parallelism, GPUs are better suited for Single Instruction Multiple Data (SIMD) parallelism. GPU computing has gained popularity in the past decade due to its suitability for high-throughput computations. Applications and algorithms that can take advantage of the massive parallelism provided by the GPU can greatly reduce their runtime.

The increased core count has also fragmented the memory hierarchy, particularly with new architectures having non-uniform memory access across cores. This requires careful design and implementation of data structures and algorithms that need to consider memory movement to maximize the overall performance. The latest GPUs and CPUs are increasingly hierarchical from both compute and memory perspectives. Algorithms, therefore, need to take advantage of hierarchical parallelism by reducing memory movement and synchronization across levels.

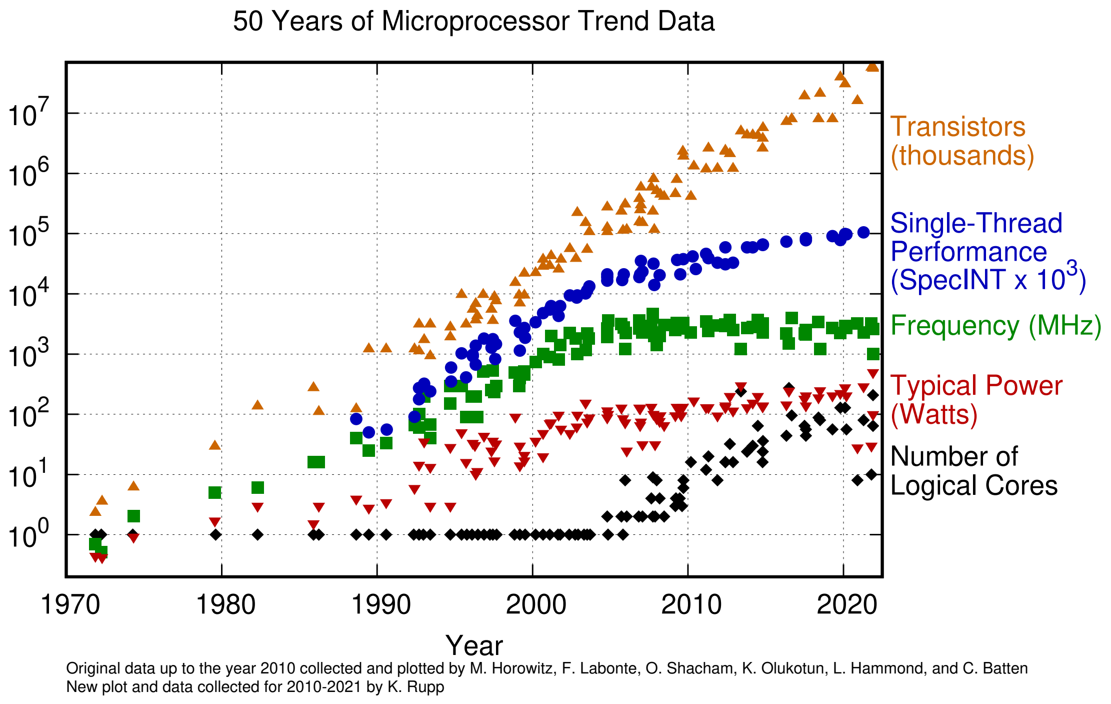


Figure 1.1: 50 years of trends for processors

Efficient large-scale distributed computation requires algorithms that can harness the fine-grained parallelism that these massively parallel architectures provide and scale this across multiple compute nodes. Most large supercomputers contain multiple GPUs per node and thousands of these multi-GPU nodes. Therefore, in addition to efficient fine-grained computation, it is essential to design algorithms that contain coarse-level parallelism that can be efficiently distributed across these nodes.

1.2 The role of synchronization

Bulk-synchronous algorithms require concurrent processing elements to synchronize to exchange information. This synchronization requirement restricts scalability. With increasing hierarchical architectures, synchronization across levels forms a bottleneck at fine-grained and coarse-grained levels.

For example, any global reduction (e.g., summing up values of a distributed array) requires global synchronization. Algorithms that use these global reductions are not amenable to task-based parallelism, as these global reductions lie in the critical path, making task-based approaches essentially bulk-synchronous.

Synchronization-free and asynchronous algorithms aim to tackle this challenge of synchronization-related bottlenecks to harness the parallelism across all levels of the computing hierarchy. If we can re-frame our algorithms and our implementations into independent parallel computations that do not require explicit communication, then we avoid these synchronization bottlenecks.

1.3 Accelerating computational science

Computational science enables rapid prototyping and exploration of ideas, particularly in fields requiring complex and expensive experimental analysis equipment. The two core components of computational science are the mathematical models that aim to simulate physical phenomena and the computing hardware used to run these models. Usually, to understand the physical phenomena and harness them for various applications, running these models for different parameters and with as small a resolution as possible is necessary.

Computational science has been at the center of scientific progress in the past few decades in fields such as combustion, plasma physics, aerodynamics, biophysics, neuroscience, bioinformatics, and many others. There has been rapid development of mathematical models and computational methods that, in combination with the rapidly improving computing infrastructures has provided us with exciting discoveries in each of these fields.

The synergy between mathematical methods, algorithms, and computing hardware is essential to keep this trend of rapid innovation and enable these scientific fields to run larger models and explore new ideas. The essential component of this synergy is computational efficiency. Efficient computation that maximizes the available resources, minimizes energy utilization, and produces physically relevant results is of utmost importance. Therefore, the central challenge for computational scientists is to improve computational efficiency.

1.4 Contributions of this work

This work focuses on developing novel mathematical methods that enable efficient computation, primarily by overcoming synchronization bottlenecks. We implement them on state-of-the-art hardware and show how applications can benefit from utilizing these ideas in their software. These methods tackle the aforementioned central challenge of computational science: improving computational efficiency. To this end, the salient contributions of this work are:

1. A detailed theoretical study of batched iterative methods and their data structures, including computational complexity, memory complexity and cache optimality.
2. Novel batched iterative solver algorithms and implementations that provide significant speedups over state-of-the-art batched direct solvers.
3. Production runs of scientific applications using the batched iterative solvers for: Combustion: PeleLM+SUNDIALS and Plasma physics: XGC.
4. A novel probabilistic model that studies and analyses the asynchronous iterative methods and enables tuning for a given system architecture.
5. A high-performance implementation and detailed analysis of the asynchronous Richardson iteration on the GPU that significantly outperforms the synchronous version.
6. A high-performance implementation and analysis of single-stage and two-stage asynchronous Schwarz methods for multi-GPU systems that provides significant performance benefits over the synchronous version.

Chapter 2 overviews some of the important mathematical and computing preliminaries used in the subsequent chapters. In Chapter 3 we elaborate on the features and capabilities of Ginkgo, the software used in this work. Chapter 4 elaborates on the synchronization-free batched iterative methods, including data structures, solvers, and application integrations. Chapter 5 looks at our high-performance implementations of preconditioners for the batched iterative solvers on GPUs. Chapter 6 studies the asynchronous iterative methods and introduces a probabilistic model and an implementation of the asynchronous iterative method for the GPU. We extend these asynchronous methods to distributed settings, particularly for multi-GPU clusters with the asynchronous Schwarz methods in Chapter 7. We finally summarize our work and provide an outlook in Chapter 8.

2 Preliminaries and Background

This chapter covers some of the mathematical, algorithmic, and computational preliminaries used in this work. It explores existing literature on linear solvers, preconditioners, and basic algorithms, aiming to provide a background for the methods, analysis, and implementations explored in the later chapters. We look at some important basic mathematical definitions and results in Section 2.1 and study some methods for solving of linear systems in Section 2.2. We also look at the hardware architecture and the programming models important to this work in Section 2.3. We look at some important data structures and their computational and memory complexities in Section 2.4. In Section 2.5 we briefly recap some common performance analysis terminologies. Finally, in Section 2.6, we elaborate on the characteristics of the High-Performance Computing (HPC) systems used in this work.

2.1 Mathematical preliminaries

We define some commonly used mathematical terms and state some theorems and results, which will be of use in the later chapters. The mathematical definitions, lemmas, and theorems have been chiefly taken and reframed from the following resources: Golub and van Loan [GVL13], Greenbaum [Gre97], Horn and Johnson [HJ85], Ortega and Rheinboldt [OR00], Saad [Saa03] and Strang [Str23]. Proofs where simple and essential have been shown; otherwise they have been referred back to the original resource.

Definition 2.1 (Vector Space)**.** *A vector space $\mathcal{V}$ over some field $\mathcal{F}$, is a set that is closed under addition, is associative and commutative, has an additive identity ("0") and has additive inverses in the set. The set is also closed under left multiplication by elements of the field $\mathcal{F}$, with the following properties holding for all $a, b \in \mathcal{F}$ and all $x, y \in \mathcal{V}$:*

$$a(x + y) = ax + ay \tag{2.1}$$

$$(a + b)x = ax + bx \tag{2.2}$$

$$a(bx) = (ab)x \tag{2.3}$$

Definition 2.2 (Subspace)**.** *A subspace $\mathcal{U}$ of a vector space $\mathcal{V}$ is a subset of $\mathcal{V}$ such that the space $\mathcal{U}$ is a vector space in itself over the same scalar field as $\mathcal{V}$.*

Definition 2.3 (Span). *For a subset S, of some vector space, the span, $\mathcal{S}$ is defined as*

$$\mathcal{S} \equiv \{a_1 v_1 + a_2 v_2 + \cdots + a_k v_k : a_1, \ldots, a_k \in \mathcal{F}, v_1, \ldots, v_k \in S, k = 1, 2 \ldots\} \tag{2.4}$$

Definition 2.4 (Linear independence). *A set of vectors $\{x_1, x_2, \ldots, x_k\}$ in a vector space is said to be linearly independent if there exist coefficients $a_1, a_2, \ldots, a_k$ not all 0, in the scalar field $\mathcal{F}$ such that*

$$a_1 x_1 + a_2 x_2 + \cdots + a_k x_k = 0 \tag{2.5}$$

Definition 2.5 (Inner product). *Given a vector space $\mathcal{V}$ and a field $\mathcal{F} \in \mathbb{C}$, the inner product is defined as the map*

$$\langle ., . \rangle : \mathcal{V} \times \mathcal{V} \to \mathcal{F}$$

that satisfies the following properties:

1. *Conjugate symmetry:* $\langle x, y \rangle = \overline{\langle y, x \rangle} \quad \forall x, y \in \mathcal{V}$
2. *Linearity:* $\langle ax + by, z \rangle = a\langle x, z \rangle + b\langle y, z \rangle \quad \forall a, b \in \mathcal{F}$ *and* $\forall x, y, z \in \mathcal{V}$
3. *Positivity:* $\langle x, x \rangle > 0 \quad \forall x \in \mathcal{V} \backslash 0$

Definition 2.6 (Basis). *A subset $\mathcal{S}$ of a vector space, $\mathcal{V}$, is said to span $\mathcal{V}$ if every element of $\mathcal{V}$ can be represented as a linear combination of the elements of $\mathcal{S}$. If the subset $\mathcal{S}$ is also linearly independent, then it is called a basis of $\mathcal{V}$. Bases are non-unique and every vector space has a basis. Basis of a vector space $\mathcal{U}$ is denoted by $\mathcal{B}_{\mathcal{U}}$.*

Definition 2.7 (Isomorphism). *Given two vector spaces, $\mathcal{U}$ and $\mathcal{V}$ over the same field $\mathcal{F}$, an invertible function $f : \mathcal{U} \to \mathcal{V}$ such that $f(ax+by) = af(x)+bf(y)$ $\forall x, y \in \mathcal{U}$ and $\forall a, b \in \mathcal{F}$ is said to be an **isomorphism**; The vector spaces, $\mathcal{U}$ and $\mathcal{V}$ are also said to be isomorphic.*

Definition 2.8 (Matrix : array). *Matrices can be seen as a rectangular array of scalars of size $m \times n$ from a field $\mathcal{F}$. If $m = n$, then the matrix is said to be square. On a field $\mathcal{F}$, the set of all matrices of size $m \times n$ is denoted by $M_{m,n}(\mathcal{F})$.*

Definition 2.9 (Matrix : Linear transformation). *Given an m-dimensional space $\mathcal{V}$ and an n-dimensional space $\mathcal{U}$ over the scalar field $\mathcal{F}$, consider two basis, $\mathcal{B}_{\mathcal{U}}$ and $\mathcal{B}_{\mathcal{V}}$ for the vector spaces $\mathcal{U}$ and $\mathcal{V}$ respectively.*

*A **linear transformation** is defined as a function $\mathcal{T} : \mathcal{U} \to \mathcal{V}$ such that $\mathcal{T}(a_1 x_1 + a_2 x_2) = a_1 \mathcal{T}(x_1) + a_2 \mathcal{T}(x_2)$ for any arbitrary scalars a_1, a_2 and vectors x_1 and x_2.*

Consider two vectors u and v in $\mathcal{U}$ and $\mathcal{V}$ which are n-tuples and m-tuples respectively. A matrix $A \in M_{m,n}(\mathcal{F})$ is a linear transformation $\mathcal{T} : \mathcal{U} \to \mathcal{V}$ that transforms the vectors from the domain space, $u \in \mathcal{U}$ to a vector from the range space $v \in \mathcal{V}$, that is $v = \mathcal{T}(u)$. This holds if and only if $[v]_{\mathcal{B}_{\mathcal{V}}} = A[u]_{\mathcal{B}_{\mathcal{U}}}$, that is if the domain and range vectors are isomorphic to the basis of the vector spaces $\mathcal{U}$ and $\mathcal{V}$.

Definition 2.10 (Positivity and non-singularity of matrices). *A matrix $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ $(i, j) \in 1, .., n$ is said to be positive if all its entries are **positive**, $(a_{ij}) > 0$ and is said to be non-negative if $(a_{ij}) \geq 0$.*

*A matrix $A \in \mathbb{R}^{n \times n}$ is said to be **non-singular** if there exists some matrix $A^{inv} \in \mathbb{R}^{n \times n}$ such that*

$$AA^{inv} = A^{inv}A = I$$

with $I \in \mathbb{R}^{n \times n}$, the identity matrix. The matrix A^{inv} is called the multiplicative inverse of A and is generally denoted as A^{-1}.

Definition 2.11 (M-matrices). *A matrix $A \in \mathbb{R}^{n \times n}$ is said to be a **M-matrix** if*

1. $a_{ii} > 0 \quad \forall i = 1, \ldots, n$

2. $a_{ij} \leq 0 \quad \forall i, j = 1, \dots, n; i \neq j$
3. *A is nonsingular*
4. $A^{-1} \geq 0$

Definition 2.12 (Adjoint and Transpose). *For a matrix $A \in \mathbb{C}^{m \times n}$, the* ***adjoint*** *$A^*$ is defined such that*

$$\langle Ax, y \rangle = \langle x, A^* y \rangle$$

with $\langle ., . \rangle$ denoting the inner product.

The ***transpose*** *of a matrix $A \in \mathbb{C}^{m \times n}$ is denoted by $A^T \in \mathbb{C}^{n \times m}$ and is such that*

$$(a_{ij}^T) = (a_{ji})$$

Definition 2.13 (Orthogonality). *A set of vectors, $\mathcal{V} = \{x_1, x_2, \dots, x_k\} \in \mathbb{R}^n$ is said to be* ***orthogonal*** *if $x_i^T x_j = 0, \ \forall i \neq j$ and* ***orthonormal*** *if $x_i^T x_j = \delta_{ij}, \ i \neq j$,*
with $\delta_{ij} = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } i \neq j \end{cases}$. If two vectors are orthogonal, then they are maximally independent.

A set of subspaces $\mathcal{S} = \{\mathcal{S}_1, \dots \mathcal{S}_k\} \in \mathbb{R}^n$ is mutually orthogonal if $x^T y = 0$ for $x \in \mathcal{S}_i$ and $y \in \mathcal{S}_j$ for $i \neq j$.

A matrix $M \in \mathbb{R}^{n \times n}$ is said to be orthogonal if $M^T M = I$. Therefore, if $M = [m_1 | \dots | m_n]$ is orthogonal, then the column spaces, m_i, form a orthogonal basis for $\mathbb{R}^n$.

Definition 2.14 (Eigenvalues). *The* ***characteristic polynomial*** *of a matrix A is defined as*

$$p(x) = \det(A - xI)$$

where det *denotes the determinant. The* ***eigenvalues*** *are then the zeros of this polynomial.*

Every matrix $A \in \mathbb{R}^{n \times n}$ therefore has n eigenvalues. The set of eigenvalues of a matrix A is denoted by

$$\lambda(A) = \{x : \det(A - xI) = 0\} \in \mathbb{C}^n$$

If the eigenvalues are real, then they can be ordered:

$$\lambda_n(A) \leq \cdots \leq \lambda_1(A)$$

$\lambda_{\max}(A)$ and $\lambda_{\min}(A)$ are sometimes used to denote the maximal and the minimum eigenvalues of the matrix A.

Definition 2.15 (Eigenvector). *Given the eigenvalues of a matrix $A \in \mathbb{C}^{n \times n}$, $\lambda \in \lambda(A)$, there exists a non-zero vector x such that $Ax = \lambda x$. This vector is called an* ***eigenvector*** *for A associated with the eigenvalue, λ.*

Definition 2.16 (Singular values). *Given the eigenvalues of a matrix $A \in \mathbb{C}^{n \times n}$, $\lambda \in \lambda(A)$, the* ***singular values****, σ_i are defined as the square roots of the eigenvalues of its self-adjoint operator AA^*, with A^* denoting the adjoint of A:*

$$\sigma_i(A) = \sqrt{\lambda_i(AA^*)}$$

Definition 2.17 (Spectal radius). *The* ***spectal radius*** *of a matrix A is given by*

$$\rho(A) = \max_{\lambda \in \lambda(A)} |\lambda|$$

Theorem 2.1.1 (Perron-Frobenius theorem). *Let $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ be a positive matrix. Then there exists a positive real number λ_{pf}, called the Perron root or the Perron-Frobenius eigenvalue, that is the leading/dominant eigenvalue of A, that is $|\lambda| < \lambda_{pf}$, with $\lambda \in \lambda(A)$. Hence the spectral radius is equal to λ_{pf}, $\rho(A) = \lambda_{pf}$. Additionally the following properties hold:*

1. *There exists a positive eigenvector v of A associated with λ_{pf}, that is $(v_j) > 0$ that is unique, upto positive multiples and is known as the Perron-Frobenius eigenvector.*
2. *The Perron-Frobenius eigenvalue satisfies the condition,*

$$\min_i \sum_j a_{ij} \leq \lambda_{pf} \leq \max_i \sum_j a_{ij}$$

Proof. See Strang [Str23]. □

Lemma 2.1.2. *Given an M-matrix, $A \in \mathbb{R}^{n\times n}$, we have $\rho(B) < 1$, where $B = I - D^{-1}A$, where D is the matrix with diagonal entries of A.*

Definition 2.18 (Similarity and Diagonalizability)**.** *If a matrix $X \in \mathbb{C}^{n\times n}$ is non-singular and $B = X^{-1}AX$, then the matrices A and $B \in \mathbb{C}^{n\times n}$ are said to be* ***similar****. Similar matrices have the exact same eigenvalues.*

If $A \in \mathbb{C}^{n\times n}$ has n independent eigenvectors, $x_1, \ldots, x_n$ and $Ax_i = \lambda_i x_i,\ i \in (1, \ldots, n)$, then A is said to be ***diagonalizable****. Therefore, if $X = [x_1|\ldots|x_n]$,*

$$X^{-1}AX = diag(\lambda_1, \ldots, \lambda_n)$$

While not all matrices are diagonalizable, for a symmetric matrix, $A \in \mathbb{R}^{n\times n}$ we have an orthogonal matrix Q such that

$$Q^T AQ = diag(\lambda_1, \ldots, \lambda_n)$$

which is called a ***Schur decomposition****. The maximal and minimum eigenvalues are then given by*

$$\lambda_{\max} = \max_{x\neq 0} \frac{x^T Ax}{x^T x}$$

$$\lambda_{\min} = \min_{x\neq 0} \frac{x^T Ax}{x^T x}$$

Definition 2.19 (Definiteness of a matrix)**.** *A matrix $A \in \mathbb{R}^{n\times n}$ is said to be* ***positive semi-definite*** *if*

$$x^T Ax \geq 0, \qquad \forall x \in \mathbb{R}^n \tag{2.6}$$

and ***positive definite*** *for the case of strict inequality (> 0) whenever $x \neq 0$.*

Lemma 2.1.3. *If A is* ***symmetric and positive (semi) definite (SPD)****, then all its eigenvalues are (non-negative) positive. If A is not symmetric, then A is positive (semi) definite if and only if $A + A^T$ is positive (semi) definite.*

Proof. Let λ be an eigenvalue of the SPD matrix A. If $\lambda = 0$, then there exists some eigenvector x such that $Ax = 0$. This is a contradication, since, $x^T Ax > 0$. Let $\lambda < 0$. There exists some x such that $Ax = \lambda x$. Therefore, we have $x^T Ax = \lambda|x|^2$, which is negative since $|x|^2 > 0$ and $\lambda < 0$. Therefore, this is also a contradiction. Hence for a SPD matrix, we have $\lambda > 0$. A non-strict equality with $\lambda = 0$ holds for the semi-definite case. □

Definition 2.20 (Vector Norm)**.** *A* ***vector norm*** *on $\mathbb{R}^n$ is a function $f : \mathbb{R}^n \to \mathbb{R}$ that satisfies:*

$$\begin{aligned} &f(x) \geq 0, && x \in \mathbb{R}^n, \\ &f(x+y) \leq f(x) + f(y), && x, y \in \mathbb{R}^n, \\ &f(\alpha x) = |\alpha| f(x), && \alpha \in \mathbb{R}, x \in \mathbb{R}^n, \end{aligned}$$

Such a function $f(x)$ is denoted by $\|x\|$. A general norm is the p-norm

$$\|x\|_p = (|x_1|^p + \cdots + |x_n|^p)^{\frac{1}{p}}, \quad p \geq 1 \tag{2.7}$$

In particular, the most useful norms are with $p = 1, 2, \infty$

$$\begin{aligned}
\|x\|_1 &= |x_1| + \cdots + |x_n| \\
\|x\|_2 &= (|x_1|^2 + \cdots + |x_n|^2)^{\frac{1}{2}} = (x^T x)^{\frac{1}{2}} \\
\|x\|_\infty &= \max_{1 \le i \le n} |x_i|
\end{aligned}$$

A unit vector w.r.t a norm $\|\cdot\|$ is a vector x that satisfies $\|x\| = 1$.

Property 2.1.4 (Hölder inequality). *For p-norms, the following inequation holds:*

$$|x^T y| \le \|x\|_p \, \|y\|_q \qquad \frac{1}{p} + \frac{1}{q} = 1 \tag{2.8}$$

A special case for $p = 2$ is called the Cauchy-Schwarz inequality

$$|x^T y| \le \|x\|_2 \, \|y\|_2 \tag{2.9}$$

Theorem 2.1.5 (Norm equivalence theorem). *Given two p-norms on $\mathbb{R}^n$, $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$, there exist positive constants, c_1 and c_2 such that*

$$c_1 \|x\|_\alpha \le \|x\|_\beta \le c_2 \|x\|_\alpha \quad \forall x \in \mathbb{R}^n \tag{2.10}$$

For example,

$$\|x\|_2 \le \|x\|_1 \le \sqrt{n} \, \|x\|_2 \quad \forall x \in \mathbb{R}^n \tag{2.11}$$
$$\|x\|_\infty \le \|x\|_2 \le \sqrt{n} \, \|x\|_\infty \quad \forall x \in \mathbb{R}^n \tag{2.12}$$
$$\|x\|_\infty \le \|x\|_1 \le n \, \|x\|_\infty \quad \forall x \in \mathbb{R}^n \tag{2.13}$$
$$\tag{2.14}$$

Proof. See Ortega [OR00]. □

Definition 2.21 (Matrix norms). *The **matrix norm** is defined in an equivalent fashion to the vector norm, since $\mathbb{R}^{m \times n} \cong \mathbb{R}^{mn}$ (isomorphism). Therefore given matrices $A, B \in \mathbb{R}^{m \times n}$, the function $f : \mathbb{R}^{m \times n} \to \mathbb{R}$ is a matrix norm if the following properties hold:*

$$\begin{aligned}
&f(A) \ge 0, && A \in \mathbb{R}^n, \\
&f(A + B) \le f(A) + f(B), && A, B \in \mathbb{R}^n, \\
&f(\alpha A) = |\alpha| f(A), && \alpha \in \mathbb{R}, A \in \mathbb{R}^n,
\end{aligned}$$

Similar to the vector norms, we use $\|A\| = f(A)$ to denote the matrix of the matrix A.

Some commonly used norms are the Frobenius norm,

$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$$

and the matrix p-norms,

$$\|A\|_p = \sup_{x \ne 0} \frac{\|Ax\|_p}{\|x\|_p}$$

We note that

$$\|A\|_p = \sup_{x \ne 0} \left\| A \left(\frac{x}{\|x\|_p} \right) \right\|_p = \max_{\|x\|_p = 1} \|Ax\|_p$$

Property 2.1.6 (Matrix norm properties)**.** *For the matrix Frobenius and p-norms, the following properties hold:*

$$\|A\|_2 \leq \|A\|_F \leq \sqrt{\min\{m,n\}}\,\|A\|_2 \tag{2.15}$$

$$\max_{i,j} |a_{ij}| \leq \|A\|_2 \leq \sqrt{mn}\max_{i,j} |a_{ij}| \tag{2.16}$$

$$\|A\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{m} |a_{ij}| \tag{2.17}$$

$$\|A\|_\infty = \max_{1\leq i\leq m} \sum_{j=1}^{n} |a_{ij}| \tag{2.18}$$

$$\frac{1}{\sqrt{n}}\|A\|_\infty \leq \|A\|_2 \leq \sqrt{m}\,\|A\|_\infty \tag{2.19}$$

$$\frac{1}{\sqrt{m}}\|A\|_1 \leq \|A\|_2 \leq \sqrt{n}\,\|A\|_1 \tag{2.20}$$

For any submatrix A_{sub} of a matrix A,

$$\|A_{sub}\|_p \leq \|A\|_p \tag{2.21}$$

For two matrices A, B,

$$\|AB\|_p \leq \|A\|_p \|B\|_p, \quad 1 \leq p \leq \infty \tag{2.22}$$

Lemma 2.1.7. *For any matrix $A \in \mathbb{R}^{n\times n}$, let $\|\cdot\|$ denote a matrix norm, then*

$$\rho(A) \leq \|A\|$$

Proof. Let $\Lambda = \operatorname{diag}\{\lambda_i(A)\}$ be the diagonal matrix with the eigenvalues of A and let $X \neq 0$ be the corresponding eigenvectors. We then have

$$AX = \Lambda X, \quad X = [x_1|x_2|\dots|x_n] \in \mathbb{R}^{n\times n}\backslash 0$$

Therefore

$$|\Lambda|\,\|X\| = \|\Lambda X\| = \|AX\| \leq \|A\|\,\|X\|$$

With $\|X\| > 0$, we have

$$|\Lambda| \leq \|A\|$$

Taking the maximum over the eigenvalues, gives us

$$\rho(A) \leq \|A\|$$

□

Lemma 2.1.8 (Gelfand's formula)**.** *For any matrix $A \in \mathbb{R}^{n\times n}$, let $\|\cdot\|$ denote a matrix norm, then*

$$\rho(A) = \lim_{k\to\infty} \left\|A^k\right\|^{\frac{1}{k}}$$

Proof. See Ortega [OR00]. □

Definition 2.22 (Condition number)**.** *Given a matrix $A \in \mathbb{R}^{n\times n}$, and some matrix norm $\|\cdot\|$, the* ***condition number*** *of matrix A is defined as*

$$\kappa(A) = \|A\|\,\|A^{-1}\|$$

The 2-norm condition number of a matrix A is

$$\kappa_2(A) = \|A\|_2\,\|A^{-1}\|_2 = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

The condition number of a matrix is scaling-invariant and is an useful indication of the degree of singularity of the matrix. A matrix is said to be ***ill-conditioned*** *if it has a large condition number and* ***well-conditioned*** *if it has a small condition number.*

Property 2.1.9. *Condition numbers are norm dependent, but due to the norm-equivalence theorem, can be related through some constants c_1 and c_2. Therefore, for two condition numbers defined with two norms α and β,*

$$c_1\kappa_\alpha(A) \leq \kappa_\beta(A) \leq c_2\kappa_\alpha(A) \quad A \in \mathbb{R}^{n\times n}$$

For example, we have

$$\frac{1}{n}\kappa_2(A) \leq \kappa_1(A) \leq n\kappa_2(A)$$
$$\frac{1}{n}\kappa_\infty(A) \leq \kappa_2(A) \leq n\kappa_\infty(A)$$
$$\frac{1}{n^2}\kappa_1(A) \leq \kappa_\infty(A) \leq n^2\kappa_1(A)$$

Additionally, for any p-norms we have that $\kappa_p(A) \geq 1$

Definition 2.23 (Convergence of vector/matrix sequences)**.** *We say that a sequence of n vectors, denoted by $\{x_{(k)}\}$ converges if*

$$\lim_{n\to\infty} \left\| x_{(k)} - x \right\| = 0$$

and a sequence of n matrices, $\{A_{(k)}\}$ converges if

$$\lim_{n\to\infty} \left\| A_{(k)} - A \right\| = 0$$

For both cases the choice if norm is not important due to the Norm equivalence theorem.

Theorem 2.1.10 (Matrix series convergence)**.** *Given a matrix $A \in \mathbb{C}^n$, we have*

$$\lim_{k\to\infty} A^k = 0 \quad \Longleftrightarrow \quad \rho(A) < 1$$

Proof. See Ortega [OR00]. □

Theorem 2.1.11 (Matrix 2-norm)**.** *Given a matrix, $A \in \mathbb{R}^{m\times n}$, there exists a unit 2-norm vector $z \in \mathbb{R}^n$ such that*

$$A^T Az = \nu z$$

where $\nu = \|A\|_2^2$

Proof. See Golub [GVL13]. □

Corollary 2.1.11.1. *Using the theorem above, we have that ν is a zero of $p(\lambda) = \det(A^TA - \lambda I)$, therefore,*

$$\|A\|_2 = \sqrt{\lambda_{\max}(A^TA)}$$

where $\lambda = \lambda(A)$ is the set of eigenvalues of A and $\lambda_{\max}$ is the maximal eigenvalue.

As estimating the eigenvalues is an expensive operation, the matrix 2-norm can also be estimated using the 1-norm and the ∞-norm.

Corollary 2.1.11.2. *Given a matrix $A \in \mathbb{R}^{m\times n}$,*

$$\|A\|_2 \leq \sqrt{\|A\|_1 \|A\|_\infty}$$

Definition 2.24 (Reducibility). *A matrix $A \in \mathbb{C}^{n\times n}$ is said to be* ***reducible*** *if there exists a permutation matrix P such that*

$$PAP^T = \begin{pmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{pmatrix}$$

A matrix is said to be ***irreducible*** *if it is not reducible.*

If a matrix has all its nonzeros, then it is irreducible.

Definition 2.25 (Diagonal dominance). *A matrix $A \in \mathbb{C}^{n\times n}$ is said to be* ***diagonally dominant*** *if*

$$\sum_{j=1, j\neq i} |a_{ij}| \leq |a_{ii}| \;\; i = 1, .., n$$

and strictly diagonal dominant for strict inequality for all i.

Additionally, the matrix is said to be ***irreducibly diagonally dominant*** *if it is irreducible, diagonally dominant and the strict inequality holds for atleast one row.*

Theorem 2.1.12 (Diagonal dominance theorem). *A matrix $A \in \mathbb{C}^{n\times n}$ that is strictly or irreducibly diagonally dominant is invertible.*

Proof. See Ortega [OR00]. □

Theorem 2.1.13 (Gershgorin circle theorem). *Let $A \in \mathbb{C}^{n\times n}$ and let $\mathbb{S}$ be the set of complex numbers defined as*

$$\mathbb{S} = \bigcup_{i=1}^{n} \left\{ z \;\; s.t \;\; |a_{ii} - z| \leq \sum_{j\neq i} |a_{ij}| \right\}$$

Then every eigenvalue of A, $\lambda \in \lambda(A)$ is in the circle $\mathbb{S}$.

Proof. If λ is an eigenvalue of A and assume that it is not in $\mathbb{S}$: $\lambda \notin \mathbb{S}$, then

$$|a_{ii} - \lambda| > \sum_{j\neq i} |a_{ij}| \;\; i = 1, \ldots, n$$

But we know that $A - \lambda I$ is strictly diagonally dominant and therefore by Theorem 2.1.12, nonsingular. Therefore, this is a contradiction and all eigenvalues of A must be in $\mathbb{S}$. □

Corollary 2.1.13.1. *If $A \in \mathbb{R}^{n\times n}$ is symmetric, irreducibly diagonally dominant and has positive diagonal elements, then A is positive definite (SPD).*

Proof. The eigenvalues, λ_i of A are real, from Theorem 2.1.13 and diagonal dominance, we have

$$\lambda_i \geq 0 \;\; i = 1, \ldots, n$$

As A is also irreducible and diagonally dominant, from Theorem 2.1.12, we have that it is invertible, therefore $\lambda_i > 0 \;\; \forall i$ and hence it is SPD. □

Definition 2.26 (Subinverse and superinverse). *For a given matrix $A \in \mathbb{R}^{n\times n}$, a matrix $B \in \mathbb{R}^{n\times n}$ is called its left subinverse (left superinverse) if*

$$BA \leq I \quad (BA \geq I)$$

and its right subinverse (right superinverse) if

$$AB \leq I \quad (AB \geq I)$$

B is a subinverse (superinverse) if both conditions hold.

Theorem 2.1.14 (Neumann theorem)**.** *For some matrix $B \in \mathbb{R}^{n\times n}$, assume that $\rho(B) < 1$. Then $(I - B)^{-1}$ exists and is given by*

$$(I - B)^{-1} = \lim_{k\to\infty} \sum_{i=0}^{k} B^i$$

Proof. See Ortega [OR00]. □

Definition 2.27 (Matrix splitting)**.** *Let $A, B \in \mathbb{R}^{n\times n}$ be two matrices, then $A = B - C$ is called a **regular splitting** of A if B is invertible, $B^{-1} \geq 0$ and $C \geq 0$. It is called a **weak regular splitting** if instead of $C \geq 0$, we have $B^{-1}C \geq 0$ and $CB^{-1} \geq 0$.*

A regular splitting is weak, but the reverse is not true.

Theorem 2.1.15. *Given a matrix $B \in \mathbb{R}^{n\times n}$ that is invertible and that $B^{-1} \geq 0$, we have that B^{-1} is a subinverse of some matrix $A \in \mathbb{R}^{n\times n}$ if and only if $A = B - C$ is a weak regular splitting.*

Proof. See Ortega [OR00]. □

Definition 2.28 (Contracting operators)**.** *An operator $\mathcal{L}$ from $\mathbb{R}^n \to \mathbb{R}^n$ is a **contracting operator** on a subset D on $\mathbb{R}^n$ if there exists an $\alpha < 1$ such that $\|\mathcal{L}x - \mathcal{L}y\| \leq \alpha \|x - y\| \quad \forall x, y \in D$.*

We see that a linear operator, $A \in \mathbb{R}^{n\times n}$ is contracting if and only if $\|A\| < 1$. Note that this is norm-dependent.

Lemma 2.1.16 (Bounds on contracting operator norm)**.** *Let $A \in \mathbb{R}^{n\times n}$ be some contracting operator and let $u > 0 \in \mathbb{R}^n$ and $\theta > 0$ be some positive vector and some positive constant respectively such that*

$$|A|u \leq \theta u$$

Then we have

$$\|A\|_u \leq \theta$$

and in particular

$$\|Ax\|_u \leq \theta \|x\|_u \, \forall x \in \mathbb{R}^n$$

Theorem 2.1.17 (Fixed point theorem)**.** *If $\mathcal{L}$ is a operator from $\mathbb{R}^n \to \mathbb{R}^n$ and is contracting on some closed set $D_0 \subset D$ and $\mathcal{L}D_0 \subset D_0$ then it is guaranteed to have a **unique fixed point** in the subset D_0.*

Proof. Let x_0 be an arbitrary point in D_0. Define a sequence

$$x_k = \mathcal{L}x_{k-1} \quad k = 1, 2, \ldots$$

Since $\mathcal{L}D_0 \subset D_0$, the sequence $\{x_k\}$ is well-defined and lies in D_0. We also have

$$\|x_{k+1} - x_k\| = \|\mathcal{L}x_k - \mathcal{L}x_{k-1}\| \leq \alpha \|x_k - x_{k-1}\|$$

giving us

$$\|x_{k+p} - x_k\| \leq \sum_{i=1}^{p} \|x_{k+i} - x_{k+i-1}\| \leq (\alpha^{p-1} + \cdots + 1) \|x_{k+1} - x_k\| \leq \frac{\alpha^k}{(1-\alpha)} \|x_1 - x_0\|$$

Therefore, $\{x_k\}$ is a Cauchy sequence and has a limit x^* in D_0. Moreover, with the continuity of $\mathcal{L}$, we have $\lim_{k\to\infty} \mathcal{L}x_k = \mathcal{L}x^*$, therefore x^* is a fixed point of $\mathcal{L}$. □

2.2 Solution of linear systems

In this section, we define some common methods that can be used to solve linear systems, including those that are used in the later chapters. In general, the methods for the solution of linear systems can be divided into two broad categories: Direct methods, which compute the solution in a single step, and iterative methods, which successively approximate the solution. The main linear system that forms our focus is

$$Ax = b \tag{2.23}$$

with a non-singular matrix $A \in \mathbb{R}^{n\times n}$ and the vectors $x, b \in \mathbb{R}^n$. We do not specify the sparsity of the matrix at this stage and all methods are applicable regardless of their sparsity. This linear system is generally assembled from the application of a linear operator $\mathcal{L}$ or through the linearization of a non-linear operator.

2.2.1 Direct methods and factorizations

The direct solution of the linear system in Equation (2.23) in most cases is algebraically equivalent to the Gaussian elimination method [DER17]. Direct methods, particularly for square matrices rely on two main steps: a factorization step, that aims to generate factors of the matrix, and a solve step, which uses the factors and produces a solution, given a right-hand side. The central concept behind these direct methods is that the solution using these factors is significantly cheaper than the solution with the full matrix (by computing its inverse).

Definition 2.29 (Triangular matrices). *A matrix is called* ***upper triangular*** *if all entries below the main diagonal are zero.*

$$U = u_{ij} = 0 \quad \forall i < j \;\; \mathbb{R}^{n\times n}$$

Similarly a matrix is called ***lower triangular*** *if all entries above the main diagonal are zero*

$$L = l_{ij} = 0 \quad \forall i > j \;\; \mathbb{R}^{n\times n}$$

A lower (upper) triangular matrix is said to be unit lower (upper) triangular if it has ones on the main diagonal.

$$l_{ii} = 1 \quad \forall i = \{1, .., n\}$$

Definition 2.30 (Upper triangular solve: Back-substitution). *Given an upper triangular matrix, a triangular solve aims to solve the equation,*

$$Ux = c$$

This can be solved in two steps:

$$x_n = \frac{c_n}{u_{nn}}$$

$$x_k = \frac{c_k - \sum_{j=k+1}^{n} u_{kj}x_j}{u_{kk}}, \quad k = n-1, n-2, \ldots, 1$$

We additionally require that $u_{kk} \neq 0 \;\; k = 1, \ldots, n$.

Definition 2.31 (Lower triangular solve: Forward-substitution). *Given a lower triangular matrix, a triangular solve aims to solve the equation,*

$$Lc = b$$

This can be solved in two steps:

$$c_1 = \frac{b_1}{l_{11}}$$

$$c_k = \frac{b_k - \sum_{j=1}^{k-1} l_{kj} c_j}{l_{kk}}, \quad k = 2, 3, \dots, n$$

We additionally require that $l_{kk} \neq 0 \;\; k = 1, \dots, n$.

Definition 2.32 (Factorized solve). *Given a matrix* $A \in \mathbb{R}^{n \times n}$*, and its upper and lower triangular factors* $L, U \in \mathbb{R}^{n \times n}$ *such that* $A = LU$*, we can then solve the linear system in Equation (2.23) with two sequential steps:*

$$Ax = b$$

$$L(Ux) = b$$

$$Lc = b \quad \textit{(step 1: Forward substitution)}$$

$$Ux = c \quad \textit{(step 2: Backward substitution)}$$

Definition 2.33 (LU factorization). *A matrix* $A \in \mathbb{R}^{n \times n}$ *can be factorized into two triangular factors* $L, U \in \mathbb{R}^{n \times n}$ *such that* $A = LU$*, where* L *is a lower triangular matrix and* U *is an upper triangular matrix. The component wise update is given by*

$$l_{ii} = 1 \quad i = 1, \dots, n$$

$$u_{ij} = a_{ij} \quad j = i, \dots, n$$

$$l_{ji} = \frac{a_{ji}}{u_{ii}} \quad j = i+1, \dots, n$$

$$a_{jl} = a_{jl} - \sum_{k=i+1}^{n} l_{jk} u_{kl} \quad j, l = i+1, \dots, n$$

With these steps being performed sequentially in a loop for each row. This is called the ***right-looking LU****.*

Corollary 2.2.0.1 (Uniqueness of LU factorization). *The LU factorization of a matrix,* $A = LU$ *is unique if* L *is strictly lower triangular, i.e, it has a unit main diagonal.*

Definition 2.34 (Cholesky factorization). *A symmetric matrix* $A \in \mathbb{R}^{n \times n}$, $A^T = A$ *can be factorized into two triangular factors* $L, U \in \mathbb{R}^{n \times n}$*, where* L *is a lower triangular matrix and* U *is an upper triangular matrix, with* $U = L^T$

Lemma 2.2.1 (Cost of factorization methods). *The LU factorization of a non-symmetric matrix,* $A = LU$ *is*

$$\mathscr{C}_{LU} = \frac{2}{3} n^3 + \mathcal{O} n^2$$

In case of the factorization of symmetric matrix, $A = LL^T$*, the cost*

$$\mathscr{C}_{chol} = \frac{1}{3} n^3 + \mathcal{O} n^2$$

A triangular solve costs

$$\mathscr{C}_{tri-solve} = n^2 + \mathcal{O}(n)$$

Proof. See Duff [DER17]. □

2.2.2 Iterative methods

In contrast to the direct methods, iterative methods produce successive approximations that aim to converge to a fixed point. We first define a general iterative process.

Definition 2.35 (General iterative process)**.** *Consider a family of operators* $\{\mathcal{L}_k\}$*,*

$$\mathcal{L}_k : D_k \subset \mathbb{R}^{n\times(k+p)} \to \mathbb{R}^n, \quad k = 0, 1, \dots$$

defines an iterative process $\mathcal{I} = (\{\mathcal{L}_k\}, D^*, p)$ *with* p *initial points and with domain* $D^* \subset D_0$*, if the set* D^* *is not empty and if for any point* $x_0, \dots, x^{-p+1} \in D^*$*, the sequence* $\{x_k\}$ *generated by*

$$x_{k+1} = \mathcal{L}_k(x_k, \dots, x_{-p+1}), \quad k = 0, 1, \dots$$

exists. A point x^* *such that* $\lim_{k\to\infty} x_k = x^*$ *is called the* ***limit of the process****, akin to a fixed point defined in Theorem 2.1.17.*

Definition 2.36 (Iteration matrices)**.** *The general iterative process in Definition 2.35 can be written as*

$$x_{k+1} = Gx_k + f \tag{2.24}$$

The matrix G *is called the* ***iteration matrix*** *for the linear fixed point iteration.*

Definition 2.37 (Iteration matrices for general splitting)**.** *Consider a general splitting,* $A = M - N$ *and the general linear fixed point iteration in Equation (2.24). Let*

$$G = M^{-1}N = M^{-1}(M - A) = I - M^{-1}A$$

and

$$f = M^{-1}b$$

then we can write the fixed point iteration as

$$x_{k+1} = M^{-1}Nx_k + M^{-1}b$$

Definition 2.38 (Convergence rate and iteration count)**.** *Given some matrix* $A \in \mathbb{R}^{n\times n}$*, the* ***convergence rate*** *of an iterative method is denoted by*

$$\varrho(A) = f(\kappa(A))$$

where $\kappa(A)$ *is the condition number of the matrix* A*.*

Given some initial residual, $\tau_{initial} = \|b - Ax_0\|_2$ *and a final tolerance,*

$$\tau_{final} \approx C\epsilon_{machine}, \quad \in (0, 1)$$

for some $C > 1$*, we would like to converge to, the number of iterations to convergence,* $\imath$*, is given by*

$$\tau_{initial}(\varrho(A))^{\imath} = \tau_{final}$$

Therefore, without loss of generality, assuming initial residual norm, $\tau_{initial} = 1$*, we have*

$$\imath = \frac{\log(\tau_{final})}{\log(\varrho(A))}$$

We see that $0 < \varrho(A) \leq 1$*.* $\varrho(A)$ *close to* 0 *implies* ***fast convergence*** *and a* $\varrho(A)$ *close to* 1 *implies* ***slow convergence*** *or stagnation.*

We first consider some simple relaxation methods, and then cover the state-of-the-art methods such as the Krylov methods.

Definition 2.39 (Jacobi iteration). *Let $A \in \mathbb{R}^{n \times n}$ be a matrix and $x, b \in \mathbb{R}^n$ be vectors in the equation $Ax = b$, for which we want to solve for x. The* ***Jacobi relaxation method*** *is a simple diagonal relaxation method that aims to reduce the residual vector, $r = b - Ax$. Therefore, the i-th component can be written as*

$$(b - Ax_{k+1})_i = 0$$

giving us the iteration (written component wise),

$$x_{k+1,i} = \frac{1}{a_{ii}}\Big(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_{k,j}\Big), \quad i = 1, \dots, n$$

If we split the input matrix $A = D - E - F$, with D, a diagonal matrix, and E and F triangular matrices, we can write the Jacobi iteration as

$$x_{k+1} = D^{-1}(E + F)x_k + D^{-1}b$$

the iteration matrix can then be written as

$$G_{Jacobi} = I - D^{-1}A$$

We can also write the Jacobi iteration with a general splitting with $M = D, N = D - A$.

We note that in the Jacobi iteration, we can update each element of the solution vector independently, as there are no dependencies between the elements in each iteration. Therefore, we use only data from the previous iteration.

If we use the latest data for each element, we get the Gauss-Seidel iteration.

Definition 2.40 (Gauss-Seidel iteration). *Let $A \in \mathbb{R}^{n \times n}$ be a matrix and $x, b \in \mathbb{R}^n$ be vectors in the equation $Ax = b$, for which we want to solve for x. The* ***Gauss-Seidel relaxation method*** *aims to utilize the latest data within each iteration. The component wise iteration can therefore be written as*

$$x_{k+1,i} = \frac{1}{a_{ii}}\Big(b_i - \sum_{j=1}^{i-1} a_{ij} x_{k+1,j} - \sum_{j=i+1}^{n} a_{ij} x_{k,j}\Big), \quad i = 1, \dots, n$$

If we split the input matrix $A = D - E - F$, with D, a diagonal matrix, and E and F triangular matrices, we can write the Gauss-Seidel iteration as

$$x_{k+1} = (D - E)^{-1}Fx_k + (D - E)^{-1}b$$

Instead of updating components in sequence from $i = 1$ to $i = n$, we can also do it in a reverse fashion, starting from $i = n$, which gives us the backward Gauss-Seidel iteration

$$x_{k+1} = (D - F)^{-1}Ex_k + (D - F)^{-1}b$$

In terms of the $A = D - E - F$, we can write the iteration matrix for the Gauss-Seidel iteration as

$$G_{Gauss-Seidel} = I - (D - E)^{-1}A$$

With a more general splitting, $A = M - N$, the Gauss-Seidel iteration is equivalent to $M = D - E$ and $N = M - A = F$.

Definition 2.41 (Richardson iteration)**.** *The* ***Richardson iteration*** *is equivalent to the Jacobi method, with an additional relaxation parameter, ω. It can be written as*

$$x_{k+1} = x_k + \omega(b - Ax_k)$$

where ω is a nonnegative scalar. Writing it in a fixed point iteration form,

$$x_{k+1} = (I - \omega A)x_k + \omega b$$

we see that the iteration matrix, $G_{Richardson} = I - \omega A$ and the convergence factor is hence $\rho(G) = \rho(I - \omega A)$.

Definition 2.42 (Preconditioning)**.** *The linear fixed point iteration in Equation (2.24) essentially solves the system,*

$$(I - G)x = f$$

Therefore, G has the form $G = I - M^{-1}A$, which is the iteration matrix and the system can be written as

$$M^{-1}Ax = M^{-1}b$$

This modified system has the same solution as the original linear system equation, $Ax = b$. The idea of solving the modified linear system is that if the matrix M approximates the original matrix A, and its inverse application is cheap, then we reduce the cost of solution of the linear system. This modified system is called a ***preconditioned system****.*

The relaxation methods mentioned above can be used as preconditioners as well.

$$M_{Jacobi} = D$$

$$M_{Gauss-Seidel} = D - E$$

Theorem 2.2.2 (Fixed point iteration convergence)**.** *Let $G \in \mathbb{R}^{n\times n}$ be a matrix such that $\rho(G) < 1$, then $(I - G)$ is non-singular and the iteration in Equation (2.24) converges for any x_0, f.*

Additionally, from Lemma 2.1.7 we have that $\rho(G) \leq \|G\|$. Therefore, in practice as the spectral radius is expensive to compute, the sufficient condition for convergence is $\|G\| < 1$.

Proof. See Ortega [OR00]. □

Theorem 2.2.3 (Convergence for a regular splitting)**.** *Consider a regular splitting, $A = M - N$. We associate with it, the iteration*

$$x_{k+1} = M^{-1}Nx_k + M^{-1}b$$

This iteration converges, i.e $\rho(M^{-1}N) < 1$ if and only if A is non-singular and A^{-1} is nonnegative.

Proof. Defining the iteration matrix, $G = M^{-1}N$, we have that for convergence $\rho(G) < 1$. We also have the relation $A = M(I - G)$, which means that A is non-singular. From Theorem 2.1.14, we have that $(I - B)^{-1}$ is nonnegative. Due to the regular splitting, we have that M^{-1} is also nonnegative. Therfore, we have A^{-1} is nonnegative as $A^{-1} = (I - G)^{-1}M^{-1}$. For the proof of the sufficient condition, see Saad [Saa03]. □

Theorem 2.2.4 (Convergence of the Richardson iteration)**.** *The Richardson iteration, defined in Definition 2.41 has the iteration matrix, $G_{Richardson} = I - \omega A$. Let $\lambda_i \in \{\lambda(A)\}$ be the eigenvalues of the matrix A and let $\lambda_{\min}, \lambda_{\max}$ be its minimal and maximal eigenvalues. The Richardson iteration converges if*

$$0 < \omega < \frac{2}{\lambda_{\max}}$$

Additionally, the optimal value of the relaxation parameter for fastest convergence is

$$\omega_{optimal} = \frac{2}{\lambda_{\min} + \lambda_{\max}}$$

Proof. We see that the eigenvalues, μ_i of the iteration matrix, $G_{Richardson}$ are such that

$$1 - \omega\lambda_{\max} \leq \mu_i \leq 1 - \omega\lambda_{\min}$$

If $\lambda_{\max} > 0$ and $\lambda_{\min} < 0$, we have atleast one eigenvalue > 1, therefore $\rho(G_{Richardson,\omega}) > 1 \;\; \forall\omega$, implying divergence. Now assume that all eigenvalues are positive, $\lambda_{\min} > 0$. Then the condition for convergence is

$$1 - \omega\lambda_{\min} < 1$$

$$1 - \omega\lambda_{\max} > -1$$

The first condition implies that we require $\omega > 0$ and the second gives us that $\omega \leq \frac{1}{\lambda_{\max}}$. Therefore, the method converges if

$$0 < \omega < \frac{2}{\lambda_{\max}}$$

For the optimal value of ω, we look at the spectral radius of $G_{Richardson,\omega}$.

$$\rho(G_{Richardson,\omega}) = \max\{|1 - \omega\lambda_{\min}|, |1 - \omega\lambda_{\max}|\}$$

This is acheived when

$$-1 + \lambda_{\max}\omega = 1 - \lambda_{\min}\omega$$

Solving for ω, we get the optimal value of the relaxation parameter as

$$\omega_{optimal} = \frac{2}{\lambda_{\min} + \lambda_{\max}}$$

□

Theorem 2.2.5 (Convergence rate estimates: Relaxation methods)**.** *Given some matrix $A \in \mathbb{R}^{n \times n}$, the convergence rate of an relaxation based iterative method is bounded by the inverse of its condition number*

$$\varrho(A) \approx 1 - \frac{C}{\kappa(A)}$$

where $\kappa(A)$ is the condition number of the matrix A and C is some constant, $\mathcal{O}(1)$.

Proof. Let G be the iteration matrix for some relaxation method with system matrix A. For a relaxation method, we have $G = I - M^{-1}A$, with some regular splitting $A = M - N$.

$$\begin{aligned}
\varrho(A) &= \rho(G) \\
&\leq \left\| I - M^{-1}A \right\| \\
&\leq 1 - \left\| M^{-1}A \right\| \\
&\leq 1 - \frac{\|A\|}{\|A\| - \|N\|} \\
&\leq 1 - \frac{C}{\|A\| \, \|A^{-1}\|} \\
&\leq 1 - \frac{C}{\kappa(A)}
\end{aligned} \tag{2.25}$$

□

2.2.2.1 Krylov subspace methods

Definition 2.43 (Projection methods). *Let $A \in \mathbb{R}^{n\times n}$ be a matrix and let $\mathcal{K}, \mathcal{L} \in \mathbb{R}^m \subset \mathbb{R}^n$ be subspaces. A **projection method** aims to find an approximate solution $\tilde{x}$ to $Ax = b$, by searching for the approximate solution in the subspace $\mathcal{K}$, and imposing the condition that the residual vector $b - A\tilde{x}$ be orthogonal to the subspace $\mathcal{L}$*

$$\text{Find } \tilde{x} \in \mathcal{K} \qquad \text{such that} \quad b - A\tilde{x} \perp \mathcal{L}$$

Writing $\tilde{x} = x_0 + \delta$, and defining the initial residual vector r_0 as $r_0 = b - Ax_0$, we have the condition

$$b - A(x_0 + \delta) \perp \mathcal{L}$$

$$r_0 - A\delta \perp \mathcal{L}$$

Therefore the approximate solution is obtained as

$$\tilde{x} = x_0 + \delta, \quad \delta \in \mathcal{K} \tag{2.26}$$

*imposing the **orthogonality condition***

$$\langle r_0 - A\delta, w\rangle = 0 \quad \forall w \in \mathcal{L} \tag{2.27}$$

*commonly called the **Petrov-Galerkin condition**.*

***Orthogonal projection methods** are those for which the two subspaces, $\mathcal{K}$ and $\mathcal{L}$ are identical and for these methods, the orthogonality condition is called the Galerkin condition.*

Definition 2.44 (Krylov subspace). *Given a matrix A and an initial residual vector $r_0 = b - Ax_0$, a **Krylov subspace** of dimension m is the subspace*

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\}$$

A Krylov subspace method aims to approximate the solution to the linear system $Ax = b$ and obtains the solution of the form

$$A^{-1}b \approx x_m = x_0 + q_{m-1}(A) r_0$$

where q_{m-1} is some polynomial of degree $m - 1$.

Definition 2.45 (Arnoldi's method). ***Arnoldi's method** is an orthogonal projection method onto $\mathcal{K}_m$ for general Hermitian matrices. This procedure is useful in obtaining accurate approximations to some eigenvalues of the matrix. It builds an orthogonal basis of the Krylov subspace $\mathcal{K}_m$ and does the following steps recursively for each basis vector.*

1. *Compute $h_{ij} = \langle Av_j, v_i\rangle \quad i = 1, 2, \ldots, j$*
2. *Compute $w_j := Av_j - \sum_{i=1}^{j} h_{ij} v_i$*
3. *$h_{j+1,j} = \|w_j\|_2$*
4. *stop if $h_{j+1,j} = 0$ else continue*
5. *$v_{j+1} = \frac{w_j}{h_{j+1,j}}$*

The procedure multiplies the previous basis vector, v_j by A and then orthonormalizes the resulting vector w_j against all the previous basis vectors using a Gram-Schmidt procedure.

Theorem 2.2.6. *The Arnoldi procedure defined in Definition 2.45 generates the vectors $v_1, v_2, \ldots, v_m$ such that these vectors form an orthonormal basis for the Krylov subspace*

$$\mathcal{K}_m = span\{v_1, Av_1, \ldots, A^{m-1} v_1\}$$

Proof. We see that the vectors generated from the Arnoldi procedure are of the form $v_j = q_{j-1}(A)v_1$, where q_{j-1} is a polynomial of degree $j-1$. Given that we use a Gram-Schmidt procedure, the vectors are orthonormal by construction. Therefore, these Arnoldi vectors form a orthonormal basis for the Krylov subspace. □

Definition 2.46 (Hessenberg matrix). *A square matrix $H \in \mathbb{R}^{n\times n}$ is said to be a* ***upper Hessenberg matrix*** *if*

$$h_{ij} = 0 \quad \forall i > j+1$$

Similarly, a lower Hessenberg matrix is defined as

$$h_{ij} = 0 \quad \forall j > i+1$$

Theorem 2.2.7 (Hessenberg decomposition). *Given a matrix $V_m \in \mathbb{R}^{n\times m}$ with column vectors $v_1, v_2, \dots, v_m$ and a Hessenberg matrix $\hat{H}_m \in \mathbb{R}^{m+1\times m}$ with nonzero entries of h_{ij} obtained from the Arnoldi procedure, and the matrix $H_m \in \mathbb{R}^{m\times m} = \hat{H}_m(1:m, 1:m)$. Then the matrix can be decomposed to get a upper Hessenberg matrix and the following relations hold:*

1. $AV_m = V_m H_m + w_m e_m^T$
2. $AV_m = V_{m+1}\hat{H}_m$
3. $V_m^T A V_m = H_m$

Proof. The Arnoldi procedure can be written as

$$Av_j = \sum_{i=1}^{j+1} h_{ij} v_i, \quad j = 1, 2, \dots, \tag{2.28}$$

Rewriting that in matrix form we have the first equation.

Multiplying both sides of the second relation by V_m^T and using the orthonormality of $\{v_1, \dots, v_m\}$, we get the Hessenberg decomposition $V_m^T A V_m = H_m$ □

Definition 2.47 (Generalized Minimum Residual (GMRES) method). *The* ***GMRES method*** *is a projection method based on taking $\mathcal{K} = \mathcal{K}_m$ and $\mathcal{L} = A\mathcal{K}_m$, where $\mathcal{K}_m$ is the $m-th$ Krylov subspace with $v_1 = \frac{r_0}{\|r_0\|_2}$.*

We see that any vector x in $x_0 + \mathcal{K}_m$ can be written as

$$x = x_0 + V_m y$$

where $y \in \mathbb{R}^m$. Taking $\beta = \|r_0\|_2$, we see that we can write the residual vector $r = b - Ax$ as

$$\begin{aligned} b - Ax &= b - A(x_0 + V_m y) \\ &= r_0 - AV_m y \\ &= \beta v_1 - V_{m+1}\hat{H}_m y \\ &= V_{m+1}(\beta e_1 - \hat{H}_m y) \end{aligned}$$

Since the columns of V_{m+1} are orthonormal, we can write a function

$$J(y) = \|b - Ax\|_2 = \left\|\beta e_1 - \hat{H}_m y\right\|_2$$

The GMRES method aims find a unique vector $x_0 + \mathcal{K}_m$ which minimizes $J(y)$. This is essentially the vector

$$x_m = x_0 + V_m y_m$$

where y_m minimizes the function $J(y) = \left\|\beta e_1 - \hat{H}_m y\right\|_2$

Therefore, the GMRES method can be written as

$$\begin{aligned} x_m &= x_0 + V_m y_m \\ y_m &= argmin_y \left\|\beta e_1 - \hat{H}_m y\right\|_2 \end{aligned} \tag{2.29}$$

The minimizer y_m requies the solution of a $(m+1) \times m$ least-squares problem where m is typically small. We note that this method above uses the Modfified Gram Schmidt method, but there are other variants which may be more numerically robust [Saa03].

Definition 2.48 (Conjugate gradient method (CG))**.** *The* ***conjugate gradient method*** *is a variant of the Lanczos method for symmetric positive definite matrices. It is widely useful for a wide range of applications that generate SPD matrices. We first express the solution vector x as*

$$x_{j+1} = x_j + \alpha_j p_j$$

where p_j can be seen as the search direction vector and α_j as the weight in that direction. The residual vectors can then be written as

$$r_{j+1} = r_j + \alpha_j A p_j$$

As we require the residual vectors to be orthogonal, we have the condition

$$\langle r_j - \alpha_j A p_j, r_j \rangle = 0$$

leading us to have

$$\alpha_j = \frac{\langle r_j, r_j \rangle}{\langle A p_j, r_j \rangle}$$

We choose the subsequent search direction as a linear combination of r_{j+1} and the previous search direction, p_j

$$p_{j+1} = r_{j+1} + \beta_j p_j$$

Therefore we have that

$$\langle A p_j, r_j \rangle = \langle A p_j, p_j - \beta_{j-1} p_{j-1} \rangle = \langle A p_j, p_j \rangle$$

because Ap_j is orthogonal to p_{j-1}. This gives us the following relations to compute the scalars α_j and β_j

$$\alpha_j = \frac{\langle r_j, r_j \rangle}{\langle A p_j, p_j \rangle}$$

and

$$\beta_j = -\frac{\langle r_{j+1}, A p_j \rangle}{\langle p_j, A p_j \rangle}$$

Given that $Ap_j = -\frac{1}{\alpha_j}(r_{j+1} - r_j)$, we have

$$\beta_j = \frac{1}{\alpha_j} \frac{\langle r_{j+1}, (r_{j+1} - r_j) \rangle}{\langle A p_j, p_j \rangle} = \frac{\langle r_{j+1}, r_{j+1} \rangle}{\langle r_j, r_j \rangle}$$

Putting these relations together, we have the CG method.

Theorem 2.2.8 (Convergence of CG)**.** *For a matrix $A \in \mathbb{R}^{n \times n}$, an initial solution vector x_0, an exact solution x^* and the solution x_m at the end of m steps of the CG method, we have*

$$\|x^* - x_m\|_A \leq 2\left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]^m \|x^* - x_0\|_A \tag{2.30}$$

Thefore we can write the convergence rate,

$$\begin{aligned} \varrho_{CG}(A) &= 2\left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]^m \\ &\approx 1 - \frac{2}{\sqrt{\kappa(A)}} \quad \textit{for } \kappa(A) >> 1 \end{aligned} \tag{2.31}$$

We note that depending on the spectral distribution of A, the convergence rate of CG and other Krylov solvers may be superlinear [Gre97].

Proof. See Greenbaum [Gre97]. □

There are more variants of the Krylov methods, in particular one with short recurrences, applicable also for general matrices (non-SPD). We do not elaborate on them here, but refer the reader to the papers which explain their derivation and analysis in detail. In particular in this work, we make use of the BiCGSTAB method [van92] for the solution of general linear systems, which can be more efficient than the GMRES method, though less robust. Some other popular Krylov methods include CGS [Son89], IDR(s) [Sv08], QMR and GCR [de 99].

2.2.2.2 Domain decomposition methods

As the name suggests, domain decomposition methods are a class of methods that decompose the domain into smaller subdomains to increase the amount of available parallelism. Decomposing into subdomains, in general, requires explicit communication between these subdomains, but the coarse-grained parallelism made available through this process can significantly improve the time to solution by making better use of the available resources. Before we look at some of the domain decomposition solvers, we need to define how we generate these decompositions.

Definition 2.49 (Partition of unity)**.** *A* ***partition of unity*** *describes the decomposition (possibly overlapping) of an open domain Ω into J open subdomains, $\Omega = \cup_{i=1}^{J} \Omega_i$. Additionally it defines the prolongation and restriction operators which equal to zero in $\Omega \backslash \Omega_i$.*

Given a set of indices of the degrees of freedom, $\mathcal{I}$, and a decomposition into J subsets, $\mathcal{I} = \cup_{i=1}^{J} \mathcal{I}_i$, the partition of unity operator corresponds to diagonal matrices of size $(\#\mathcal{I}_i \times \#\mathcal{I}_i)$ with non-negative entries such that

$$u = \sum_{i=1}^{J} R_i^T \chi_i R_i u$$

where $u \in \mathbb{R}^{\#\mathcal{I}}$. Therefore

$$I = \sum_{i=1}^{J} R_i^T \chi_i R_i$$

$R_i \in \mathbb{W}^{\#\mathcal{I} \times \#\mathcal{I}}$ is the restriction operator that maps the vector $u \in \mathbb{R}^{\#\mathcal{I}}$ from the domain Ω to its respective subdomain Ω_i. Therefore

$$u_i = R_i u$$

Similarly the prolongation operator, in this case the transpose of the restriction operator, R_i^T, maps the subdomain vector to the global vector.

$$u = R_i^T u_i$$

We denote this partition of unity as $\wp_i(R_i, \chi_i)$ or $\wp_i$ in short.

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 [Sch70]. Many variants of the method have found their use, both as iterative methods [DW87; DGG10; BCR16] and in particular as preconditioners [LH16; BDG+17].

Definition 2.50 (Schwarz methods)**.** *For a non-singular matrix $A \in \mathbb{R}^{n\times n}$, we define a partition of unity, $\wp_i$. For initial an solution, x_0 to the equation $Ax = b$, the general* ***Schwarz method*** *computes the iterates x_k with a fixed point iteration*

$$x_{k+1} = x_k + M^{-1}_{Schwarz} r_k \quad k = 0, 1, \ldots$$
$$M^{-1}_{Schwarz} = \mathcal{L}^{J}_{i=1}\left(R_i^T \chi_i (R_i A R_i^T)^{-1} R_i\right) \tag{2.32}$$

Using different operations for the general operator $\mathcal{L}$, we can represent two types of Schwarz methods:

1. *(Composition): Multiplicative Schwarz: $\mathcal{L}_i^J = \circ : \mathcal{L}(A, B)x = A(B(x))$*
2. *(Combination): Additive Schwarz: $\mathcal{L}_i^J = + : \mathcal{L}(A, B)x = A(x) + B(x)$*

We note that Equation (2.32), we require local subdomain solutions, represented by $(R_i A R_i^T)^{-1}$. Using some splitting for the subdomain matrices $A_i = M_i - N_i$, we can compute these local solution, for example with Jacobi ($M_j = D_J, N_j = A_j - D_j$), Gauss-Seidel or Krylov methods.

Definition 2.51 (Weighted and Restricted Additive Schwarz methods)**.** *From Equation (2.32), we can write the general* ***weighted additive Schwarz*** *method as*

$$M^{-1}_{weighted\_schwarz} = \sum_{i=1}^{J}\left(R_i^T \chi_i (R_i A R_i^T)^{-1} R_i\right) \tag{2.33}$$

The weighting matrix, χ_i defines the weighting of the degrees of freedom between neighboring subdomains with overlaps.

A simple modification was discovered by Cai et.al [CS99], called the ***restricted additive Schwarz method****, which shows better convergence, in addition to reduced communication volume. In this method, we restrict the subdomains such that in the solution vector, we discard the data that is non-local to the subdomain.*

In this case, the partition of unity weights the solution to only locally owned degrees of freedom, $\chi_i = \delta_{ij} \quad i, j \in \mathcal{I}_i$, with $\mathcal{I}_i$ being the proper subset of the set of global degrees of freedom, $\mathcal{I}$ such that $\mathcal{I}_i \cap \mathcal{I}_j = \emptyset \quad \forall i \neq j$

Theorem 2.2.9 (Convergence of additive Schwarz methods)**.** *Let the additive Schwarz iteration matrix be $G_{Schwarz} = I - M^{-1}_{Schwarz} A$, where $M^{-1}_{Schwarz}$ is as defined in Equation (2.33).*

Then there exists a positive constant C independent of the number of subdomains J such that

$$\kappa(M^{-1}_{Schwarz} A) \leq C J^2 (1 + \beta^{-2})$$

where β is the overlap between the subdomains.

Proof. See Chan and Mathew [CM94] and Dryja and Widlund [DW90]. □

The deterioration in the bound of the condition number above is expected due to the information exchange nature of the Additive Schwarz method: Information is exchanged only between neighbors without any global information exchange. To remedy this, there have been multi-level methods proposed, which aim to incorporate global information with an additional solution on some coarse grid(s), which can be algebraically generated from the global system matrix/domain.

Definition 2.52 (Multi-level Additive Schwarz methods)**.** *Given a matrix $A \in \mathbb{R}^{n\times n}$, and some restriction and prolongation operators that generate a hierarchy of coarse matrices, $C_l = R_l A P_l \quad l = 1, \ldots, L$ with L levels, we can define the* ***multi-level additive Schwarz*** *method as*

$$M^{-1}_{multilevel-Schwarz} = \sum_{i=1}^{J} \left(R_i^T \chi_i (R_i A R_i^T)^{-1} R_i \right) + \sum_{l=1}^{L} P_l C_l^{-1} R_l \tag{2.34}$$

We see that this fits in nicely with the additive procedure and the coarse information can be additively incorporated into the solution.

Theorem 2.2.10 (Convergence of multi-level Schwarz methods)**.** *Let the additive Schwarz iteration matrix be $G_{multilevel-Sch}$ $I - M^{-1}_{multilevel-Schwarz} A$, where $M^{-1}_{multilevel-Schwarz}$ is as defined in Equation (2.34).*

Then there exists a positive constant C independent of the number of subdomains J such that

$$\kappa(M^{-1}_{multilevel-Schwarz} A) \leq C(1 + \beta^{-2})$$

where β is the overlap between the subdomains.

Proof. See Chan and Mathew [CM94] and Dryja and Widlund [DW90]. □

2.3 Hardware architectures, Programming models

In this section, we have a brief look at the architectures of the hardware used in this work. Given that each hardware generally provides a specific programming model with which it generally obtains optimal performance, we also look at the different programming models and their characteristics.

2.3.1 CPU and GPU hardware

The proliferation of parallel computing has created new massively parallel specialized architectures. A particular type of hardware initially used mainly for graphics processing but being broadly adopted for general-purpose computing is the Graphics Processing unit (GPU).

In contrast with the general purpose CPUs, these GPUs have many parallel computing threads that work in a lock-step fashion, organized into larger independent groups. GPUs can provide a significant performance benefit for algorithms that can harness this massive parallelism. It has to be noted that CPUs are more general-purpose and can handle a wide variety of workloads. GPUs are more specialized, and particular care needs to be taken when designing data structures and algorithms to accelerate them on a GPU.

To this end, the general architecture of the CPU and GPU are different. GPUs generally have a smaller Instruction set architecture (ISA), are more hierarchical with respect to both compute and memory and generally have a lesser amount of total available main memory. They also devote more transistors to parallel data processing and less to data caching and control flow, thereby enabling massive throughput. However, they are inefficient for code that contains complex control flow or branches. Figure 2.1 shows the distribution of the resources and the architectural differences between a CPU and a GPU.

It is crucial to consider the application at hand, its algorithms, and their computational and memory complexity before choosing the hardware to obtain the overall best performance. For example, an algorithm with a computational complexity of $\mathcal{O}(n^3)$ might achieve n^3 floating point operations/sec (flops, a standard measure of performance), but if an alternative algorithm providing the same result has a computational complexity of $\mathcal{O}(n)$, it might achieve much fewer flops on the hardware, but might have a lesser run-time.

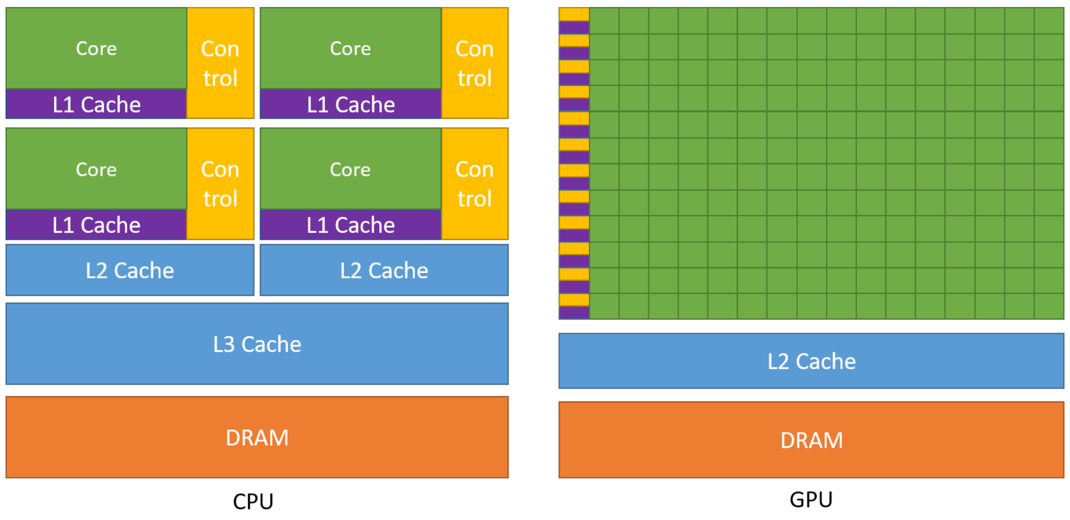


Figure 2.1: Comparison of CPU and GPU architectures [NVI23b]

2.3.2 Programming on GPUs: CUDA, HIP and SYCL

As GPUs are not general-purpose computing units, in the sense that they do not have an operating system that enables user interaction and orchestrates the different components, computing on a GPU almost always requires a CPU to perform the orchestration tasks. Figure 2.2 shows the general system architecture. The CPU orchestrates the different devices and interfaces through the operating system. The GPU communicates to the CPU through a GPU driver, generally provided by the vendor. It is to be noted that due to different Instruction set architectures (ISAs), a driver is necessary to enable the CPU to move data and execute kernels on the GPU. An important aspect to note is that the CPU and GPU memories are separate. Allocating, deallocating, and moving data from the CPU to the GPU and vice versa usually needs to be explicit (except in the case of unified memory models), and care has to be taken not to access GPU resident data from the CPU and vice versa. Additionally, the GPU runtime then handles the movement of the data within the GPU and maps the kernels to the different compute and memory hierarchy levels.

Currently, three main vendors build high-performance server-grade GPUs: NVIDIA, AMD, and Intel, and they provide three different programming models: CUDA for NVIDIA GPUs, HIP/ROCm for AMD GPUs, and SYCL for Intel GPUs. The CUDA and HIP programming models are very similar concerning their programming interfaces, while Intel follows the SYCL 2020 standard developed by Khronos.

GPU hardware is hierarchical from both compute and memory perspectives. Figure 2.3 shows a schematic of this hierarchy with terminology from the CUDA programming model. The finest compute level is called a thread. Each thread has its local memory and a register file with very low latency. Multiple threads are organized into a warp, which are SIMT units that can process multiple elements in parallel using the same instruction. A collection of warps is called a thread block. Threads within one thread block can additionally communicate with each other through the local shared memory/L1 cache, which is user-configurable. Multiple thread blocks are composed to form thread block clusters, which are further composed to form grids. Threads from different thread blocks can only communicate through the main GPU DRAM.

The HIP programming model is similar to the CUDA model, with some differences, such as the warp sizes (64 in AMD v/s 32 in NVIDIA) and the kernel launch parameters, which have different semantics. This similarity can enable the user to unify the code for the two programming models [TCR+21]. The schematic in Figure 2.4 shows the kernel launch syntax for the CUDA/HIP programming model. The kernel is called with the grid and block dimensions, specifying the number of threads in a block and the number of blocks the kernel will use. The user can also specify the amount of shared memory to be allocated at runtime and pass this to the kernel. This enables maximizing the occupancy of the GPU as over-usage of shared memory reduces the number of warps scheduled on the same streaming multi-processor. Finally, to enable parallelism between different kernels, the user can also associate the kernel with a specific stream, with control over the synchronization of the streams to ensure correctness.

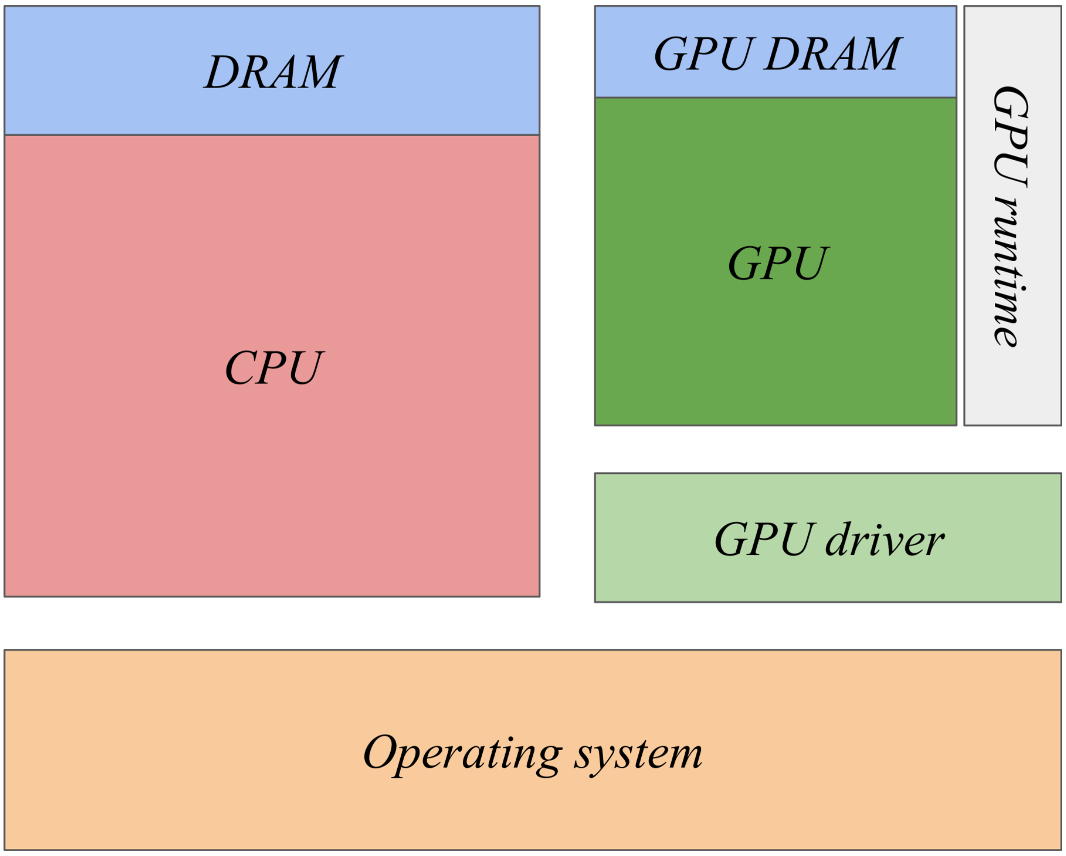


Figure 2.2: General system architecture

Table 2.1: Compute hierarchy naming differences: CUDA, HIP, SYCL and CPU

CUDA	HIP	SYCL	CPU
thread-block	thread-block	work-group	core
warp	wavefront	sub-group	SIMD lane
thread	thread	work-item	thread

The hardware hierarchy on the Intel GPUs is quite similar to NVIDIA and AMD [Int23b]. The programming model is SYCL, which is a royalty-free cross-platform abstraction layer that enables users to write C++ code for devices [Khr14]. The salient differences in the names for the different compute hierarchies are shown in Table 2.1. Some important differences between the organization of a SYCL kernel and the CUDA/HIP kernel are shown in Figure 2.4. Unlike in CUDA, SYCL kernels must be explicitly associated with a queue (which can be the default queue). The user must also allocate the shared memory beforehand and pass it into the kernel; the runtime does not handle this. As SYCL is meant to be a general portable programming model that works on different architectures such as CPUs, GPUs and FPGAs, the concepts of threads, blocks, and warps, are configurable and can be queried from the device the user would need to run on. We note that SYCL can be used for NVIDIA and AMD GPUs as well, as they aim to compile to an intermediate layer such as LLVM-IR [KB22], but widespread support and performance tests are lacking. Additionally, the build systems for these cases are more fragile.

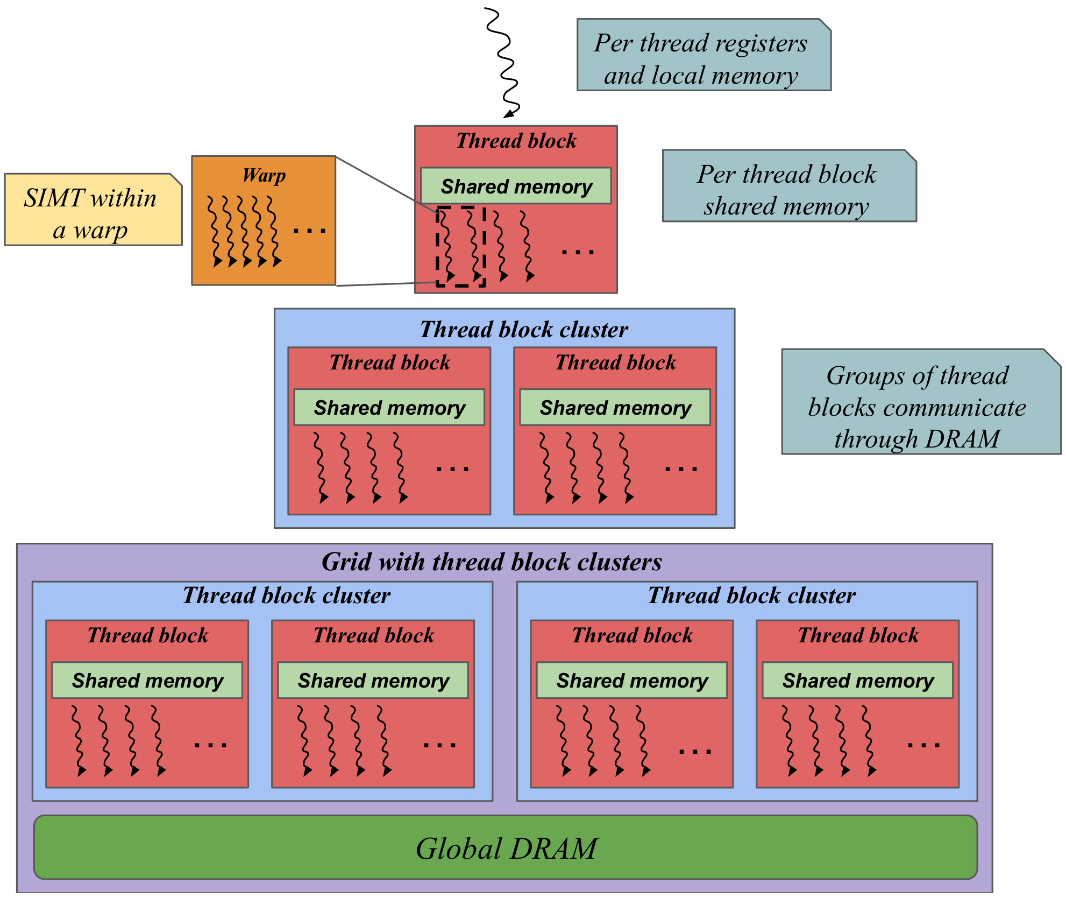

Per thread registers and local memory
Thread block
Shared memory
SIMT within a warp
Warp
Per thread block shared memory
Thread block cluster
Thread block
Shared memory
Groups of thread blocks communicate through DRAM
Grid with thread block clusters
Thread block cluster
Global DRAM

Figure 2.3: Compute and memory hierarchy for NVIDIA GPUs [NVI23b]

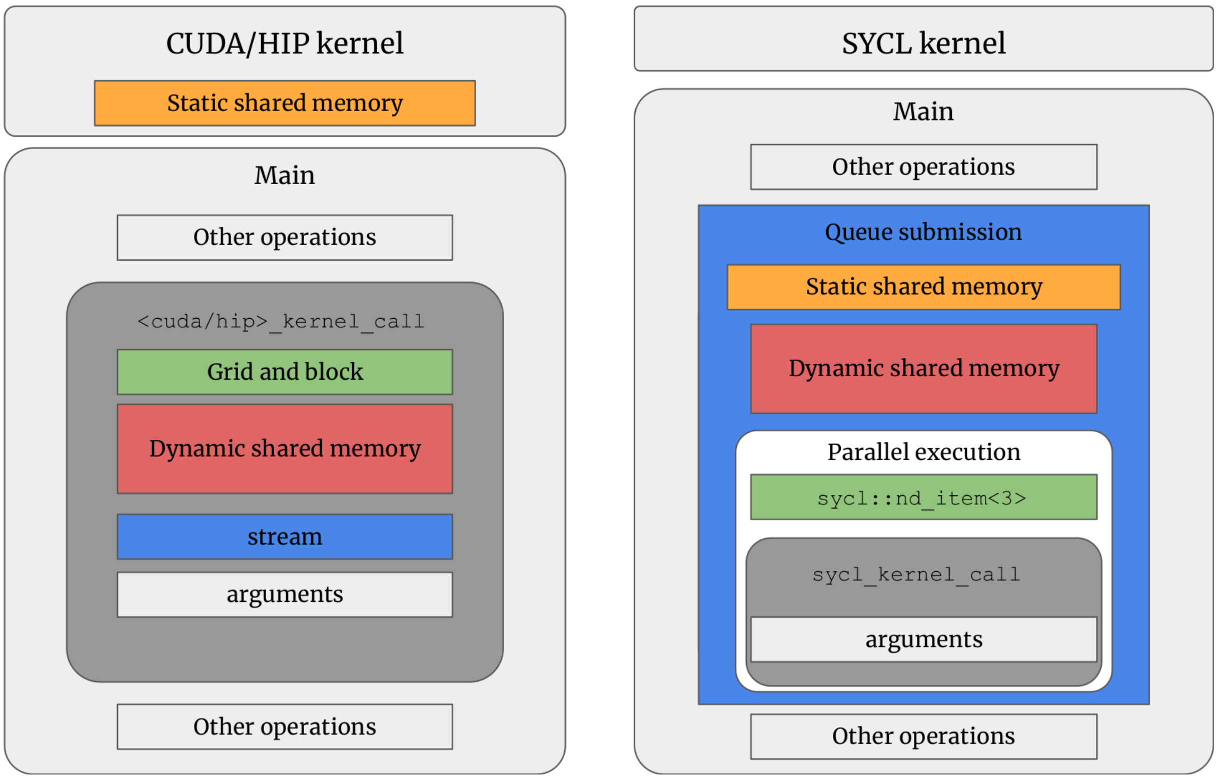


Figure 2.4: Differences in the kernel launch syntax for CUDA and HIP v/s SYCL [TCA22]

2.4 Data structures

In addition to efficient algorithms, efficient data structures form an integral part of maximizing performance on hardware. Data structures form the bridge between the mapping of the data written in an abstract algorithm to the hardware and are generally programming language agnostic. In particular, the following aspects of a generic data structure are essential to consider before using them in an algorithm implementation.

1. Memory usage: The number of bytes a data structure occupies on hardware.
2. Access patterns: The type of access to the data, the data structure can provide.
3. Memory layout: The layout of the data within the data stucture.
4. Static or dynamic: Determines if the data structure can be resized

An ideal data structure has the least memory footprint, provides constant time access to the underlying data, is cache-friendly, and can be resized as necessary without any overheads. Unfortunately, not all these objectives can be achieved simultaneously, and a trade-off must be made that depends primarily on the algorithm and the application at hand.

2.4.1 Memory and cache models

An efficient data structure must enable optimal performance. Depending on the operations that need to be performed on a data structure, we can analyze the efficiency and optimality of a data structure with some well-known models. Therefore, we define some models that will be useful for our analysis.

Definition 2.53 (Hiererchical memory model)**.** *The* ***hierarchical memory model*** *is a uniform model with different memory levels whose locations take different amount of time to access. Accessing location x takes time $f(x)$. In general, the levels are discrete and as shown in Figure 2.5 [VS94]. We assume that each level $l, \quad l \geq 1$ contains 2^{l-1} locations at addresses $2^{l-1}, 2^{l-1}+1, \dots, 2^{l}-1$.*

Assume well behaved access costs, $f(x)$, with a nondecreasing $f(x)$ and constants $c_0, c_1 \in \mathbb{R}$ such that $f(2x) \leq c_0 f(x) \quad x \geq c_1$. Then the access for any location at level l takes $\mathcal{O}(f(2^l))$ time.

Most compute units (CPUs and GPUs included) can be modeled with such a memory hierarchy. Level 1 is closest to the compute unit but of relatively small size. With increasing levels, the amount of memory increases, but the access costs also accordingly increase. Of course, Figure 2.5 shows only one compute unit, but in actual hardware, we have multiple compute units (such as CPU cores or GPU streaming multiprocessors) sharing memory levels.

Definition 2.54 (Ideal cache model)**.** *The* ***ideal cache model*** *models a system with a two level memory hierarchy consisting of an ideal data cache of Z words (data elements) and an arbitrarily long DRAM (main memory) [FLP+12]. We assume that the word size is constant. The cache is partitioned into cache lines, each consisting of L consecutive words that are always moved together between the DRAM and the cache. In most cases, $L > 1$, and it pays to move multiple cache lines at once particularly when we can benefit from spatial locality. In general, we can assume that the cache is tall, which is often the case in practice.*

$$Z = \mathcal{O}(L^2)$$

The compute unit can only reference words that reside in the cache. If the referenced word is in a line of the cache, then a ***cache hit*** *occurs. If not, then a* ***cache miss*** *occurs and the line needs to be fetched from the DRAM into the cache. We assume a fully associative cache [HP12]: Cache lines can be stored anywhere in cache. If the cache is full, we need to evict a cache line before fetching a new cache line from DRAM.*

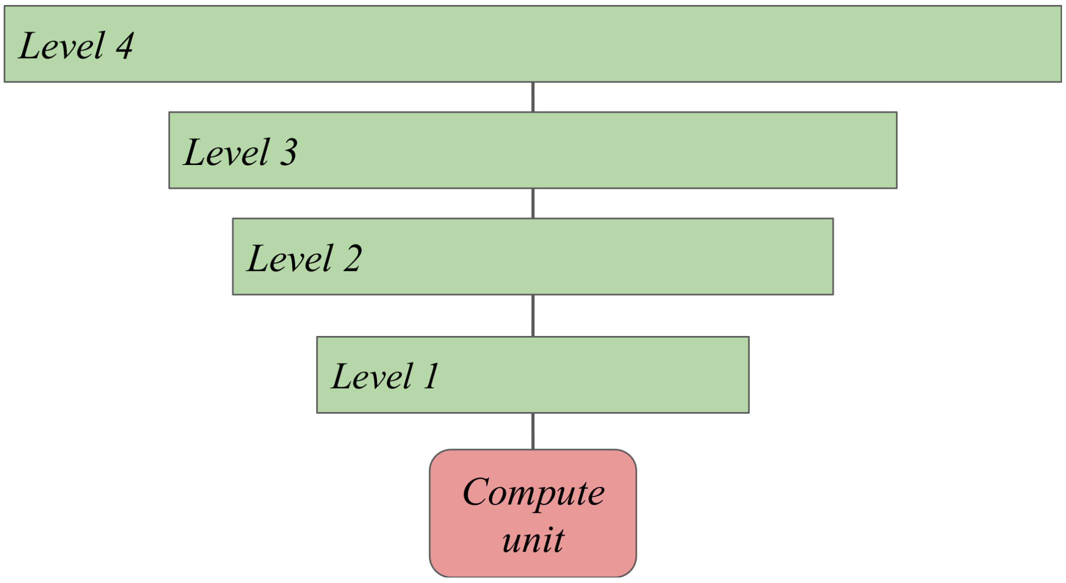


Figure 2.5: The hierarchical memory model

Definition 2.55 (Work and cache complexity)**.** *Consider an algorithm with n input elements that need to be processed. Its work or* ***computational complexity*** *with the ideal cache model, denoted by $\mathscr{W}(n)$ is defined as its convential runtime in a standard RAM model [AH74].*

The ***cache complexity*** *of the algorithm, denoted by $\mathscr{Q}(n; Z, L)$ is defined as the number of cache misses that occur as a function of the cache size Z and cache line length, L.*

Definition 2.56 (Memory complexity)**.** *Consider an algorithm with n input elements. Its* ***memory complexity****, denoted by $\mathscr{M}(n)$ is defined as the number of elements that need to be read from main memory and the number of writes that need to be written back to main memory.*

2.4.2 Data layouts

With hierarchical memory layouts in state-of-the-art computing hardware, choosing an appropriate data layout for the data structure that maximizes its cache usage (minimizes the amount of cache misses) is extremely important.

In general, data structures in this work are either two-dimensional (matrices), one-dimensional (vectors, arrays) or collection of one-dimensional objects (multi-vectors, which can be seen as pseudo-two-dimensional). In actual memory, data is laid out in one dimension and striding (offset) enables a two-dimensional view.

Some common data layouts for two-dimensional arrays are shown in Figure 2.7. (a) shows a row-major layout, with rows being stored consecutively. (b) shows a column-major layout, with columns being stored consecutively. (c) and (d) show tile based layouts where data is coarse partitioned into tiles and in each tile, stored in either a row-major or column-major fashion. These tiled layout have been shown to be more cache-friendly [PHP03] for some algorithms.

2.4.3 Multi-vectors and Dense matrices

An integral data-structure in numerical linear algebra libraries is the multi-vector data structure. For example, given a vector space $\mathscr{V} = v_1, v_2, \ldots, v_m \in \mathbb{C}^{n \times m}$, the multivector essentially stores $n \times m$ elements. The choice of the data layout is library-dependent and can be either row-major (store corresponding elements of the vectors in a consecutive fashion) or column-major (store each complete vector in a consecutive fashion).

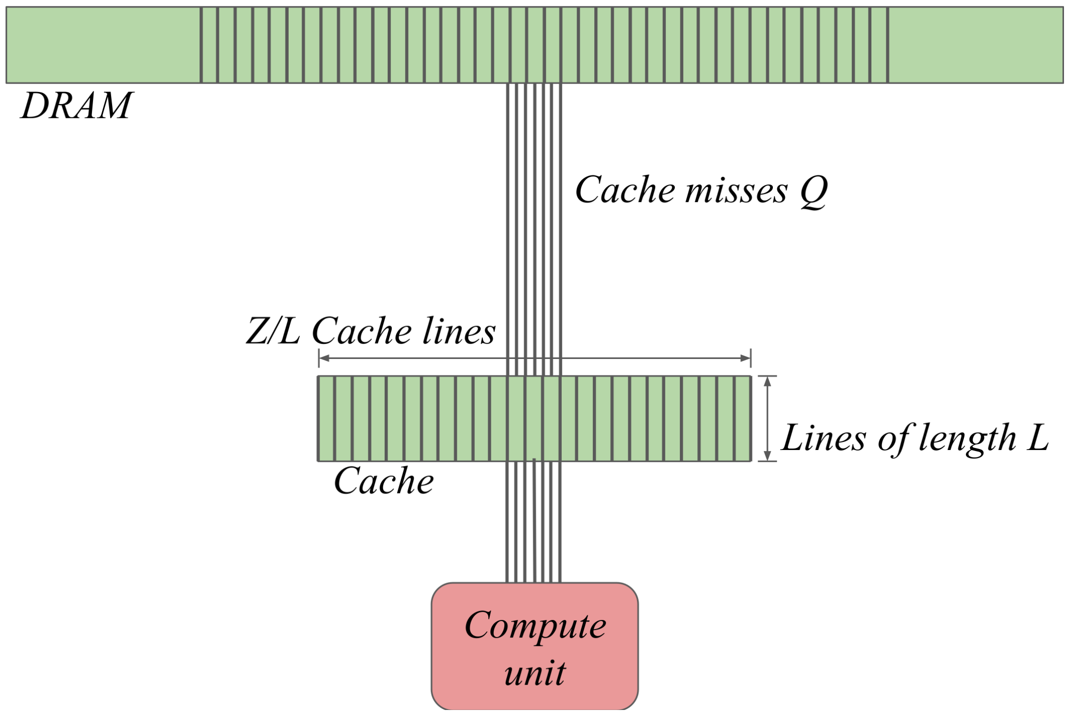


Figure 2.6: The ideal cache model

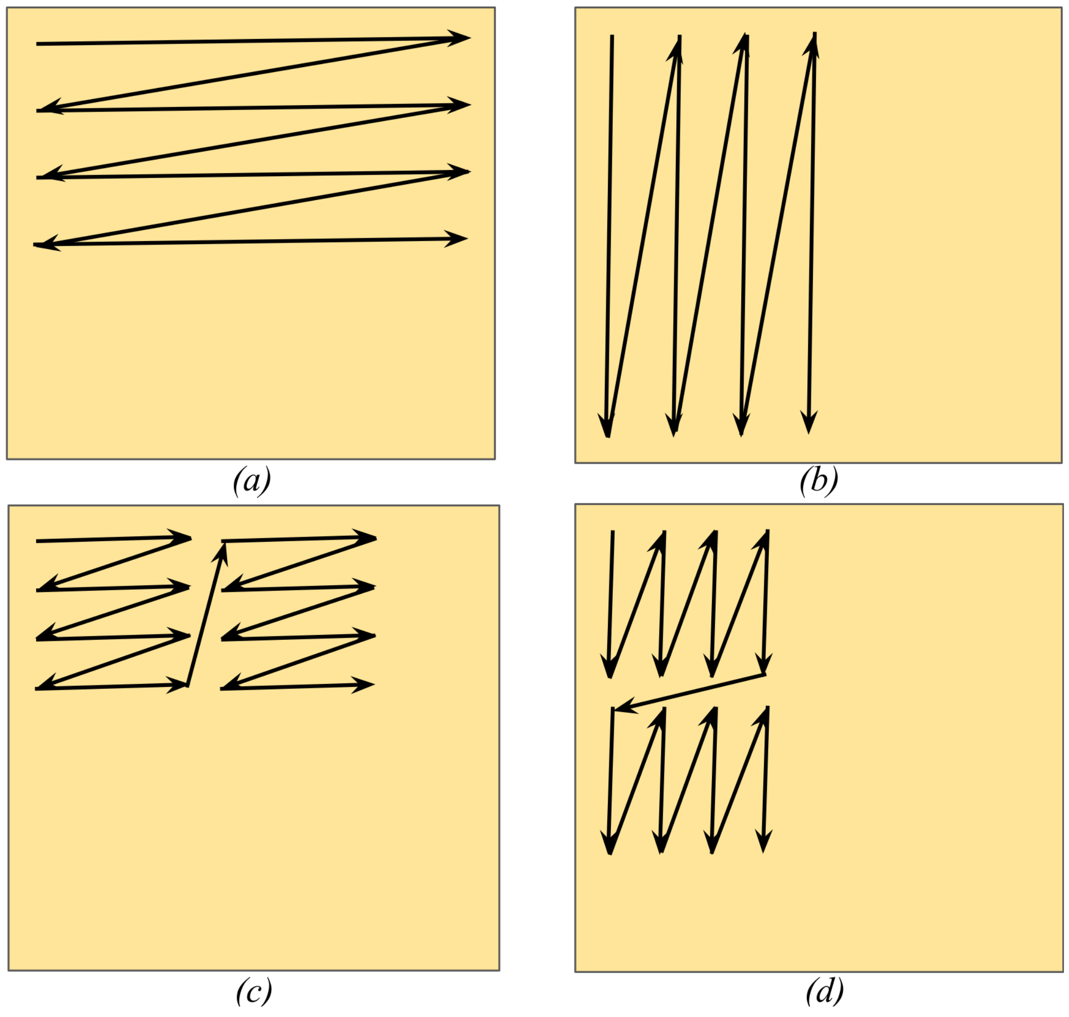


Figure 2.7: Some common data layouts

A dense matrix, $A \in \mathbb{C}^{n\times m}$ is also generally stored in a similar fashion given that from a storage perspective it is equivalent to a multi-vector.

For our work-complexity estimations, we make the following assumptions:

1. A standard RAM model [AH74].
2. Multiplications, additions and divisions are of the same cost, even though in practice divisions are more expensive and multiply-add units are usually available that can perform both multiplication and addition with a single instruction.

We now give some computational complexity estimations for some common operations on multi-vectors and dense matrices that are used in this work.

Lemma 2.4.1 (Dot product work complexity). *Given two multi-vectors, $\mathcal{V}, \mathcal{U} \in \mathbb{C}^{n\times m}$, the computational complexity of the dot product, represented by*

$$\langle \mathcal{V}, \mathcal{U} \rangle = \sum_{i=1}^{n} v_k u_k, \quad k = 1, \ldots, m$$

is equal to

$$\mathscr{C}_{dot} = 2m(n-1)$$

Proof. From the represented operation, we clearly see that for each vector, we need to perform n multiplications and $n-1$ reductions (additions). Given that we have m vectors, we have that

$$\mathscr{C}_{dot} = 2m(n-1)$$

□

Lemma 2.4.2 (Dot product memory complexity). *Given two multi-vectors, $\mathcal{V}, \mathcal{U} \in \mathbb{C}^{n\times m}$, the memory complexity of the dot product, represented by*

$$\langle \mathcal{V}, \mathcal{U} \rangle = \sum_{i=1}^{n} v_k u_k, \quad k = 1, \ldots, m$$

is equal to

$$\mathscr{M}_{dot} = 2mn + m$$

Proof. From the represented operation, we see that we need to read $2mn$ elements and overall write m elements.

$$\mathscr{M}_{dot} = 2mn + m$$

□

Lemma 2.4.3 (2-norm work complexity). *Given a multi-vector, $\mathcal{V} \in \mathbb{C}^{n\times m}$, the computational complexity of the 2-norm operation, represented by*

$$\|\mathcal{V}\|_2 = \sqrt{\sum_{i=1}^{n} v_k^2}, \quad k = 1, \ldots, m$$

is equal to

$$\mathscr{C}_{2-norm} = 2nm$$

Proof. From the represented operation, we clearly see that for each vector, we need to perform n multiplications and $n-1$ reductions (additions) and a final square root. Given that we have m vectors, we have that

$$\mathscr{C}_{2-norm} = 2nm$$

□

Lemma 2.4.4 (2-norm memory complexity). *Given a multi-vector, $\mathscr{V} \in \mathbb{C}^{n\times m}$, the memory complexity of the 2-norm operation, represented by*

$$\|\mathscr{V}\|_2 = \sqrt{\sum_{i=1}^{n} v_k^2}, \quad k = 1, \dots, m$$

is equal to

$$\mathscr{M}_{2-norm} = (n+1)m$$

Proof. From the represented operation, we clearly see that we need to read nm elements and write m elements. Therefore,

$$\mathscr{M}_{2-norm} = (n+1)m$$

□

Lemma 2.4.5 (Dense matrix vector product work complexity). *Given a multi-vector, $\mathscr{B} \in \mathbb{C}^{n\times m}$ and a dense matrix, $A \in \mathbb{C}^{n\times n}$ and an output vector $\mathscr{X} \in \mathbb{C}^{n\times m}$, the computational complexity of the matrix vector product, represented by $\mathscr{X} = A\mathscr{B}$:*

$$x_k = \sum_{j=1}^{n} a_{ij} b_{jk}, \quad i = 1, \dots, n; k = 1, \dots, m$$

is equal to

$$\mathscr{C}_{dense-mv} = 2mn^2 - mn$$

Proof. From the represented operation, we clearly see that for each solution vector component, we need to perform n multiplications and $n-1$ reductions (additions). As we have n vector components, this leads us to a total of $2n^2 - n$ work per vector. Given that we have m vectors, we have that

$$\mathscr{C}_{dense-mv} = 2mn^2 - mn$$

□

Lemma 2.4.6 (Dense matrix vector product memory complexity). *Given a multi-vector, $\mathscr{B} \in \mathbb{C}^{n\times m}$ and a dense matrix, $A \in \mathbb{C}^{n\times n}$ and an output vector $\mathscr{X} \in \mathbb{C}^{n\times m}$, the memory complexity of the matrix vector product, represented by $\mathscr{X} = A\mathscr{B}$:*

$$x_k = \sum_{j=1}^{n} a_{ij} b_{jk}, \quad i = 1, \dots, n; k = 1, \dots, m$$

is equal to

$$\mathscr{M}_{dense-mv} = 2mn^2 + m$$

Proof. From the represented operation, we clearly see that for each solution vector component, we need to read n elements of A, and n of B and write 1 element of X. Given that we have m vectors, we get

$$\mathscr{M}_{dense-mv} = 2mn^2 + m$$

□

2.4.4 Sparse matrices

In contrast to dense matrices where a matrix $A \in \mathbb{C}^{n\times m}$ stores nm elements, sparse matrices are a class of data structures which store φn number of nonzeros, where φ is the average number of non-zeros per row and in general $\varphi < n$. We do not require that each row have the same number of non-zeros.

Many applications generate sparse matrices. For example, in finite differences, where one generally uses a stencil to assemble a matrix, we only have as many nonzeros per row as the unrolled stencil width. In particular, sparse matrices are necessary when scaling up applications. When n is large, storing zeros and processing them both from a memory and computation perspective is highly inefficient. For simplicity, we only concern ourselves with square matrices, but this can be easily generalized to non-square matrices. We now look at some popular sparse matrix formats.

Definition 2.57 (Coordinate format (COO))**.** *Given a matrix $A \in \mathbb{C}^{n\times n}$, the coordinate matrix format stores the triplet* $(i, j, a_{ij}) \quad \forall i | j \neq 0, i, j = 1, \ldots, n$.

In general, for efficiency, a structure of arrays (SoA) is preferable; i.e one generally stores three arrays, each of size equal to the number of nonzeros in the matrix: one for the row index, i, one for the column index j and one for the value a_{ij}.

Therefore the memory storage complexity is

$$\mathscr{M}_{coo} = 3n_{nz}$$

where n_{nz} is the total number of nonzeros in the matrix.

Definition 2.58 (Compressed sparse row format (CSR))**.** *Given a matrix $A \in \mathbb{C}^{n\times n}$, the compressed sparse row matrix format stores the column index and the value of the nonzero element. Additionally, it stores a compressed form of the row index called the row pointers array (in a cumulative fashion). This array points to the start of the nonzero element for each row.*

Therefore the memory storage complexity is

$$\mathscr{M}_{csr} = 2n_{nz} + n + 1$$

where n_{nz} is the total number of nonzeros in the matrix.

Definition 2.59 (ELLPack format)**.** *Given a matrix $A \in \mathbb{C}^{n\times n}$, the ELLPack matrix format stores a constant number of nonzeros per row eliminating the need to store an additional row index or row pointers array as the offsets are now equal to the number of nonzeros stored in a row, $n_{nz-per-row}$ (padded with zeros if necessary).*

Therefore the memory storage complexity is

$$\mathscr{M}_{ell} = 2n \times n_{nz-per-row}$$

where $n_{nz-per-row}$ is the number of nonzeros per row (same across all rows) in the matrix.

The sparse matrix vector product(SpMV) is an important operation that we need to consider, particularly as it forms the workhorse of the Krylov solvers that we consider in the later chapters. The operation itself is similar to the dense matrix vector product shown in Lemma 2.4.5, but with A being sparse.

Lemma 2.4.7 (Work complexity of SpMV)**.** *Given a matrix $A \in \mathbb{C}^{n\times n}$ and a vector $b \in \mathbb{C}^n$, the work complexity of the sparse matrix vector product, $x = Ab$:*

$$x_i = a_{ij} b_j \quad \forall a_{ij} \neq 0$$

is equal to

$$\mathscr{C}_{spmv} = 2n_{nz}$$

where n_{nz} is the number of nonzeros in the matrix.

Proof. We see that we need to perform n_{nz} number of multiplications and in each row we need to perform some reductions equal to $n_{nz,row}$. Given that we have n rows, the total number of operations is

$$\mathscr{C}_{spmv} = n_{nz} + \sum_{row=1}^{n} n_{nz,row}$$

$$\mathscr{C}_{spmv} = 2n_{nz}$$

□

Lemma 2.4.8 (Memory complexity of SpMV). *Given a matrix $A \in \mathbb{C}^{n\times n}$ and a vector $b \in \mathbb{C}^n$, the memory complexity of the sparse matrix vector product, $x = Ab$:*

$$x_i = a_{ij}b_j \quad \forall a_{ij} \neq 0$$

is equal to

$$\mathscr{M}_{spmv} = 2n_{nz} + n$$

where n_{nz} is the number of nonzeros in the matrix.

Proof. We see that we need to n_{nz} elements of the matrix once. For the right-hand side as well, overall, we read n_{nz} elements. We need to write overall n elements. Therefore, the memory complexity is

$$\mathscr{M}_{spmv} = 2n_{nz} + n$$

□

The choice of the optimal sparse matrix format for a certain matrix depends on the following factors and is mainly influenced by the performance of the SpMV with the chosen matrix format.

1. Number of nonzeros per row: If the matrix has a similar number of nonzeros per row, then using the ELL matrix format can provide the best performance.
2. Load balance: The variation in the number of nonzeros per row. An imbalanced matrix is one which has a high variation in the number of nonzeros per row.
3. Matrix size: For a small matrix, a dense matrix might be more suitable as the overhead of the additional arrays and the indirection, leading to cache misses is not amortized over the time spent in the SpMV operation.

2.5 Performance and scaling analysis

In this section we define some terminologies and models commonly used to analyze and study performance of algorithms and implementations.

Definition 2.60 (Arithmetic intensity). *The **arithmetic intensity** of an algorithm is defined as the ratio of the number of arithmetic operations (multiply, add etc) to number of memory operations (read and write).*

$$AI = \frac{n_{arith}}{n_{mem}}$$

An algorithm with high arithmetic intensity performs a high number of arithmetic operations for fewer memory operations. For example, the dense matrix matrix multiplication has an AI of $\mathcal{O}(\frac{2n^3}{n^2}) = \mathcal{O}(2n)$, thereby scaling with n.

The arithmetic intensity is generally measured in Flops/byte.

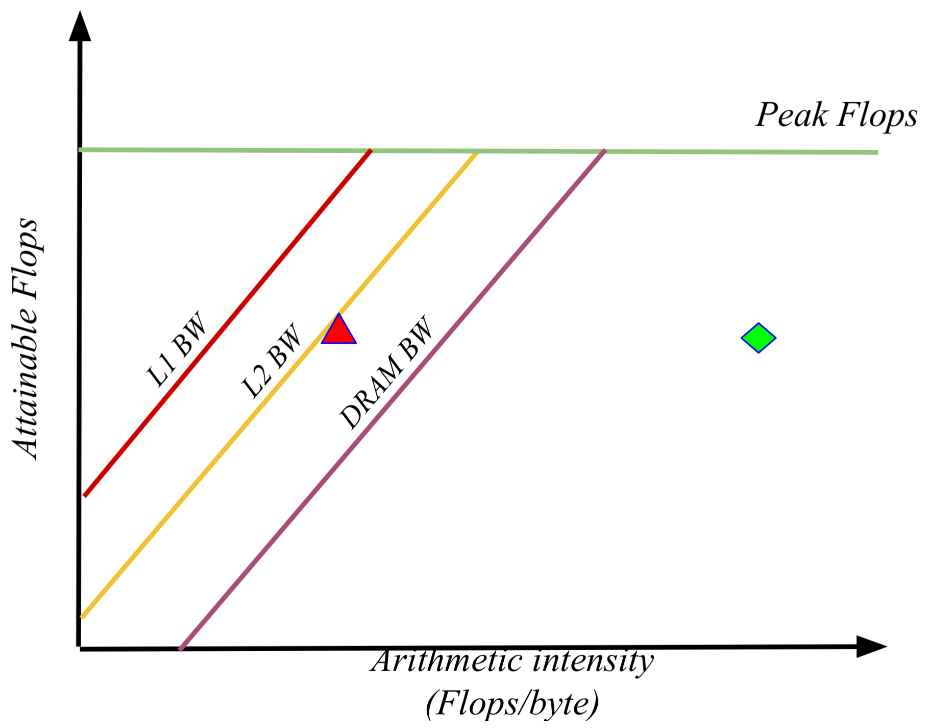


Figure 2.8: The roofline model

Definition 2.61 (Roofline model). *The **roofline model**, shown in Figure 2.8 models the achievable performance of an algorithmic implementation. For a specific algorithmic implementation, it plots the amount of Flops attained by the algorithm against its measured arithmetic intensity(AI). In particular, it showcases if the implementation is limited by the memory bandwidth of a certain memory level (DRAM, L2, L1 etc) or is limited by the amount of operations that the compute unit can perform.*

*Implementations that are limited by the compute, or algorithms with a very high arithmetic intensity (shown in Figure 2.8 in the green diamond) are said to be **compute bound**.*

*Implementations that are limited by the available memory bandwith, be it the main memory bandwidth or one of the cache bandwidths (shown in Figure 2.8 by the red triangle), are said to be **memory bound**.*

Definition 2.62 (Amdahl's law). *Consider an algorithm with both parallel and serial components. Let the proportion of the parallel component be equal to w. The serial component proportion is then equal to $(1 - w)$. Let the number of parallel resources on a system be p.*

***Amdahl's law** [Amd67] states that the theoretical speedup that can be achieved by running this algorithm using all p parallel resources is equal to*

$$S_{amdahl} = \frac{1}{(1-w) + \frac{w}{p}}$$

Moreover we see that

$$\lim_{p \to \infty} S_{amdahl} = \frac{1}{1-w}$$

which means that the serial component limits the total achievable speedup.

*Amdahl's law measures the speedup or parallel efficiency with a constant problem size. This type of scaling is called **strong scaling**.*

Amdahl's law gives a pessimistic estimate of the speedup as it assumes that the number of parallel resources is independent of the amount of work. This is in general not true and a more realistic estimate is available through Gustafson's law.

Definition 2.63 (Gustafson's law)**.** *Consider a algorithm with both parallel and serial components. Let the proportion of the parallel component be equal to w. The serial component proportion is then equal to $(1 - w)$. Let the number of parallel resources on a system be p.*

Gustafson's law *[Gus88] states that the theoretical speedup that can be achieved by running this algorithm using all p parallel resources is equal to*

$$S_{gustafson} = (1 - w) + pw$$

$$S_{gustafson} = 1 + (p - 1)w$$

Gustafson's law measures the speedup or parallel efficiency with an increasing problem size. This type of scaling is called ***weak scaling****.*

2.6 Overview of HPC systems

In this section, we give an overview of the four supercomputers we perform our experiments on.

2.6.1 The HoreKa supercomputer at KIT

Hochleistungsrechner Karlsruhe (HoreKa) [Hor] is an innovative hybrid system with nearly 60,000 Intel processor cores, and 668 NVIDIA A100 GPUs. It is housed at the Karlsruhe Institute of Technology's Campus North. The nodes that are used in this work are a subset of the full system and is called the HoreKa Green system.

The HoreKa Green system has 167 nodes with 2 Intel Xeon Platinum 8368 CPUs per node with each CPU having 76 cores. Each node additionally has 4 NVIDIA A100 (40GB- HBM) GPUs connected together with NVLINK. The interconnect used is the InfiniBand 4X HDR 200 GBit/s which provides a very low latency (~1 μs).

2.6.2 The Perlmutter supercomputer at NERSC

Perlmutter [NER23] is a HPE Cray EX supercomputer hosted at the National Energy Research Scientific Computing Center (NERSC), Berkeley, USA. Based on the HPE Cray Shasta platform, it is a heterogenous system comprised of 3072 CPU-only nodes and 1792 GPU-accelerated nodes.

In this work, we only make use of the GPU-accelerated nodes, which consists of 1 AMD EPYC 7763 (Milan) CPU with 64 CPU cores. Each node contains 4 NVIDIA A100 (40GB-HBM) GPUs connected together with 3rd generation NVLINKs. The interconnect used is the 4X HPE Slingshot 11 with 200Gbits/s bandwidth and the nodes are connected in a Dragonfly topology.

2.6.3 The Summit supercomputer at ORNL

Summit [Sum] is an IBM system located at the Oak Ridge Leadership Computing Facility with a theoretical peak double precision performance of approximately 200 petaflops. It consists of approximately 4600 heterogenous compute nodes, each of which contain 2 sockets of IBM Power 9 CPUs, each with 22 SIMD cores (SIMD width of 4). Each socket is connected to 3 NVIDIA V100 (16GB) GPUs with NVLINKs between GPUs on the same socket for a total of 6 V100 GPUs per node. The summit nodes are connected together with a dual rail EDR InfiniBand network providing a node injection bandwidth of 23 GB/s, in a Non-blocking Fat Tree topology. A node schematic of the summit node is shown in Figure 2.9.

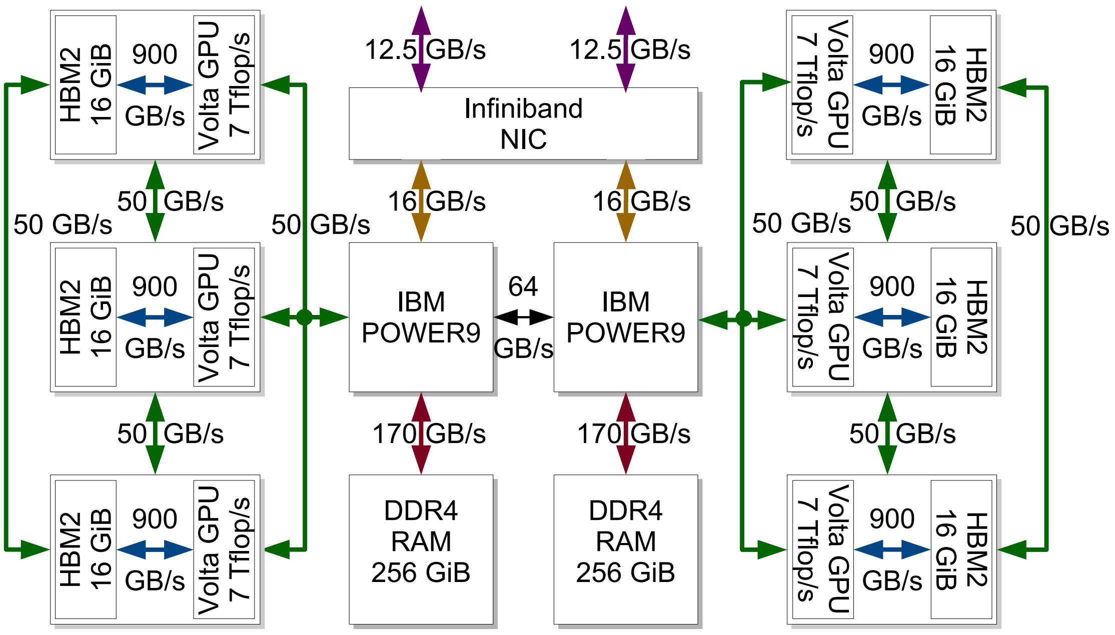


Figure 2.9: ORNL's Summit supercomputer node schematic [Sum].

2.6.4 The Frontier supercomputer at ORNL

Frontier [Fro] is a HPE Cray EX supercomputer located at the Oak Ridge Leadership Computing Facility. With a theoretical peak double precision performance of 2 Exaflops, it is top 1 on the Top500 list [Top]. It is also a heterogenous system with both CPUs and multiple GPUs per node. In total it has 9,408 compute nodes. Each compute node consists of 1 AMD 3rd Generation EPYC CPU with 64 CPU cores and 4 AMD MI250X GPUs (8 GCDs in total, 2 GCDs per GPU), each with 64 GB of HBM2E. The CPU is connected to each GCD via an Inifinity Fabric enabling CPU-GPU transfers of 36GB/s. On the same GPU, the GCDs are also connected via the Infinity Fabric with BW of 200GB/s. The GPU-GPU data trasnfers is also through the Infinity Fabric. A notable aspect is that the network cards are connected directly to the GPUs instead of the CPUs and hence off node transfers are staged through the GPU memory. The schematic of a Frontier node is shown in Figure 2.10.

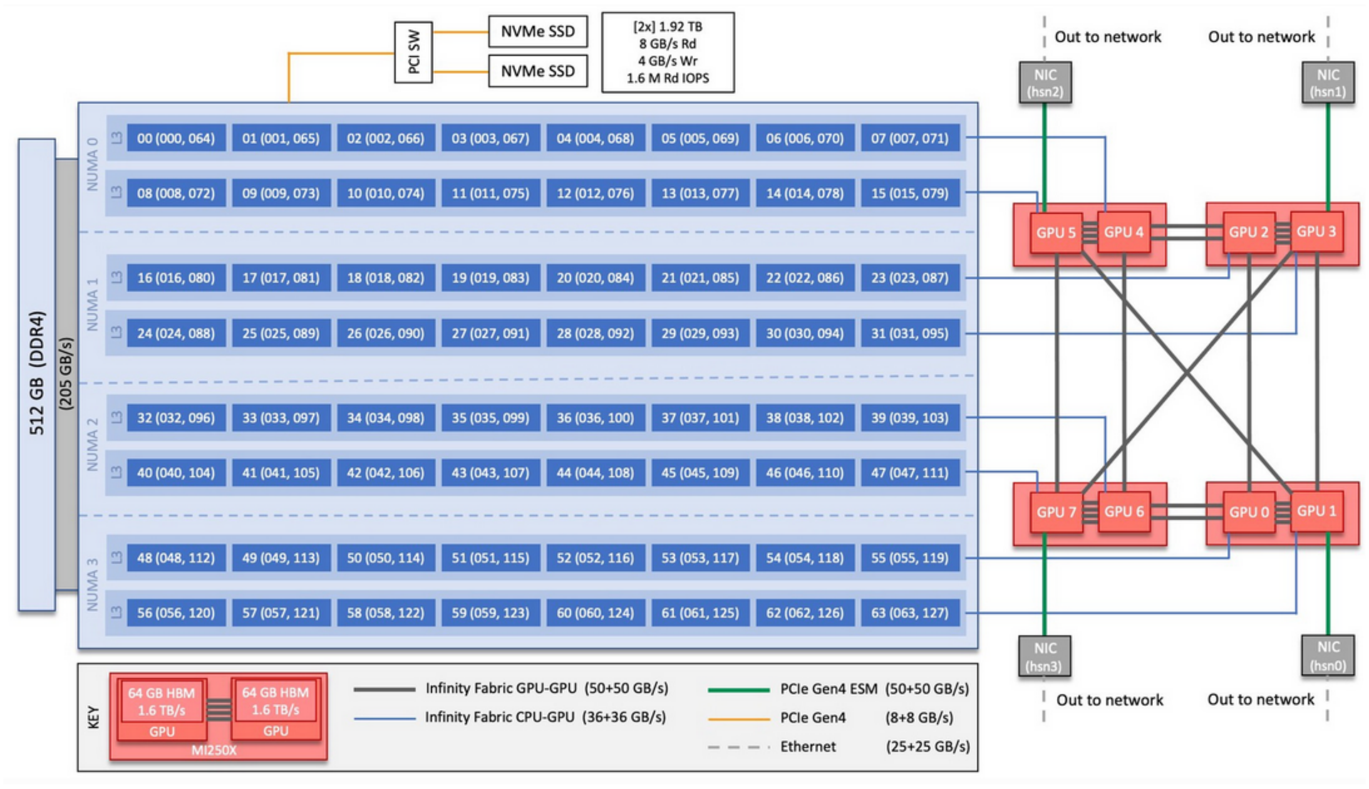


Figure 2.10: Frontier node diagram [Fro]

3 Ginkgo: A high performance numerical linear algebra library

Mathematical software lies at the center of computational science. Most applications, for example, simulations for plasma physics, combustion, weather, and so forth, need to perform various operations, which can be expressed through simple linear algebra. Even non-traditional HPC applications, such as Machine Learning and Artificial Intelligence need to use linear algebra operations ranging from matrix-matrix multiplications to eigensolvers and linear solvers.

The explosion of hardware architectures from different vendors, with differing programming models have brought about the challenge of performance portability. In particular, most supercomputers now have GPUs, where most of the computational power is concentrated. There have been many efforts in trying to unify programming models for these different architectures, but that has proved challenging [Khr14; AMD23], with architecture-specific quirks only supported by the architecture-specific vendor programming model; Maximizing performance on architectures requires the usage of the vendor-specified programming model.

Ginkgo [ACF+22; ACC+20b] is a high-performance numerical linear algebra library that aims to leverage the hardware vendor's native programming model to implement highly tuned architecture-specific kernels. Separating the core algorithm from these architecture-specific kernels enables high performance while enhancing the readability and maintainability of the software.

Ginkgo provides linear solvers such as CG, BiCGSTAB, GMRES, and preconditioners such as Incomplete LU, block-Jacobi, and sparse approximate inverse, which help accelerate the solution of the linear solvers. A particular strength of Ginkgo is its support for myriad sparse matrix formats (CSR, ELL, COO, Sell-P, etc.), each providing different architecture-tuned sparse matrix vector product kernels, which form the workhorse of the iterative linear solvers [ACC+20a].

We elaborate on Ginkgo's software design in Section 3.1 and on the `LinOp` concept on which Ginkgo's objects are based, in Section 3.2. We detail Ginkgo's matrix, solver, and preconditioner functionalities in Section 3.3, also showing their performance. In Section 3.4, we showcase Ginkgo's distributed functionalities and analyze the performance and scaling on large supercomputers. In Section 3.5, we show how Ginkgo can help accelerate applications and maximize their performance. Finally, we conclude in Section 3.6.

We note that some contents of this chapter has been published in the following publications [ACC+20b; ACF+22; ACC+20a].

3.1 Designing software for high performance

The lifecycles of software often exceed those of hardware, particularly in computational science [AHL+11]. Software, therefore, has to be designed with both extensibility and backward compatibility in mind. A particular challenge for mathematical software that plugs into applications and runs on supercomputers is the need to be able to maximize performance on various hardware backends. There are two main approaches to tackle this challenge: A single source portability layer such as Kokkos [CETS14], RAJA [BSB+19], SYCL [Khr14] etc. which aim to provide the users with a single software layer, which in turn calls the hardware-specific programming model. This portability layer enables the user to program in a uniform fashion without worrying about hardware and vendor-specific quirks. The alternative approach is to write tuned kernels for each hardware backend using the vendor's native programming model. It has been shown that due to the differing nature of the vendor programming models, the generalization achieved with a single source software layers may not always provide the highest performance [EVM+20; GVD+23]. GINKGO, therefore, utilizes the second strategy of implementing highly tuned kernels for each hardware backend using the vendor-native programming model.

With software sustainability and maintainability as its central principles, GINKGO enables high performance with the following practices:

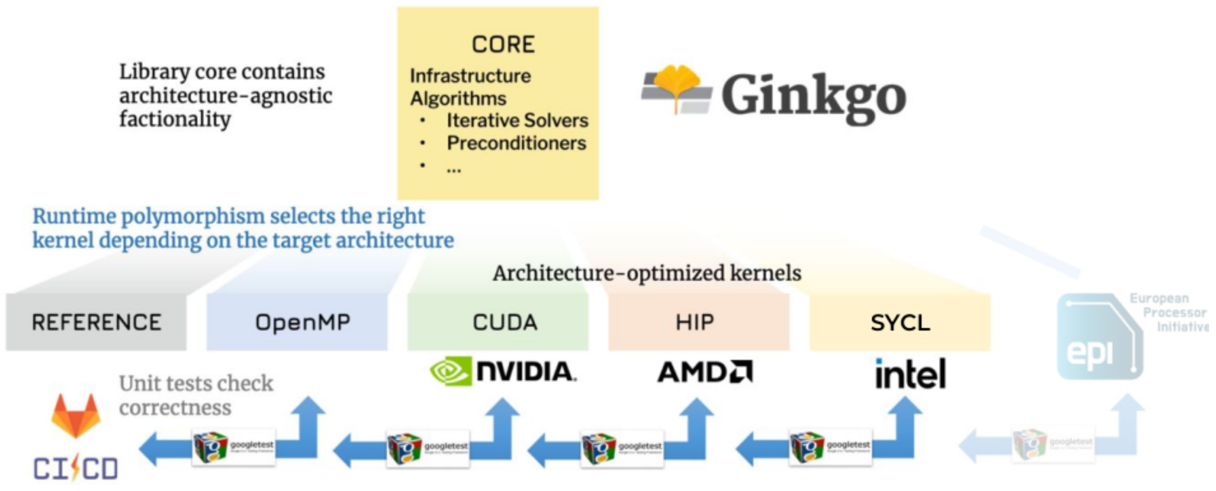


Figure 3.1: A schematic showing GINKGO's software architecture

1. Abstraction of complex algorithms (solvers, preconditioners) into a backend-agnostic core library composed of tuned kernels.
2. Highly tuned architecture-aware kernels for basic algorithms that are shared between the more complex algorithms.
3. An extensible `executor` interface that enables easy addition of new hardware architectures with minimal modification of the core algorithms.
4. An MPI layer that composes the existing `executor` interface and enables distributed computation on large supercomputers.
5. A single threaded reference implementation with verifiable output.
6. Comprehensive unit tests that ensure the robustness of both the basic kernels and the more complex algorithms.
7. A continuous integration pipeline that ensures thorough testing on different compiler suites and hardware platforms.
8. A code peer review framework that fosters collaboration, promotes the development of high quality software and encourages software sustainability.

Figure 3.1 shows the schematic for GINKGO's software architecture. A sub-library is compiled for each hardware backend. This sub-library consists of kernels tuned for that specific hardware using the hardware vendor's native programming model, enabling maximum performance. The core sub-library consists of algorithms and infrastructure and is hardware agnostic. At runtime, depending on the hardware available and chosen by the user, the backend sub-library provides high-performance implementations of the kernels with minimal overhead using modern C++ and runtime polymorphism.

3.1.1 High performance kernels using vendor-native programming models

Utilizing the vendor-native programming models enables GINKGO to maximize performance for each hardware architecture. Using a composition approach, that is, building the complex algorithms from their simpler building blocks, enables us to write kernels that can be shared between various algorithms. This is crucial to minimize code duplication and keep the code maintainable. For example, a **`prefix_sum`** kernel computes a cumulative sum of the elements in an array and is useful for index calculations in many algorithms. GINKGO implements a tuned **`prefix_sum`** kernel for each architecture that can then be composed with similar building blocks to write a more complex algorithm.

The **`Executor`** class handles the execution of the kernel on the correct hardware using runtime polymorphism. For each backend, GINKGO provides a separate executor class (**`CudaExecutor`** for NVIDIA GPUs, **`HipExecutor`** for AMD GPUs, and so forth). These executors override the **`run`** method from the base **`Executor`** class, hence calling the appropriate named kernel for that specific concrete executor. This enables seamless execution of the kernel on the hardware with minimal overhead [ACF+22].

3.1.2 Managing memory movement

With GINKGO focusing mostly on sparse linear algebra, careful memory management is crucial for optimized performance. With most sparse linear algebra algorithms being memory bound (few compute operations per fetched memory element), keeping data as close to the computing unit as possible and maximizing data reuse is essential. Additionally, in the context of distributed computing and for heterogeneous algorithms, it is necessary to minimize the memory transfers between host and device [CFW+21].

Most of GINKGO's objects are based on the host/device managed **`array`** class. Composing different data structures from this **`array`** class enables GINKGO to prevent unnecessary host-to-device transfers. Additionally, to enable seamless plugging into applications, most data structures can be constructed from device resident memory by creating **`array<ValueType>::view(...)`** wrappers over the data and providing the GINKGO objects with these array views. This view performs no deep copies but allows GINKGO to view the memory and update it if necessary.

3.1.3 Extracting fine-grained level parallelism

Current GPU architectures are highly hierarchical, both from a compute perspective and a memory perspective. To maximize the compute performance, extracting the fine-grained level of parallelism from the algorithms and mapping them to the appropriate level of the compute hierarchy is necessary. Similarly, with multiple layers of memory and cache levels, it is vital to design data structures that maximize cache usage and minimize memory traffic.

The computational hierarchy consists of four levels. At the finest level is the single work-item (thread). These work-items are then organized into sub-groups (warps). All the work-items in a sub-group execute the same instruction. These sub-groups are then organized into work-groups (thread blocks), which are mapped to the streaming multi-processor on the GPU. It has to be noted that all work items within a work-group can share data using the L1 cache/shared local memory. Work-items between two distinct work-groups can communicate only through the GPU global memory, which can be significantly more expensive.

To maximize the performance of the GPU, we should aim to reduce the GPU's main memory accesses and cache as much data in the shared memory as possible. Additionally, utilizing the fine-grained level of parallelism within a warp can significantly improve performance. To this end, GINKGO provides an abstraction layer that emulates the cooperative groups introduced in CUDA 9 for all the GPU hardware backends. This cooperative groups interface enables GINKGO to utilize sub-group and work-group level shuffle, ballot, and masking operations, enabling us to minimize memory movement while maximizing the available fine-grained parallelism on all GPUs in a unified interface [TCR+21].

3.1.4 Enabling mixed-precision computations

The proliferation of different precision formats has challenged software support for these various precision formats. Hardware support is generally available for only a few standard formats, such as those defined in the IEEE 754 standard. In the past, software libraries were usually compiled for a single format, and mixing different formats within a single library was impossible.

Numerical analysis has shown that various methods are robust enough to use mixed and multi-precision [HM21] and a performance assessment has shown that they can provide significant benefits [AAB+20]. Through modern C++ and templating, GINKGO supports mixed precision computations. All data structures, solvers and preconditioners are instantiated for the IEEE 754 single and double precision and some special algorithms make use of mixed precision for different kernels, maximizing performance without losing accuracy [AAG+22; AAG+21].

A novel concept in GINKGO that enables easy usage of mixed-precision is the accessor concept. This enables GINKGO to separate the storage and the compute precision, utilizing lower precision for storage, thereby saving on the memory transfers and higher precision for the computation, minimizing round-off errors in registers. The conversion between precisions has been shown to have minimal overhead [GAQ23] and provide benefits, particularly for memory bound computations.

3.2 The LinOp concept

A central concept in GINKGO is its **`LinOp`** class. This class forms the base class that provides the **`apply`** method, which represents the application of a linear operator,

$$\mathcal{L} \to apply(b, x) \mapsto x = \mathcal{L}(b) \tag{3.1}$$

Most of GINKGO's operations can be represented as linear operations of this type. A (sparse) matrix is applied to a vector to map it into another vector (SpMV), a (sparse) matrix is applied to a (sparse) matrix to obtain another (sparse) matrix (SpGEMM), a solver is applied to the right-hand-side to obtain a solution, a preconditioner is applied to a vector to map it into another vector, and so on. In combination with vector-based operations such as norm, dot, and scale, all of GINKGO's solvers can be composed of these building blocks.

3.2.1 Generating a LinOp

Listing 3.1: Generating a Cg solver

```
using cg = gko::solver::Cg<>;
using iter = gko::stop::Iteration;
using residual_norm = gko::stop::ResidualNorm<>;
auto cg_factory  =
            cg::build()
                    .with_criteria(
                        iter::build()
                            .with_max_iters(20u).on(exec),
                        residual_norm::build()
                            .with_reduction_factor(tolerance)
                            .on(exec))
                   .on(exec);
    // generate the solver with an input LinOp.
    auto cg_solver = cg_factory->generate(system_matrix);
```

Different **LinOps** can have different user-defined parameters, and different algorithms are generally used to create the various **LinOps**. Ginkgo, therefore, defines a **LinOpFactory** class that provides a **generate** method that each **LinOp** can use to generate its operator.

Ginkgo also provides a simple macro, **GKO_CREATE_FACTORY_PARAMETERS**, which automatically wraps the user-settable parameters of the class into a struct for easy use in the generate and apply methods. Therefore, the user has to just call the **build** method of the factory and provide any parameters they wish to specify and subsequently call the **generate** method to generate the **LinOp**. For example, generating a CG solver for the solution on a specified executor with Ginkgo is shown in Listing 3.1. The same **cg_factory** can generate different solvers given different system matrix objects.

3.2.2 Composing and combining LinOps

In addition to the **apply** method, **LinOps** represent the mathematical equivalent of a linear operator in fixed precision.

A **Composition** object represents the composition operator, $\circ$, of two linear operators, representing the operation,

$$\begin{gathered}\mathcal{L}_{composition} = \mathcal{L}_1 \circ \mathcal{L}_2 \\ \mathcal{L}_{composition} \rightarrow apply(b,x) \mapsto x = \mathcal{L}_1(\mathcal{L}_2(b))\end{gathered} \tag{3.2}$$

Similarly, a **Combination** object represents the combination, $+$, of two linear operators with the operation,

$$\begin{gathered}\mathcal{L}_{combination} = \mathcal{L}_1 + \mathcal{L}_2 \\ \mathcal{L}_{combination} \rightarrow apply(b,x) \mapsto x = \mathcal{L}_1(b) + \mathcal{L}_2(b)\end{gathered} \tag{3.3}$$

Creating these composition and combination objects is simple with the `create` method, which takes the **LinOp** objects as input and stores a `std::vector` to these **LinOps**. As these composition and combination objects are themselves **LinOps**, they can be applied to another **LinOp**, which recursively calls the `apply` method of the stored **LinOp**, unwrapping them in a fashion as shown above.

3.3 Matrices, solvers and preconditioners

Maximizing performance on a GPU requires careful design of data structures to minimize memory movement, maximize cache re-use and maximize the available compute resources. With Ginkgo's primary focus on Krylov solvers, we provide a wide variety of bespoke matrix data structures with the SpMV operation, which form the work-horse of all Krylov solvers.

3.3.1 Sparse data structures in Ginkgo

For a particular system matrix, it is important to first consider the sparsity pattern of the matrix, and choose an appropriate sparse data structure that can maximize the SpMV performance. Therefore, we provide the following sparse matrix data structures:

1. **`Coo`** : A coordinate format that stores the triplet: row index, column index and value for each non-zero element in separate arrays.
2. **`Csr`** : A compressed sparse row matrix format that stores the column index and value for each nonzero and a cumulative sum of the pointers to start of each row's nonzero values.
3. **`Ell`** : An ELLPACK matrix format that stores the same number of nonzeros per row (padding with zeros for structural zeros) removing the need for the pointer array.
4. **`SellP`** : A sliced version of the ELLPACK format, that stores blocks of **`Ell`** matrices each with different stride values (number of nonzeros per row, with padding).
5. **`Hybrid`** : A combination of the **`Coo`** and the **`Ell`** that combines the benefits of both these data-structures.

For balanced matrices with a constant number of nonzeros per row, the **`Ell`** format usually gives the best performance. For highly imbalanced matrices (a wide range of non-zeros per row), the **`Hybrid`** matrix format can provide optimal performance by storing rows with smaller number of nonzeros in a **`Ell`** format, while using **`Coo`** for the more imbalanced rows.

The **`Csr`** matrix format is the most versatile, and suitable for a very wide range of operations, such as sparse factorizations, global to local index computations and hence it is the most widely used matrix format. To this end, we provide a wide range of strategies for the **`Csr`** matrix SpMV operation that can perform well for a wide range of sparsity patterns [ACC+20a].

3.3.2 Solvers and preconditioners in Ginkgo

Ginkgo supports a wide range of solvers and preconditioners that enable solution for a wide array of problems. These include Krylov solvers that can be composed from simple BLAS operations:

1. **`Cg`** : A short recurrence Krylov solver that is optimal for symmetric positive definite matrices [FS12] (See Definition 2.48).
2. **`Bicgstab`** : A short recurrence Krylov solver that extends the ideas from **`Cg`** to enable solution for non-SPD matrices [van92].
3. **`Gmres`** : A generalized minimal residual method that is optimal and monotonic in exact arithmetic and can be used to solve general matrices [SS86](See Definition 2.47).
4. **`Idr(s)`**: A short recurrence induced dimension reduction method that constructs residuals in shrinking subspaces and can be used for solution of general matrices [Sv08].

One option to improve the convergence of Krylov solvers is using preconditioners. Preconditioners aim to reduce the effective condition number of the iteration matrix, consequently enabling faster convergence. Preconditioner choice depends on the application, and therefore, Ginkgo supports a wide variety of preconditioners:

1. **`Jacobi`** : A (block-) diagonal preconditioner that inverts the (block-) diagonal and is effective for (block-) diagonally dominant matrices [FAC+21].
2. **`Ilu(k)`** : A general preconditioner that produces lower and upper triangular factors (with control on sparsity of these factors) that in combination with triangular solvers can be used for preconditioning general matrices [Doi91].
3. **`Isai`** : A sparse approximate inverse preconditioner that aims to produce an approximate inverse of the matrix which can then be applied relatively cheaply with an SpMV [BMT96].

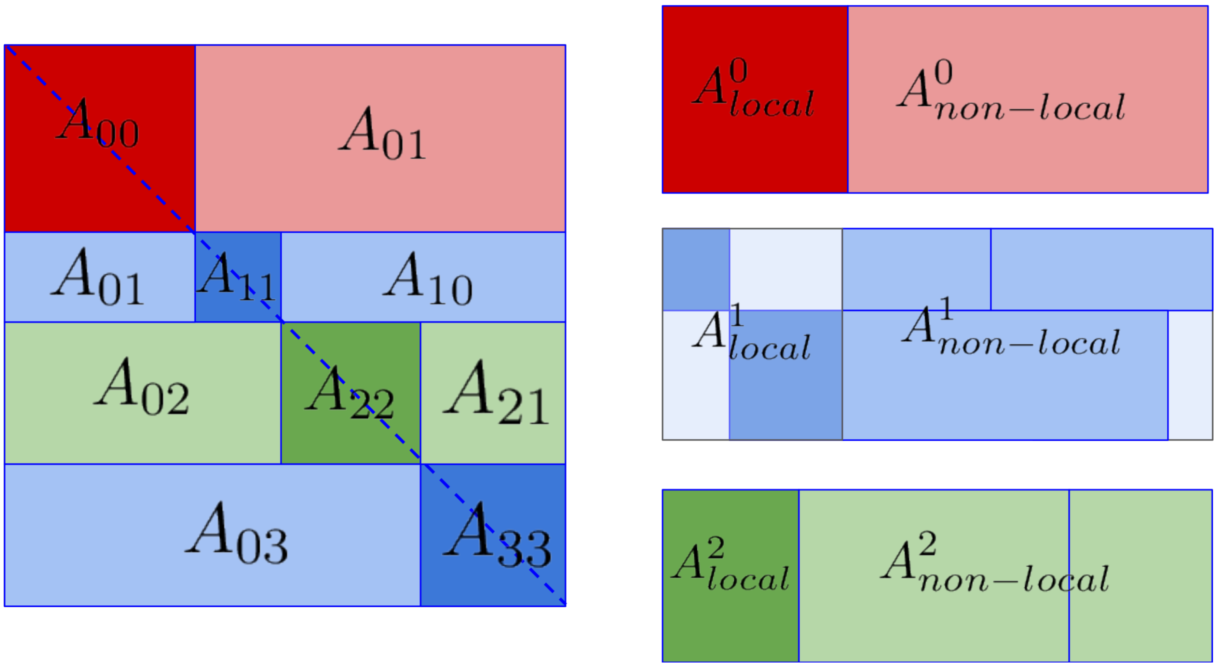


Figure 3.2: GINKGO's **`distributed::Matrix`** storage schematic

4. **`Multigrid`**: A multi-resolution method that constructs a hierarchy of matrices from the input matrix to isolate and minimize the high-frequency errors. GINKGO also provides an algebraic parallel graph match algorithm to construct matrices for different the different hierarchical levels [NAC+15; TBA23].

For a detailed analysis of the solver performance, we refer the reader to the GINKGO paper [ACF+22].

3.4 Enabling efficient distributed computations

Solving large-scale problems with billions of unknowns/parameters requires using distributed memory machines, as the problem size may exceed the available shared memory capacity. For problems with inherent parallelism, scaling up the parallel resources allows for acceleration within the limits of Amdahl's law (Definition 2.62).

Efficient distributed computation requires efficient data structures, particularly on the latest supercomputers that are highly hierarchical within a single node. Data structures need to be designed for coarse partitioning to enable the highly parallel, on-node accelerators to harness their compute, while minimizing off-node communication, as these off-node latencies and bandwidths can be an order of magnitude worse than the on-GPU and on-node ones.

For distributed computation, GINKGO primarily uses the MPI model and provides a precision templated thin layer over MPI. Each MPI rank then instantiates an executor and can execute operations on the hardware the executor is associated with. This enables users to perform heterogeneous computing, with some MPI ranks associated with GPUs and some with CPUs. For the distributed sparse matrix data structures, GINKGO provides a general **`distributed::Matrix`** class that enables users to mix and match the different single executor sparse matrix data structures depending on the application's sparsity requirements. Row and column partitioning are available, allowing the user to partition their matrix in a general fashion. The **`distributed::Matrix`** class internally stores a local and non-local sparse matrix in each rank.

Figure 3.2 shows the storage schematic GINKGO's **`distributed::Matrix`** object. Consider a row partition that partitions the matrix as shown on the left. If we consider three MPI ranks and assign the 0th block to rank 0, the 1st and third blocks to rank two, and the second block to rank 3, we obtain a block partition mapped to ranks as shown in the right. Each rank, therefore, stores a local diagonal block and a non-local off-diagonal block. We note that each of these local and non-local matrices can be stored in any format that a single executor GINKGO supports. This is particularly important because the non-local matrix blocks are usually hyper-sparse and possibly imbalanced, while the local diagonal blocks are more structured. The local and non-local matrices are stored with local indices for all ranks. If necessary, the global indices can be obtained from the class metadata, which additionally stores information about the neighbors of the rank, the indices, and the number of elements to exchange with its neighbors.

3.4.1 Scaling the distributed SpMV on large supercomputers

Efficient scaling on large supercomputers for our linear solvers requires optimization of the distributed SpMV. Using our **`distributed::Matrix`** format, the distributed SpMV can be split into a local SpMV and a non-local SpMV.

The **`distributed::Vector`** is also partitioned in a similar fashion to the distributed matrix class and only stores the local vector in a **`matrix::Dense`** format. In Figure 3.3a, the colors represent the owners for the respective vector elements. In Figure 3.3b for the right-hand side, the lighter shades represent the vector elements that we need to gather from other ranks. In the solution vector, we only write into our locally owned elements (the non-hashed parts), and hence, we only need one communication step per distributed SpMV. The distributed SpMV can be represented as

$$\begin{aligned} x^k_{local} &= A^k_{local} b^k_{local} \\ x^k_{local} &= x^k_{local} + A^k_{non-local} b^k_{non-local} \end{aligned} \tag{3.4}$$

We must communicate the non-local parts of the distributed right-hand side vector, but we only write into the rank-local x vector. This enables us to split the distributed SpMV into three steps, as shown in Algorithm 1. We note that lines 2 and 3 are independent and can be performed in parallel, enabling us to overlap computation and communication.

Algorithm 1 GINKGO's distributed sparse matrix vector product

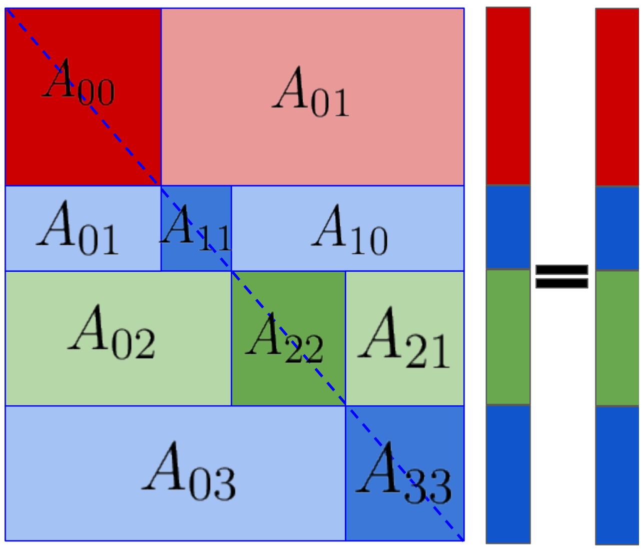
```
1: local_x ← $x_{local}$, local_b ← $b_{local}$
2: $A_{local}$ → APPLY(local_b, local_x)                    ▷ Local SpMV
3: $b$→ GATHER_NON_LOCAL(buffer)                           ▷ Communicate the non-local vector
4: $x_{local}$ += $A_{non-local}$ → APPLY(buffer, local_x)   ▷ Non-local SpMV
```

Figure 3.4 shows the weak scaling of GINKGO's distributed SpMV on Frontier, the first exascale supercomputer. We fix the local problem size (number of rows) and increase the number of ranks. Each rank is mapped to 1 GPU, which for Frontier is one GCD of the MI250X 2GCD GPU. We show the achieved performance for four types of problems originating from a stencil problem for up to 16,384 GPUs, which corresponds to 2048 nodes of Frontier. The 27-point stencil has a significant communication footprint due to the large stencil bandwidth. We observe that the SpMV has an excellent linear scaling for the larger problem sets where the single GPU is saturated and obtains a weak scaling efficiency of over 80% with respect to a single GPU.

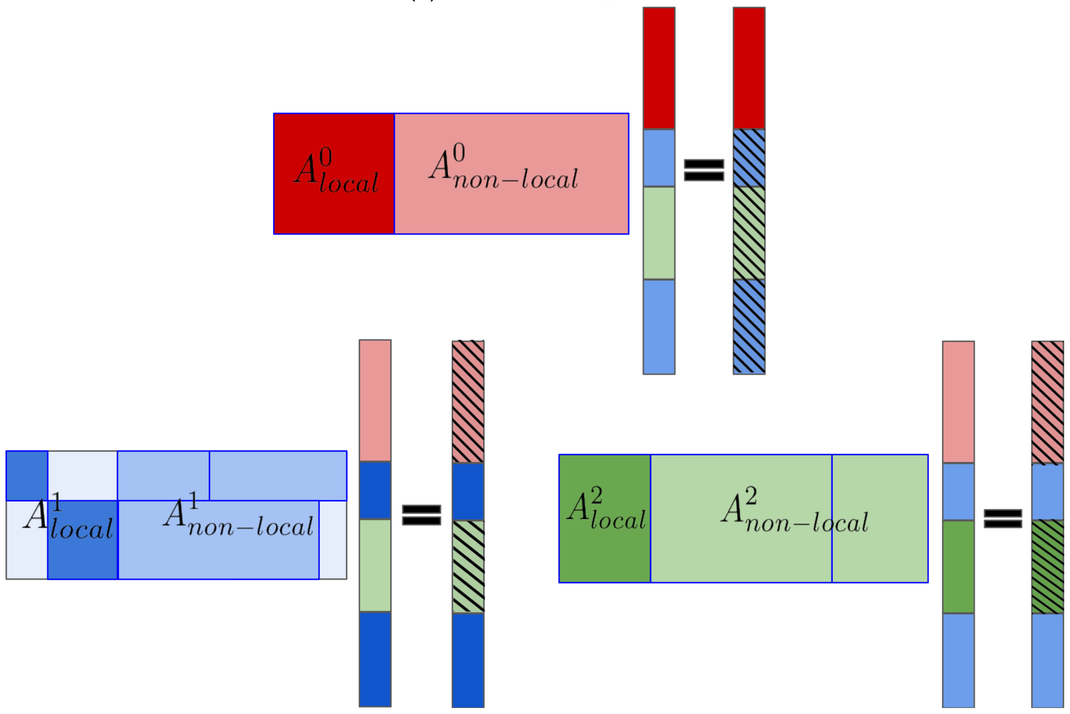
3.4.2 Scaling distributed solvers on large supercomputers

In addition to the distributed SpMV, for the Krylov solvers, additional bottlenecks include global reduction, which is necessary when computing the dot products and norms. In this case, we rely on the **`MPI_All_reduce`** implementation of the underlying MPI implementation on the machine.

Figure 3.5 shows the strong scaling of the GINKGO's Krylov solvers on Frontier for upto 1024 GPUs [Koc23]. Again, we fix the global problem size and increase the amount of parallel resources (GPUs). We show scaling for two Krylov solvers: CG with a distributed Schwarz preconditioner with a Block-Jacobi preconditioner. We observe that the algorithms show strong scalability as long as the local problem size is large enough to saturate the compute power of the GPUs. As soon as the communication starts dominating the runtime, the scalability degrades.



(a) Distributed SpMV



(b) Rank specific view of the distributed SpMV

Figure 3.3: Schematic of SpMV for Ginkgo's **`distributed::Matrix`**.

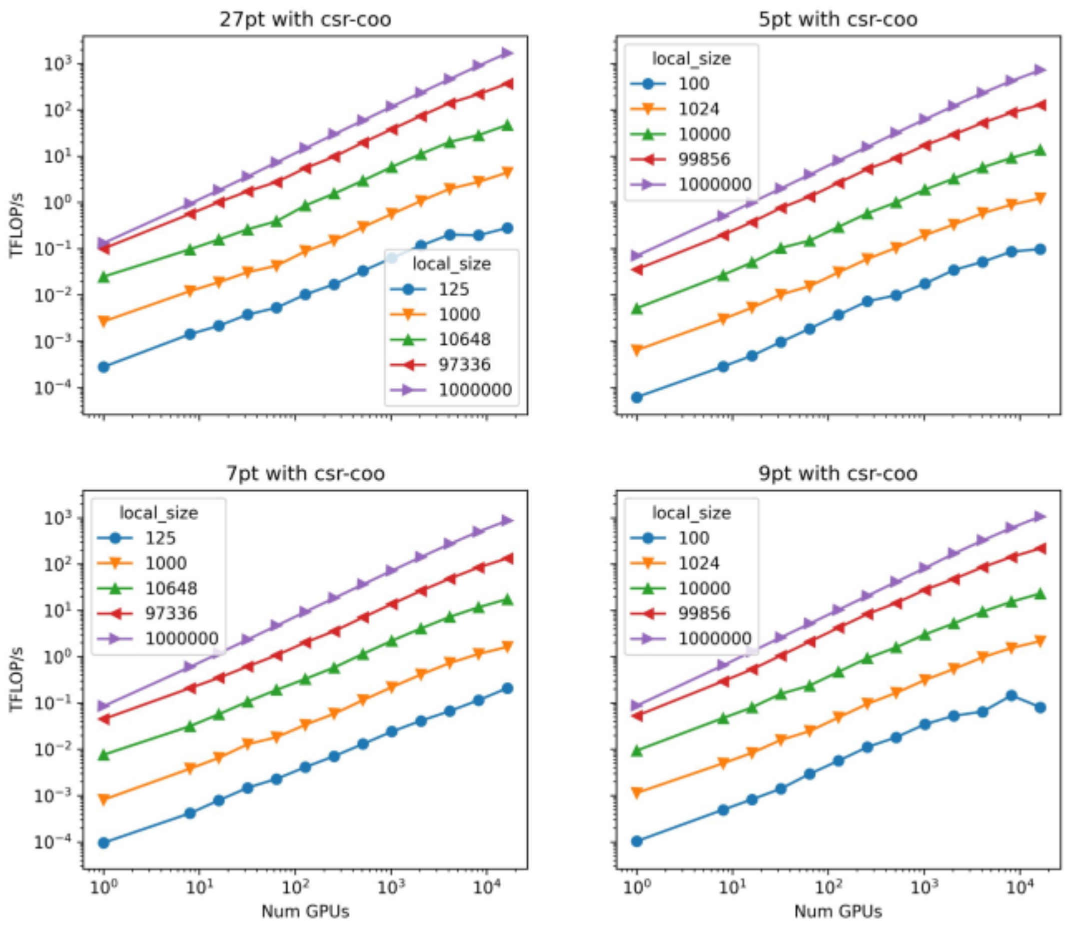


Figure 3.4: Weak scaling of GINKGO's distributed SpMV on the Frontier supercomputer.

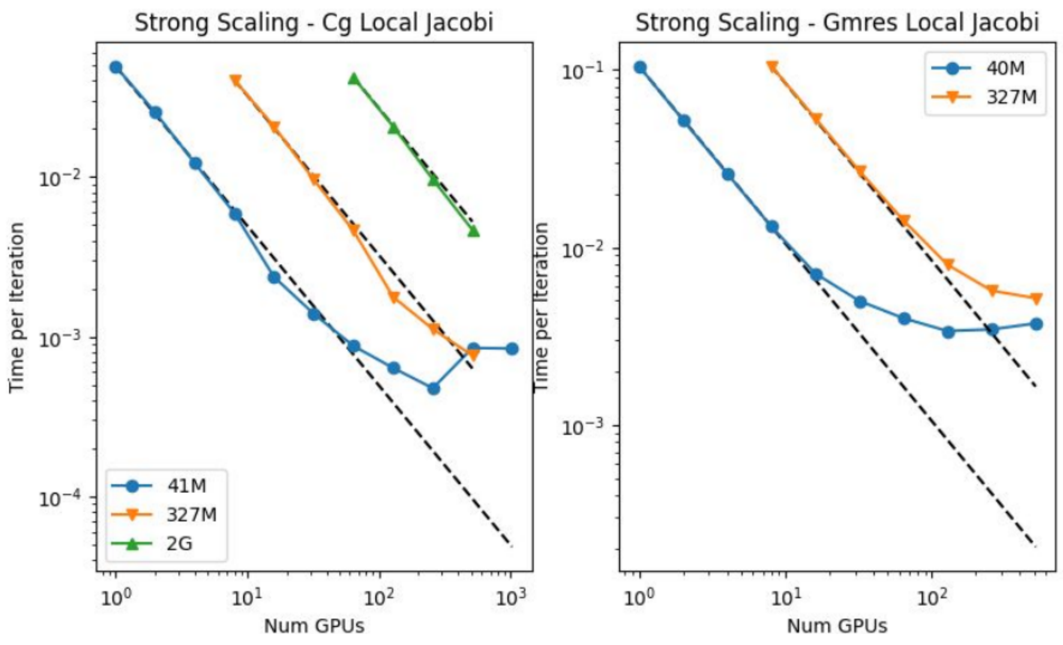


Figure 3.5: Strong scaling the iterative solvers (CG and GMRES) on the Frontier supercomputer

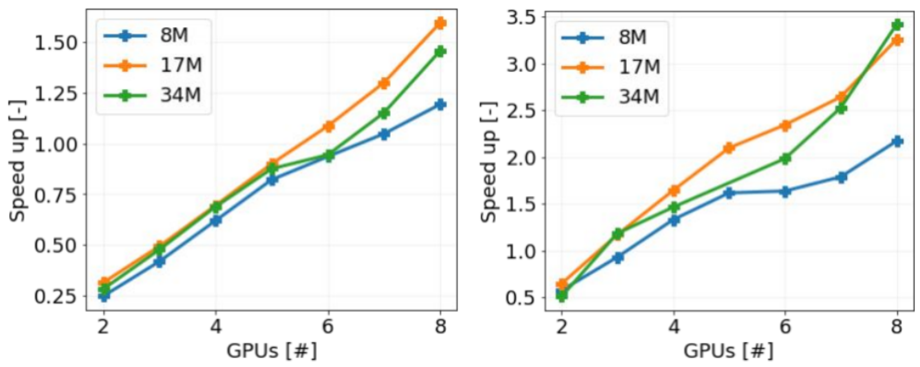


Figure 3.6: Accelerating OpenFOAM with distributed GINKGO GPU solvers.

3.5 Integrating into applications

GINKGO provides integrations into many applications, enabling them to accelerate their science on machines with different GPU architectures. Applications using GINKGO for accelerating scientific research include finite element libraries such as MFEM, deal.ii; XGC, a plasma physics application; SUNDIALS, an ODE solver library; HyKKT, an electric grid optimization library; OpenCARP, a cardiac modeling library; OpenFOAM, a fluid simulation software, with many other application integrations coming soon.

To ease integration into applications, GINKGO provides the following functionalities:

1. Data views: For efficient memory usage, GINKGO can operate on views from the application without needing to copy the data over.
2. Stream/queue and allocator interfaces: To reduce unnecessary synchronizations, GINKGO can use an application-provided stream/queue. GINKGO can also use pool allocators from the application, reducing memory allocation/deallocation overheads.
3. Matrix assembly: Two different classes with structures of array (SoA) and array of structures (AoS) data layouts enable easy matrix assembly for applications.
4. Logging functionality: An easily extensible logging interface that enables applications to look under the hood of GINKGO and record, log, and write solver, matrix, and preconditioner data to tune and debug applications.
5. Profiling annotations: GINKGO hooks into different GPU vendor profiler plugins such as NVTX, rocTX, and Intel VTune in addition to popular profilers such as TAU, enabling them to annotate GINKGO's functions for easy lookup and analysis.

Figure 3.6 shows the speedup with OpenFOAM using the OpenFOAM-Ginkgo layer [OA23]. The base case is the OpenFOAM default CPU solver with the Lid driven test case on a machine with AMD EPYC CPUs and 8 AMD MI100 GPUs. The left figure shows the speedup using GINKGO's distributed CG solver with a distributed Schwarz preconditioner with local ISAI preconditioner and on the right with the same solver except using a Multigrid preconditioner within the distributed Schwarz preconditioner.

3.6 Summary

Accelerating science requires software that can maximize the computing capabilities of the hardware. GINKGO is a mathematical numerical linear algebra library framework that enables applications to accelerate their computations. GINKGO provides efficient methods and algorithms that can harness the parallelism on different GPU and CPU architectures and also enable scaling to thousands of nodes on large supercomputers.

In this chapter, we looked at GINKGO's software architecture that enables high performance while focusing on sustainability and good software practices. We gave an overview of GINKGO's features, such as matrix formats, solvers, and preconditioners, that applications can take advantage of. We also looked at GINKGO's support for large-scale distributed computation and showcased its performance on large supercomputers such as Frontier. We finally elaborated on the various applications that utilize GINKGO and the features that GINKGO provides to enable seamless integration with these applications.

4 Batched Iterative solvers on GPUs

Batching is a technique that enables massive levels of parallelism through independent concurrent computations that have no data dependencies. The aim is to process multiple problems with the same algorithm, with these individual problems being too small to occupy the compute resources fully. Batched methods have been utilized for sparse and dense matrix-vector multiplications [OYI+19; ACD+17], for solution of pairwise-independent linear systems [ADF+17b] and for singular value decomposition [BTL+18], to name a few common algorithms. Many applications have also utilized batched methods to accelerate their computations, ranging from high-order Finite Element and Discontinuous Galerkin methods and metabolic networks to quantum chemistry and image compression. Batched dense matrix-matrix multiplications have also been used [AHT+16] in machine learning algorithms to accelerate the training and inference for deep neural networks.

Batching techniques have been widely used in linear algebra, and a community-driven standard interface was also developed [DDG+16] to encourage vendors to provide high-performance and tuned implementations of the commonly used BLAS and LAPACK routines [NVI23a; Int23a].

Batched computations are often useful in large sparse linear algebra operations to maximize throughput. In particular, when sparse matrices contain small dense blocks, as when assembled through high-order finite element discretizations, batched methods can accelerate the computations involved in the solution of these matrices [BAT+20]. Factorization of large sparse matrices, which are used in direct solvers, also makes use of batched operations to parallelize computations and maximize compute usage [CDH+08; DER17]. In sparse iterative solvers, preconditioners can help accelerate convergence by reducing the working condition number. For example, the widely-used block-Jacobi preconditioner consists of applying a block-diagonal matrix to a global vector. Utilizing batched methods in generating these inverted block-diagonals [ADF+17a] and their application can provide significant runtime benefits.

In addition to maximizing throughput for large sparse matrices by breaking them into batched computations, many applications need to solve a large number of independent and small linear systems. Batched direct methods have been the most popular solution for these linear systems. LAPACK [And99] contains many routines widely used by applications for the solution of batched linear systems. For linear systems with commonly occurring sparsity patterns, such as tri-diagonal, penta-diagonal, and general banded matrices, tailored algorithms have been developed for CPUs and some GPU architectures [PS07; VMS+18; GÓNP19].

Work on batched iterative solvers, on the other hand, has been very limited. The dogma in the community has been that direct solvers are well-suited for these small linear systems; hence, research on iterative linear systems has been lacking. The configurability and adaptability of the iterative solvers make them a good candidate to be used in a batched fashion. The hierarchical parallelism provided by GPUs, which form the workhorse of the current and upcoming largest supercomputers, can be well utilized by the iterative solvers due to the large amount of parallelism in the building blocks of the iterative algorithms, mainly consisting of matrix-vector products, scalar and vector updates and reductions.

To this end, this chapter elaborates on the following contributions to the development of batched iterative solvers:

1. Theoretical memory and work complexity analysis for batched methods.
2. A flexible and composable design that eases integration into applications.
3. Novel high performance batched dense and sparse matrix formats to minimize storage and maximize cache utilization.
4. Novel high performance portable solver kernels tuned for different GPU backends, orchestrated by backend-agnostic core side algorithms.
5. Detailed performance analysis of the different batched methods, showcasing their scalability and efficiency.
6. Integration into and examples from two real-world applications:
    - Combustion simulation to study reactive flow evolution through the ODE solver library SUNDIALS and PeleLM,
    - A Gyrokinetic Particle-in-Cell (PIC) code, XGC, used to simulate plasma close to the edge region of a toroid.

In Section 4.1, we analyze the computational efficiency of the batched methods. We elaborate on the interface design in Section 4.2 and study the data structures and their optimality in Section 4.3. We discuss batched kernel implementation details in Section 4.4 and analyze the performance of the batched iterative solvers in Section 4.5. We also showcase the usage of these batched methods in applications in Section 4.6 and conclude in Section 4.7.

We note that some contents of this chapter have been published in the following publications [KNK+22; KNK+23; AKN+21; NA23; NNA23].

4.1 Maximizing compute and memory throughput with batched methods

For the solution of multiple independent linear systems, there are three main strategies:

1. $\mathscr{S}_1$: Solving the systems one after another, in a serial fashion.
2. $\mathscr{S}_2$: Assembling the linear systems in a block-diagonal form as shown in Equation (4.1) and solving for the block-diagonal ensemble system.

$$\mathbf{A} = \begin{bmatrix} A_1 & 0 & \cdots & 0 \\ 0 & A_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_B \end{bmatrix}. \tag{4.1}$$

3. $\mathscr{S}_3$: Solving the independent linear systems in a batched fashion, in parallel.

Theorem 4.1.1 (Optimality of batched methods). *Let $P \geq 1$, be the number of available parallel resources. With some measure of computational cost, $\mathscr{C}_{j\in 1,2,3}$ for each of the 3 strategies, $\mathscr{C}_1$, $\mathscr{C}_2$ and $\mathscr{C}_3$ respectively, representing the time to solution, strategy $\mathscr{S}_3$ is optimal, and the computational costs of the three strategies are ordered in the form,*

$$\begin{aligned} \mathscr{C}_2 \geq \mathscr{C}_1 = \mathscr{C}_3 \quad & \textit{for } P = 1 \\ \mathscr{C}_1 \geq \mathscr{C}_2 \geq \mathscr{C}_3 \quad & \textit{for } P > 1 \end{aligned} \tag{4.2}$$

Proof. Assume that we have B independent linear systems to be solved, with condition numbers $\kappa_b, \quad b = 1, \dots, B$ with P amount of parallel resources. A higher condition number implies a lower convergence rate and hence a higher number of iterations for convergence.

Let $\imath(A)$ represent the number of iterations for some iterative method as defined in Definition 2.38. The cost per iteration, $\mathcal{c}(A)$, is dependent on the structure of the matrix, particularly on the size of the matrix, n. For simplicity, assume that all the batch entries have the same size, $n \times n$. With the matrix-vector product being the most expensive operation in an iteration, we can assume that for a matrix $A \in \mathbb{R}^{n\times n}$, $\mathcal{c}(A) = \mathcal{O}(n^2)$.

The overall cost of the solution is, therefore, equal to the cost per iteration times the total number of iterations. Therefore we can write,

$$\mathscr{C}(A) = \imath(A)\mathcal{c}(A)$$

For strategy 1, the cost of the solution is equal to the sum of the cost of the individual linear system solutions.

$$\mathscr{C}_1 = \sum_{b=1}^{B} \imath(A_b)\mathcal{c}(A_b)$$

For strategy 2, the cost of the solution is equal to the cost of the solution of the block diagonal ensemble system, $\mathbf{A}$.

$$\mathscr{C}_2 = \imath(\mathbf{A})\mathcal{c}(\mathbf{A}) \tag{4.3}$$

Given a system matrix A, for a general iterative method, the convergence rate can be written as is

$$\varrho(A) = 1 - \frac{C_2}{\kappa(A)^{1/m}}$$

where $\kappa(A)$ is the condition number of the matrix A (See Theorem 2.2.5 and Theorem 2.2.8). For example, $m = 1$ for relaxation methods and $m = 2$ for CG [Gre97]. From Definition 2.38, the iteration count for an iterative method can be written as,

$$\begin{aligned} \imath(A) &= \frac{\log(\tau_{final})}{\log(\varrho(A))} \\ &= \frac{-C_1}{\log(\varrho(A))} \\ &= \frac{-C_1}{\log(1 - \frac{C_2}{\kappa(A)^{1/m}})} \end{aligned} \tag{4.4}$$

Let $\varkappa = \frac{C_2}{\kappa(A)^{1/m}}$. We see that as $\varrho(A) < 1$, $|\varkappa| < 1$. Therefore, we can use the series expansion,

$$\log(1-x) = -\sum_{k=1}^{\infty} \frac{(-1)^k(-x)^k}{k}, \quad |x| < 1$$

$$\log(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \dots$$

we have

$$\begin{aligned} \imath(A) &= \frac{-C_1}{\log(1-\varkappa)} \\ &\leq \frac{C_1}{\varkappa} \\ &\leq C\kappa(A)^{1/m} \end{aligned} \tag{4.5}$$

with $C = C_1/C_2$ being some constant, independent of the system matrix, A. For our cases, we can assume $m = 1$, as the choice of the iterative solver is same for all three strategies.

$$i(A) \leq C\kappa(A) \tag{4.6}$$

We see that the eigenvalues of the block-diagonal ensemble system, $\lambda_i(\mathbf{A})$ are the union of the eigenvalues of the individual batch entries.

$$\lambda_i(\mathbf{A}) \in \cup_{b=1}^{B}\{\lambda_i(A_b)\}$$

Therefore,

$$\kappa(\mathbf{A}) \geq \max_B(\kappa(A_b)) \tag{4.7}$$

Therefore, for strategy 2, we can write,

$$\begin{aligned} \mathscr{C}_2 &= i(\mathbf{A})c(\mathbf{A}) \\ &\leq \kappa(\mathbf{A})c(\mathbf{A}) \\ &\leq \kappa(\mathbf{A})Bn^2 \end{aligned} \tag{4.8}$$

Given P parallel resources and perfect parallelization of the block matrix vector product, we have

$$\mathscr{C}_2 \leq n^2\frac{B}{P}\kappa(\mathbf{A})$$

For strategy 2, the condition number of the block-diagonal ensemble matrix is minimized when the spectrum of the batch entries is contained within the bounds of the spectrum one batch entry. Therefore,

$$\kappa_{\min}(\mathbf{A}) = \max_B(\kappa(A_b))$$

$$\begin{aligned} \mathscr{C}_2 &\leq n^2\frac{B}{P}\kappa(\mathbf{A}) \\ &\leq n^2\frac{B}{P}\max_B(\kappa(A_b) \end{aligned} \tag{4.9}$$

For strategy 3, the total cost of solution is equal to the sum of the cost of the individual linear system solutions.

$$\mathscr{C}_3 = \sum_{b=1}^{B} i(A_b)c(A_b)$$

With the cost per iteration mainly dependent on the matrix structure, we can without loss of generality assume $c(A_i) = c(A_j) = n^2 \quad \forall i,j \in 1,...,B$. Therefore,

$$\begin{aligned} \mathscr{C}_3 &= n^2\sum_{b=1}^{B} i(A_b) \\ &\leq n^2\sum_{b=1}^{B}\kappa(A_b) \end{aligned} \tag{4.10}$$

With strategy 3, we split our set of systems $b = \{1,\ldots,B\}$ to be solved into B/P subsets, $B_i, \quad i = 1,\ldots,B/P$ each of size P. Therefore, we have

$$\mathscr{C}_3 \leq n^2\sum_{i=1}^{B/P}\max_{B_i}(\kappa(A_{b_i})) \tag{4.11}$$

In the worst case, we have that each subset (of size P) has the overall worst condition number, $\max_{B_i}(\kappa(A_{b_i})) = \max_B(\kappa(A_b))$. Therefore,

$$\mathscr{C}_3 \leq n^2\frac{B}{P}\max_B(\kappa(A_b)) \tag{4.12}$$

We see that the encompassing eigenvalue assumption made in Equation (4.9) is the worst case for strategy 3, with one system being the hardest to solve. Therefore comparing Equation (4.9) and Equation (4.12), we have that the best case condition number for strategy 2 is the worst case for strategy 3, in which case we have equality of the computational cost estimates. Therefore,

$$\mathscr{C}_3 \leq \mathscr{C}_2$$

Therefore, for $P > 1$, we have

$$\mathscr{C}_3 \leq \mathscr{C}_2 \leq \mathscr{C}_1 \tag{4.13}$$

With $P = 1$, (only one computing thread, no parallel resources available), we clearly see that $\mathscr{C}_1 = \mathscr{C}_3$. Additionally, from Equation (4.7), we have that $\kappa(\mathbf{A}) \geq \max_B(\kappa(A_n))$. Therefore, for $P = 1$, $\mathscr{C}_2 \geq \mathscr{C}_1 = \mathscr{C}_3$.

□

Therefore, among the three strategies, batched methods are well-suited for the solution of multiple independent linear systems.

4.2 A flexible and composable design

Software sustainability requires software that can run on and utilize different hardware architectures, not only of the present but of the future. This requires careful attention to the design of the sustainable and maintainable software. The software design of the batched iterative solvers and preconditioners was based on the principles of

1. High performance, with kernels tuned for efficient execution on multiple GPU architectures,
2. Composability, with the ability to efficiently compose different solvers, preconditioners, stopping criteria, and loggers,
3. User-friendliness, enabling users to obtain maximum performance without sacrificing code readability, and
4. Extensibility, enabling developers to add functionality easily with a stable interface, providing the necessary building blocks.

4.2.1 The BatchLinOp hierarchy

Batched solvers and preconditioners can be expressed as batches of linear operators. This abstraction allows us to combine the expressibility provided by the **`LinOp`** concept, introduced in Chapter 3 with the high performance of the particular data structures required for batched operators.

Figure 4.1 shows the class hierarchy with the **`BatchLinOp`** concept. **`BatchLinOp`** contains the **`apply`** method, which the concrete operators, such as the batch solvers, preconditioners, or matrix formats, need to implement. This concept allows us to compose and combine different **`BatchLinOp`** s through additional classes such as **`Combination`** or **`Composition`**. Functions that are common to linear operators, such as application to another linear operator (**`apply`**), transposition (**`transpose`**) and conjugate transposition (**`conj_transpose`**) are defined in the **`BatchLinOp`** class, which all of the derived classes, such as solvers, preconditioners and matrix formats need to implement.

The **`BatchLinOp`** class also stores the dimensions of the batched linear operator. An object of type **`batch_dim<dimension>`** stores the dimensions of the individual entry linear operators. To optimize storage and dimension verification, this class specializes when all the batch entries have identical dimensions.

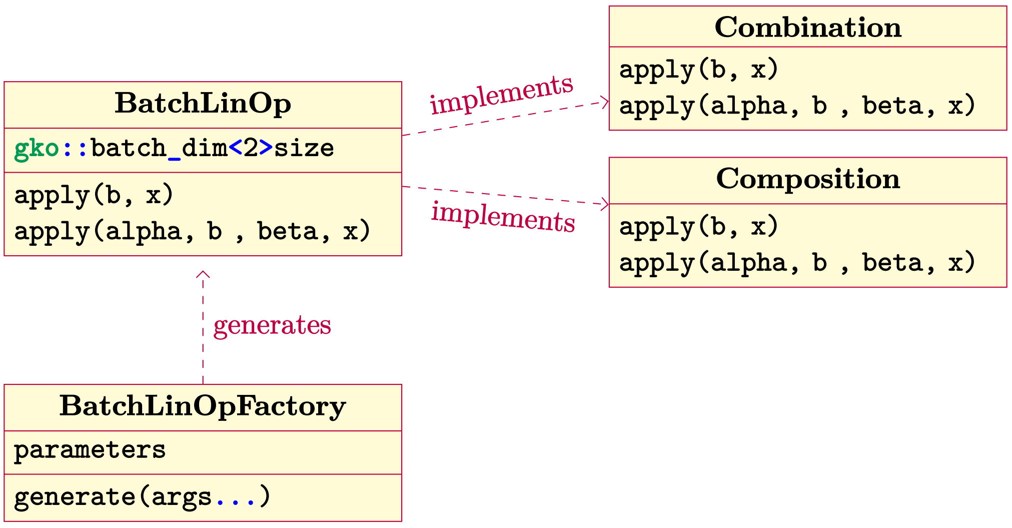


Figure 4.1: The **BatchLinOp** hierarchy and its salient members

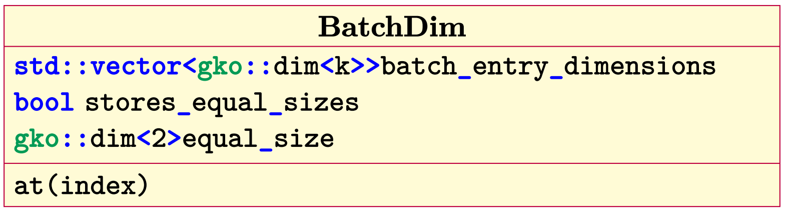


Figure 4.2: The **batch_dim** class and its salient members

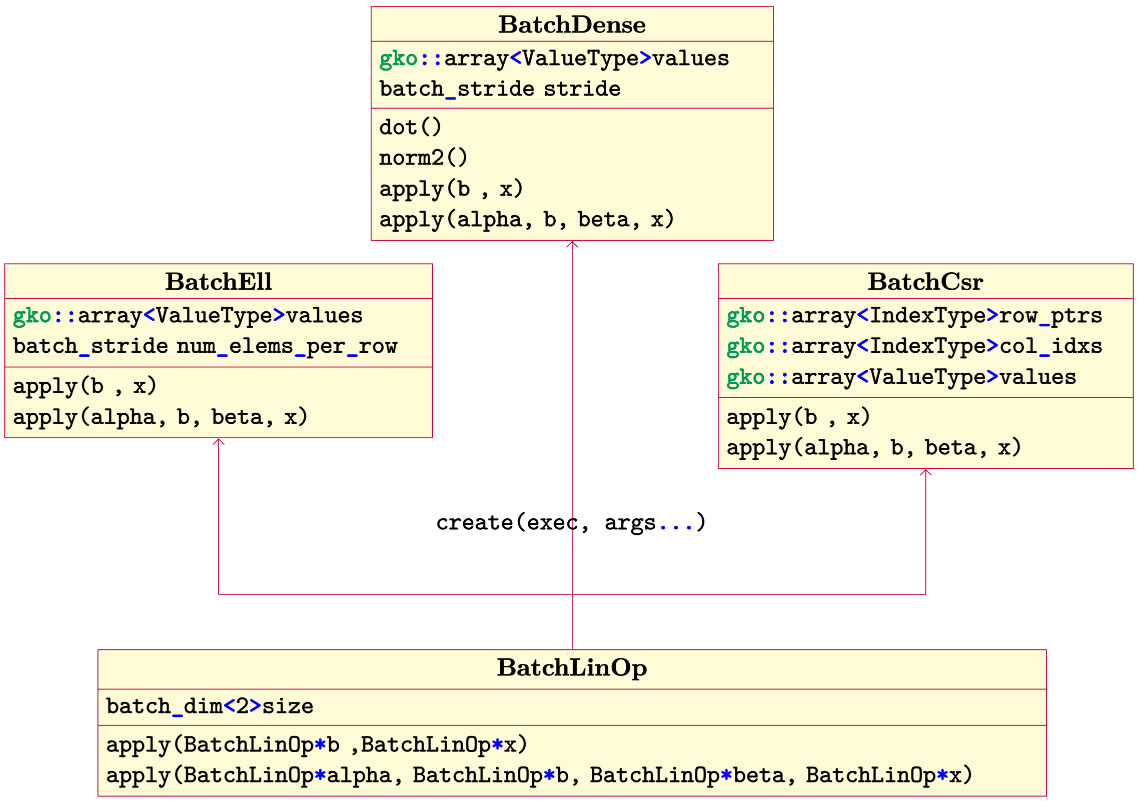
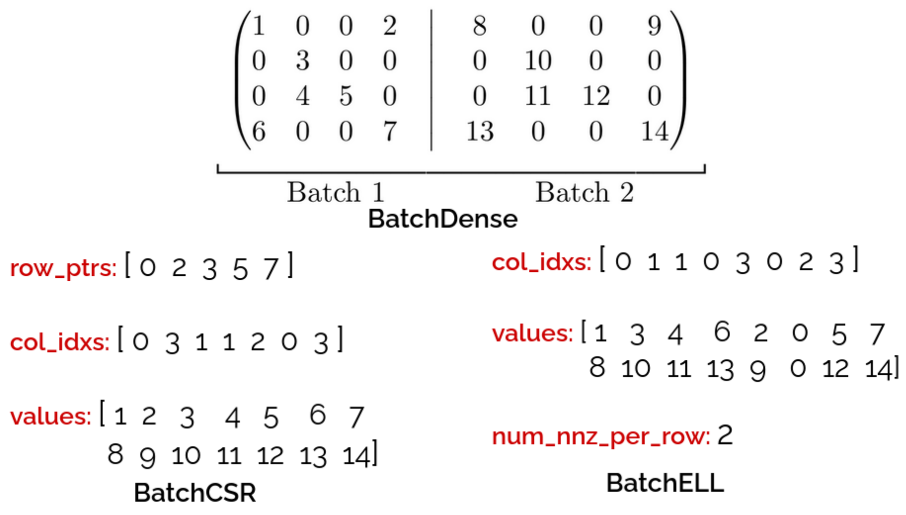

BatchDense
gko::array<ValueType>values
batch_stride stride
dot()
norm2()
apply(b , x)
apply(alpha, b, beta, x)
BatchEll
gko::array<ValueType>values
batch_stride num_elems_per_row
apply(b , x)
apply(alpha, b, beta, x)
BatchCsr
gko::array<IndexType>row_ptrs
gko::array<IndexType>col_idxs
gko::array<ValueType>values
apply(b , x)
apply(alpha, b, beta, x)
create(exec, args...)
BatchLinOp
batch_dim<2>size
apply(BatchLinOp*b ,BatchLinOp*x)
apply(BatchLinOp*alpha, BatchLinOp*b, BatchLinOp*beta, BatchLinOp*x)

Figure 4.3: Batched matrix formats and their salient members



$\left(\begin{array}{cccc|cccc} 1 & 0 & 0 & 2 & 8 & 0 & 0 & 9 \\ 0 & 3 & 0 & 0 & 0 & 10 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 & 11 & 12 & 0 \\ 6 & 0 & 0 & 7 & 13 & 0 & 0 & 14 \end{array}\right)$
Batch 1
Batch 2
BatchDense
row_ptrs: [0 2 3 5 7]
col_idxs: [0 3 1 1 2 0 3]
values: [1 2 3 4 5 6 7 8 9 10 11 12 13 14]
BatchCSR
col_idxs: [0 1 1 0 3 0 2 3]
values: [1 3 4 6 2 0 5 7 8 10 11 13 9 0 12 14]
num_nnz_per_row: 2
BatchELL

Figure 4.4: Storage scheme for an example sparse matrix with the three batch matrix formats, **`BatchDense`**, **`BatchCsr`**, **`BatchEll`**

Table 4.1: Storage requirements for general Batch dense, Batch Csr and Batch Ell matrix formats.

Matrix Format	*Memory requirements*	*Storage layout*
Batch dense	$\sum_{k=1}^{K}(n_{rows}^{k} \times n_{cols}^{k})$	Row-major
Batch Csr	$\sum_{k=1}^{K}(2n_{nz}^{k} + n_{rows}^{k} + 1)$	Row-major
Batch Ell	$\sum_{k=1}^{K}(2n_{rows}^{k} \times n_{nz}^{k,row})$	Column-major

4.3 Efficient access through optimized data structures

To maximize performance, it is crucial to store objects and their underlying data so that their accesses are coalesced: parallel work-items read data contiguous in memory. It is also necessary to minimize memory movement by caching frequently used data to have as few cache misses as possible. To this end, we design custom data structures for the batched sparse iterative solvers.

Many of the applications we target, as elaborated in Section 4.6, consists of hundreds of thousands of linear systems sharing a sparsity pattern. Given the shared sparsity pattern, we have designed specialized matrix storage formats to minimize the storage requirements and allow for maximum cache usage. Given K batch entries with each matrix entry of size $n \times n$ and storing n_{nz} nonzeros, Table 4.1 summarizes the memory and the storage layout we use for general matrices without sharing a sparsity pattern. Table 4.2 shows the storage requirements and the memory savings obtained with our custom matrix storage formats. The **`BatchDense`** format stores the batch entries consecutively in memory, storing each batch entry in a row-major fashion. Vectors also use this format, enabling operation with multiple right-hand sides if necessary. The **`BatchCsr`** matrix format stores one copy of the sparsity pattern, with the row pointers and the column indices, and the values are then stored in a row-major fashion by batches. The **`BatchEll`** format stores a constant number of nonzeros per row, storing some explicit zeros if necessary. The column index array and the number of nonzeros per row describe the common sparsity pattern. It is stored once, with the values of the different batch entries being stored consecutively and in a column-major fashion to improve coalesced accesses.

Table 4.2: Saving memory with a shared sparsity pattern for **BatchDense**, **BatchEll** and **BatchCsr** for square matrices ($n_{rows} = n_{cols} = n$).

Matrix Format	*Memory requirements*	*Approximate savings w.r.t* **BatchDense**
BatchDense	Kn^2	–
BatchCsr	$(K+1)(n_{nz}+n+1)$	$\frac{n_{nz}}{n^2}$
BatchEll	$(K+1)(n \times n_{nz}^{row})$	$\frac{n \times n_{nz}^{row}}{n^2}$

4.3.1 Computational, memory and cache complexity for batched matrix formats.

We consider the matrix vector product that forms the workhorse of Krylov subspace solvers. The algorithm for a general sparse matrix vector product(SpMV) is shown in Algorithm 2.

Algorithm 2 The general sparse matrix vector product

```
1: INPUT: $A \in \mathbb{C}^{\sum_{k=1}^{K}(n_{rows}^k \times n_{cols}^k)}, x \in \mathbb{C}^{\sum_{k=1}^{K}(n_{rows}^k \times 1)}, b \in \mathbb{C}^{\sum_{k=1}^{K}(n_{rows}^k \times 1)}$
2: $K$ = A.get_num_batch_entries()
3: for $k$ < $K$ do
4:     $n_{rows}^k$ = A[k].get_size()[0]
5:     for row $\in (1, n_{rows}^k)$ do
6:         x[k][row] = A[k][row,($\forall$ cols $\in$ A[k][row,:] $\setminus$ {0} ))]
7:                         * b[k][:]
8:     end for
9: end for
```

Theorem 4.3.1 (RAM model: Batched SpMV)**.** *Consider a batched sparse matrix with K entries, with the k-th entry having n_{nz}^k nonzeros, and of size $(n_{rows}^k \times n_{cols}^k)$. Within the RAM model, the batched sparse matrix vector product has a computational complexity of $\mathcal{O}(\sum_{k=1}^{K} n_{nz}^k)$, and a memory complexity of $\mathcal{O}(\sum_{k=1}^{K}(2 \times n_{nz}^k + n_{rows}^k))$.*

Proof. From Algorithm 2, for the k-th batch entry, we see that we have for each row, operations equal to the number of nonzeros in that row, $\mathcal{O}(n_{nz}^{k,row})$. Therefore the computational complexity for one batch entry is equal to $\mathcal{O}(n_{nz}^{k,row} \times n_{rows}^k)$ and considering that we have K batch entries, the computational complexity is equal to $\mathcal{O}(\sum_{k=1}^{K} n_{nz}^k)$, with $n_{nz}^k = n_{nz}^{k,row} \times n_{rows}^k$.

To calculate the memory complexity, we look at the number of reads and writes the algorithm needs to perform. We assume that the reads and writes have the same cost, which might not be true on hardware. We also assume no caching and that the cache is flushed in each iteration, and data needs to be read again.

From Algorithm 2, we see each inner loop needs to read the b^k vector, and in n_{rows}^k iterations, we need to read all the nonzeros of the matrix. In addition, we write 1 value of x^k in each iteration, with a total of n_{rows}^k writes. Therefore, the overall memory complexity with a general RAM model is $\mathcal{O}(\sum_{k=1}^{K}(2 \times n_{nz}^k + n_{rows}^k))$. □

With the sparse matrix-vector operation being memory bound (due to its low arithmetic intensity), minimizing the number of I/O operations, such as reads from global memory, and maximizing the cache utilization is essential. To better understand the optimality of the different matrix formats and their suitability, we analyze the cache complexity of the different matrix storage layouts with the matrix-vector product algorithm. For simplicity, from now on, we will assume square matrices, $n_{rows} = n_{cols} = n$, but the following analysis can be easily extended to rectangular matrices.

Table 4.3: Nonzero and column retrieval functions for **BatchDense**, **BatchEll** and **BatchCsr** formats.

Matrix Format	*Retrieve nonzero*	*Retrieve column*

BatchCsr	`IndexType get_nnz_start(int row){`
	`  return get_row_ptrs()[row];`
	`}`
		`IndexType get_col(int nnz){`
		`  return get_col_idxs()[nnz];`
		`}`

BatchDense	`IndexType get_nnz_start(int row){`
	`  return num_cols*(row-1);`
	`}`
		`IndexType get_col(int nnz){`
		`  return nnz;`
		`}`

BatchEll	`IndexType get_nnz_start(int row){`
	`  return num_elems_per_row*(row-1);`
	`}`
		`IndexType get_col(int nnz){`
		`  return get_col_idxs()[nnz];`
		`}`

Additionally, we will restrict ourselves to the case of shared sparsity patterns between all batch entries. We want to note that minor variations in size and sparsity pattern between the batch entries can still be modeled with our approach by storing the sparsity pattern of the largest and densest batch entry. Algorithm 3 shows the algorithm used for the batched matrix-vector product. The nonzero and column retrieval functions are abstracted away and defined separately for each matrix format depending on their storage layout and are shown in Table 4.3.

Algorithm 3 The generalized batched matrix vector product

```
1: INPUT: $A \in \mathbb{C}^{(n\times n)\times K}, x \in \mathbb{C}^{(n\times 1)\times K}, b \in \mathbb{C}^{(n\times 1)\times K}$
2: for k < K do
3:     x ← x[k], values ← A[k].values, b ← b[k]
4:     for row ∈ (1, n) do
5:         temp ← 0
6:         for nnz ∈ (GET_NNZ_START(row), GET_NNZ_START(row+1)) do
7:             temp += values[nnz]*b[GET_COL(nnz)]
8:         end for
9:         x[row] ← temp
10:    end for
11: end for
```

Theorem 4.3.2 (Cache complexity: **BatchDense**). *For K batch entries, each with a size $n \times n$, utilizing the* **BatchDense** *matrix storage format for the matrix vector product, which has an operational complexity of $\mathcal{O}(Kn^2)$, the asymptotic worst-case cache complexity is*

$$\mathcal{Q}_{\texttt{BatchDense}}(n; Z, L) = \mathcal{O}(K(2n^2(\frac{L}{Z}) + 1))$$

Proof. Consider the ideal cache model (See Definition 2.54) with a tall cache assumption. From Algorithm 2, we have to access in each loop, one row of the matrix A^k and the complete column of b. Given a cache line size of L and a cache size of Z words, the cache complexity for each batch entry is $\mathcal{O}(2n^2(\frac{L}{Z}) + 1)$. ☐

Theorem 4.3.3 (Cache complexity: **BatchCsr**). *For K batch entries, each with a size of $n \times n$, utilizing the* **BatchCsr** *matrix storage format, which has an operational complexity of $\mathcal{O}(Kn_{nz})$, the asymptotic worst-case cache complexity is*

$$\mathcal{Q}_{\texttt{BatchCsr}}(n; Z, L) = \mathcal{O}(K(n(n_{nz} + 2)(\frac{L}{Z}) + 2))$$

Proof. Consider the ideal cache model (See Definition 2.54) with a tall cache assumption. From Algorithm 2, we have to access in each loop, n_{nz}^{row} nonzeros of the column-index vector and the values; $\frac{2n_{nz}^{row}}{L}$ loads. The index accesses of b are dependent on the values of the column indices and in the worst case we have to load $\frac{n\times n_{nz}^{row}}{L}$ with each loop loading $2L$ elements for the row pointers giving a worst case cache complexity for each batch entry of $\mathcal{O}(n(n_{nz}+2)(\frac{L}{Z})+2)$. ☐

Theorem 4.3.4 (Cache complexity: **BatchEll**). *For K batch entries, each with a size of $n \times n$, utilizing the* **BatchEll** *matrix storage format, which has an operational complexity of $\mathcal{O}(Knn_{nz}^{row})$, the asymptotic worst-case cache complexity is*

$$\mathcal{Q}_{\texttt{BatchEll}}(n; Z, L) = \mathcal{O}(Kn_{nz}^{row}(n^2 + 2n)(\frac{L}{Z}))$$

Proof. Consider the ideal cache model (See Definition 2.54) with a tall cache assumption. From Algorithm 2, we have to access in each loop, n_{nz}^{row} nonzeros of the column-index vector and the values; $\frac{2n_{nz}^{row}}{L}$ loads. The index accesses of b are dependent on the values of the column indices and in the worst case we have to load $\frac{nn_{nz}^{row}}{L}$ giving a worst case cache complexity for each batch entry of $\mathcal{O}(n_{nz}^{row}(n^2 + 2n)(\frac{L}{Z}))$. □

Consider K batch entries, each with a size of $n \times n$ and n_{nz} nonzeros. Consider an architecture with an L1 cache and main memory. This L1 cache can also be split into a constant cache and a data cache. The constant cache holds read-only data that is not modified through the kernel run. The user can control this behavior. The data cache is a standard cache that can handle both read-only and read-and-write data. This is the case for the latest generation of GPUs. With these assumptions, the batched sparse matrix formats have the following cache optimality benefits.

Theorem 4.3.5 (Cache optimality of **`BatchCsr`** and **`BatchEll`**). *Assuming a shared sparsity pattern between different batch entries and a cache size $Z = \mathcal{O}(\mathcal{k} n_{nz})$ with some constant $\mathcal{k} > 1$, the **`BatchCsr`** and the **`BatchEll`** matrix storage format have fewer cache misses of the order of $\mathcal{O}(\frac{n(\mathcal{k}-1)}{n_{nz}^{row}(\mathcal{k})})$ compared to the the **`BatchDense`** format.*

Proof. Let us assume that all matrix formats have to load the same b and x vectors. The x vector needs to be written into and read from and, hence, needs to be in the L1 cache for all formats. The b vector is a read-only vector and hence can be cached in the constant data cache (available in the GPUs at the L1 level). For simplicity, let us assume that we prioritize the b vector as it has the potential for non-coalesced accesses. Therefore, the amount of memory available at the L1 level is $Z_{base} = Z - n$. The **`BatchDense`** format needs to load n elements in each loop iteration, with each loop iteration being new data. This implies $\mathcal{Q}_{dense} = n\frac{L}{Z_{base}}$ cache misses in each iteration.

For both the **`BatchCsr`** and the **`BatchEll`** format, assume that we cache the sparsity pattern in the constant cache, leaving an effective L1 cache size of $Z_{csr} = Z_{base} - n_{nz} - 2n$ and $Z_{ell} = Z_{base} - n - n_{nz}$, respectively. The **`BatchCsr`** format needs to fetch n_{nz}^{row} elements per iteration, giving $\mathcal{Q}_{csr} = n_{nz}^{row}\frac{L}{Z_{csr}}$ cache misses and the **`BatchEll`** format needs to fetch n_{nz}^{row} elements per iteration, giving $\mathcal{Q}_{ell} = n_{nz}^{row}\frac{L}{Z_{ell}}$ cache misses. Therefore, assuming that we need to do n iterations for K batch entries for all the three formats, we have a factor of

$$\frac{\mathcal{Q}_{dense}}{\mathcal{Q}_{csr}} = \frac{n}{n_{nz}^{row}}(\frac{Z_{csr}}{Z_{base}}) \approx \frac{n}{n_{nz}^{row}}\frac{Z_{base} - n_{nz} - 2n}{Z_{base}} = \frac{n}{n_{nz}^{row}}\frac{(\mathcal{k}-1)n_{nz} - 3n}{\mathcal{k}n_{nz} - n} \approx \frac{n(\mathcal{k}-1)}{n_{nz}^{row}(\mathcal{k})} > 1$$

fewer cache misses for **`BatchCsr`** and a factor of,

$$\frac{\mathcal{Q}_{dense}}{\mathcal{Q}_{ell}} \approx \frac{n}{n_{nz}^{row}}\frac{Z_{base} - n_{nz} - n}{Z_{base}} = \frac{n}{n_{nz}^{row}}\frac{(\mathcal{k}-1)n_{nz} - 2n}{\mathcal{k}n_{nz} - n} \approx \frac{n(\mathcal{k}-1)}{n_{nz}^{row}(\mathcal{k})} > 1$$

fewer cache misses for **`BatchEll`**. □

4.3.2 Optimizing the batched matrix-vector product

With the batched matrix-vector product being the workhorse of the batched iterative solvers, it is necessary to select a performance-optimal data format for the problem at hand. As we aim to support many different applications, with possibly a wide variety of sparsity patterns, we support three different matrix formats: **`BatchDense`**, **`BatchCsr`**, and **`BatchEll`**. The storage layouts of these matrix formats are designed for maximum cache utilization and parallel throughput for the matrix-vector product on many-core architectures.

The **`BatchCsr`** storage format is one of the most general sparse matrix formats, and we develop two parallel algorithms for the batched sparse matrix-vector product. With strategy 1, shown in Algorithm 4, we assign one work-item to one row of the matrix, updating each element of the x vector independently, in parallel. This strategy works well for matrices with a similar nonzero count across rows, where each work-item performs a similar amount of work.

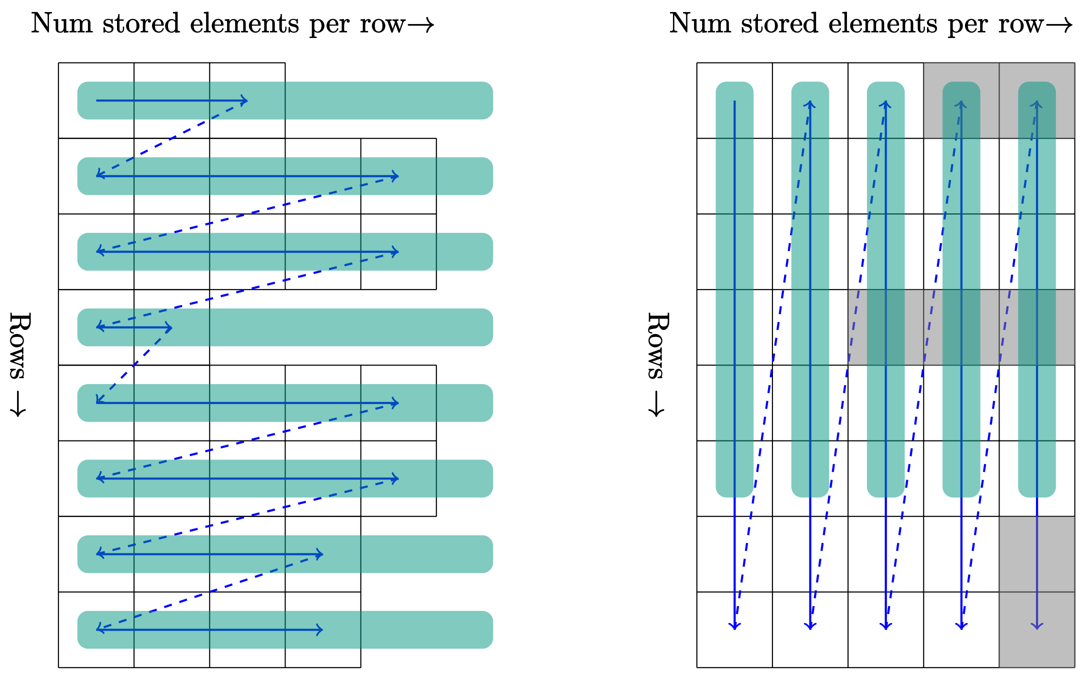
For more unbalanced matrices (more imbalance in nonzeros per row), assigning one subgroup to a row provides higher parallelism due to the larger nonzero count that more work-items can now handle. As each work-item inside a subgroup

Algorithm 4 The generalized parallel batched matrix vector product, strategy 1: thread per row.

```
INPUT: $A \in \mathbb{C}^{(n\times n)\times K}, x \in \mathbb{C}^{(n\times 1)\times K}, b \in \mathbb{C}^{(n\times 1)\times K}$
parallel for k < K do                                        ▷ Assign one workgroup to one batch entry
    x ← x[k], values ← A[k].values, b ← b[k]
    parallel for row ∈ (1, n) do                             ▷ Assign one work-item to one row
        temp ← 0
        for nnz ∈ (GET_NNZ_START(row), GET_NNZ_START(row+1)) do
            temp += values[nnz]*b[GET_COL(nnz)]
        end for
        x[row] ← temp
    end parallel for
end parallel for
```

Algorithm 5 The generalized parallel batched matrix vector product, strategy 2: subgroup per row.

```
INPUT: $A \in \mathbb{C}^{(n\times n)\times K}, x \in \mathbb{C}^{(n\times 1)\times K}, b \in \mathbb{C}^{(n\times 1)\times K}$
parallel for k < K do                                        ▷ Assign one workgroup to one batch entry
    x ← x[k], values ← A[k].values, b ← b[k]
    parallel for row ∈ (1, n) do                             ▷ Assign one subgroup to one row
        temp ← 0
        for nnz ∈ (GET_NNZ_START(row), GET_NNZ_START(row+1)) do
            temp += values[nnz]*b[GET_COL(nnz)]
        end for
        temp ← SUBGROUP_REDUCE(temp)
        x[row] ← ATOMIC_WRITE(temp)
    end parallel for
end parallel for
```



(a) CSR (b) ELL

Figure 4.5: Two possible layouts of the non-zero coefficients' array of a matrix with some arbitrary sparsity pattern. Green bars show how the warps are oriented for a subgroup length of 6.

works in parallel, we first need to reduce the partial results from the work-items and then have only one work-item write the corresponding reduced value to the x vector. This is shown in Algorithm 5.

The matrix-vector product for all three matrix formats can be expressed as shown in Algorithm 4 and Algorithm 5, with specialized non-zero and column retrieval functions as shown in Table 4.3.

The **BatchEll** matrix format is suitable for matrices with a similar number of nonzeros across rows (possibly non-uniform). This allows for strided access to the matrix values at the cost of storing some explicit zeros. Additionally, for the **BatchEll** matrix format, as shown in Figure 4.4, the values are stored in a column-major format, with strided access for each thread, with constant strides enabling better cache locality. This is showcased in Figure 4.5, where we compare the processing of each row with the two matrix storage formats. We observe that for the **BatchEll** format, each work-item has a balanced load with coalesced accesses, which can lead to better performance.

For matrices that are wholly or almost dense (very few zeros), sparse matrix formats such as **BatchCsr** and **BatchEll** provide no benefits; hence, we also provide a **BatchDense** format. This format stores the nonzeros in a contiguous manner in a row-major fashion, again with a constant stride, by default equal to the number of columns in the matrix. The **BatchDense** matrix format also has the advantage of coalesced accesses for the b vector, which can benefit smaller, dense batched matrices.

4.3.3 The workhorse: BatchCsr and BatchEll SpMV kernels

We want to avoid communication between thread blocks to reduce the data movement to and from the global memory. Therefore, we assign one thread block to solve one system. With the fine-grained parallelism in GPUs, each thread block should contain the number of threads proportional to the size of an individual linear system. We must tune our thread block sizes according to the problem size. However, based on the register usage by the kernel, there is a limit to how many threads can be used to solve one batch entry.

Figure 4.6 shows the relative speedup of the **BatchCsr** and the **BatchEll** matrix formats, against the standard **BatchDense** format. Seven different problems with varying matrix sizes and sparsity patterns have been considered. For

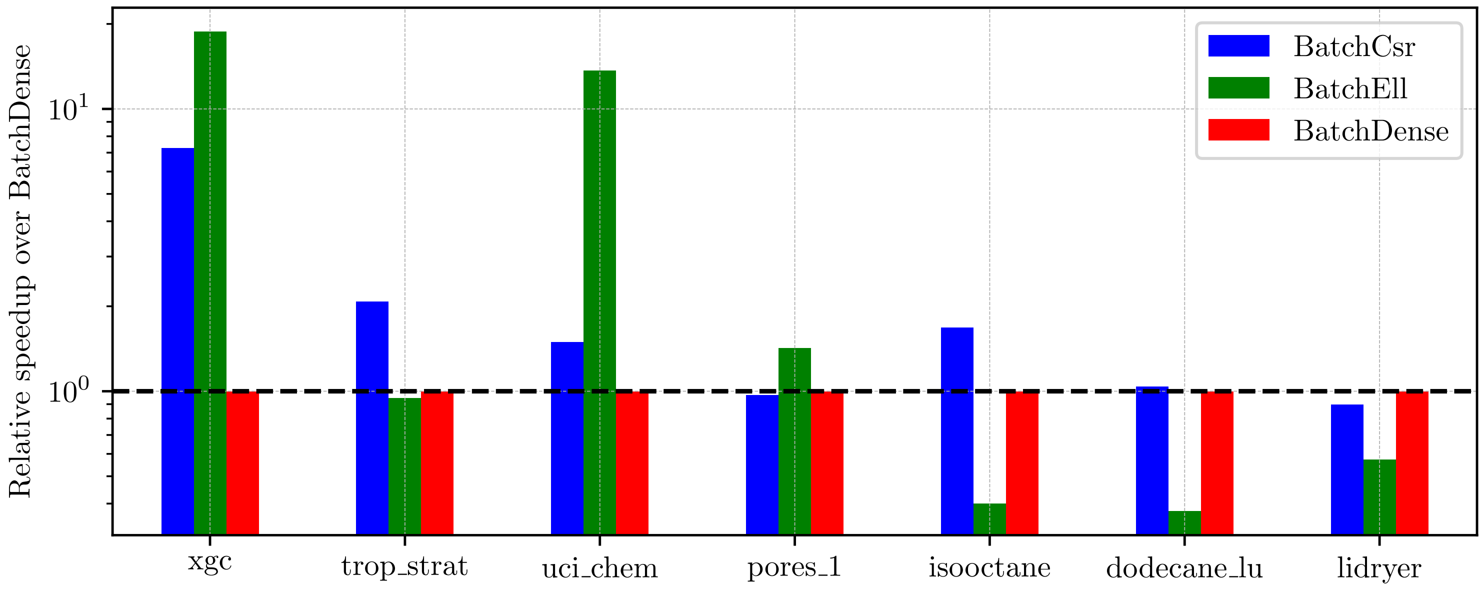


Figure 4.6: Relative speedup of **BatchCsr** and **BatchEll** SpMV compared to **BatchDense**.

Matrix name	**Size**	**Non-zeros (%)**
lidryer	10	91 (91%)
pores_1	30	180 (20%)
uci_chem	53	262 (9.3%)
dodecane_lu	54	2332 (80%)
trop_strat	130	1232 (7.2%)
isooctane	144	6135 (30%)
xgc	992	8554 (0.8%)

Table 4.4: Characteristics of matrices with their sizes and non-zero counts

very sparse matrices, **BatchCsr** is better than **BatchDense**. For matrices having a balanced number of nonzeros per row (xgc and uci_chem), the **BatchEll** format can outperform both **BatchCsr** and **BatchDense**. Finally, for mostly dense matrices, **BatchDense** can be slightly better than **BatchCsr**. Table 4.4 shows the corresponding matrix properties.

4.4 Implementing efficient iterative solver kernels

Generally, an iterative solver algorithm does not have a pre-determined execution order, and its execution, particularly its number of iterations, is problem-dependent. In addition, iterative solvers come with various options, enabling the user to tune the solver to the problem at hand. This multitude of options, in combination with the need for flexibility in choosing the matrix storage format, problem-independent stopping criteria, and preconditioners, make the design of iterative solvers challenging.

The batched iterative solver can be expressed as the solution of k independent linear systems, $A_i x_i = b_i$, $i = 1 \cdots k$, with the possibility of having a left-preconditioner M_i individually generated for each linear system.

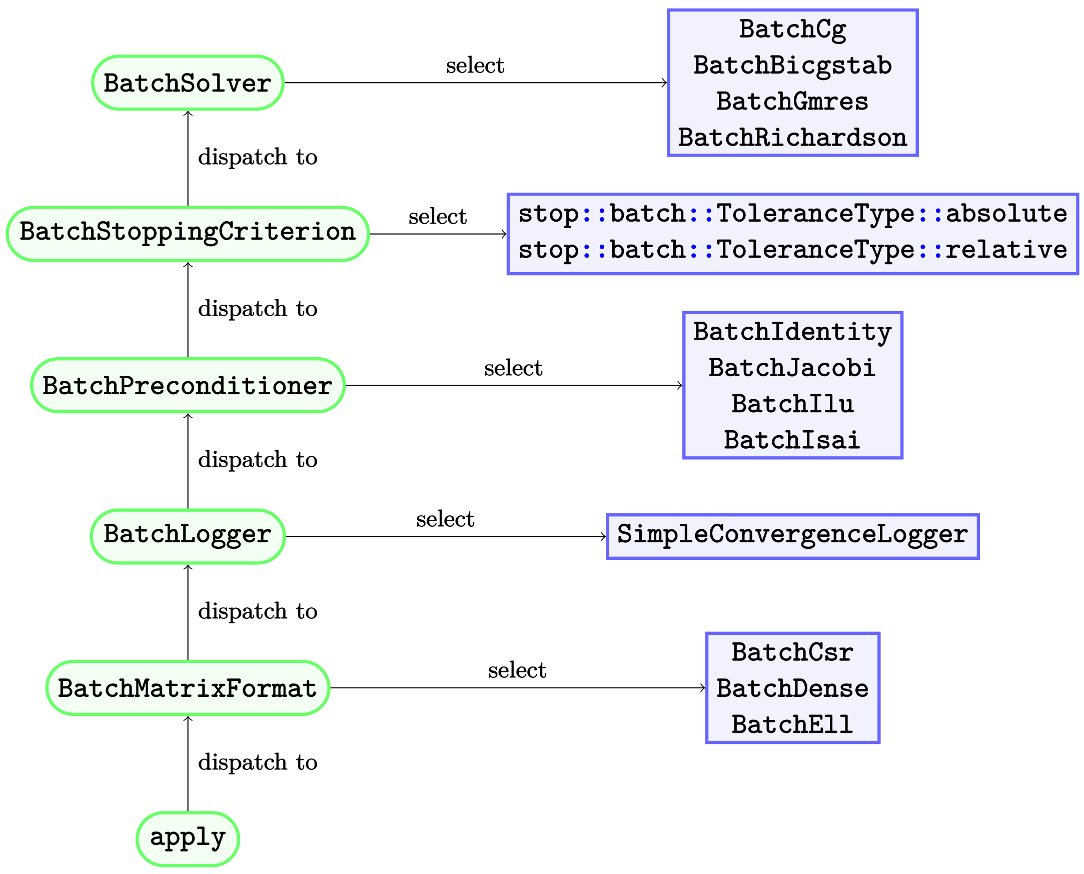


Figure 4.7: Multi-level dispatch mechanism for batched iterative solvers

4.4.1 The multi-level dispatch mechanism

Our goal is to provide applications with the ability to choose between different options, allowing problem-specific optimizations while minimizing the overhead for selection between these options. The explosion of parameters and features is handled with a multi-level dispatch mechanism. This mechanism, shown in Figure 4.7, goes through the different class hierarchies, selects the appropriate matrix format, the logger, stopping criterion, preconditioner, and finally, the solver. The solver application is templated on these classes, minimizing the runtime penalty for the solver call.

4.4.2 Batched solution of symmetric positive definite matrices

As we have seen in Definition 2.48, the conjugate gradient method is well-suited for solutions of symmetric positive definite (SPD) matrices. The **BatchCg** algorithm is shown in Algorithm 6. We loop over each batch entry and can independently compute the solution in each batch entry.

Theorem 4.4.1 (RAM model:Batched CG solver)**.** *The computational complexity of the batched conjugate gradient solver, given K batch entries, with linear systems of size $(n \times n)$ is*

$$\mathcal{W}_{\texttt{BatchCg}} = \mathcal{O}(\sum_{k=1}^{K}(l_k \times (n^2 + 5 \times n) + n^2 + n))$$

where l_k is the number of iterations the k-th batch entry needs to converge to some prescribed tolerance τ. Additionally, the memory complexity is

$$\mathcal{M}_{\texttt{BatchCg}} = \mathcal{O}(\sum_{k=1}^{K}(l_k \times (17 \times n + n^2) + 10 \times n + n^2))$$

Algorithm 6 The batched preconditioned conjugate gradient solver.

```
1: $A \leftarrow A[k], b \leftarrow b[k], x \leftarrow x[k]$
2: for each batch entry do
3:     $r \leftarrow b - Ax, z \leftarrow W^{-1}r, p \leftarrow z, t \leftarrow \mathbf{0}$
4:     $\rho \leftarrow r \cdot z, \alpha \leftarrow 1, \hat{\rho} \leftarrow 1$
5:     for $i < N_{iter}$ do
6:         if $|\rho| < \tau$ then
7:             break
8:         end if
9:         $t \leftarrow Ap$
10:        $\alpha \leftarrow \frac{\rho}{p \cdot t}$
11:        $x \leftarrow x + \alpha p$
12:        $r \leftarrow r - \alpha t$
13:        $z \leftarrow$ PRECOND($r$)
14:        $\hat{\rho} \leftarrow r \cdot z$
15:        $p \leftarrow z + \frac{\hat{\rho}}{\rho} \cdot p$
16:        $\rho \leftarrow \hat{\rho}$
17:    end for
18: end for
```

Proof. We consider the computational complexity of one iteration of the CG solver. We need to perform 2 dot products and 3 vector updates which are a total of $5n$ operations. In addition we have one matrix-vector product with n^2 operations. Additionally, during initialization, we compute the residual, which involves a matrix-vector product and a vector update, with $n^2 + n$ operations. Assuming that the k-th batch entry takes l_k number of iterations, the total computational cost for the k-th batch entry is $\mathcal{O}(l_k \times (n^2 + 5 \times n) + n^2 + n)$. Hence, the computational cost of K batch entries is $\mathcal{O}(\sum_{k=1}^{K}(l_k \times (n^2 + 5 \times n) + n^2 + n))$

Similarly, in each iteration of the conjugate gradient solver, we need to read 12 vectors of size n, one matrix of size n^2 and write 5 vectors of size n to memory, incurring a memory volume of $n^2 + 17n$. Assuming l_k iterations as before and the need to read 7 vectors of size n, one matrix of size n^2 from memory and write 3 vectors of size n to memory, the overall memory complexity for K batch entries is $\mathcal{O}(\sum_{k=1}^{K}(l_k \times (17 \times n + n^2) + 10 \times n + n^2))$. □

4.4.3 Batched solution of general matrices

For solution of general matrices, we implement two iterative methods, the BiCGSTAB method and the GMRES method as defined in Section 2.2.2.1. Their algorithms are shown in Algorithm 7 and Algorithm 8 respectively. We additionally implement a relaxation method, Richardson as shown in Algorithm 9.

Theorem 4.4.2 (RAM model:Batched BiCGSTAB solver)**.** *The computational complexity of the batched bi-conjugate gradient(stabilized) solver, given K batch entries, with linear systems of size $(n \times n)$ is*

$$\mathcal{W}_{\texttt{BatchBicgstab}} = \mathcal{O}(\sum_{k=1}^{K}(l_k \times (2 \times n^2 + 11 \times n) + n^2 + n))$$

where l_k is the number of iterations the k-th batch entry needs to converge to some prescribed tolerance τ. Additionally, the memory complexity is

$$\mathcal{M}_{\texttt{BatchBicgstab}} = \mathcal{O}(\sum_{k=1}^{K}(l_k \times (31 \times n + 2 \times n^2) + 7 \times n + n^2))$$

Proof. Can be deduced in a similar fashion to Theorem 4.4.1 and from Algorithm 7. □

Algorithm 7 The batched preconditioned bi-conjugate gradient(stabilized), BiCGSTAB solver.

```
1: $A \leftarrow A[k], b \leftarrow b[k], x \leftarrow x[k]$
2: for each batch entry do
3:     $r \leftarrow b - Ax, \hat{r} \leftarrow r, p \leftarrow \mathbf{0}, v \leftarrow \mathbf{0}$
4:     $\rho' \leftarrow 1, \omega \leftarrow 1, \alpha \leftarrow 1$
5:     for $i < N_{iter}$ do
6:         if $\|r\| < \tau$ then
7:             break
8:         end if
9:         $\rho \leftarrow r \cdot r'$
10:        $\beta \leftarrow \frac{\rho' \alpha}{\rho \omega}$
11:        $p \leftarrow r + \beta(p - \omega v)$
12:        $\hat{p} \leftarrow$ PRECOND($p$)
13:        $v \leftarrow A\hat{p}$
14:        $\alpha \leftarrow \frac{\rho'}{\hat{r} \cdot v}$
15:        $s \leftarrow r - \alpha v$
16:        if $\|s\| < \tau$ then
17:            $x \leftarrow x + \alpha\hat{p}$
18:            break
19:        end if
20:        $\hat{s} \leftarrow$PRECOND($s$)
21:        $t \leftarrow A\hat{s}$
22:        $\omega \leftarrow \frac{t \cdot s}{t \cdot t}$
23:        $x \leftarrow x + \alpha\hat{p} + \omega\hat{s}$
24:        $r \leftarrow s - \omega t$
25:        $\rho' \leftarrow \rho$
26:    end for
27: end for
```

Algorithm 8 The batched generalized minimal residual (with restarts), GMRES solver.

```
1: $A \leftarrow A[k], b \leftarrow b[k], x \leftarrow x[k]$
2: for each batch entry do
3:     $D \leftarrow max\_restart, r \leftarrow b - Ax, z \leftarrow W^{-1}r$
4:     $V \leftarrow \mathbf{0}, H \leftarrow \mathbf{0}, W \leftarrow \mathbf{0}$
5:     while $i < N_{iter}$ do
6:         if $\|r\| < \tau$ then
7:             break
8:         end if
9:         $V(:,0) \leftarrow z/\|z\|, s \leftarrow \mathbf{0}, s[0] = \|z\|$
10:        for $d < D$ do
11:            ARNOLDI($A, V, H, W, s$)
12:            if $|s[d]| < \tau$ then
13:                $y \leftarrow$ SOLVE($H(0:D-1, 0:D-1), s(0:D-1)$)
14:                $x = x + V(:, 0:D-1) \times y$
15:                break and break
16:            end if
17:        end for
18:        $y \leftarrow$ SOLVE($H(0:D-1, 0:D-1), s(0:D-1)$)
19:        $x = x + V(:, 0:D-1) \times y$
20:        $r \leftarrow b - A \times x$
21:        $z \leftarrow$ PRECOND($r$)
22:    end while
23: end for
```

Algorithm 9 The batched Richardson method

```
1: $A \leftarrow A[k], b \leftarrow b[k], x \leftarrow x[k], \omega$
2: for each batch entry do
3:     $r \leftarrow b - Ax, dx \leftarrow \mathbf{0}$
4:     for $i < N_{iter}$ do
5:         $r \leftarrow b - Ax$
6:         if $\|r\| < \tau$ then
7:             break
8:         end if
9:         $dx \leftarrow$ PRECOND($r$)
10:        $x \leftarrow x + \omega dx$
11:    end for
12: end for
```

Theorem 4.4.3 (RAM model: Batched Richardson solver)**.** *The computational complexity of the batched Richardson solver, given K batch entries, with linear systems of size $(n \times n)$ is*

$$\mathcal{W}_{\texttt{BatchRichardson}} = \mathcal{O}(\sum_{k=1}^{K}(l_k \times (n^2 + 4n) + n^2 + n))$$

The memory complexity is

$$\mathcal{M}_{\texttt{BatchRichardson}} = \mathcal{O}(\sum_{k=1}^{K}(l_k \times (n^2 + 8n) + n^2 + 3n))$$

Proof. From Algorithm 9, we see that each iteration has to compute a matrix vector product, 2 vector updates and a norm computation giving us a total of $\mathcal{O}(\sum_{k=1}^{K}(l_k \times (n^2+4n)+n^2+n))$ operations for K linear system solves with each having l_k iterations. Additionally, in each iteration we need to read 5 vectors and write back 3 vectors to memory and read the matrix from memory, giving a total memory complexity of $\mathcal{O}(\sum_{k=1}^{K}(l_k \times (n^2+8n)+n^2+3n))$ for the batched Richardson solver.

□

Theorem 4.4.4 (RAM model: Batched GMRES solver)**.** *The computational complexity of the batched GMRES solver, given K batch entries, with linear systems of size $(n \times n)$ and a maximum restart of D is*

$$\mathcal{M}_{\texttt{BatchGmres}} = \mathcal{O}(\sum_{k=1}^{K}(l_k \times (\sum_{d=1}^{D}(2d^2 + 2n + (n+4)d + 4D) + 3n + n^2 + \frac{D^3 + 4D^2}{3})) + n^2 + n)$$

where l_k is the number of iterations the k-th batch entry needs to converge to some prescribed tolerance τ. Additionally, the memory complexity is

$$\mathcal{M}_{\texttt{BatchGmres}} = \mathcal{O}(\sum_{k=1}^{K}(l_k \times (\sum_{d=1}^{D}(2d^2 + 2n + (2n+8)d + 6D) + 5n + (2n+3)D + n^2 + \frac{D^2 + D}{2})) \\ + n^2 + 3n + (n + 2 + D)D)$$

Proof. The Arnoldi iteration(See Definition 2.45) has a computational complexity of $\mathcal{O}(2d^2 + 2n + (n+4)d + 4D)$ and a memory complexity of $\mathcal{O}(2d^2 + 2n + (2n+8)d + 6D)$. The triangular solve has a computational complexity of $\mathcal{O}(\frac{D^3+D^2}{3})$ and a memory complexity of $\mathcal{O}(\frac{D^2+D}{2})$. Following a similar process as from Theorem 4.4.1 for the vector and matrix vector operations with Algorithm 8, the overall computational and memory complexity of the restarted GMRES solver can be computed to be $\mathcal{O}(\sum_{k=1}^{K}(l_k \times (\sum_{d=1}^{D}(2d^2 + 2n + (n+4)d + 4D) + 3n + n^2 + \frac{D^3+4D^2}{3})) + n^2 + n)$ and $\mathcal{O}(\sum_{k=1}^{K}(l_k \times (\sum_{d=1}^{D}(2d^2+2n+(2n+8)d+6D)+5n+(2n+3)D+n^2+\frac{D^2+D}{2}))+n^2+3n+(n+2+D)D)$ respectively. □

4.4.4 Enhancing convergence with a scalar Jacobi preconditioner

As we have seen in Definition 2.42, preconditioners can accelerate the convergence of an iterative method. With our multi-level dispatch mechanism, as shown in Figure 4.7, we provide the user with an option to specify a preconditioner. An effective preconditioner must be cheap to generate and apply while reducing the working condition number of the system matrix.

We elaborate on the different preconditioners we provide in Chapter 5, but for the applications we consider here, we showcase a scalar Jacobi preconditioner, which is a diagonal preconditioner, where the diagonal of the system matrix is extracted, and the inverse of the diagonal is applied in the preconditioner apply step. It can hence be seen that the cost of the preconditioner generation is minimal, $\mathcal{O}(n_{rows})$, and the apply, where one needs to apply a diagonal matrix to a vector, is also cheap with a computational cost of $\mathcal{O}(n)$. The scalar Jacobi preconditioner is a very effective preconditioner for diagonal-dominant matrices and matrices with large diagonal elements.

4.4.5 System independent solver convergence and logging

A particular advantage of the iterative solvers is the ability to control the solution quality. Batched iterative solvers that monitor solver convergence on an individual system level can allow the distinct solver invocations to run differing iteration counts for the distinct problems. This system-individual convergence is crucial in two main aspects: 1) To maximize occupancy of the compute device and not perform any wasteful computations, and, 2) To prevent solution and solver degradation by over-iteration of the solution. For ensemble approaches for which batched solvers are used, performing a global ensemble solution is non-optimal [LBP+20] (See Theorem 4.1.1). This is mainly because a global solution needs to resolve the union of the eigenvalues of all the systems while the system-individual batched solvers have to resolve only the eigenvalues of its system.

To enable this system-independent convergence and to be able to monitor and log the convergence at this level, we provide the user with two stopping criteria; one based on the relative residual norm and the other based on the absolute residual norm. In addition, a simple convergence logger logs the number of iterations required to converge and the final residual norm for each batch system.

4.4.6 Reducing kernel latency and maximizing bandwidth

An essential aspect for obtaining optimal performance for the solution of hundreds of thousands of small linear systems is minimizing latency and the kernel launch overhead. On GPUs, which require the presence of a host for kernel launch orchestration, if we take the strategy of launching one kernel for the solution of every linear system in the batch, we incur the latency of the launch, which involves transferring some data and objects from the host memory to the GPU memory and communication between the two runtimes. When solving hundreds of thousands of these linear systems, this kernel launch latency is prohibitive, making it extremely important to alleviate these latencies. To this end, we use only one kernel launch to solve all the batches while still having system-independent convergence.

A by-product of the single kernel launch is the maximization of bandwidth. Due to a single kernel launch, the GPU runtime does not need to keep transferring data from the host to the device and vice versa. Additionally, GPU runtimes generally only flush caches after the completion of a kernel, and hence the larger caches, such as the L2 and L3, where the constant read-only data is generally cached, is not flushed, providing us with a much higher cache hit rate.

4.4.7 Maximizing cache locality

Batched iterative solvers, like their monolithic counterparts, are memory-bound (due to the low arithmetic intensity of the operations). Minimizing memory movement and maximizing cache utilization is crucial for obtaining good performance across all compute architectures. Therefore, with highly hierarchical architectures such as GPUs, it is essential to provide contextual information to the compiler regarding which objects and data to cache at which level of its memory hierarchy.

With the knowledge of the algorithm and the hardware architecture, we can provide the compiler and the runtime with hints to maximize this cache utilization. We develop a preferential caching strategy to improve our performance.

From the memory complexity bounds, we know that the sparse matrix-vector product (SpMV) forms the workhorse of the iterative solver process. Any vector involved with a SpMV, if not cached, must be fetched from global memory, greatly increasing the memory traffic. Any other vectors frequently read and written from must also be in the cache. Algorithm 10 shows this preferential caching strategy for the **`BatchCg`** solver. Vectors in **red** are 'intermediate vectors' involved in matrix-vector products; these are the most preferred to be allocated in the L1 cache,as SpMVs account for a large part of the batched solver execution time. Those in **blue** are other intermediate vectors that are allocated in the L1 cache only if space remains after all the **red** vectors have been allocated. **Green** indicates read-only constant data, which we would like to have cached in the L2 cache. We also implement similar strategies for both the **`BatchBicgstab`** and the **`BatchGmres`** version but skip their details for brevity.

Algorithm 10 Preferential caching strategy for the **BatchCg** solver

```
$A \leftarrow A[k], b \leftarrow b[k], x \leftarrow x[k]$
for each batch entry do
    $\boldsymbol{r} \leftarrow \boldsymbol{b} - \boldsymbol{Ax}, \boldsymbol{z} \leftarrow \boldsymbol{Mr}, \boldsymbol{p} \leftarrow \boldsymbol{z}, \boldsymbol{t} \leftarrow \boldsymbol{0}$
    $\rho \leftarrow \boldsymbol{r} \cdot \boldsymbol{z}, \alpha \leftarrow 1, \hat{\rho} \leftarrow 1$
    for $i < N_{iter}$ do
        if $|\rho| < \tau$ then
            break
        end if
        $\boldsymbol{t} \leftarrow \boldsymbol{Ap}$
        $\alpha \leftarrow \frac{\rho}{\boldsymbol{p} \cdot \boldsymbol{t}}$
        $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \boldsymbol{p}$
        $\boldsymbol{r} \leftarrow \boldsymbol{r} - \alpha \boldsymbol{t}$
        $\boldsymbol{z} \leftarrow$ PRECOND($\boldsymbol{r}$)
        $\hat{\rho} \leftarrow \boldsymbol{r} \cdot \boldsymbol{z}$
        $\boldsymbol{p} \leftarrow \boldsymbol{z} + \frac{\hat{\rho}}{\rho} \cdot \boldsymbol{p}$
        $\rho \leftarrow \hat{\rho}$
    end for
end for
```

4.4.8 Overcoming synchronization bottlenecks

The massive parallelism that many-core processors such as GPUs provide is generally organized hierarchically. We have already seen the parallel architecture of GPUs in Section 2.3. The GPU comprises multiple compute units, each having independent fast memory, which can be seen as a cache.

Due to their bulk-synchronous nature, iterative solvers generally require multiple global synchronizations. We need to solve multiple of these linear systems with batched iterative solvers. Our problem space consists of moderate-sized linear systems with the requirement to solve hundreds of thousands of these matrices in parallel. Therefore, we choose to always schedule one linear system on one work-group and tune the work-group size based on the problem size. This has multiple advantages:

1. Synchronization is restricted to the work-group level, thereby providing higher parallelism,
2. Enough parallelism available through subgroups to accelerate the building blocks of the iterative method,
3. The occupancy of the GPU can be maximized due to one work-group being scheduled on one compute unit, and the independent linear systems can be scheduled on all available compute units.

4.4.9 Maximizing compute utilization

To maximize performance, in addition to minimizing the memory traffic and reducing synchronization, maximizing the occupancy on the GPU is essential. The theoretical occupancy depends on three main aspects: 1) The number of registers being used per work-item, 2) the amount of shared memory being utilized per work-group, and 3) the size of the work-group.

To maximize theoretical occupancy, we should reduce the register usage, reduce the amount of local memory, and have a minimum size for the work-group to have enough subgroups available to be scheduled. But this goal conflicts with our previous goal to maximize cache utilization. Therefore, it is necessary to balance these aspects on a problem-by-problem basis. To this end, given the size and other parameters of the linear system and the iterative solver, we allocate only as much shared memory as necessary and aim to optimize our work-group sizes at runtime, given the problem at hand.

4.4.10 Minimizing host based orchestration

A typical workflow when using GPU for simulations is to transfer the data from host memory, launch the GPU kernel with the appropriate data, and transfer any data back to the host, if necessary. This approach is suitable for traditional serial and bulk-synchronous algorithms. However, for algorithms that aim to take advantage of the embarrassing parallelism, removing transfers to the host and having data reside only on the GPU without communicating with the host is better. This is particularly important for cases where the linear solver forms a step of the outer non-linear loop. Batched iterative solvers again provide some advantages here: 1) There is no need for host transfers and orchestration apart from the launching of the kernel, 2) All the steps, including the convergence checks, are performed on the GPU, making them completely host-communication free. This may not be the case for direct solvers, which require pre-processing of the matrix to improve the parallelism, and these pre-processing steps, due to their sequential nature, are better suited on the CPU.

4.4.11 Optimizing the runtime scheduling for the parallel workgroups

We need to enable the GPU to handle the perfect parallelism offered by the batched methods, which involves the solution of independent linear systems. The runtime scheduler has no information on the time to solution of the linear systems. Therefore, in the worst case, we schedule the most expensive system at the last, reducing the overall utilization of the GPU.

The runtime scheduler schedules systems sequentially in the batch, without synchronization, until all the compute units have been scheduled. Therefore, to prevent a worst-case scenario, we can order the systems in the batch from hardest (the ones that take most iterations) to easiest, ensuring that harder systems are scheduled first. This ordering requires an estimate of the computational cost of the solution of each of the systems, which can be obtained either by a condition number estimate or by performing an initial solve and recording the number of iterations for convergence.

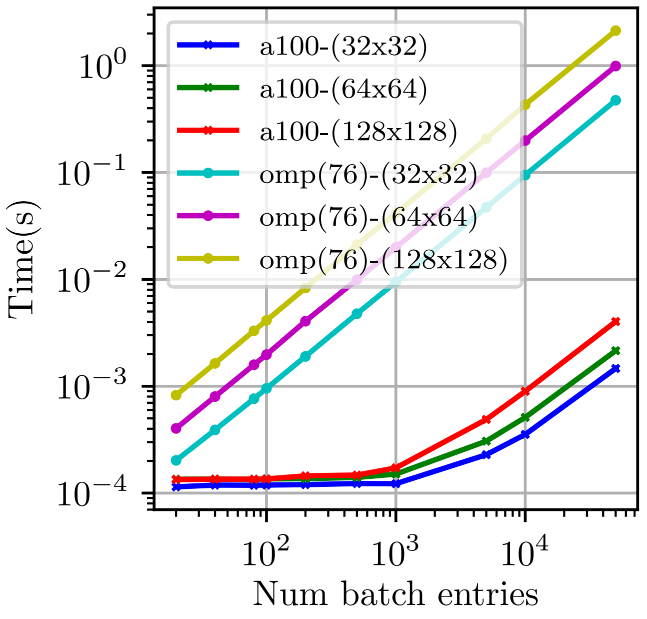
From a practical standpoint, this reordering can be expensive to compute. In addition to an initial estimate of the computational cost, reordering may involve reallocation of memory of the objects involved in the batched solution, which usually involves host orchestration, which can significantly increase the overhead and hence the overall time to solution.

4.5 Analyzing the behaviour and performance of the batched iterative solvers

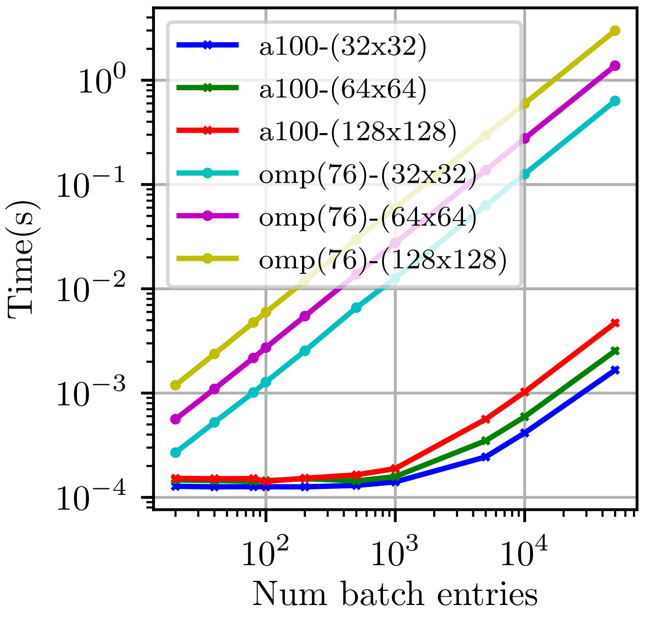
In this section, we showcase the performance of the batched iterative solvers on different architectures.

4.5.1 Evaluating performance: The 3-point stencil Laplacian

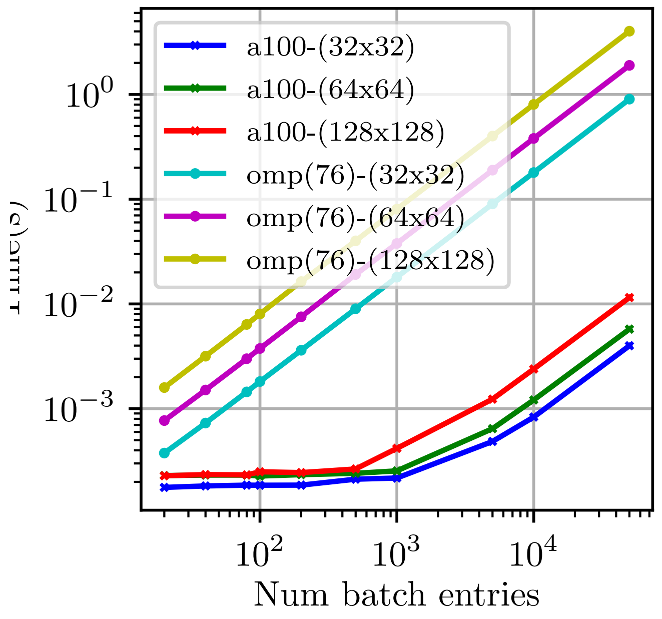
To benchmark the performance of the batched iterative solvers, we consider a matrix derived from the stencil used to discretize the 1D Laplacian problem. We generate the matrix for different grid sizes, obtaining matrices of sizes 32×32, 64×64, and 128×128. The solvers do not use any preconditioners, except the Richardson solver, which uses a scalar Jacobi preconditioner as an inner solver to guarantee its convergence. All solvers are run until convergence to a tolerance of 1e-6. In Figure 4.8, we show the different batched iterative solvers' performance compared on two different architectures: the A100 GPU and the Intel Xeon with 76 cores. We see clear benefits of the batched methods on GPUs. We can hide the latency until almost 1000 batch entries for all solvers and matrix sizes. Table 4.5 shows the achieved metrics for cache hit rates, memory throughput, and occupancy for the different batched solvers. As expected, we see that **`BatchCg`** is the most suitable for this SPD matrix. We have a very high L1 hit rate due to the aforementioned preferential caching strategy. By reducing our register usage and using an optimal block size, we also maximize the occupancy of our GPU and are very close to the theoretical occupancy limit, showing that the implementation does not have many subgroup stalls.



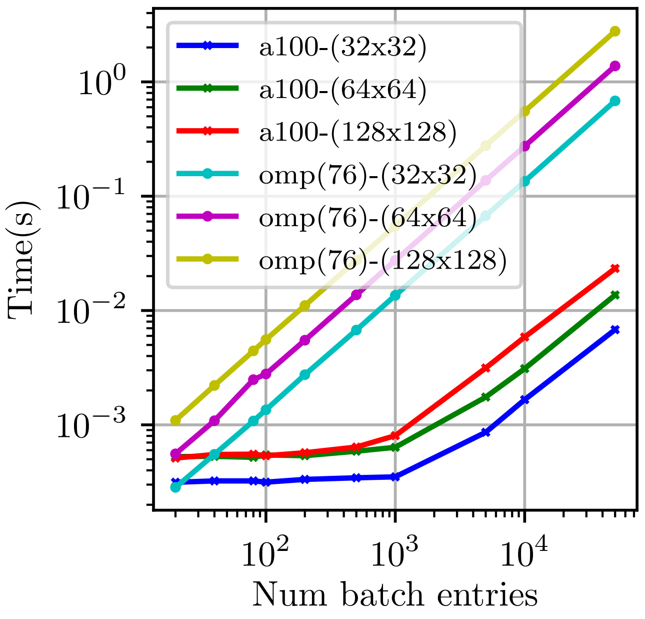
(a) **BatchCg**



(b) **BatchBicgstab**



(c) **BatchGmres**



(d) **BatchRichardson** with Jacobi

Figure 4.8: Time to solution for a 3 point stencil, with A100 GPU and on Intel Xeon(76 cores) CPUs

Table 4.5: Cache and compute usage for the different solvers on NVIDIA A100 with the 3pt stencil problem.

Solver	*L1 hit rate (%)*	*L2 hit rate (%)*	*Performance (TFlops)*	*Theoretical/Achieved occupancy (%)*
`BatchCg`	97.5	50.5	1.92	75/68
`BatchBicgstab`	97.9	51.2	1.35	75/68
`BatchGmres`	97.8	51.5	0.723	50/46.6
`BatchRichardson`	99.2	86.8	0.939	75/67

4.5.2 Evaluating performance: SuiteSparse matrix collection

To evaluate performance for general matrices, we use the SuiteSparse matrix collection [DH11]. This matrix collection hosts matrices contributed by different applications, giving us a wide variety of matrix properties to benchmark our solvers on. The solvers do not use any preconditioners. Each solver is run until convergence to a tolerance of 1e-6 on three different architectures, the NVIDIA A100, the AMD MI250X, and an Intel Xeon CPU (with 76 cores) with 5000 batch entries to occupy the hardware fully. From Figure 4.9, we see that GPUs outperform the CPUs for all solvers. Between the two GPUs, for almost all cases, the NVIDIA A100 performs better than the MI250X due to the larger L1 + shared memory available, which is crucial for good performance. Solvers that did not converge for certain matrices have missing bars in Figure 4.9.

The **`BatchGmres`** solver has an additional restart parameter that specifies the number of subspaces to store and orthogonalize against. A larger restart implies more storage and more pressure on the memory due to the need to store it in the L1 cache, while a smaller restart can hinder convergence. Figure 4.10 shows the effects of using three different restart parameters and the necessity of choosing the correct restart parameter for the problem at hand. We see that a restart parameter of 50 is robust but can also increase the overall time to solution. For most problems, a restart of 10 outperforms the other two but can sometimes lead to solver divergence or instability, which is signified by missing bars in Figure 4.10.

4.5.3 Comparing effectiveness of the different batched iterative solvers

Choosing the most effective solver for the problem at hand is also essential. To this end, we compare the three batched iterative solvers on problems where all of them can converge (SPD matrices). Figure 4.11 shows both the iteration count and the time per iteration required for the three batched iterative solvers to converge to a final residual error of 1e-6. We observe that **`BatchGmres`** is the most robust (all solvers converge here, and missing bars indicate convergence in 1 or 2 iterations). The **`BatchCg`** solver is the cheapest with the lowest time per iteration. In contrast, the **`BatchGmres`** is more expensive but can converge in fewer iterations, and the total time to solution is problem-dependent. We note that as we consider no preconditioners here, while all matrices may not converge, none diverge.

4.6 Accelerating applications

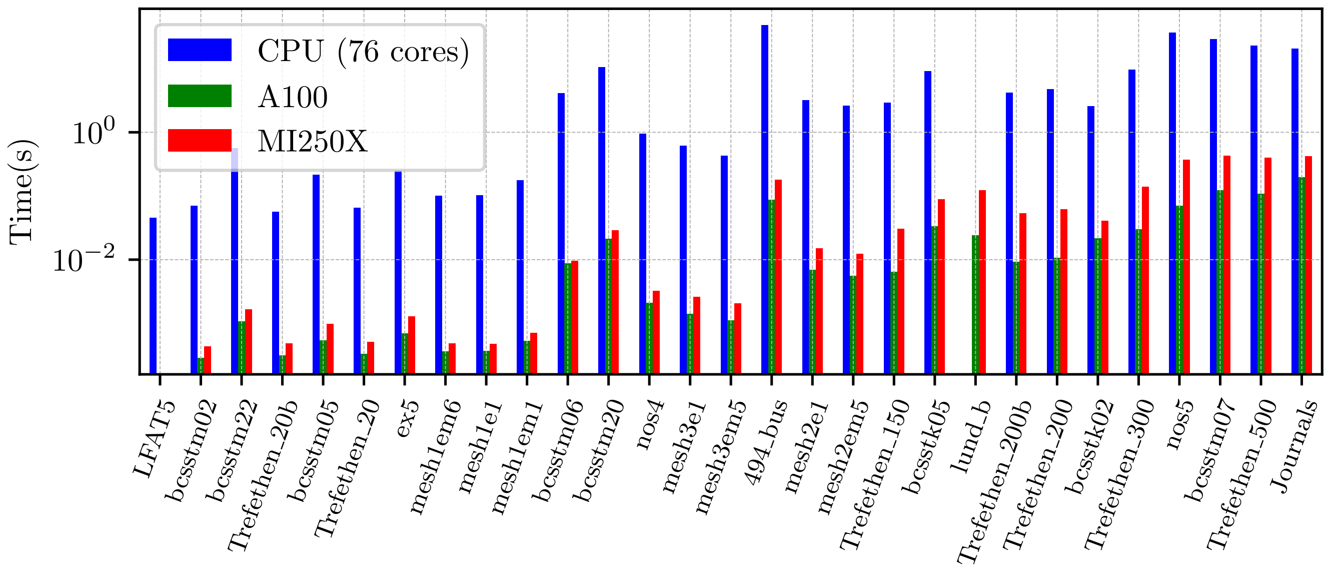
Application requirements have primarily driven the development of the batched methods. The flexibility provided by the iterative solvers in terms of early stopping, re-use of initial guess, and adaptability to matrix properties can make them very attractive for solving a large number of relatively small problems. This section showcases the benefits of utilizing the batched iterative solvers in real-world applications. We look at two main application domains, namely:

1. A combustion simulation application, where batched solvers are used to solve chemical reaction equations in each cell.
2. A plasma physics application, where batched solvers help compute the velocities and accelerations of species inside the collision kernel, that is used to simulate the edge region of the plasma within a gyrokinetic particle-in-cell simulation.

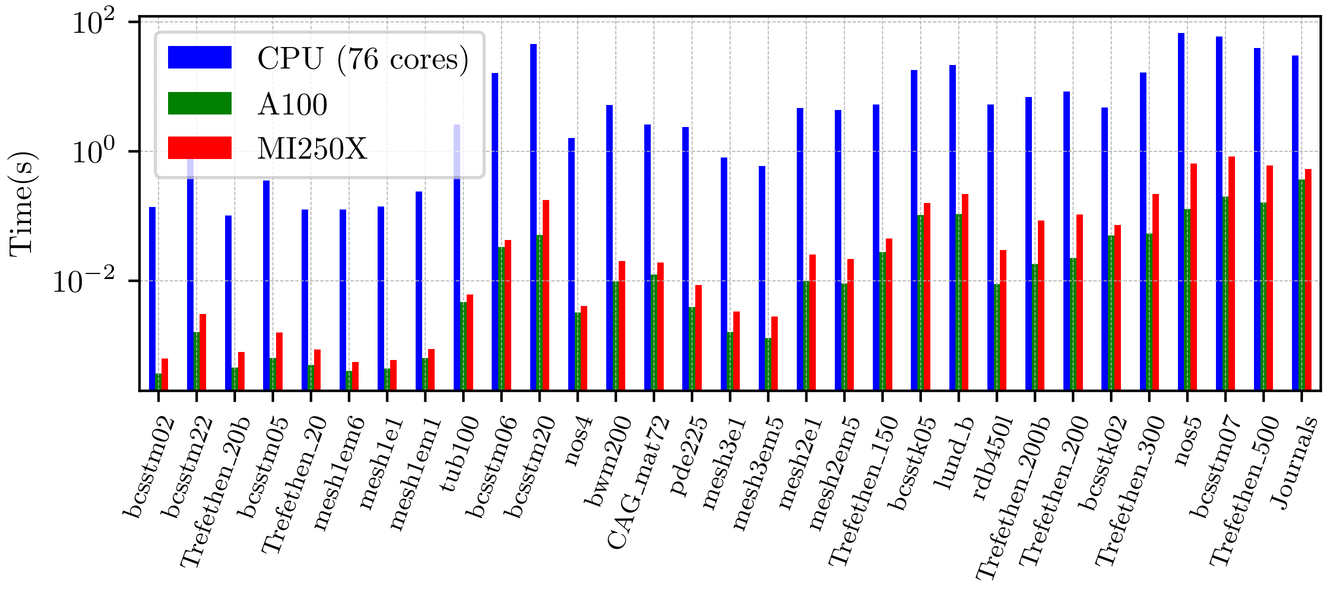
4.6.1 Accelerating combustion reaction simulations

The simulation of hydrodynamic processes with reaction systems occur frequently in combustion and astrophysical applications. These systems typically operator split the reactions from the hydrodynamics and thus require the solution of many *independent* chemical reaction ordinary differential equations (ODEs) systems,

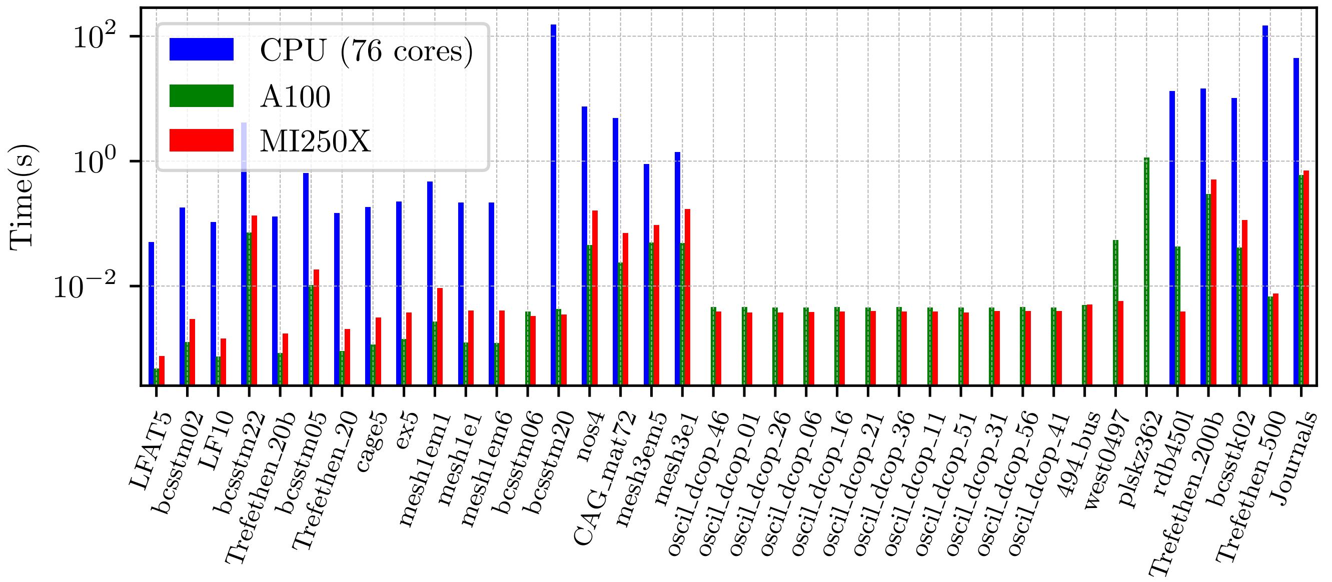
$$\phi'(t) = R(t, \phi(t)) + F(t, \phi(t)), \;\; \phi(t) \in \mathbb{R}^N, \tag{4.14}$$



(a) **BatchCg**



(b) **BatchBicgstab**



(c) **BatchGmres** (30)

Figure 4.9: Time to solution for different SuiteSparse matrices, sorted by number nonzeros on Intel Xeon(76 cores) CPUs, NVIDIA A100 and AMD MI250X, 5000 batch entries

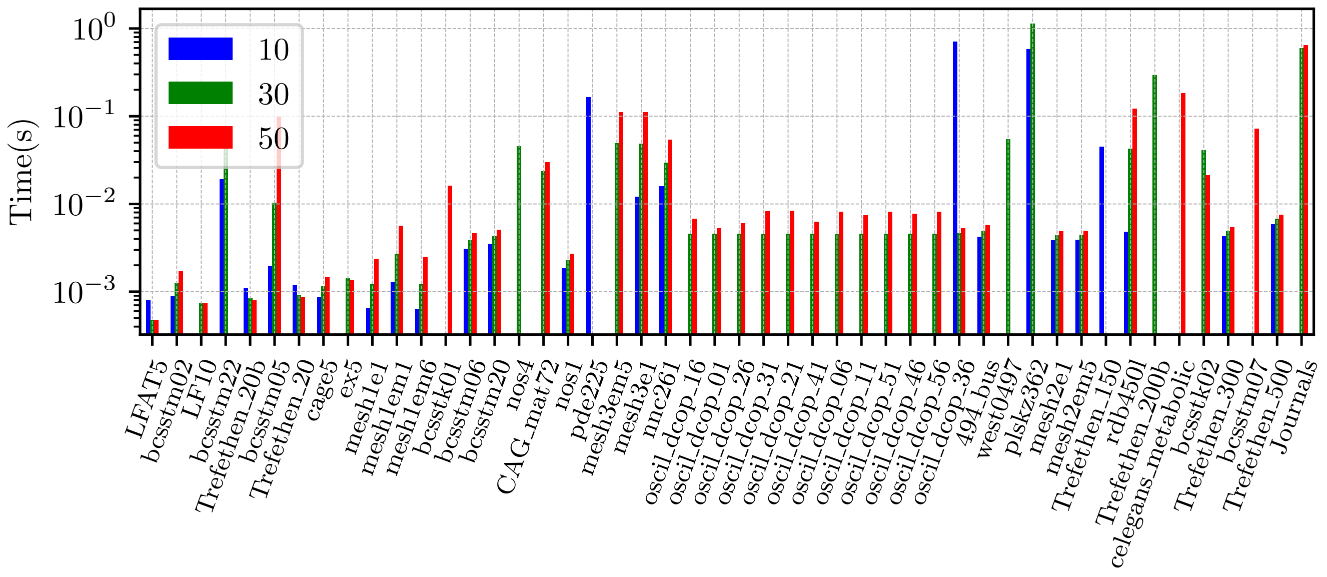


Figure 4.10: Comparing effectiveness of the restart parameter for **`BatchGmres`** on an NVIDIA A100, 5000 batch entries

where $R(t, \phi(t))$ is the reaction term and $F(t, \phi(t))$ is some forcing term, in each of the spatial discretization cells. Due to the numerical stiffness associated with the chemical kinetics, implicit time-stepping schemes, usually Backwards Differentiation Formula (BDF) linear multi-step methods, are typically utilized to evolve the solution of Equation (4.14) in time. Letting ϕ_n approximate the solution of $\phi(t_n)$, the application of the BDF method to Equation (4.14) gives rise to a nonlinear system at every time step,

$$G(\phi_n) \equiv \phi_n - h_n \beta_{n,0} f(t_n, \phi_n) - a_n = 0 \tag{4.15}$$

where $f(t_n, \phi_n) \equiv R(t_n, \phi_n) + F(t_n, \phi_n)$, h_n is the temporal step-size, $\beta_{n,0}$ are coefficients of the method, and a_n is comprised of known data from prior time steps. Solving this nonlinear system with $m + 1$ Newton iterations requires the solution of the linear systems,

$$(I - h_n \beta_{n,0} \frac{df}{d\phi})[\phi^{n(m+1)} - \phi^{n(m)}] = -G(\phi^{n(m)}). \tag{4.16}$$

The sparsity of the matrix $\mathbf{A} \equiv (I - h_n \beta_{n,0} \frac{df}{d\phi})$ is dependent on the chemical reaction mechanism.

This implies that for each spatial discretization cell, a linear system has to be solved, where all the systems share the same sparsity pattern.

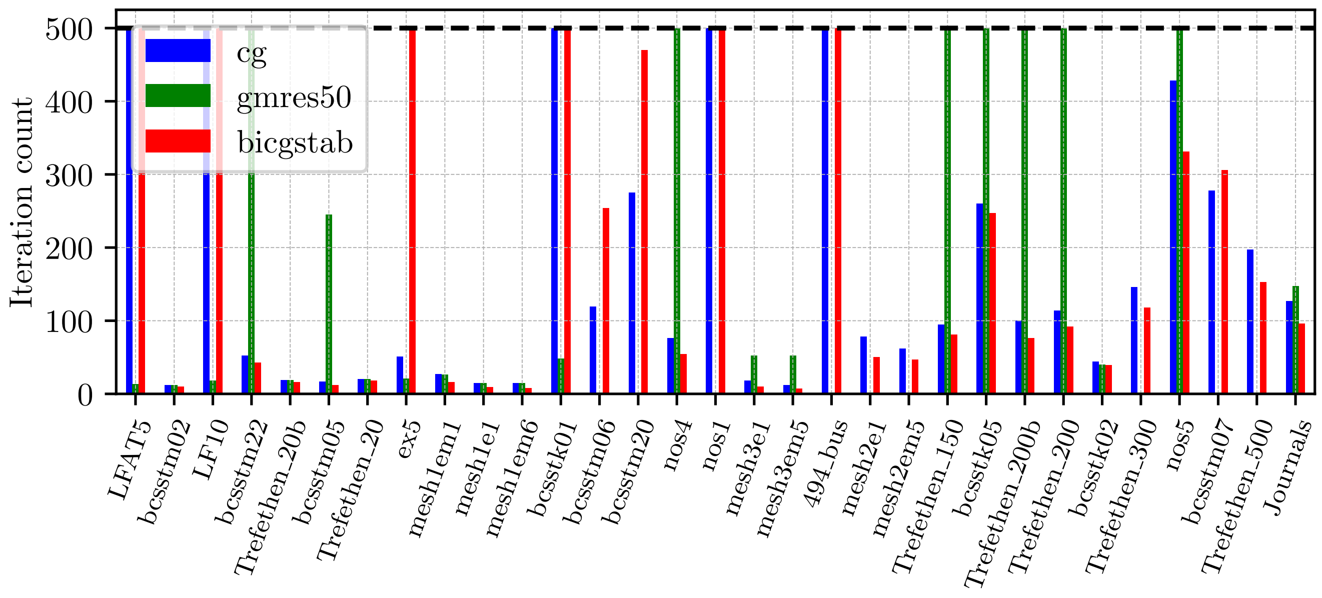
4.6.1.1 Improving matrix conditioning with scaling

For some problems, to improve conditioning and to reduce the range of values, it is necessary to scale the system by row and by column. Instead of solving $\boldsymbol{Ax} = \boldsymbol{b}$, we solve

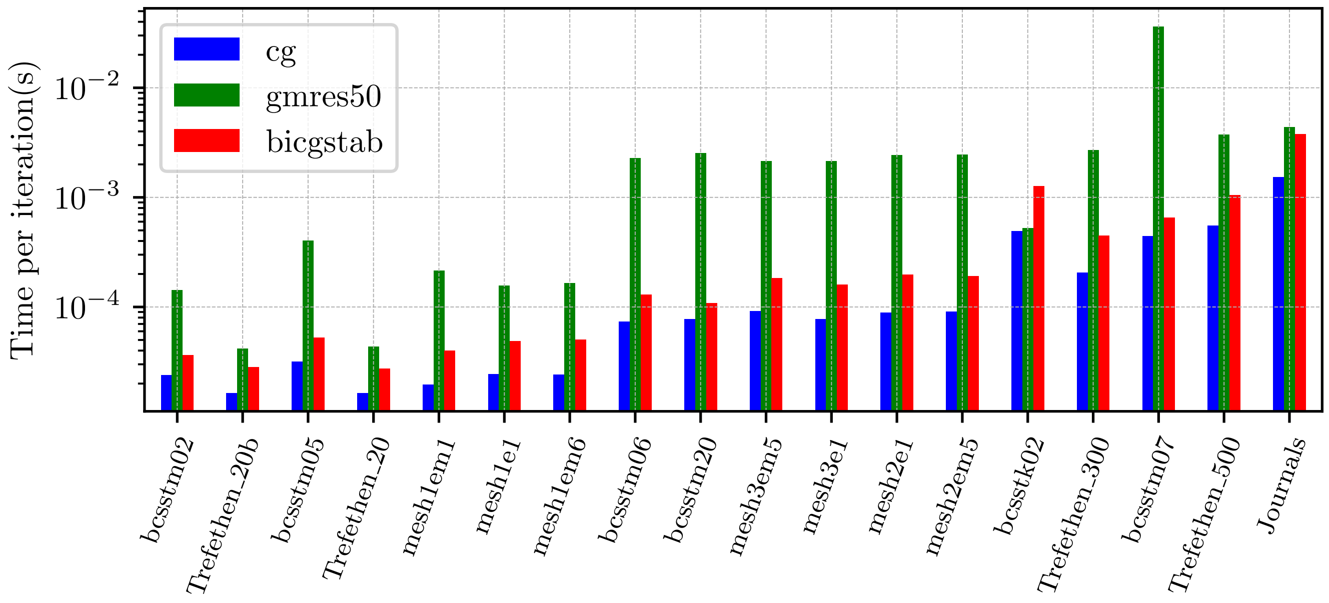
$$\boldsymbol{S}_1 \boldsymbol{A} \boldsymbol{S}_2 (\boldsymbol{S}_2^{-1} \boldsymbol{x}) = \boldsymbol{S}_1 \boldsymbol{b} \tag{4.17}$$

where $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ are diagonal matrices, which can be seen as left and right preconditioning operators. These scaling operators are generally computed using information derived externally and not available from the matrix. For example, for some ODE integrators, the step size and the weights can be used to provide better conditioning for the solution of the resulting ODE linear system [Hin02; BGW+21].

This scaling operation is required only once per linear system solution, and hence, we choose to perform this scaling as a pre-processing step instead of within the solver apply. Within this pre-processing step, we take both the scaling vectors,



(a) Iteration count for a final residual error of $1e - 6$



(b) Time per iteration (s)

Figure 4.11: Comparing the three different solvers, **`BatchCg`**, **`BatchBicgstab`** and **`BatchGmres`** on an NVIDIA A100, 5000 batch entries

Problem	Size	Non-zeros (A)	Non-zeros (L+U)
dodecane_lu	54	2,332 (80%)	2,754 (94%)
drm19	22	438 (90%)	442 (91%)
gri12	33	978 (90%)	1,018 (93%)
gri30	54	2,560 (88%)	2,860 (98%)
isooctane	144	6,135 (30%)	20,307 (98%)
lidryer	10	91 (91%)	91 (91%)

Table 4.6: Key matrix and sparsity characteristics of the benchmark reaction mechanisms extracted from PeleLM chemistry simulation runs.

and use them to left and right multiply the matrix for all the batch entries within one kernel. Computing this within a pre-processing step and not in the solver allows us to maximize the GPU occupancy by not requiring us to allocate shared memory for the system matrix, which is not read-only but read-write. Note that for consistency to get back the original solution, we need to perform a post-processing scaling for the solution vector, which is also done similarly after the solution process has been completed.

4.6.1.2 Assembling the linear system matrices for different mechanisms

From Equation (4.16), we have seen that different chemical reaction mechanisms give us different linear systems that need to be solved on each spatial discretization cell, all sharing the same sparsity pattern.

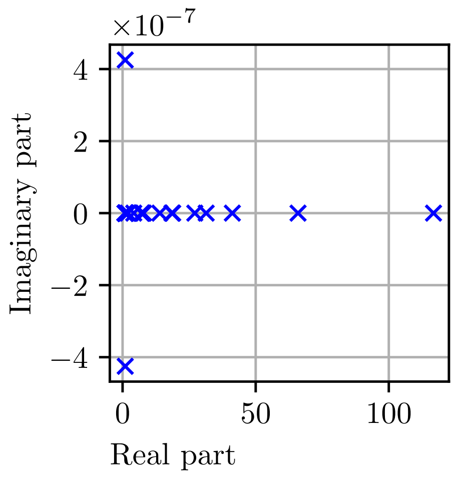
With the PeleLM application code [DB99; NDB18], which aims to simulate reactive flow in the Low Mach number regime, these flow simulations are dynamic and evolve in both time and space. The PeleLM code uses the SUNDIALS [Hin02] software library to solve for the resulting reaction ODEs in each cell.

To study the behavior of the iterative solvers and to choose the optimal solver for a reaction mechanism, we extract the matrices from the PeleLM code and perform an external analysis. We consider six different reaction mechanisms, with matrix sizes ranging from 10 to 144, as shown in Table 4.6. We also include the performance of a direct solver (shown in the last column), which can significantly increase the fill-in, thereby increasing the memory requirements.

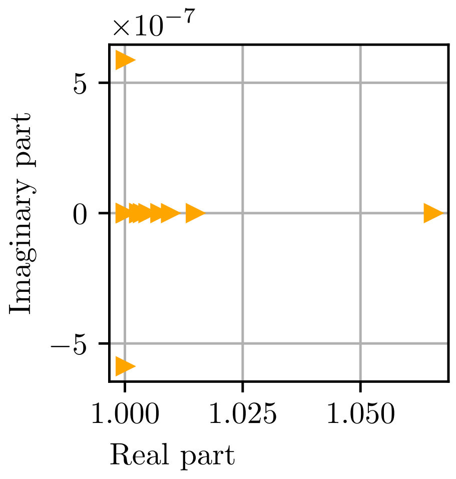
In Figure 4.12, we show the eigenvalue distribution of one of the matrices for each of the six reaction mechanisms we consider. These eigenvalue distributions are an indicator of the condition number of the matrix, and the larger the spread of the spectrum (ratio between the largest and the smallest eigenvalue), the harder the problem to solve and the requirement for a more robust method or the requirement for a large number of iterations to ensure convergence. From Figure 4.12, we see that of the six mechanisms, dodecane_lu and isooctane have the largest condition numbers, and the other mechanisms are relatively easy to solve.

4.6.1.3 A posteriori convergence verification

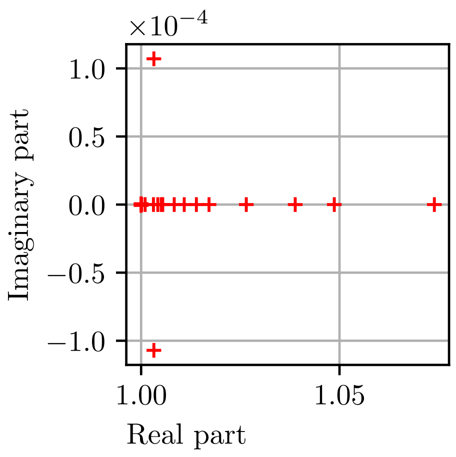
In some cases, the internal convergence detection of an iterative method may diverge from the true error of the solution, defined by $r = b - Ax$. To ensure that our implementations have converged to the prescribed tolerance, we perform an a-posteriori error check for the matrices we use, where we explicitly compute the true residual norm to ensure that it is lesser than the prescribed tolerance. Figure 4.13 shows the relative residual errors of each matrix in the batch for each mechanism. We observe that the direct method (we use the cuBLAS batch dense LU solver) converges to machine precision. In contrast, the other solvers, which are iterative, have a higher error with the tolerance being set to 1e-8. Except for a few matrix entries that have a high residual error and diverge for the **`BatchRichardson`** solver, for the rest of the mechanisms, all methods have the a-posteriori error as prescribed by the input tolerance.



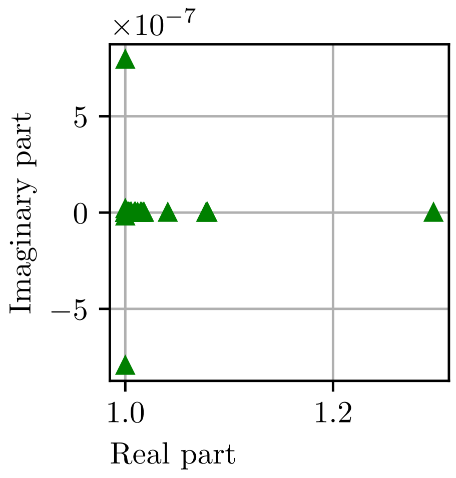
(a) dodecane_lu



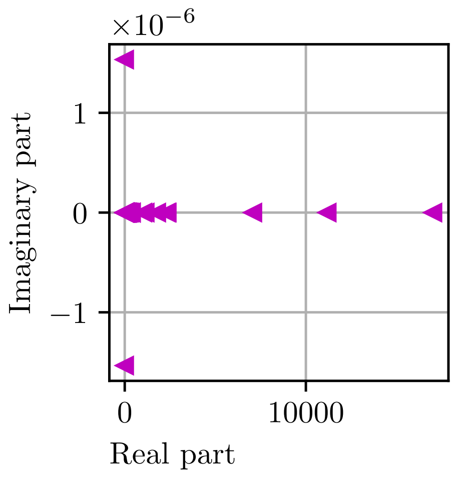
(b) drm19



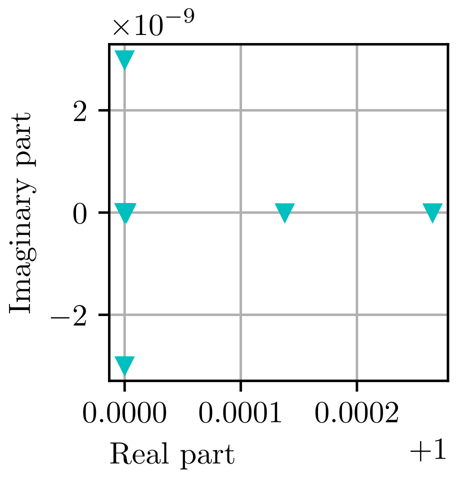
(c) gri12



(d) gri30

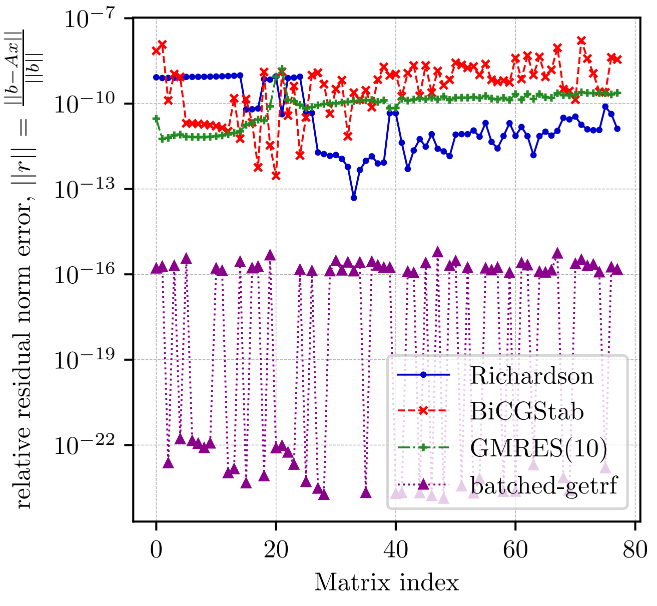


(e) isooctane

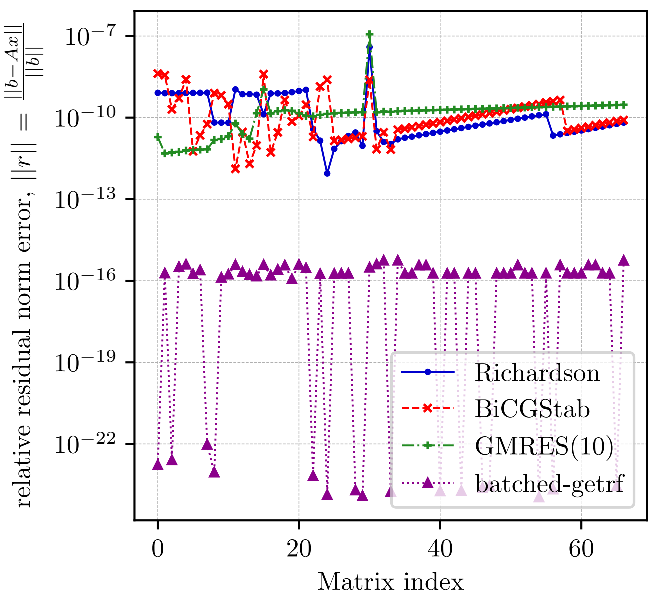


(f) lidryer

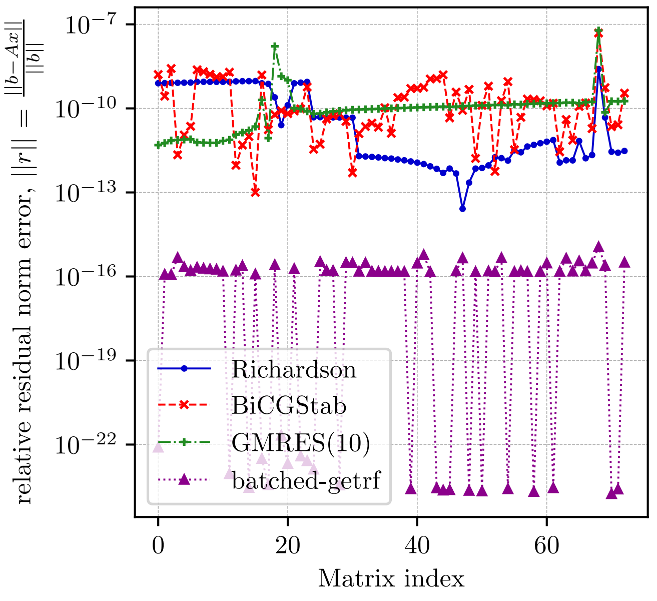
Figure 4.12: Eigenvalues of the matrices



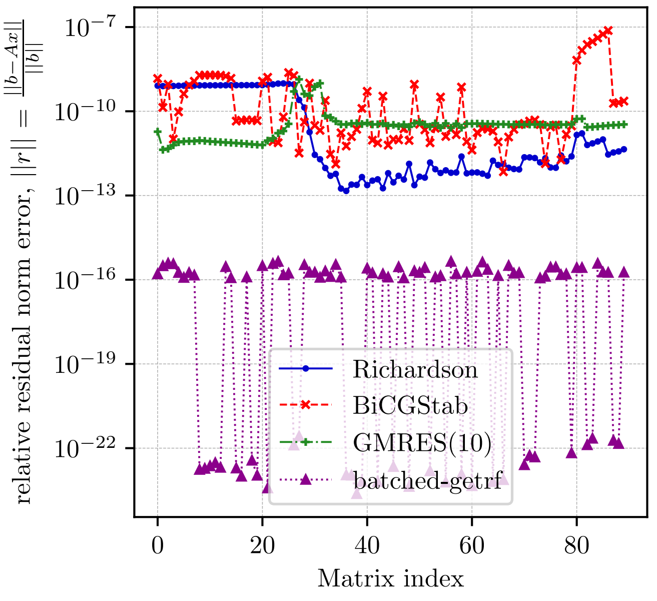
(a) dodecane_lu



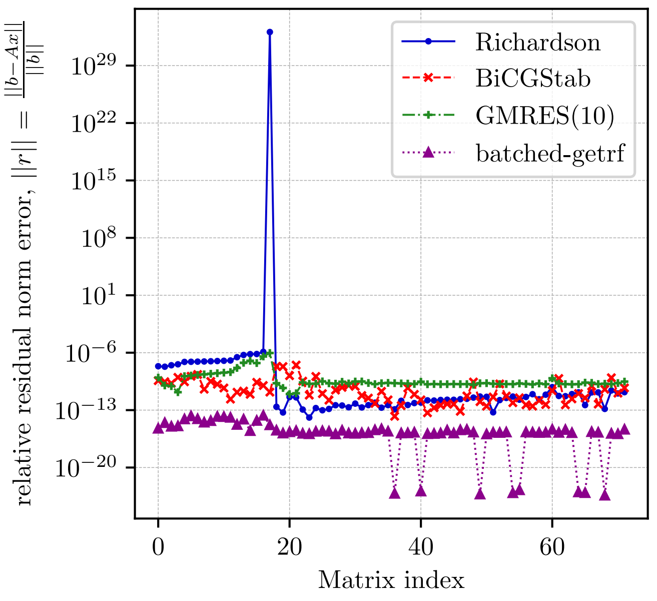
(b) drm19



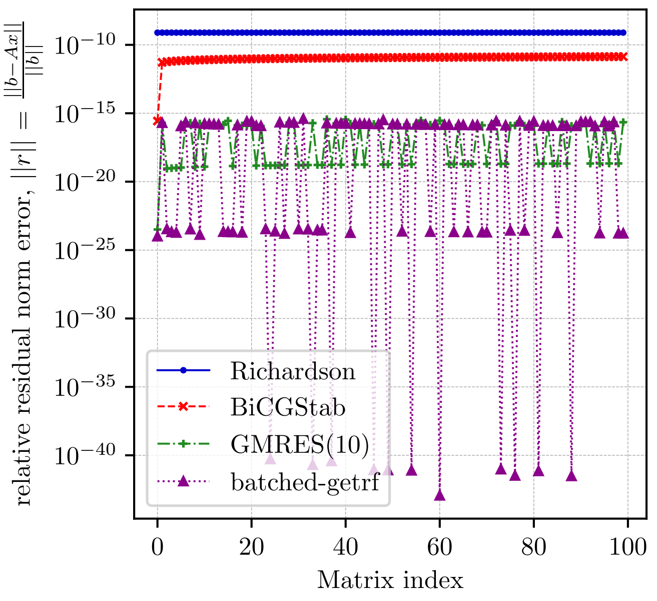
(c) gri12



(d) gri30



(e) isooctane



(f) lidryer

Figure 4.13: Verifying correctness with an a-posteriori residual norm error evaluation.

4.6.1.4 Benchmarking the batched iterative solvers against the direct methods

Finally, in Figure 4.14, we compare the time to solution for the different reaction mechanisms with different batched iterative solvers. We observe that we outperform the batched dense direct for all problems, even for the smallest problem we consider (lidryer: 10×10). For these problems, the **BatchRichardson** coupled with the scalar Jacobi solver is remarkably efficient, with a very cheap per-iteration cost. For the isooctane problem, we require a more robust solver, and only the **BatchBicgstab** and **BatchGmres** converge, with the **BatchBicgstab** performing better. We also observe that the direct method runs out of memory for the dodecane_lu and the gri30 problems beyond $\mathcal{O}(10^5)$ batch entries.

The number of iterations required to converge to a tolerance of 1e-8 is shown in Figure 4.15. While it can be advantageous to use **BatchGmres** with a lower restart, this may not be sufficient for problems such as the isooctane mechanism.

Finally, Figure 4.16 shows the speedups of the different batched iterative solvers against the cuBLAS dense direct solver, and we obtain an average speedup of around $2.5\times$ considering all mechanisms and the best solver for each mechanism.

4.6.1.5 Integration into SUNDIALS and Pele: The ReactEval benchmark

The PelePhysics [Pelc] is a repository of physics databases and implementation that is used within other Pele codes such as PeleC [Pela] and PeleLMeX [Pelb]. It allows the user to efficiently manage the different chemistry and transport models, which allows for state-of-the-art simulation. A central component of the PelePhysics code is the evolution of the reactive flow in time. PelePhysics relies on the CVODE module from the SUNDIALS suite, which implements many different time integrators. In many cases, these reactive flow simulations are numerically stiff, which requires an implicit time-stepping scheme to enforce stability. Assembling these reaction mechanisms so that they can be integrated over time with an implicit time-stepping scheme gives us linear systems that need to be solved at each time step.

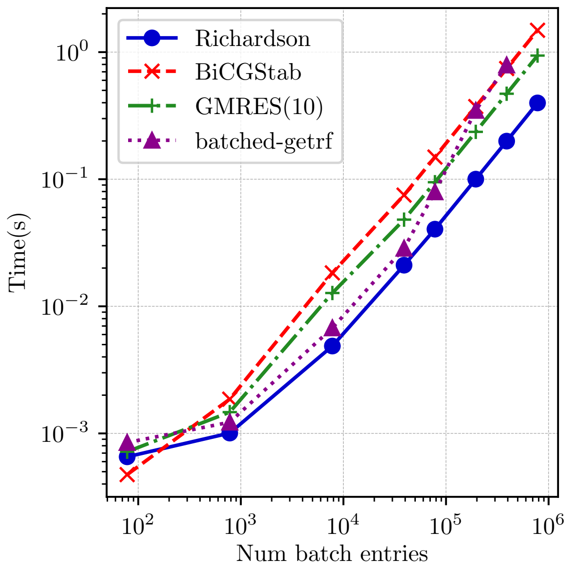
The batched solvers are particularly useful here as they take advantage of the perfect parallelism of the independent linear system solution on each grid cell. PelePhysics through SUNDIALS-CVODE has a few options for the solution of the linear systems, including our batched iterative approach that has been newly integrated:

1. Direct solution with the MAGMA library [HDT+15]: Direct methods are desirable for the batched solution, particularly for mechanisms with a few species.
2. Block-based ensemble GMRES is available with SUNDIALS, which assembles a block diagonal matrix and solves it with a monolithic solver.
3. Ginkgo batched iterative solvers, which makes available the **BatchBicgstab** and **BatchGmres** methods to solve the systems in a batched fashion.

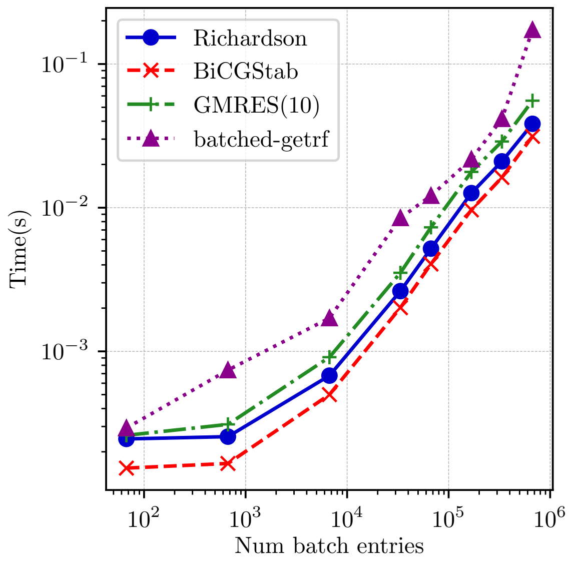
Figure 4.17 shows the speedup obtained with the batched iterative solvers with the **BatchCsr** format and a scalar Jacobi preconditioner against the highly tuned MAGMA dense direct solver [HDT+15]. We showcase the performance of the batched iterative solvers with two mechanisms, the dodecane_lu with a matrix size of (54×54) and the dodecane_lu_qss mechanism with matrices of size (35×35). For the smaller mechanism, **BatchGmres** performs slightly better than **BatchBicgstab**, but both the batched iterative solvers outperform the batched LU solver from MAGMA. For the larger mechanism, we see that the **BatchBicgstab** is the best choice with an average speedup of around 2x over MAGMA.

4.6.2 Accelerating Plasma physics simulations

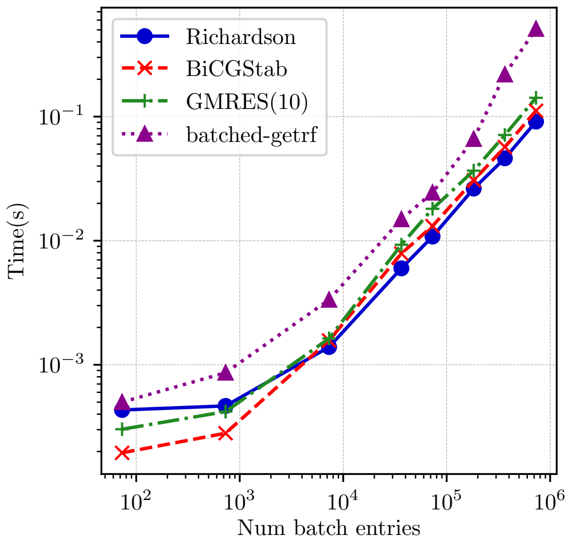
Renewable energy sources based on magnetically confined fusion plasma, such as the International Tokamak Experimental Reactor (ITER), currently cannot study the required parameter space experimentally. Hence, projects like the WDMApp aim to provide high-fidelity numerical simulations. For accurate high-resolution and high-fidelity simulations, these applications must scale to tens of thousands of nodes, each containing multiple GPUs. The WDMApp



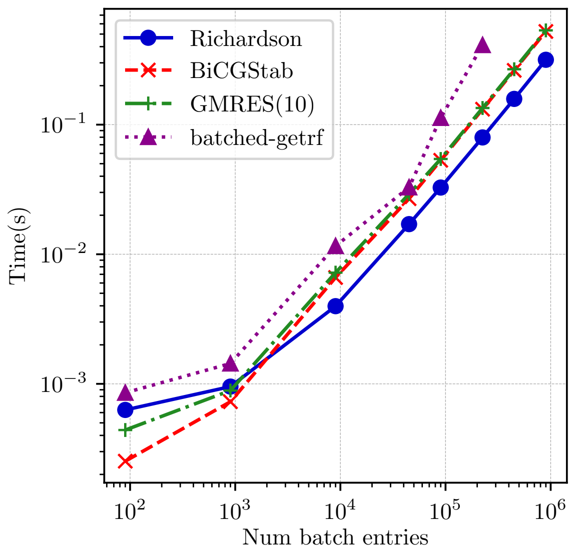
(a) dodecane_lu



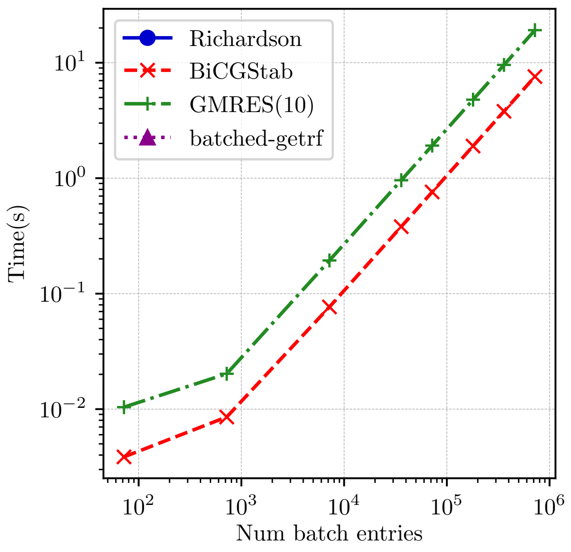
(b) drm19



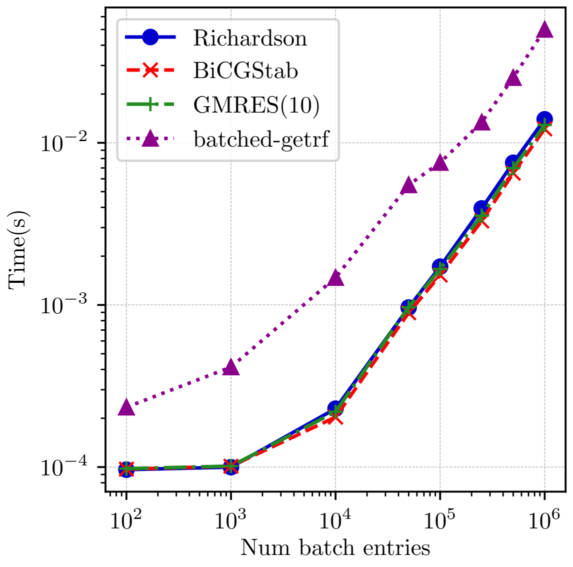
(c) gri12



(d) gri30



(e) isooctane



(f) lidryer

Figure 4.14: Time to solution with different solvers, **BatchRichardson**, **BatchGmres**, **BatchBicgstab** and the cuBLAS dense direct solver

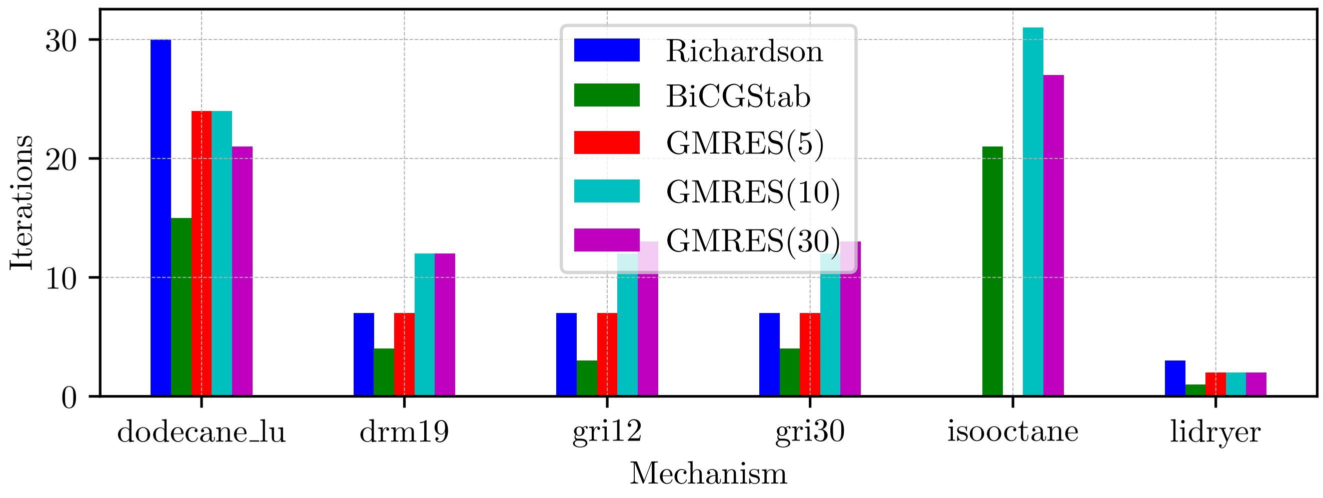


Figure 4.15: Iteration counts for the different solvers for different reaction mechanisms

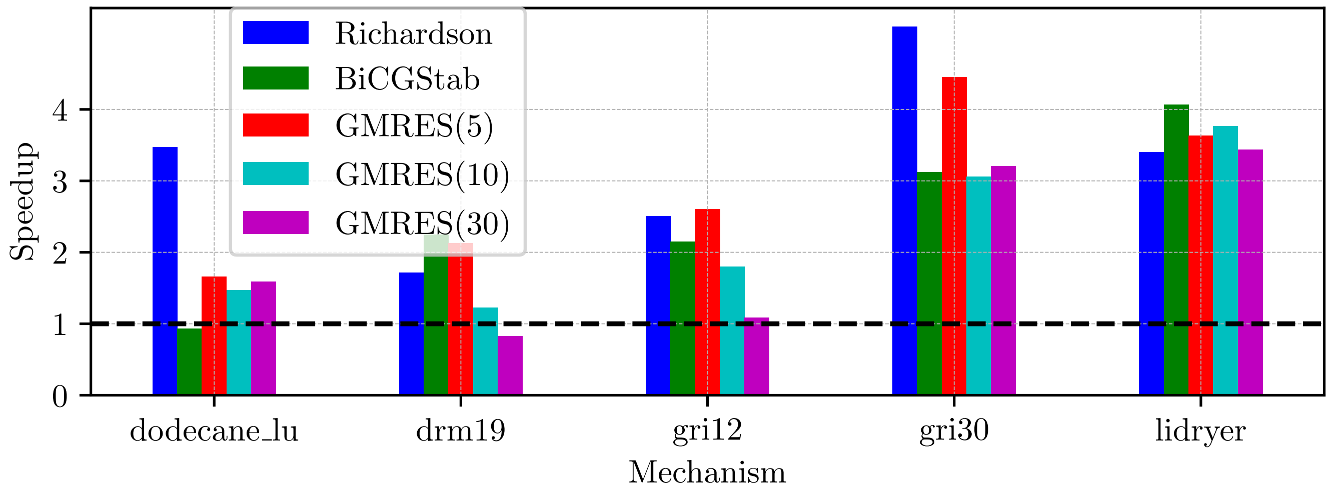
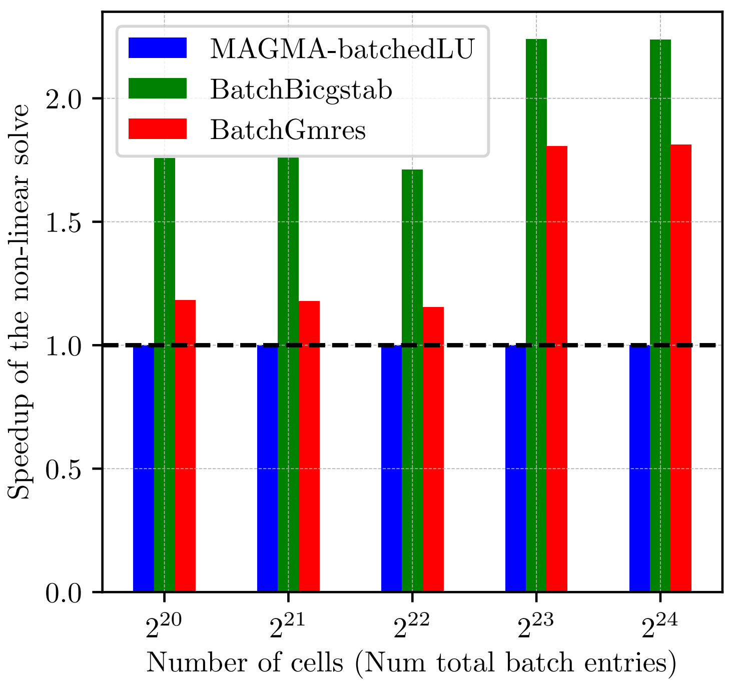
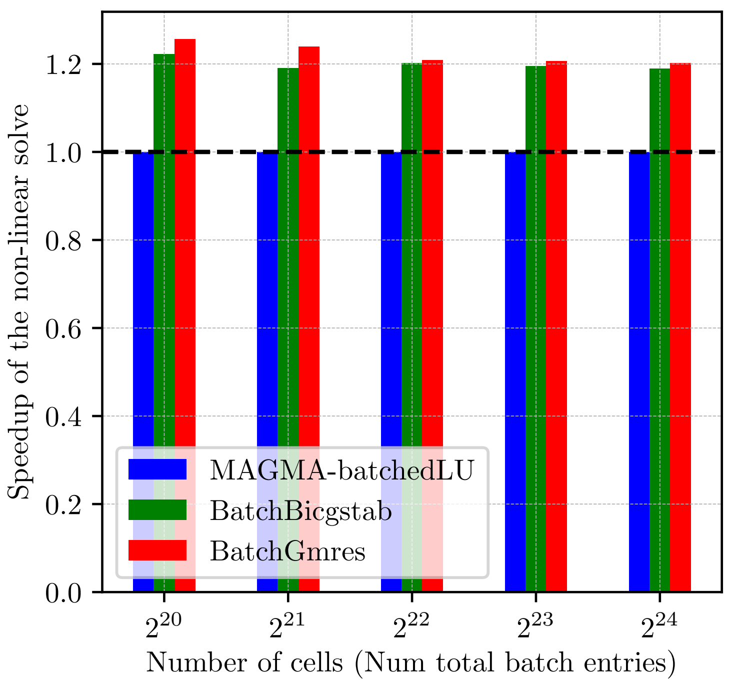


Figure 4.16: Speedup of the iterative solver against the cublas batched solver for different reaction mechanisms



(a) dodecane_lu mechanism (54 × 54)



(b) dodecane_lu_qss mechanism (35 × 35)

Figure 4.17: Speedup for the entire non-linear solver with Ginkgo's batched solvers over MAGMA batched dense direct solver for 2 mechanisms

application contains different codes focussing on different aspects, modeling different parts of the plasma and the device. For example, XGC is a gyrokinetic Particle in Cell (PIC) code used to model the plasma close to the edge.

XGC is a 5D full-function gyrokinetic particle-in-cell (PIC) application code that numerically simulates fusion edge plasmas. A nonlinear collision operator is required to model edge plasmas accurately. Therefore, XGC employs a nonlinear Fokker-Planck-Landau operator in the 2D guiding-center velocity space for multiple particle species. Coulomb collisions between particles in the plasma have been identified as a bottleneck in XGC. An implicit time integration method is employed, and the Picard method is used for the nonlinear solver. At each configuration space grid node, we must solve the nonlinear operator on the 2D velocity space grid to evolve the different species through time. In each cell of the velocity grid, we must first solve for the collisions involving a linear solution. As the collision operator is based on a stencil, each linear solve in each cell shares the same sparsity pattern. Hence, to compute the evolution of these species, we need to solve multiple independent linear systems, all sharing a sparsity pattern. Therefore, this forms a good candidate where utilizing the batched solvers can improve the parallelism and reduce the time to solve for the overall simulation.

The collision kernel utilizes MPI for multiple CPU nodes, and Kokkos [CETS14] to offload to GPUs as well as utilize OpenMP for intra-node CPU parallelism.

4.6.2.1 The XGC proxy application

Production simulations with XGC employ the LAPACK banded solver **`dgbsv`** on the CPU for the linear solve. However, as more and more of XGC is ported to GPU, the **`dgbsv`** time on the CPU becomes a larger and larger fraction of the run time, hence the need for a fast and efficient linear solver on the GPU.

A proxy app for the collision kernel has been developed using Kokkos to provide a performance portable layer for GPU offloading. The proxy app is parallelized over spatial mesh nodes and is perfectly parallel. While the future XGC application is expected to simulate multiple ion species ($\sim$10) and electrons, the proxy app currently simulates a plasma

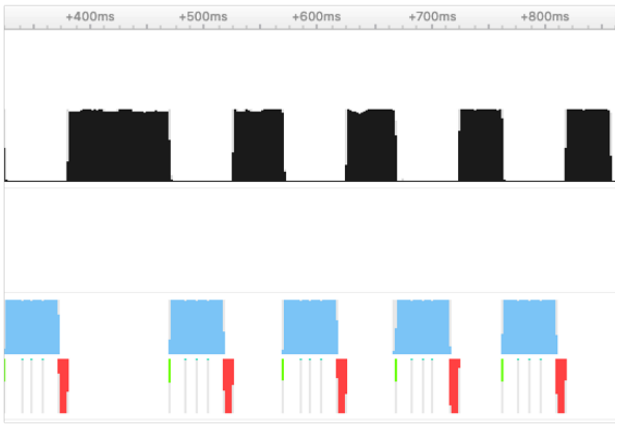


Figure 4.18: Profile of one Picard loop of the collision kernel proxy app showing time spent on CPU (black), GPU (blue), and memory transfer (red: Device to Host, green: Host to Device)

with one ion species (along with the electrons). A backward Euler time discretization and Picard iteration are employed for the two species for every mesh node. The Picard loop typically requires five iterations for convergence. Figure 4.18 shows the execution timeline of one such Picard iteration captured on one MPI rank with multiple OpenMP threads on the CPU and one GPU - the top half of the figure (black rectangles) shows CPU execution of the linear solver employed in the Picard iteration and associated processing. In contrast, the bottom half shows GPU execution (blue rectangles) and data transfer (red and green rectangles). As seen in Figure 4.18, a significant portion of the execution time for the Picard loop (~48%) is on the CPU - of which a majority of the time is spent in the solve (**`dgbsv`** call) itself (~66%). In addition, data on the GPU needs to be transferred back and forth between the CPU and the GPU (9%). These optimization opportunities motivate the need for a GPU-based batched solver, which can utilize the perfect parallelism that the algorithm provides.

The matrix sizes utilized in the main application are on the order of 10^3 rows and possess a sparsity pattern arising from the use of a nine-point stencil (9 non-zero elements per row). For these sizes and bandwidth, using dense solvers on the GPU is insufficient to beat the gain obtained from exploiting the banded nature of the matrix on the CPU. Thus, sparse GPU solvers are required and must be batched to fully saturate the GPU. Further, as the XGC matrices exhibit a low condition number, iterative batched sparse solvers are a good candidate.

We see in Figure 4.19 that the ion and the electron matrices have quite different eigenvalue distributions. For ions, the eigenvalues are more or less clustered around 1.0 (note the log real axis), which will lead to rapid convergence. At the same time, electrons have a greater range of real parts of the eigenvalues, which hint that it may require more iterations for convergence. That being said, they are both well-conditioned enough to take good advantage of iterative solvers.

4.6.2.2 Optimized matrix storage for the XGC application

The XGC application uses a tailored 9-point non-symmetric stencil with matrices that can be considered relatively large for batched solvers (992 rows for the proxy application). The sparsity pattern of the matrix is shown in Figure 4.20. We can see that this matrix mostly has a constant number of nonzeros per row. A tailored matrix format that enables coalesced access to the matrix entries without incurring a memory storage penalty can improve the performance, particularly for the SpMV operation, which forms the workhorse of the iterative solvers.

From the matrix formats we provide, we see that the **`BatchEll`** format is a good choice for this sparsity pattern. All the matrices across the grid share the same sparsity pattern, and hence, we store one copy of the sparsity pattern store the individual values in a **`BatchEll`** format (in column-major).

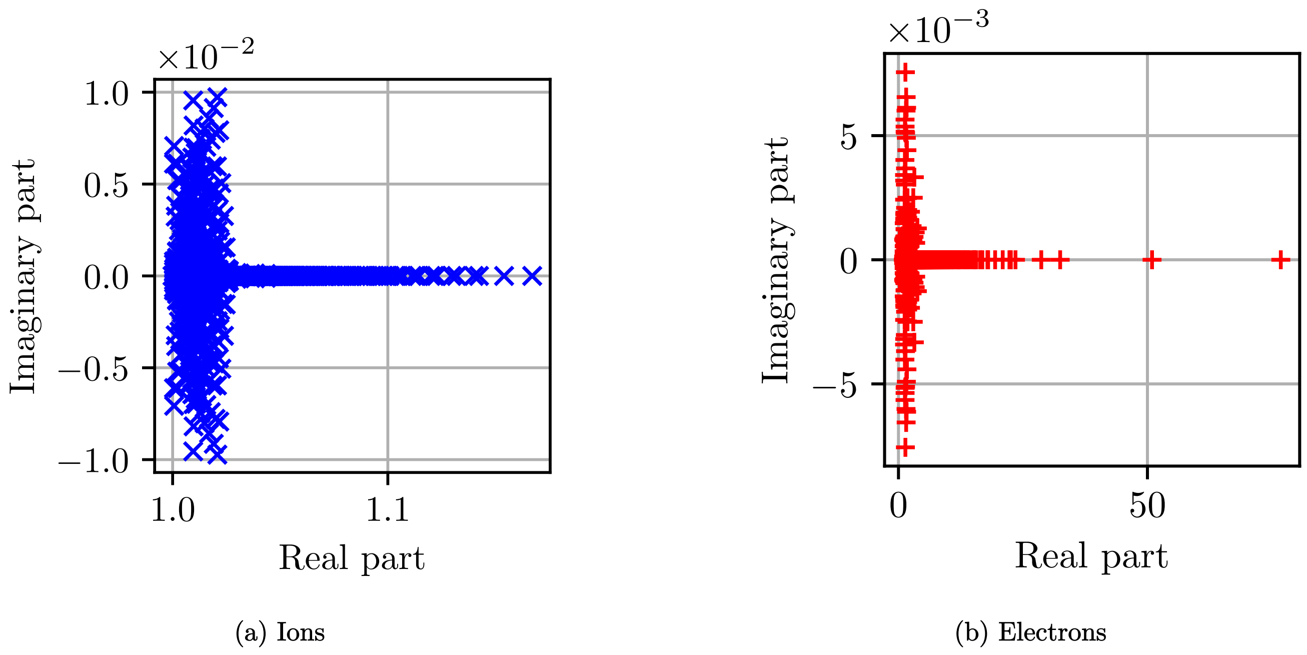


Figure 4.19: Eigenvalue distribution for the two species used in the proxy application

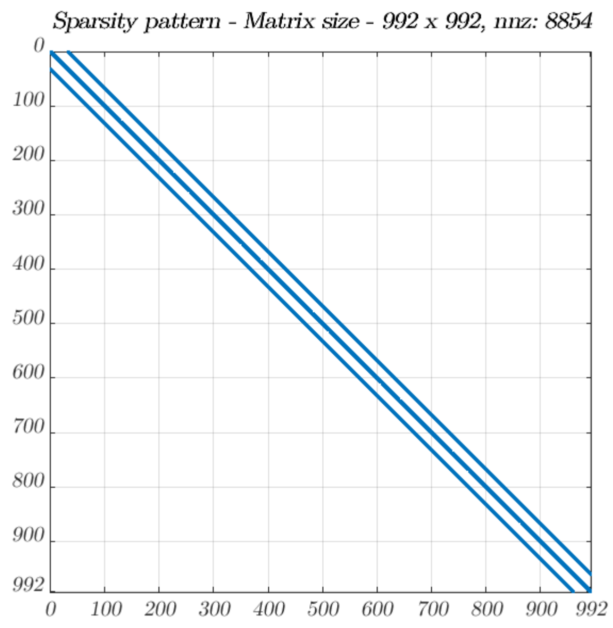


Figure 4.20: Sparsity pattern of an individual entry of the XGC matrix: 992 rows, 9 nonzeros per row.

Table 4.7: Some relevant theoretical performance numbers for different processors [NVI20; NVI17; AMD20]

Architecture	*Peak FP64* (TFlops)	*Main memory BW* (GB/s)	*(L1 + shared memory) /CU* (KB)	*L2 data cache* (MB)	# of SMs/CUs
A100-40GB	9.7	1555	192	40	108
V100-16GB	7.8	990	128	6	80
MI100-32GB	11.5	1230	16+64	8	120
Intel Xeon Gold	1.0	128	64	20	20

4.6.2.3 The choice of solvers: Direct banded or Iterative

The XGC proxy app uses the LAPACK banded solver, **`dgbsv`** as a batched solver on the CPU. It employs one CPU core to solve one individual system and utilizes all available cores on the CPU to solve the batched systems in parallel. On NVIDIA GPUs, the cuSOLVER batched sparse QR routine, which uses the **`BatchCsr`** matrix format, is the only available batched sparse solver for general (non-banded) matrices. Currently, no other batched sparse solvers are provided by the GPU vendors.

While direct solvers always solve the system to the full precision of the underlying type, iterative solvers come with the option of tuning the tolerance to solve the systems to the required precision. This makes iterative solvers attractive when an 'exact' solution is unnecessary; this is the case in several engineering applications and especially when the linear solve is part of a non-linear solver, as is the case in XGC. Additionally, iterative solvers can take an initial guess, which in many cases significantly reduces the time to solution. With an outer non-linear Picard iteration, for XGC, we can take the solution of the previous step as an initial guess for the subsequent linear solve, which proves to be a reasonable estimate of the solution, thereby accelerating the convergence.

Table 4.7 shows the characteristics of the three GPUs we run the batched solvers on. The number of compute units represents the independent multi-processors available on the GPU. The L1 data cache + shared memory size per compute unit signifies the memory available in each compute unit. The NVIDIA GPUs can view the L1+shared memory as a single memory level. For example, on the V100, 32 KB is reserved per CU for the L1 cache, while the shared memory per CU is configurable up to 96 KB. Memory that the kernel has not requested as shared memory is automatically used as L1 data cache, thereby increasing the amount of L1 data cache available. On the AMD MI100, the shared memory is set to 64 KB per CU, and the available L1 cache is 16 KB.

4.6.2.4 Evaluating the performance of the XGC proxy app with batched solvers

The XGC proxy app isolates the collision kernel by extracting two species, one ion and one electron. At each grid point, we need to solve for the evolution of these species, giving us two linear systems per grid point that can be arranged in a batch fashion. These batches consist of ion and electron matrix repetitions similar to XGC production runs.

At the outset, we note that we let each system converge to an absolute residual tolerance of 10^{-10}. Conservation of relevant physical quantities (within the collision kernel) in XGC to a pre-decided threshold (10^{-7}) was met with a minimum tolerance of 10^{-10} with our batched iterative solver, **`BatchBicgstab`** . Increasing the linear solver tolerance above 10^{-10} resulted in the Picard loop not converging up to 100 iterations.

Figure 4.21 shows the time to solution (per solve and per solve per matrix) using the proxy app, utilizing different solvers, matrix formats, and for three different architectures, the NVIDIA V100, NVIDIA A100, and AMD MI100, and how they compare with the LAPACK batched banded solver, parallelized over different matrices on the Intel Skylake node. We also compare with a direct sparse QR solver provided by the cuSOLVER library.

From Figure 4.21, we see that the batched sparse QR solver performs poorly for these banded matrices. Given the good conditioning of these matrices, the cheap cost of a solution with an iterative solver is more effective than its direct counterpart. Additionally, the CSR format is better suited for the sparse direct solver due to the sparse QR algorithm.

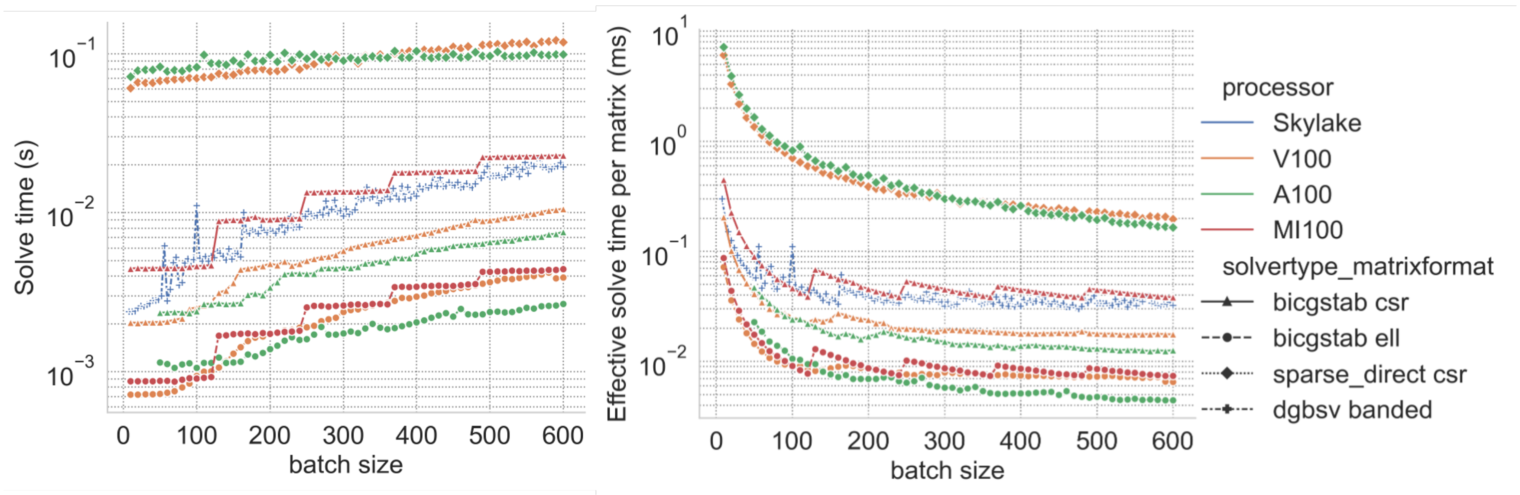


Figure 4.21: Time to solution (left: per solve, right: per solve per matrix), for different matrix formats, on different platforms, as a function of the batch size

Table 4.8: Performance metrics on different platforms with the two batch matrix formats (L1 cache data was not available for the AMD MI100)

Processor, format	**Subgroup occupancy** %	**L1 hit rate** %	**L2 hit rate** %
V100, CSR	75.1	50.7	63.1
V100, ELL	98.2	24.5	63.1
A100, CSR	72.9	76.6	97.2
A100, ELL	98.2	74.5	94.8
MI100, CSR	52	62.3	86
MI100, ELL	94	61.5	88

On the other hand, an iterative solver such as BiCGStab is only dependent on the matrix-vector product algorithm access patterns, which can hence implement tailored matrix storage formats, which in this instance is the **`BatchEll`** format. We see that we outperform the direct solver approximately 10 to 30 $\times$ for all the batch sizes.

We also observe that LAPACK's banded solver, **`dgbsv`**, is quite competitive due to the banded nature of the matrix and the large caches available on the Intel CPUs. In this case, Kokkos is used to parallelize the batched banded solve, with each CPU core performing one linear solve.

We observe a significant difference in the performance of **`BatchCsr`** and **`BatchEll`** for this problem on all three GPUs. Since this is a banded problem with nine non-zeros per row except in rows corresponding to boundary points of the grid, (1) it is well-suited to a uniform rectangular storage block with very little padding necessary (only for the boundary points of the grid) and (2) with only nine non-zeros per row, the subgroup-parallel reduction used by our **`BatchCsr`** SpMV is not able to utilize the subgroup completely. We store nine non-zeros per row for all our experiments with the **`BatchEll`** format. The subgroup is well-utilized, with different threads in a subgroup operating on different rows and with 992 rows in the matrix. Each row is processed sequentially, and hence nine subgroup-iterations are needed to process all the columns in each row. The data is stored column-major to make sure we get coalesced memory accesses.

With **`BatchCsr`**, a subgroup of 32 work-items has only five active work-items (9 divided by 2, rounded up) active in the first reduction stage. Therefore, the subgroup is not well-utilized. This is exacerbated in the AMD GPUs, which have a subgroup size of 64, thereby providing us with higher speedups for **`BatchEll`** compared to **`BatchCsr`**. The subgroup utilization metrics from the ROCm and NVIDIA profilers corroborate this. On the entire BiCGStab solve, we observe an overall high subgroup utilization with **`BatchEll`** (Table 4.8).

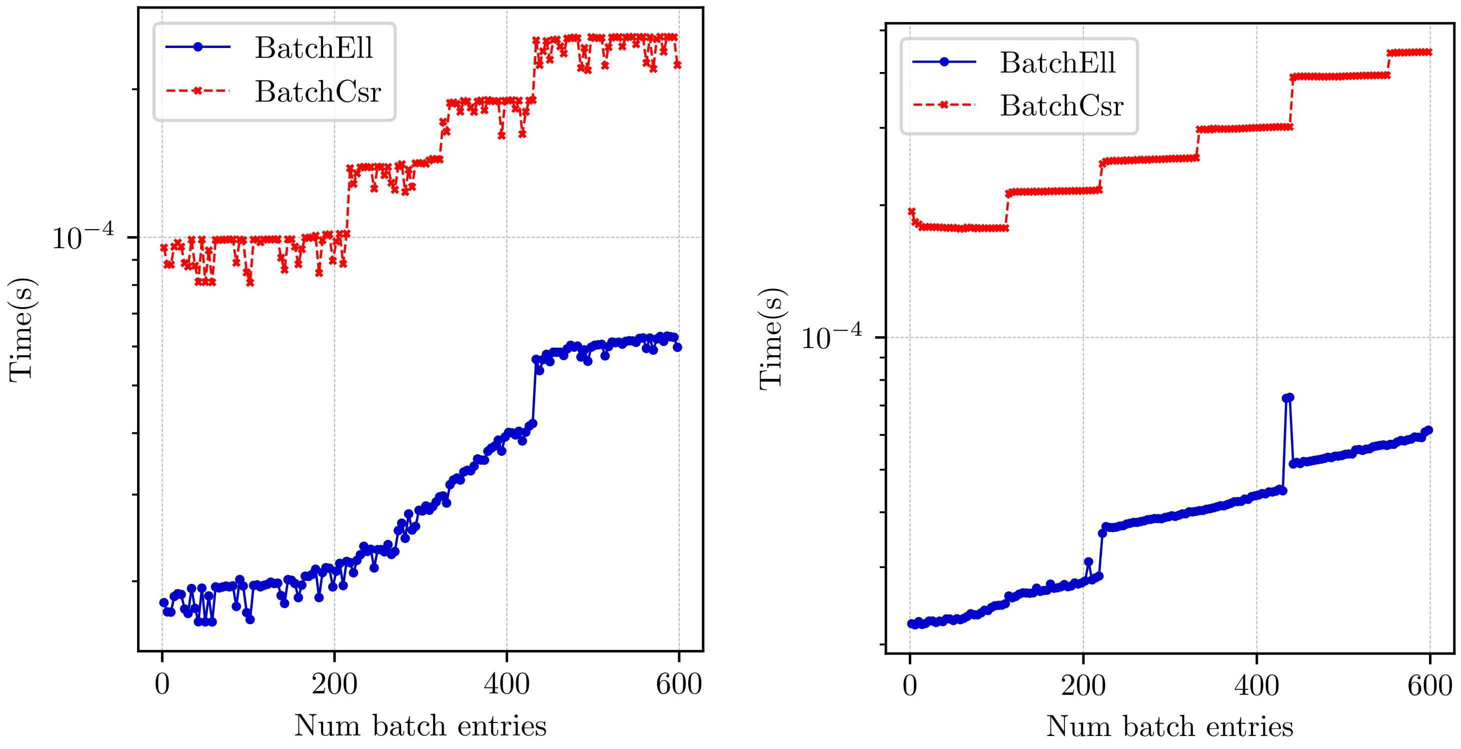


Figure 4.22: Total time taken by the **BatchCsr** and the **BatchEll** SpMV kernels on an A100 GPU and a MI250X GPU

The discrete jumps at multiples of 120 for the MI100 GPU occurs, as the MI100 has 120 compute units. To schedule the next system after a multiple of 120, the scheduler must wait for one of the compute units to be available. The V100 and the A100 have a smooth trend in the time to solution and do not exhibit these jumps at multiples of their number of compute units (80 and 108). This is likely attributed to the NVIDIA runtime scheduler, which can hide the workgroup launches without a significant overhead. This also means that we can expect the NVIDIA GPUs to perform better for non-uniform systems (with different linear systems in a batch having different iteration counts).

The right figure in Figure 4.21 shows the variation of the average time for the solution per batch matrix entry as a function of the batch size. These curves clearly show that with an increasing number of batch entries, the time to solution needed per batch entry decreases, showing that we are saturating the GPU and no overhead accumulates over increasing the number of batch entries.

In order to isolate the impact of the sparse matrix format, we show in Figure 4.22, the timing plots for the sparse-matrix vector kernel for both the **BatchCsr** and the **BatchEll** formats on the A100 GPU. We observe that due to factors previously mentioned, the **BatchEll** format is the superior format for the problem at hand.

With iterative solvers, we can provide contextual information about the problem to the algorithm in the form of an initial guess. In Figure 4.23, we investigate the impact of using an initial guess for the linear solves inside the non-linear Picard iteration. An excellent initial guess can significantly reduce the iteration count needed. Hence, iterative solvers accepting an initial guess have a clear advantage for scenarios where they can be used as inner solvers for a non-linear solver.

With batched iterative solvers, when solving independent linear systems with possibly different convergence properties, the effects of initial guess are pronounced only for those systems with a higher iteration count. In our case, the electron system requires a moderate number of iterations, around 35 with an initial guess of all zeros. Using the previous Picard iteration solution, we can reduce the iteration count for successive linear solves in the non-linear solver. In the XGC proxy-app, we have 5 Picard iterations, and the linear solver iteration counts for successive Picard iterations are shown in Table 4.9. We see a significant reduction in iteration count, translating to a faster time to solution.

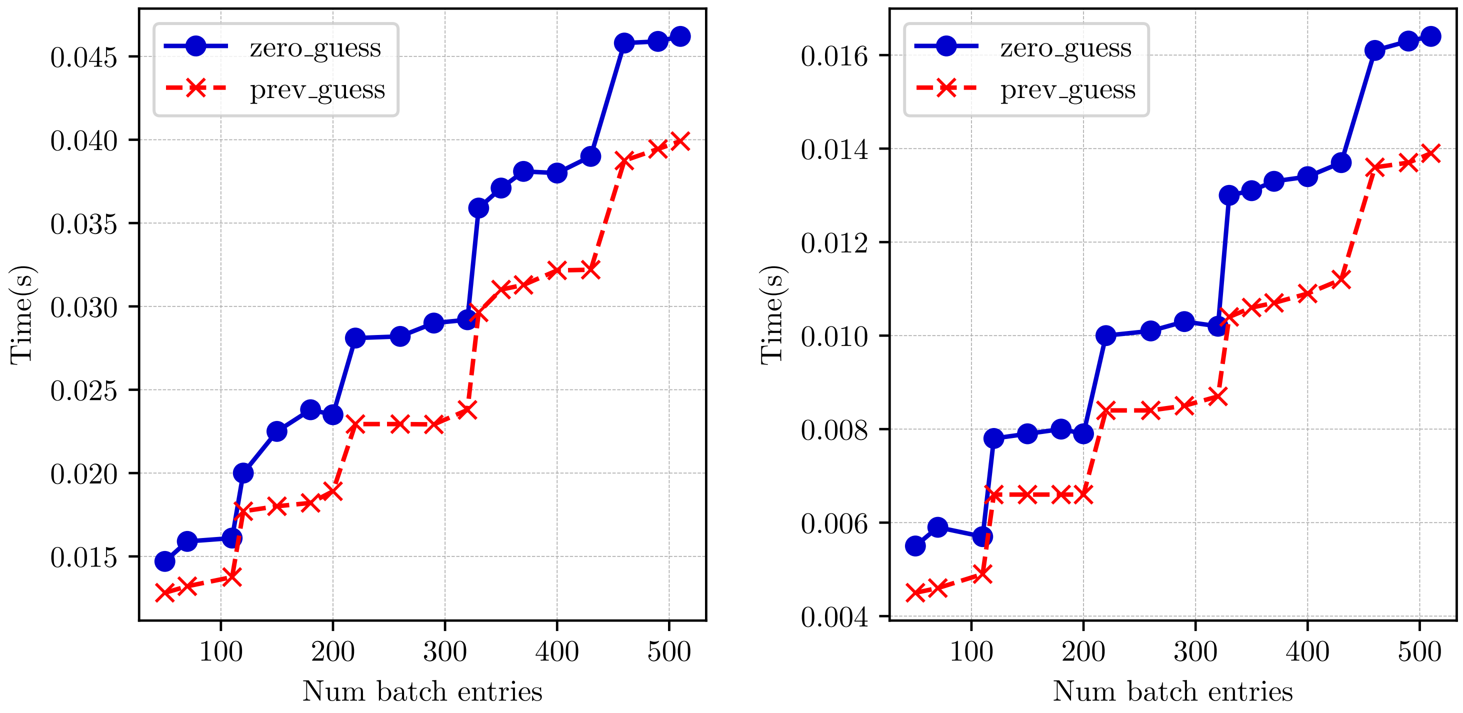


Figure 4.23: Effect of using previous Picard iteration solution as an initial guess for the subsequent iteration (cumulative over all the Picard iterations), *Left*: with **`BatchCsr`** format, *Right*: With **`BatchEll`** format.

Table 4.9: Number of iterations needed for the linear solve inside successive Picard iterations using the previous Picard iteration solution as initial guess (With **`BatchEll`** format and an absolute tolerance of 10^{-10}).

Picard iteration	#iters for electron species	#iters for ion species
0	30	5
1	28	4
2	20	3
3	16	2
4	12	2

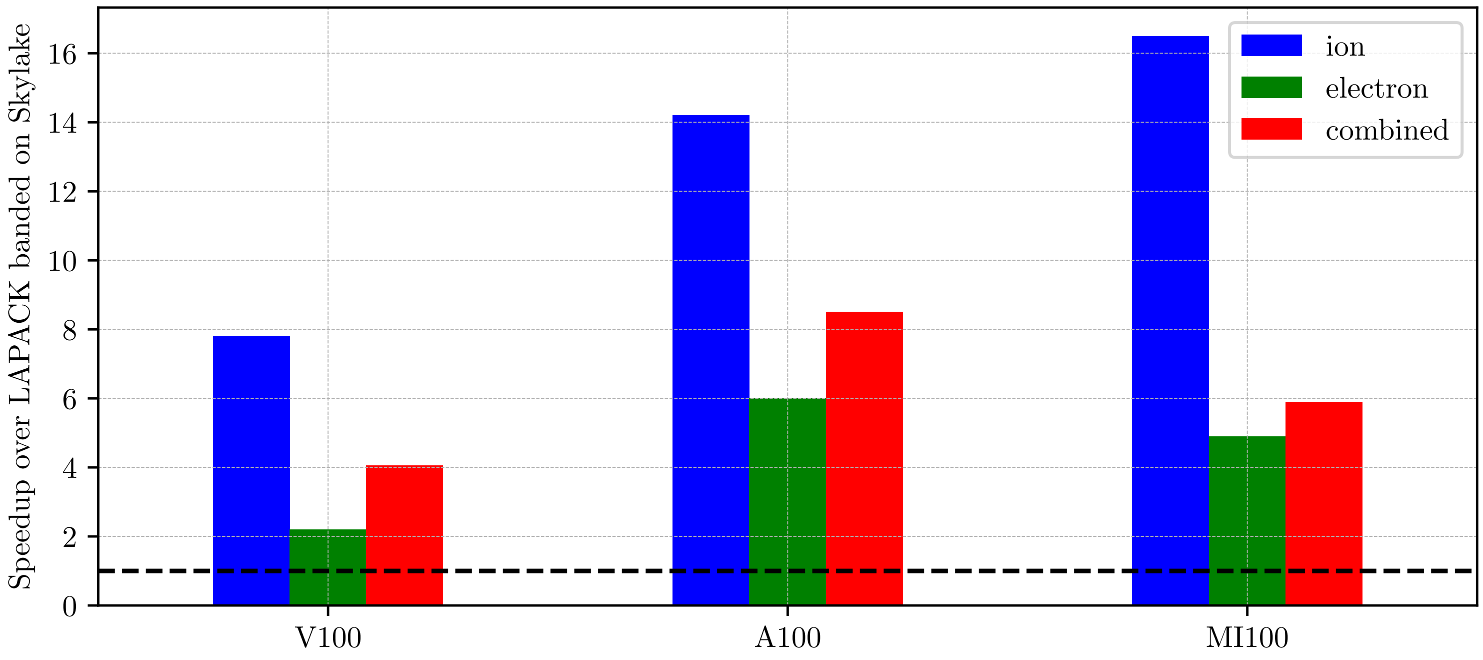


Figure 4.24: Speedup for 5 Picard iterations using batched BiCGStab on GPUs over the banded solver on CPU

In Figure 4.23, we see the time to solution for two different initial guesses. With the solution of the previous Picard iteration as the initial guess for the linear solve of the subsequent Picard iteration, we obtain a significant speedup due to a reduction in the number of linear solver iterations for the same solution quality. For the CSR format, we see speedups of $\sim 1.15\times$ to $\sim 1.25\times$ in terms of total time, while for ELL format, we see speedups between $\sim 1.2\times$ up to about $\sim 1.6\times$ compared to using a zero initial guess for the A100 GPU with the batched BiCGStab solver.

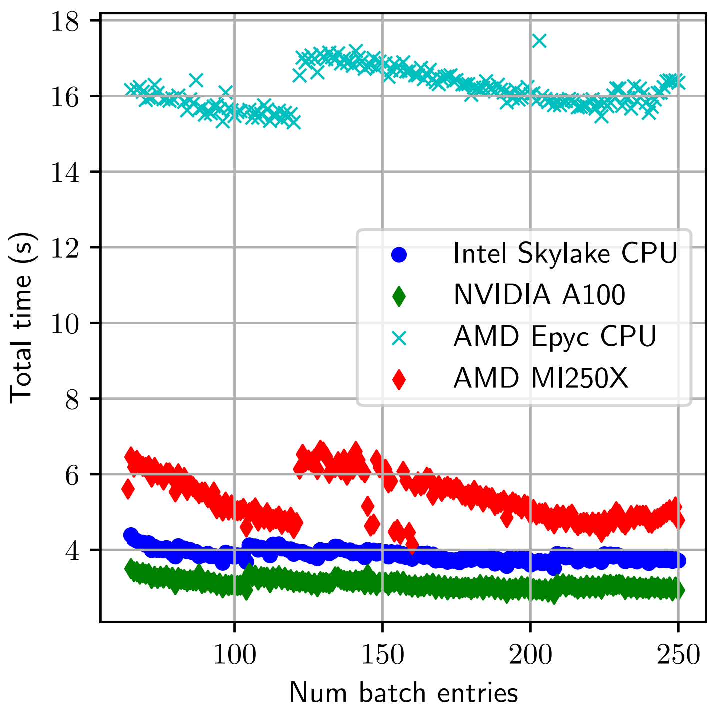
Finally, in Figure 4.24, we show the speedups obtained with the batch iterative solvers on the GPU platforms over the **`dgbsv`** solver on the Skylake CPU. The total time required for all 5 Picard iterations is used for this plot. As explained in the previous paragraph, we use the solution of the previous Picard iteration as the initial guess for the batched iterative linear solver in the subsequent Picard iteration. The **`BatchEll`** format is used for these runs.

As expected, the speedup for the ion systems is the largest because they need few iterations. For the combined batches with equal numbers of ion and electron matrices, we get effective speedups between 4x and almost 9x, depending on the GPU architecture without sacrificing the accuracy required by the application.

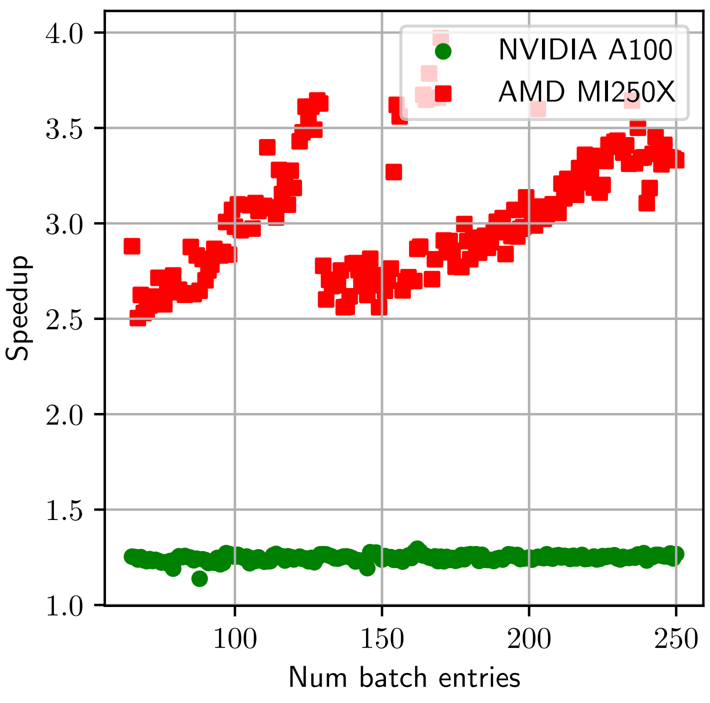
4.6.2.5 Bringing it all together in a full XGC simulation

XGC is a tokamak plasma physics code specializing in edge physics, aiming to simulate realistic geometries. It is a 5D gyrokinetic electrostatic and electromagnetic Particle-in-cell(PIC) code using an unstructured 2D mesh for the poloidal planes and structured mesh in the toroidal dimension. The code is parallelized using domain decomposition, with each processing element (MPI rank) handling one or many toroidal slices.

The collision operator in XGC is a fully non-linear multi-species Fokker-Planck-Landau operator. The velocity space is discretized for each mesh vertex with a 35×35 velocity grid. Using a 2D nine-point stencil produces matrices of the $\mathcal{O}(1000)$ rows on each mesh vertex with nine nonzeros per row, sharing a sparsity pattern across all mesh vertices. An outer Picard iteration is used to resolve the non-linear operator, and within each Picard iteration, we need to solve many independent linear systems. Due to the typically larger nature of the matrices, traditional dense direct solvers are unsuitable. XGC used the LAPACK banded solver, **`dgbsv`**, which assigned one linear system to one CPU core and parallelized the available linear systems over the number of CPU cores available.

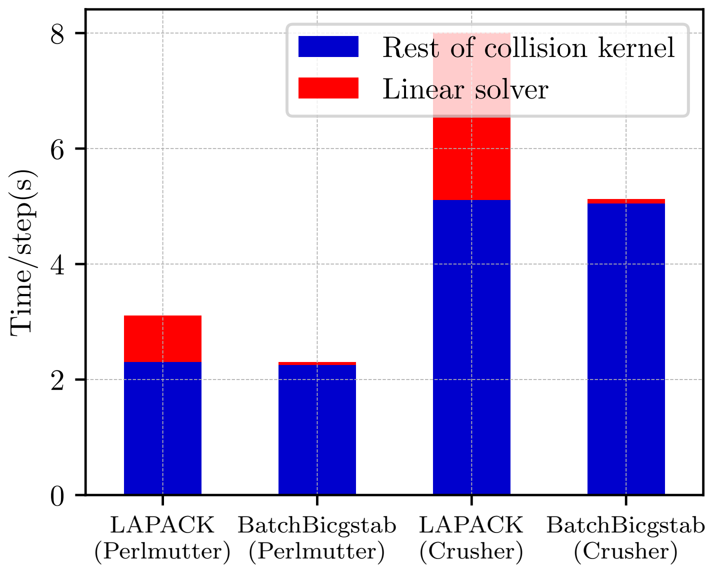


(a) Total runtime (s)

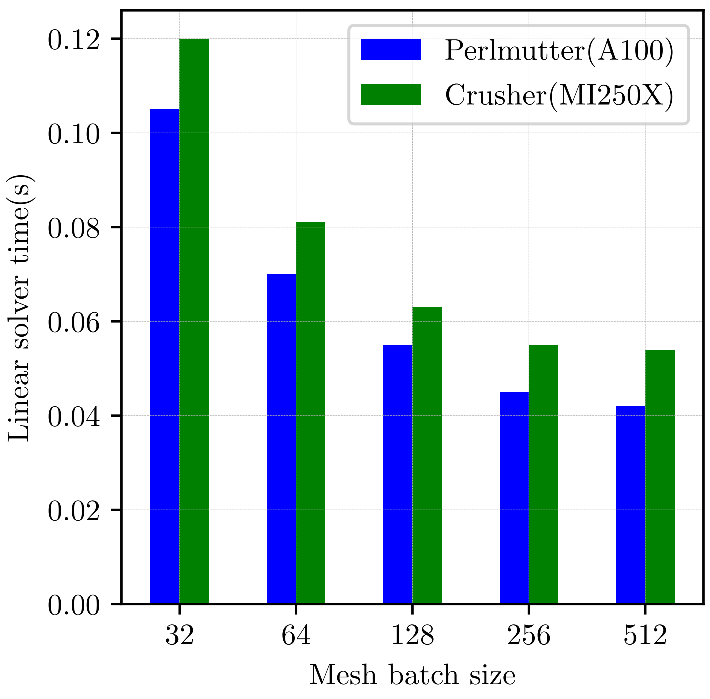


(b) Speedup

Figure 4.25: Comparing runtimes and speedup of the Ginkgo **BatchBicgstab** over the LAPACK banded solver on CPUs.



(a) LAPACK v/s Ginkgo-batched



(b) Increasing batch size

Figure 4.26: XGC run on 32 nodes of Perlmutter and Frontier TDS(Crusher) with 128 A100 and MI250X GPUs respectively

Figure 4.26 shows the timings and scaling of the DIII-D Tokamak Electromagnetic test case. The test case contains 432,000 mesh vertices, with each vertex needing to solve for two species. The velocity grid is of size (33×39) leading to matrices of size (1278×1278), each with 9 nonzeros per row as before in the proxy app. In total, there are 22.4 million particles per species per GPU. This leads to 864,000 linear solver calls for every call to the collision operator solve. The test case runs for 20 time steps, with the collisions being calculated every other step.

We observe that using the **`BatchBicgstab`** solver reduces the linear solver time by about 90% on both the Frontier TDS and the Perlmutter systems. We also observe that as we increase the batch size, which enables us to solve more systems at once, the batched solver is more efficient due to the reduction in the launch, allocation, and synchronization overheads.

4.7 Summary

In this chapter, we showed the optimality of the batched methods for problems that require the solution of multiple independent linear systems from both a theoretical and experimental perspective. We saw that the design needs careful consideration to ensure optimal usage of the compute resources while providing applications with different options, enabling them to adapt the solvers for their specific use case.

We developed optimized data structures that maximize cache usage and occupancy. Using these tailored data structures, we implemented high-performance batched iterative solvers, benchmarked them with different matrices, and observed a performance improvement of 2-10x across the benchmarking dataset compared with the vendor-provided batched solvers.

Finally, we also accelerated real-world applications such as PeleLM and XGC, providing them with a minimal overhead interface to utilize the batched solvers observing an overall speedup of around 2x for PeleLM and a speedup of 10x for the linear solution within the XGC application.

With most supercomputers providing massive amounts of parallelism through GPUs, batched solvers with their ability to harness the perfect parallelism form an excellent candidate to accelerate applications.

5 Preconditioners for Batched Iterative solvers

Iterative methods are effective solvers due to their cheap cost, configurability, and ability to incorporate user-provided initial information. One of the main criticisms of the iterative solvers, batched and monolithic alike, is their sensitivity to the problem characteristics and the lack of robustness compared to their direct solver counterparts.

For example, plain iterative solvers such as Krylov subspace methods are dependent on the system matrix of the linear system having a moderate condition number. For very ill-conditioned matrices, iterative methods may either require a lot of iterations, increasing the cost, or might not converge and require the user to fall back to direct methods.

Both the efficiency and robustness of iterative solvers can be improved through preconditioning. The term *Preconditioning* broadly describes transforming the original linear system into one with the same solution but being easier to solve using iterative methods. For large and sparse linear systems, preconditioners are often necessary for the convergence of iterative methods.

For batched iterative solvers, where we generally deal with the solution of thousands of small linear systems, preconditioners can effectively minimize the overall cost by reducing the number of iterations required. In some cases, where a few linear systems in the batch might have high condition numbers, preconditioners might be necessary.

In Section 5.1, we look at the main idea behind preconditioning and some common preconditioning strategies. We then define some popular preconditioners in literature in Section 5.2. We elaborate on our batched preconditioner implementation in Section 5.3 and evaluate the performance on a wide range of datasets in Section 5.4. We provide a brief summary in Section 5.5.

We note that some contents of this chapter have been published in [ANK+22].

5.1 Preconditioning strategies

Given some matrix $A > 0$, we have $\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$. Consider the Richardson iteration, given a certain optimal relaxation parameter, ω_{opt}. We have seen that the optimal convergence rate, ϱ_{opt} can be defined in terms of the spectral condition number,

$$\varrho(G_{Rich,\omega_{opt}}) = \frac{\kappa(A) - 1}{\kappa(A) + 1} \tag{5.1}$$

where $G_{Rich,\omega_{opt}}$ is the iteration matrix for the Richardson iteration with the optimal relaxation parameter, ω_{opt}. Therefore, if the condition number, $\kappa(A)$ is close to 1, we have very fast convergence. On the other hand, if the condition number is large, we have

$$\varrho(G_{Rich,\omega_{opt}}) = 1 - \frac{2}{\kappa(A)} + \mathcal{O}(\kappa(A)^{-2}) \tag{5.2}$$

which implies that the convergence rate is inversely proportional to the condition number.

The core principle behind preconditioning is to find some operator, $\mathcal{W}$ such that its application to the matrix, A may have

$$\hat{A} = \mathcal{W}(A) \text{ with a positive spectrum} \tag{5.3a}$$

$$\kappa(\mathcal{W}(A)) \text{ is as small as possible} \tag{5.3b}$$

which gives us convergence rate of the preconditioned iteration,

$$\varrho(G_{Rich,Prec,\omega_{opt}}) = \frac{\kappa(\mathcal{W}(A)) - 1}{\kappa(\mathcal{W}(A)) + 1} \tag{5.4}$$

We note that given an iteration matrix G, for example for the Richardson iteration, $G_{Rich} = I - A$, the preconditioned iteration matrix is expressed as $G_{Rich,Prec} = I - W^{-1}A$, where W^{-1} represents the application of the preconditioning operator.

Therefore, assuming the operator application is equivalent to multiplication by some preconditioning matrix inverse, W^{-1}, we have three different strategies of preconditioning, expressed in terms of application of the inverse to the linear system:

$$W^{-1}Ax = W^{-1}b \text{ (Left preconditioning)} \tag{5.5a}$$

$$AW^{-1}u = b \text{ with } x = W^{-1}u \text{ (Right preconditioning)} \tag{5.5b}$$

$$W_L^{-1}AW_R^{-1}u = W_L^{-1}b \text{ with } x = W_R^{-1}u \text{ (Split/Two-sided preconditioning)} \tag{5.5c}$$

5.2 State of the art preconditioners

The main objective of preconditioners is to reduce the effective condition number of the iteration matrix. In addition, from a practical standpoint, for a preconditioner to be effective, we require the preconditioner to be cheap. We look at preconditioners for SPD matrices, which occur commonly in applications, and general matrices. While we deal with preconditioners for the batched solvers, the properties and ideas for the generation carry over from the monolithic case. Hence, we elaborate on them without loss of generality in that context. In Section 5.3, we elaborate on the techniques we use to implement these in a batched fashion.

Definition 5.1 (Effective Preconditioner). *A preconditioner, defined by the application of $\mathcal{W}$ or equivalently W^{-1}, can be considered effective with*

$$\textit{Cheap application cost: } \mathcal{C}_{app}^{prec} = \mathcal{C}(G_{prec}) = \mathcal{O}(C_1 \frac{l_{no-prec}}{l_{prec}} \mathcal{C}(G_{no-prec})), C_1 < 1 \tag{5.6a}$$

$$\textit{Cheap generation cost: } \mathcal{C}_{gen}^{prec} = \mathcal{C}(\mathcal{W} :\to W^{-1}) = \mathcal{O}(C_2 l_{prec} \mathcal{C}_{app}^{prec}), C_2 < 1 \tag{5.6b}$$

with $l_{prec}, l_{no-prec}$ denoting the number of iterations for convergence with and without the preconditioner repsectively and C_1 and C_2 being some constants < 1. The first condition bounds the application cost in terms of the number of iterations required with and without a preconditioner. The second condition bounds the cost of the preconditioner generation to be some constant factor times lesser than the complete solve time with the preconditioner. A very effective preconditioner is one whose application and generation costs are very low.

5.2.1 Preconditioners for Symmetric Positive Definite matrices

We have seen that the solution of symmetric positive definite (SPD) matrices are required in a wide array of applications. We have also seen specialized solvers that are designed for efficient solutions of SPD linear systems, for example, the Conjugate Gradient method, which is a particular class of the Krylov Subspace methods.

Consider an SPD matrix A, and some preconditioner operator $\mathscr{W}$ or the equivalent preconditioner matrix W, which aims to approximate A such that $\mathscr{W}(A) \approx I$. If we assume that the preconditioning matrix is also SPD, then we have the preconditioned linear system as $W^{-1}Ax = W^{-1}b$ with left preconditioning or $AW^{-1}u = b$ with right preconditioning. We see that the coefficient matrices are not symmetric. Hence, we must use alternative strategies in the SPD case and for solvers that require SPD-ness of the linear system.

One alternative is to use the two-sided preconditioning and split the preconditioning matrix $W = LL^T$. A second and more attractive alternative comes from the observation that $W^{-1}A$ is self-adjoint with respect to the W inner product as it is SPD. Therefore,

$$(W^{-1}Ax, y)_W = (Ax, y) = (x, Ay) = (x, W^{-1}Ay)_W \tag{5.7}$$

which leads us to the modified conjugate gradient algorithm shown in Algorithm 11.

Algorithm 11 The preconditioned conjugate gradient solver.

```
1: $r \leftarrow b - Ax, z \leftarrow W^{-1}r, p \leftarrow z, t \leftarrow \mathbf{0}$
2: $\rho \leftarrow r \cdot z, \alpha \leftarrow 1, \hat{\rho} \leftarrow 1$
3: for $i < N_{iter}$ do
4:     if $|\rho| < \tau$ then
5:         break
6:     end if
7:     $t \leftarrow Ap$
8:     $\alpha \leftarrow \frac{\rho}{p \cdot t}$
9:     $x \leftarrow x + \alpha p$
10:    $r \leftarrow r - \alpha t$
11:    $z \leftarrow$ PRECOND($r$)
12:    $\hat{\rho} \leftarrow r \cdot z$
13:    $p \leftarrow z + \frac{\hat{\rho}}{\rho} \cdot p$
14:    $\rho \leftarrow \hat{\rho}$
15: end for
```

The batched preconditioned CG solver can be expressed as shown in Algorithm 6.

5.2.2 Diagonal and Block-Diagonal preconditioners

A simple and effective preconditioner that works for both SPD and non-SPD cases, is the diagonal preconditioner,

$$W = D = diag^j(A) \tag{5.8}$$

In particular, for SPD matrices, where we have the property of diagonal dominance, a diagonal preconditioner can be very effective. Equation (5.8) shows a generalization, where depending on the size j of the block to be extracted, we have a scalar Jacobi ($j = 1$) or a block Jacobi $j \in [1, n_{rows}]$ preconditioner.

Additionally, from Van Sluis [van69], we have the following theorem which shows the optimality of a diagonal preconditioner, which says that diagonal of the system matrix is the optimal diagonal preconditioner for linear solution with that system matrix.

Theorem 5.2.1 (Diagonal preconditioner optimality). *Consider a positive definite matrix, A, its diagonal $D = diag^j(A)$ and assume that each row of the matrix contains a maximum of $q = \max_{row}(n_{nz}^{row})$ nonzeros per row. Then D is an almost optimal diagonal preconditioner up to a factor of q, in the sense that $\kappa(D^{-1}A) \leq q\kappa(\Delta^{-1}A)$ for all diagonal matrices Δ.*

Proof. See [van69]. □

5.2.3 Factorization based preconditioners

We have seen the LU factorization method (Definition 2.33) where we factorize the given matrix A into lower and upper triangular factors L and U, such that $A = LU$. A direct solver consists of this factorization step, with the second step being the forward and backward solution with the two triangular factors.

In Lemma 2.2.1, we saw that the triangular solvers are fairly cheap. For cases such as preconditioners, where we only require an approximation of the application of the system matrix inverse, if we pre-compute a factorization, application of the preconditioner would involve only the relatively cheap triangular solves.

A full factorization costs $\mathcal{O}(n_{rows}^3)$, which can be prohibitively expensive for large n_{rows}. Additionally, any fill-in that occurs when factorizing (using LU or Cholesky) a sparse matrix may further increase both the application and generation cost.

Instead of a complete factorization, we can instead compute an approximate factorization, $W = \tilde{L}\tilde{U} \approx A$. This incomplete factorization is computed by controlling for sparsity, for example, enforcing zero fill-in for the factors [Saa03; Hac16].

Definition 5.2 (Incomplete LU, ILU(k)). *Given a matrix A, the sparsity pattern can be defined by a graph $\mathcal{G}(A)$. Let E be an elimination pattern, which is defined as a product of the ordered index sets, $\mathcal{I} = \{1, 2, \dots n\}$: $E \subset \mathcal{I} \times \mathcal{I}$. An incomplete LU decomposition, ILU(k) of a matrix A is then defined as $A \approx \tilde{L}\tilde{U}$, with $L_{ij} = U_{ij} = 0, \quad \forall (i,j) \notin E$. Additionally the following requirements are imposed on E:*

$$G(A^k) \subset E, \textit{ Enforce sparsity of the factors} \tag{5.9a}$$

$$(i,i) \in E \quad \forall i \in I, \textit{ Enforce non-zero diagonal elements} \tag{5.9b}$$

with $k \geq 1$ and $k \in \mathbb{Z}_+$ defining the power of the matrix whose sparsity is to be enforced. Note that $G(A^k) \subset G(A^{k+1})$.

We see that ILU(k) preconditioner, based on the Gaussian elimination, has to deal with the dependencies among rows, though these dependencies are restricted to elements in the prescribed sparsity pattern of A^k. These dependencies reduce the amount of parallelism available, thereby increasing the overall cost of the preconditioner generation. Given that the ILU preconditioner is an approximation of the LU factors, we can compute all the nonzeros in the incomplete factors in parallel and in an asynchronous fashion [CP15].

Definition 5.3 (Parallel Incomplete LU, ParILU). *Given a matrix A, the sparsity pattern can be defined by a graph $\mathcal{G}(A)$. Let E be an elimination pattern, which is defined as a product of the ordered index sets, $\mathcal{I} = \{1, 2, \dots n\}$: $E \subset \mathcal{I} \times \mathcal{I}$. A parallel incomplete LU decomposition, ParILU(k) of a matrix A is then defined as $A \approx \tilde{L}\tilde{U}$, with*

$L_{ij} = U_{ij} = 0, \quad \forall (i,j) \notin E$ and aiming to satisfy the non-zero element wise property $\sum_{h=1}^{\min(i,j)} l_{ih} u_{hj} = a_{ij}$, with $(i,j) \in E$. We see that this gives us $\#E$ unknowns to be solved with $\#E$ equations which can be written in the following explicit form:

$$l_{ij} = \frac{1}{u_{jj}}(a_{ij} - \sum_{h=1}^{j-1} l_{ih} u_{hj}) \tag{5.10a}$$

$$u_{ij} = a_{ij} - \sum_{h=1}^{i-1} l_{ih} u_{hj} \tag{5.10b}$$

Therefore these $\#E$ unknowns can be solved with a fixed point iteration $x_{p+1} = F(x_p), \quad p = 0, 1, \ldots$. Additionally, we have the same conditions on the set E as in Definition 5.2.

Convergence for this type of element based fixed point iteration has been proved [CP15]. In addition, it has been demonstrated that in practical situations, we obtain good quality approximations in a few fixed point iterations.

Definition 5.4 (Incomplete LU thresholded, ILUT(k))**.** *Given a matrix A, the sparsity pattern can be defined by a graph $\mathcal{G}(A)$. Let E be an elimination pattern, which is defined as a product of the ordered index sets, $\mathcal{I} = \{1, 2, \ldots n\}$: $E \subset \mathcal{I} \times \mathcal{I}$. An incomplete thresholded LU decomposition, ILUT(k) of a matrix A is then defined as $A \approx \tilde{L}\tilde{U}$, with $L_{ij} = U_{ij} = 0, \quad \forall (i,j) \notin E$. Additionally the following requirements are imposed on E:*

$$G(A^k) \subset E, \textit{ Enforce sparsity of the factors} \tag{5.11a}$$

$$(i,i) \in E \quad \forall i \in I, \textit{ Enforce non-zero diagonal elements} \tag{5.11b}$$

$$A_{ij} < \tau \quad \notin E, \textit{ Threshold} \tag{5.11c}$$

with $k \geq 1$ and $k \in \mathbb{Z}_+$ defining the power of the matrix whose sparsity is to be enforced. The Equation (5.11c) drops entries that are below a certain threshold value. Note that $G(A^k) \subset G(A^{k+1})$. A parallel fixed point based method can also be used in this case to generate the incomplete thresholded factors. In general, thresholding has shown to perform better due to the increased sparsity of the incomplete factors without a decrease in the preconditioner quality [ARF+19].

After these incomplete factors have been generated, the preconditioner application takes the form $LUz = r$, where we need to solve for the vector z. This requires the solution of two triangular solutions, which are similar to the ones used in the direct methods.

5.2.4 Approximate inverse based preconditioners

The central idea behind preconditioners is to approximate the inverse of the system matrix, i.e. we would like to have $\mathcal{W}(A) = I$. With the incomplete factor-based preconditioners, two successive triangular solves are required, which can be expensive if there are many dependencies. An alternative idea is to try to generate approximate inverses of the matrix A, whose application then involves a matrix-vector product, which can be very efficient on many-core architectures.

Extending the ideas from the incomplete factorization, we can generate incomplete sparse approximate inverses for L and U, and applying the preconditioner involves two successive SpMVs. This is called the Incomplete Sparse Approximate Inverse (ISAI) [BMT96]. Another option is to generate the inverse for the complete matrix, which gives us the generalized sparse approximate inverse (General ISAI) [CS98].

Definition 5.5 (Incomplete Sparse approximate inverse, ISAI)**.** *Given a matrix A, the sparsity pattern can be defined by a graph $\mathcal{G}(A)$. Let E be an elimination pattern, which is defined as a product of the ordered index sets, $\mathcal{I} = \{1, 2, \ldots n\}$: $E \subset \mathcal{I} \times \mathcal{I}$. The sparse approximate inverse of the matrix A is defined as the matrix W which minimizes:*

$$\|I - WA\|_F \tag{5.12}$$

which can be again written as algebraic equations

$$\sum_{h\in\mathcal{I}} w_{ih}a_{hj} = \delta_{ij} \quad \forall(i,j) \in \mathcal{I} \quad \textit{and}\ \delta_{ij} = \begin{cases} 1 & \textit{if } i,j \ \in \mathcal{I} \\ 0 & \textit{else} \end{cases} \tag{5.13}$$

These algebraic equations can be assembled into a matrix of the form for each row i: $W(i,\mathcal{I})A(\mathcal{I},\mathcal{I}) = I(i,\mathcal{I})$, with $I \in \mathbb{R}^{n\times n}$, being the identity. Therefore to compute the approximate inverse, we need to solve n independent linear systems $\tilde{A}\tilde{x}_i = \tilde{b}$, $\tilde{A}_i = A^T(\mathcal{I},\mathcal{I})$ and $\tilde{b} = I(\mathcal{I},i), \quad i \in \{1,\dots n\}$ [GGR+21].

5.3 Implementing efficient batched preconditioners on GPUs

Batched preconditioners, similar to the monolithic preconditioners, are composed of two main steps: Generation of the preconditioner and application of the preconditioner within the solver. The additional constraint is that the applications we consider share a sparsity pattern among the system matrices stored in the batch. As described in Chapter 4, the solver phase also contains a **generate** step that takes in the batched system matrix and performs any pre-processing steps such as parameter initialization and in case the preconditioner has been enabled, generating the preconditioner. For most preconditioners, the preconditioner generation is more expensive than the preconditioner application.

For preconditioners such as the ILU, ISAI and the Block-Jacobi preconditioners, we generate the preconditioner outside the solver apply call. This has the added benefit that we are not limited by occupancy, register, and shared memory limits imposed on the solver kernel. Additionally, in many applications, when the batched linear solver is called within an outer non-linear loop, the solver is called multiple times with the same batch system matrix. In this case, it is beneficial to only re-generate the preconditioners if the matrix has changed. An exception to this case is the scalar Jacobi preconditioner, which involves inverting the diagonal of the batch system matrix, which, being a work-balanced and cheap operation, can be done within the solver apply kernel with the additional benefit of maximizing the cache utilization during the preconditioner apply.

The second step, which is the solver apply, uses the multi-level dispatch mechanism as shown in Figure 5.1 to select the preconditioner at runtime, with the solver being templated on the preconditioner, enabling a preconditioner agnostic solver apply. The device-only preconditioner classes are then used, as shown in Listing 5.1, which contain the aforementioned preconditioner apply and generate functions abstracted across all the preconditioner classes. These can be agnostically called from within the solver apply kernel without knowledge of the concrete preconditioner type. This allows us to template the solver on the preconditioner.

Listing 5.1: Device Preconditioner class

```
template <typename ValueType>
class preconditioner final {
public:
    __device__ __forceinline__ void generate(
        size_type batch_id,
        const gko::batch_matrix_type::BatchEntry<const ValueType>&
                                                  mat_entry,
        ValueType* const __restrict__ work);

    __device__ __forceinline__ void apply(
        const int num_rows,
        const ValueType* const __restrict__ r,
        ValueType* const __restrict__ z) const;

};
```

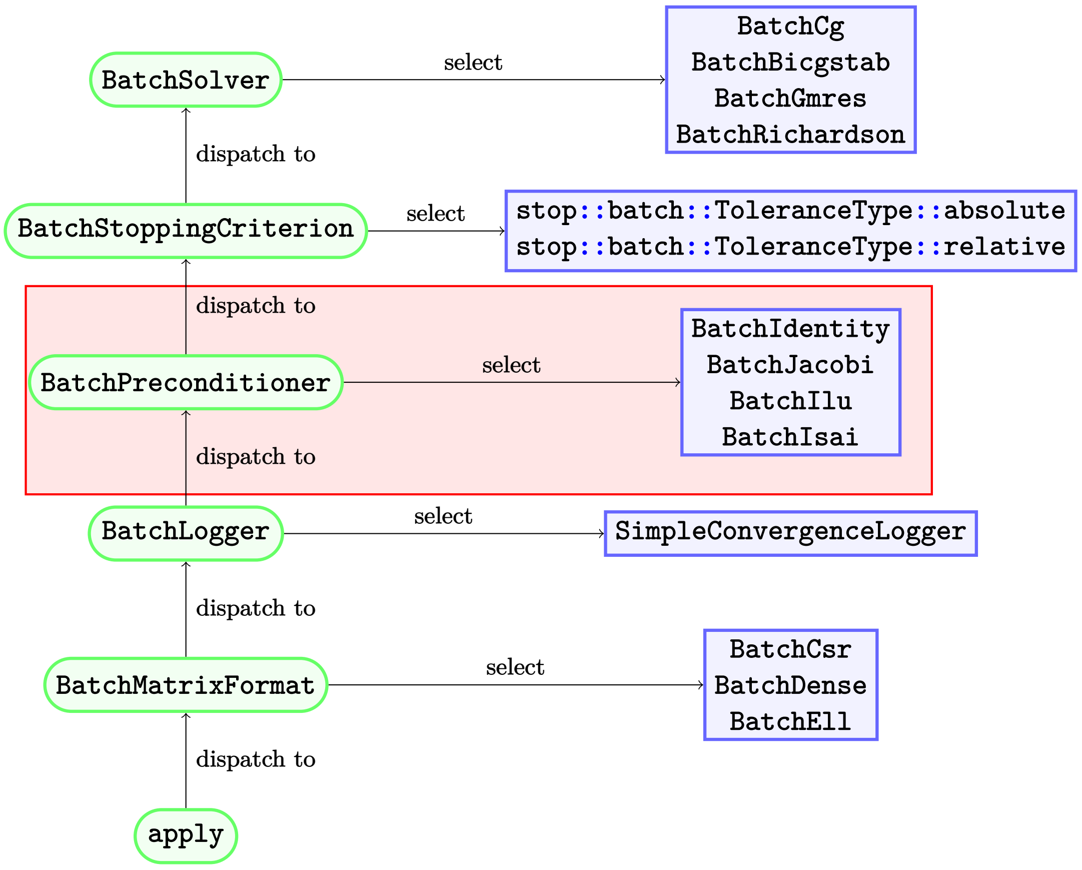


Figure 5.1: Multi-level dispatch mechanism for batched iterative solvers

The design of the preconditioners for our batched iterative solvers assumes that all matrices of the batch share the same sparsity pattern but differ in the numerical values, thereby potentially exhibiting different convergence properties. All preconditioners expect the sparse matrices of the batch to be stored in the **BatchCsr** matrix format, which stores one copy of the sparsity pattern and a contiguous array of values for the distinct matrices in the batch. This is particularly important for preconditioners that need efficient access to the sparsity pattern during both the generation and apply phase, and the **BatchCsr** forms an efficient storage format for a wide variety of sparsity patterns.

5.3.1 Batched ILU(k) preconditioner

The batched ILU preconditioner has been defined in Definition 5.2. To minimize the memory requirements, we use an in-place factorization method, where we first create a copy of the immutable batch system matrix and compute the batched L and U factors in-place. This algorithm is shown in Algorithm 12 and it follows the standard ILU(k) implementation [Doi91], but taking advantage of the shared sparsity pattern.

The batched ILU(k) algorithm performs all computations in a single kernel, assigning one workgroup to each batch entry. If the nonzeros of a row in A are not sorted in increasing column index, a sorting step is invoked to enable efficient parallel processing in the factorization step. The batched matrix must not have any missing diagonal elements (as mentioned in Definition 5.2), and any missing diagonal elements are added and set to 1, using a separate batched diagonal addition kernel. An array containing pointers to the diagonal elements of A_0 is computed beforehand and passed into the factorization kernel to avoid repeated computations.

The algorithm loops over all rows of the matrix A_b one by one. It operates on all rows below the current row in parallel, using one subgroup per row, which enables coalesced accesses to the CSR values array. Additionally, the current CSR row's elements are cached in a dense array in the local L1 cache. This avoids repeated computations searching for

Algorithm 12 The batched Exact ILU(k) preconditioning algorithm

```
INPUT: $A$
OUTPUT: Factorized (in-place) $A \approx LU$
$N \leftarrow num\_rows$
for $b = 0$ to $num\_batch\_entries - 1$ do
    for $i = 0$ to $N - 1$ do
        for $m = i + 1$ to $N - 1$ do
            $row \leftarrow 0$
            if $(m, i) \in spy(A_b^k)$ then
                $A_b(m, i) \leftarrow A_b(m, i)/A(i, i)$
                $row \leftarrow A_b(m, i)$
            end if
            for $c = i + 1$ to $N - 1$ and $(m, c) \in spy(A_b^k)$ do
                $col \leftarrow 0$
                if $(i, c) \in spy(A_b^k)$ then
                    $col \leftarrow A_b(i, c)$
                end if
                $A_b(m, c) \leftarrow A_b(m, c) - (row * col)$
            end for
        end for
    end for
end for
```

elements in the current row by their column indices, as this operation is frequently required when updating the rows below the current row.

5.3.2 Batched ParILU preconditioner

As we have seen in Definition 5.3, we can harness more parallelism while generating the ILU preconditioner using a fine-grained element-wise update approach. Generation of this ParILU factorization of a batched matrix A with all the individual entries having a typical sparsity pattern is shown in Algorithm 13. The implementation follows the standard asynchronous ParILU algorithm [CP15], but adapts it for batched matrices sharing a sparsity pattern.

The first step is to create and initialize the batched factors, L and U, which have the same sparsity pattern as that of the lower and upper triangular parts of batched matrix A respectively. The initial guess that we use for L is the lower triangular part of A with ones on the main diagonals, while the upper triangular part, including the main diagonal, is used as the initial guess for U.

To take advantage of the shared sparsity pattern and avoid similar computations while initializing the batched factors' values, we generate common patterns $\mathcal{G}(L)$ and $\mathcal{G}(U)$ for the lower and upper triangular factors, respectively, which contain pointers to the values array of matrix A_0. Following that, we fill in the values array of the individual matrices in L and U based on the stored patterns by simply reading in the values from the corresponding matrix in A.

A particular data structure, denoted in Algorithm 13 as $\mathscr{G}_{dep}(A)$ is a dependency graph for each non-zero element. This dependency graph is generated on the CPU as this process contains little parallelism and needs to be done only once for the shared sparsity pattern. To enable data locality in the factorization kernel while accessing the dependency graph, we store the dependencies for each non-zero element of A_0 in a contiguous fashion. This pre-computation of $\mathscr{G}_{dep}$ reduces global memory fetches in the factorization kernel and enables caching $\mathscr{G}_{dep}$ in the constant cache, improving cache hit rates.

The ParILU factorization kernel then employs one work-group per matrix to perform the ParILU sweeps. The sweeps are asynchronous, as they use the most recent data to update the elements without waiting for the dependencies. A separate work-item handles each element of an individual matrix. We copy the global values array of L_i and U_i to the shared memory to reduce the number of global memory writes while updating the elements.

Algorithm 13 The batched ParILU algorithm

```
INPUT: $A$
OUTPUT: $L, U$
for $b = 0$ to $num\_batch\_entries - 1$ do                                  ▷ Initialize L and U factors
    $L_b \leftarrow A_b(\mathcal{G}(L))$
    $U_b \leftarrow A_b(\mathcal{G}(U))$
end for
$\mathscr{G}_{dep}(A) \leftarrow [\ ]$
for $(i, j) \in \mathcal{G}(A_0)$ do                                  ▷ Build the dependency graph structure
    $g \leftarrow [\ ]$
    if i > j then
        g.PUSH_BACK($idx \in L_0(i, j)$)
        for $k = 0$ to $j - 1$ do
            if $(i, k) \in \mathcal{G}(L_0)$ and $(k, j) \in \mathcal{G}(U_0)$ then
                g.PUSH_BACK($idx \in L_0(i, k)$)
                g.PUSH_BACK($idx \in U_0(k, j)$)
            end if
        end for
        g.PUSH_BACK($idx \in U_0(j, j)$)
    else
        g.PUSH_BACK($idx \in U_0(i, j)$)
        for $k = 1$ to $i - 1$ do
            if $(i, k) \in \mathcal{G}(L_0)$ and $(k, j) \in \mathcal{G}(U_0)$ then
                g.PUSH_BACK($idx \in L_0(i, k)$)
                g.PUSH_BACK($idx \in U_0(k, j)$)
            end if
        end for
    end if
    $\mathscr{G}_{dep}[idx \in A_0(i, j)] \leftarrow g$
end for
for $b = 0$ to $N_b - 1$ do
    for $sweeps = 0, 1, 2 \cdots$ until convergence do                ▷ Parallel sweeps to compute the element factors
        for $nnz = 0$ to $n_{nnz}$ do
            $g \leftarrow \mathscr{G}_{dep}[nnz]$
            $sum \leftarrow 0$
            for $m = 1;\ m < len - 1;\ m = m + 2$ do
                $sum \leftarrow sum + L_b[g[m]] * U_b[g[m + 1]]$
            end for
            has_diag_dep ← modulo(g.size, 2) == 0
            if has_diag_dep == true then
                $diag \leftarrow U_b[g[g.size - 1]]$
                $L_b[g[0]] \leftarrow (A_b[nnz] - sum)/diag$
            else
                $U_b[g[0]] \leftarrow A_b[nnz] - sum$
            end if
        end for
    end for
end for
```

5.3.3 Batched ISAI preconditioner

The batched ISAI preconditioner defined in Definition 5.5 is implemented with the sparsity pattern of the incomplete factors [BMT96]. For convenience, we denote the left approximate inverse, A^{-1}_{left} by $\hat{A}$. The algorithm to compute the batched inverses for all entries is shown in Algorithm 14.

Algorithm 14 The batched ISAI algorithm.

```
1: INPUT: $A$, k
2: OUTPUT: $\hat{A}$
3: $\mathcal{G}(\hat{A}) \leftarrow \mathcal{G}(A^k)$                          ▷ Populate the sparsity pattern from the matrix
4: for $i = 0$ to $n_{rows} - 1$ do
5:     $\ell_i \leftarrow$ FIND_NON_ZERO_LOCATIONS($\hat{A}_0(i,:)$)
6:     $g_i \leftarrow$ GENERATE_PATTERN($A_0(\ell_i,\ell_i)$)
7:     $r_i \leftarrow$ FIND_LOCATION($I(i,\ell_i)$)
8:     for $b = 0$ to $n_{batch\_entries} - 1$ do          ▷ Solve the small $n \times n$ linear systems to get the inverse
9:         $\hat{A}_b(i,\ell_i)A_b(g_i) =$ GET_RHS($\ell_i$.size, $r_i$)
10:    end for
11: end for
```

In a pre-processing step, the sparsity pattern based on the power of the matrix, $\mathcal{G}(A^k)$ is computed and stored. The batched inverse is also initialized with this sparsity structure.

Next, a large number of small linear systems corresponding to each row of the matrices in the batch $\hat{A}$ are generated by extracting the target entries in the batched matrix A given by the nonzero locations of the row inverse.

All batch entries share the sparsity pattern. Therefore, unlike the ILU algorithms, for the ISAI generation, we first loop over the rows and compute all the inverse entries across the batch simultaneously. To facilitate this, we extract the row pattern in g_i, containing the pointers to the values of the batch entries, and fill the right-hand side in ℓ_i. This then gives us small linear systems that we can solve to compute the entries of the approximate inverse. These linear systems are solved with a triangular solve or a general solve, depending on the input sparsity pattern, $\mathcal{G}$; If it is triangular, then a triangular solver is used, and a general solver is used otherwise.

These smaller linear systems are stored in a row-major dense format for efficiency and contiguous access. Given that we compute the SAI of one batch entry with one thread block, we solve one small linear system within one subgroup. If a row has more nonzero elements than the number of work-items in a subgroup, for example, more than 32 on a CUDA device, we need to assemble these rows in a separate excess system, which we do in a CSR format(for each batch entry). This excess system is then solved with a GMRES solver.

In addition to generating an approximate inverse for the given system matrix A, the ISAI algorithm can also be used to generate an approximate inverse of a triangular system. Thus, it can be combined with an ILU-based preconditioner to solve the generated triangular factors.

5.3.4 Batched sparse triangular solvers

Sparse triangular solvers form an essential component of the ILU, ParILU, and ISAI preconditioners, being used in either the generation phase of the preconditioners (for ISAI based on ILU) or for the application phase. Triangular solvers consist of two main components: a symbolic phase that consists of the generation of a dependency graph, and a numerical phase that uses the numerical values and the dependency graph to solve the triangular systems.

Due to dependencies, synchronization can harm the efficient parallelization of sparse triangular solves. In the batched sparse triangular solve implementation, we use a busy-waiting loop inside a thread block, where threads are scheduled continuously. However, we only perform work once all the previous dependencies are completed. This approach has been proposed in [LLH+16] and fits the requirements of batched preconditioner application. This sync-free approach requires independent thread scheduling, available only on NVIDIA GPUs (Volta and later). For other architectures, such as AMD, we use the standard subgroup synchronous approach [ZSL+21].

Algorithm 15 The batched sparse work-item independent, sync-free triangular solve

```
INPUT: $T, u$
OUTPUT: $y = T\backslash u$
for $b = 0$ to $n_{batch\_entries} - 1$ do
   for $i = 0$ to $n_{rows} - 1$ do
      sum ← 0
      for nnz = $T_b(i,:)$.nnz_start to $T_b(i,:)$.nnz_end do
         col_index ← $T_b$.GET_COL_INDICES(nnz)
         while IS_FINITE($y$.value_at(nnz)) do          ▷ Busy wait for dependencies to be resolved
         end while
         sum += $T_b$.value_at(nnz)*$y_b$[col_index]
      end for
      $y_b$[i] = $u_b$[i] - sum / $T_b$.(value_at($T_b(i,:)$.nnz_end)          ▷ Update solution
   end for
end for
```

5.4 Evaluating the performance of batched preconditioners

For the performance evaluation, we run Ginkgo's batched iterative solvers with different preconditioners on an NVIDIA A100 GPU with a main memory bandwidth of 1555 GB/s, with 108 SMs, L1 cache of 192 KB per compute unit and an L2 cache of 40MB per compute unit. The theoretical peak performance for double precision is 9.7 TFlops/s. We perform our experiments on the HoreKa cluster with gcc-8.3.0 and with CUDA-11.4 [1].

To showcase the need, effectiveness, and scalability of the batched preconditioners, we consider problems from different applications. First, we focus on a stencil matrix that is characteristic of many particle and stencil-based codes. Then, we turn to benchmark problems from the SuiteSparse matrix collection. Finally, we look at problems from the Pele application to evaluate the effectiveness of the batched preconditioner in a real-world application.

All the experiments use a zero initial guess $x^0 = 0$. The residual stopping criterion is set to $\varepsilon = 10^{-12}$, which implies that we iterate until the residual $\left\|b - Ax^k\right\|$ in iteration k is 12 orders of magnitude smaller than the initial residual $\|b - Ax^0\|$. This reflects high accuracy requirements and a fair but pessimistic comparison against batched direct solvers. Many applications, e.g., using nonlinear solvers, can tolerate less accurate solution approximations. In the experiments with stencil and SuiteSparse matrices, we use an all-one right-hand side vector $b = 1$ for the linear systems.

5.4.1 Evaluating performance: Stencil matrices

We evaluate the performance of the 3-point stencil discretization of a 1D Laplace problem and assess the runtime of the preconditioned batched iterative solvers for increasing batch size. The size of each linear system is 64. To converge to a relative residual stopping criterion of 10^{-12}, an un-preconditioned BiCGStab solver needs 16 iterations, while an ILU-preconditioned BiCGStab completes after the first iteration. These iteration counts are noted in Table 5.1.

In Figure 5.2 (left), we increase the number of batch entries from 8 to 262,144. We note that the runtime in this graph reflects only the iteration time, that is, the BiCGStab iterative solver time and the preconditioner application time. The preconditioner generation is excluded.

The implementation assigns one thread block to each linear system in the batch. As a result, the iteration runtime is constant until the batch size exceeds the number of streaming multiprocessors (SMs) on the A100, and all SMs are busy executing a thread block. For larger batch sizes, the runtime increases with the number of linear systems in the batch.

For this problem, due to its cheap cost of application, we observe that the ISAI preconditioner wins over all other preconditioners. The scalar Jacobi preconditioner is also very effective due to the fast application of the inverse diagonal. The ILU and ParILU preconditioners suffer from the comparatively expensive triangular solves. These sophisticated preconditioners are more effective for problems with higher condition numbers, as we will see later. We also observe that the preconditioned BiCGStab variant is faster than the CUBLAS batched direct solver.

[1]This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research

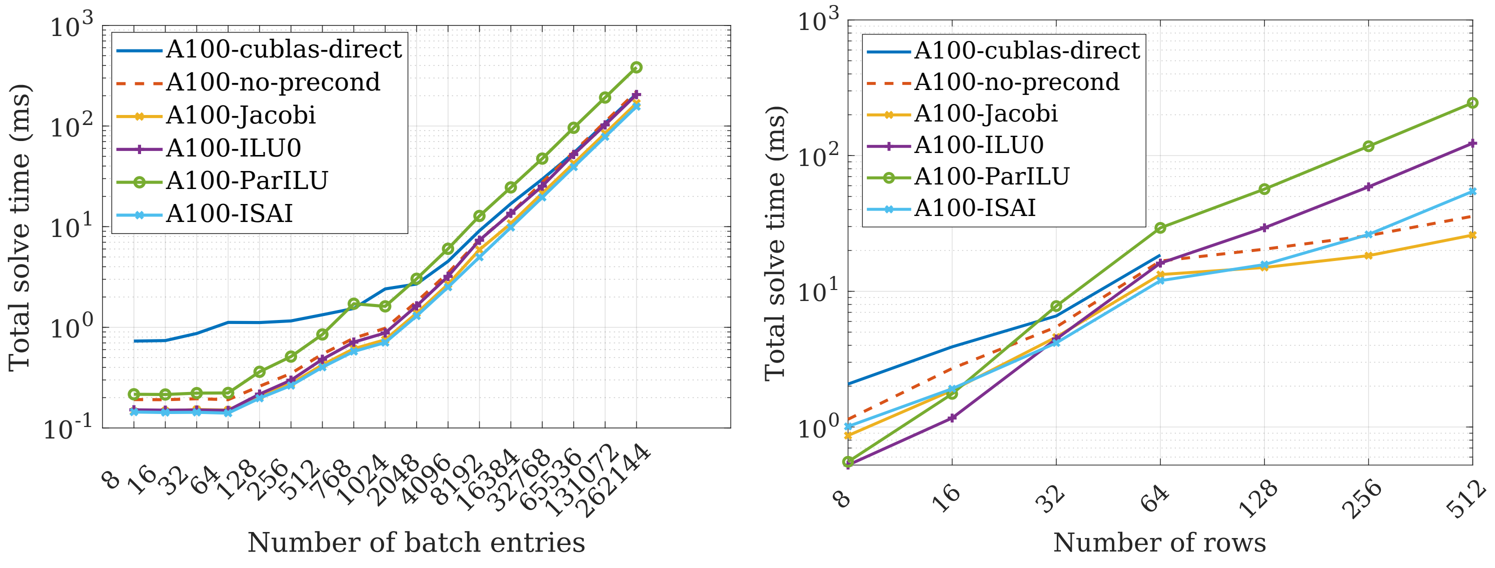


Figure 5.2: (Preconditioned) BiCGStab runtime for the 3-pt stencil matrix. Left: Fixed number of rows (64), Right: Fixed number of batch entries (20000)

In the next experiment, we fix the batch size to 20,000 and increment the size (number of rows) of the individual problems in the batch, as shown in Figure 5.2 (right). We note that the CUBLAS batched direct solver runs into memory issues (out of memory on the NVIDIA A100) for problems larger than 64 rows. For smaller problems, ILU(0) and ParILU(0) are very effective, while for larger problems, the cost of the exact triangular solves can not be compensated by faster convergence. For large problems, the inexpensive Jacobi preconditioner is the best choice. Nevertheless, the ISAI preconditioner, with its cheap application (involving only a SpMV), is also very effective.

5.4.2 Evaluating Performance: SuiteSparse matrix collection

To evaluate the performance of matrices from real-world applications, we turn to matrices selected from the SuiteSparse matrix collection [DH11]. The matrices we consider are listed in Table 5.1, with some key properties, in particular the number of rows and the number of nonzeros, along with the number of iterations required for convergence with a `BatchBicgstab` solver without a preconditioner and with different preconditioners we consider.

In Figure 5.3, we visualize the `BatchBicgstab` iteration counts for the relative residual stopping criterion 10^{-12} and an upper limit of 500 iterations. We consider configurations that do not converge within 500 iterations as failing. We note that for several problems, robust preconditioners are needed to ensure convergence, i.e., a non-preconditioned BiCGStab or a simple scalar Jacobi-preconditioned `BatchBicgstab` fails to converge. We also see that for some matrices, such as the `bcsstm02`, most preconditioners enable convergence within 1 iteration; hence, the bars are not visible.

To evaluate performance, we duplicate the matrix from the SuiteSparse collection, consider 10^3 batch entries, and run each solver with 3 warm-up iterations and ten repetitions to minimize noise.

In Figure 5.4 (top), we visualize the time needed to generate the preconditioners for the different problems. The preconditioner generation time generally increases with the size of the problem and the number of nonzeros in the batch. As expected, the ILU(0) preconditioner generation is significantly more expensive than the scalar Jacobi preconditioner generation. ParILU uses the fine-grained ParILU algorithm to approximate the incomplete factors of an ILU(0) preconditioner via fixed-point iterations. We use 20 fixed-point sweeps of the ParILU algorithm to generate good approximations of the ILU factors. In this setting, the ParILU algorithm is generally faster than the traditional ILU(0). Even though the ISAI preconditioner generation can provide a lot of parallelism during application due to the operation and memory overhead (with the need to store an explicit approximate inverse of the sparse matrix), we see the runtime is not always lesser. Increased memory requirements lead to increased pressure on the L1 cache, reducing SM

Table 5.1: Matrix and matrix set characteristics and iteration counts for the BiCGStab iterative solver using different preconditioners.

	size	nonzeros	*No Precond*	*Jacobi*	*ILU(0)*	*ParILU*	*ISAI*
1D Laplace							
3pt-stencil-64	64	190	16	11	1	1	6
SuiteSparse							
LFAT5	14	46	80	33	7	8	16
bcsstm02	66	66	11	1	1	1	1
LF10	18	82	351	–	38	34	–
Trefethen_20	20	158	19	8	5	5	6
pivtol	102	306	16	13	2	2	7
bfwb62	62	342	30	15	6	6	9
olm100	100	396	–	–	36	98	26
bcsstk22	138	696	493	229	43	42	95
cage6	93	785	–	12	4	4	7
ck104	104	992	112	118	13	15	164
494_bus	494	1666	–	–	81	81	–
mesh3em5	289	1889	14	13	1	1	10
mhdb416	416	2312	–	37	2	2	41
bcsstk05	153	2423	325	124	32	32	149
steam1	240	3762	–	–	3	3	–
PeleLM							
isooctane	144	6135	–	38	3	4	–

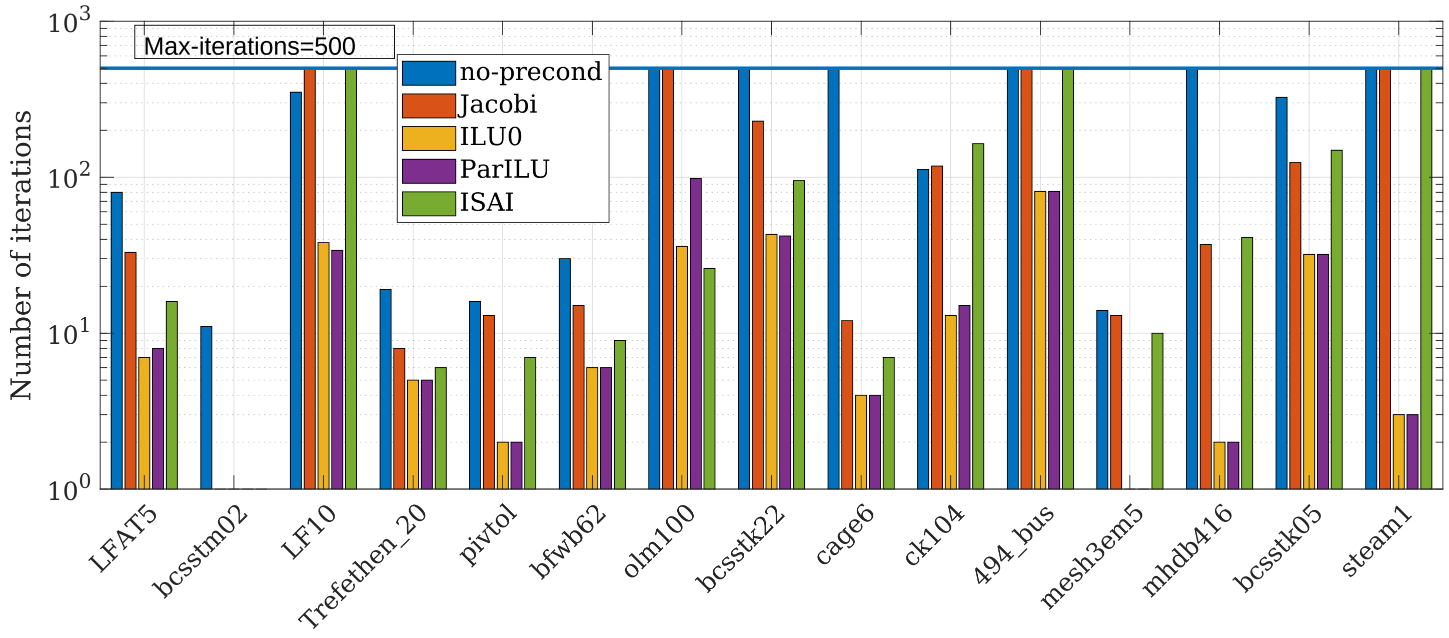
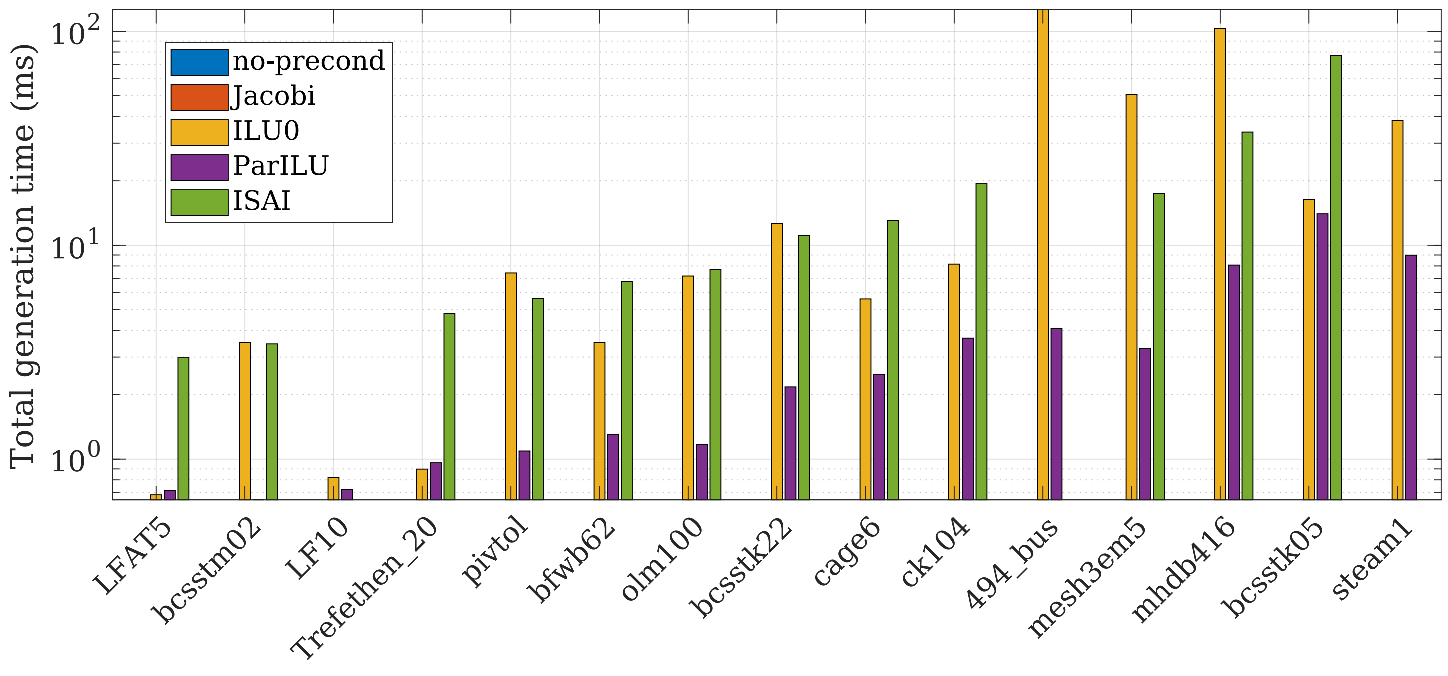


Figure 5.3: SuiteSparse matrix iteration count for the preconditioned BiCGSTAB solver.



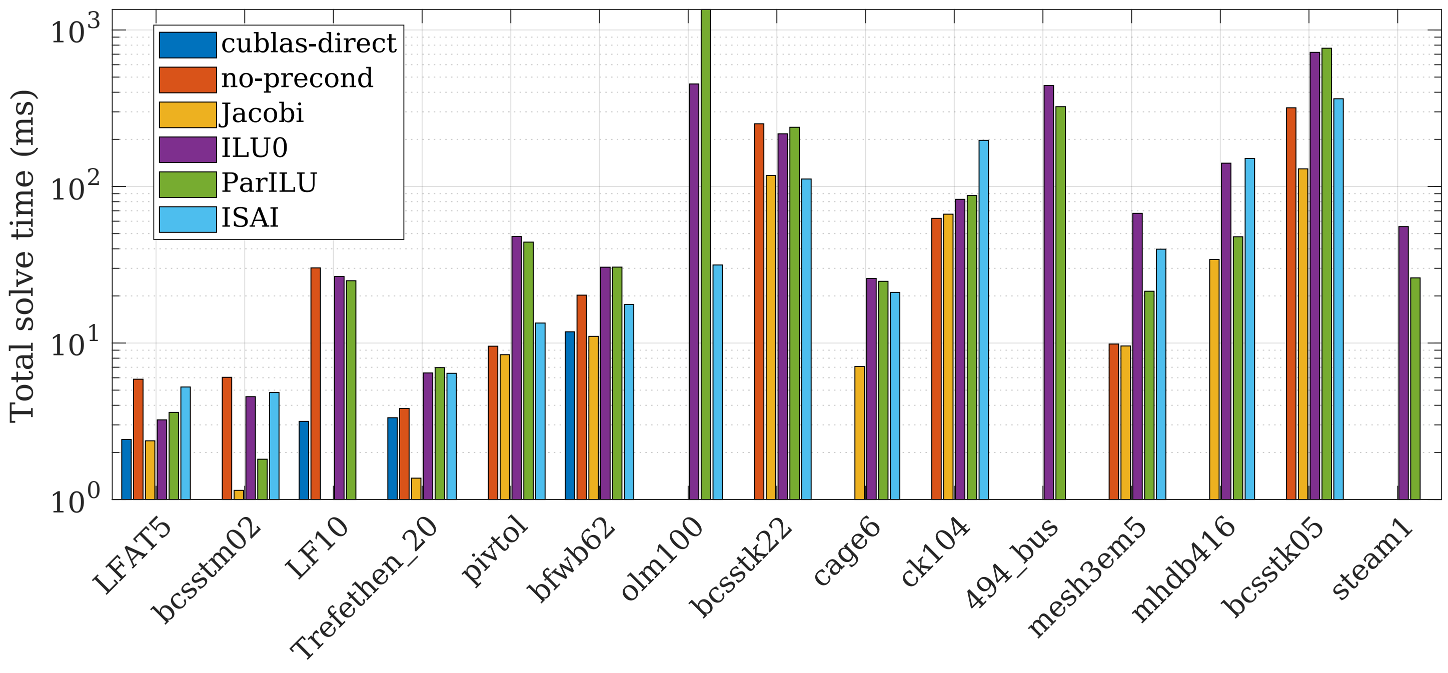


Figure 5.4: Using batched preconditioners for SuiteSparse benchmark matrices: Preconditioner generation time(top). Total solve time (including preconditioner generation) the preconditioned batched BiCGStab solver (bottom).

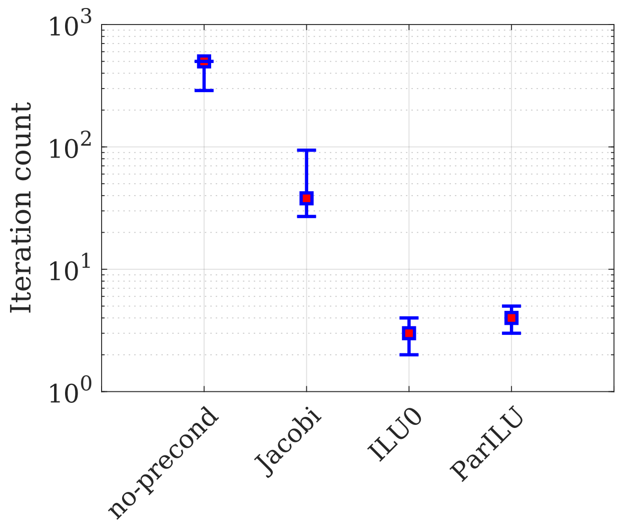


Figure 5.5: Variation in the iteration count for the isooctane problem for different preconditioners, see Table 5.1 for median iteration counts.

utilization. Thus, we observe that the generation cost of the ISAI is typically higher in comparison to ParILU. Finally, we note that the scalar Jacobi preconditioner's generation time is not visible here because its generation is carried out within the iterative solver kernel and not in a separate generation kernel as it only involves inverting the diagonal.

Figure 5.4 (bottom) visualizes the total solve time of the preconditioned BiCGStab solver for the batched problems containing 10,000 batch entries. Here, the total time to solution also includes the preconditioner generation time. The batched direct solver from cuBLAS works only for small problems but exceeds the A100 GPU memory capacity for larger problems. For the problems where it is applicable, the cuBLAS batched direct solver is competitive but not faster than all the preconditioned batched iterative solvers. For larger problems, there is no clear winner. Matrices such as olm100, 494_bus, and steam1 need the ILU type preconditioners as they do not converge with the scalar Jacobi preconditioner. But for matrices such as bcsstm02 or Trefethen_20, the fast preconditioner application makes the Jacobi preconditioner the preferred choice. In some cases, the ISAI preconditioner can be effective (olm100), while in other cases, the ILU and triangular solve-based preconditioners are more effective (mhdb416). Therefore, we can conclude that the choice of preconditioner is heavily dependent on the problem, and for poorly conditioned matrices, it is essential to use robust preconditioners such as the ILU or ParILU.

5.4.3 Application: Reaction problem from the Pele application

Finally, we consider the matrices arising from the computational chemistry application that was initially solved with batched iterative solvers [AKN+21]. Its characteristics are shown in Table 5.1. This reaction mechanism has 72 batch entries, all of whom share the same sparsity pattern but differ in the numeric values. As a result, preconditioned iterative solvers may exhibit different convergence properties for individual problems. Figure 5.5 visualizes the variation of the BiCGStab iteration counts for the distinct problems. We emphasize that the preconditioned iterative solvers are deterministic in that for a single problem, the iteration count stays constant across multiple runs. The error bars reflect the variations in the convergence for the distinct systems in the batch. All solver configurations have variations in the iteration count for the distinct problems. Figure 5.6 visualizes the overall solve time of the preconditioned batched BiCGStab solver for an increasing number of batch entries. The larger batches are generated by duplicating the 72 matrices. The solve time includes the preconditioner generation and the preconditioned batched iterative solver execution. The results show that preconditioning helps reduce the overall solve time, with Jacobi and ParILU being the best choices for this problem.

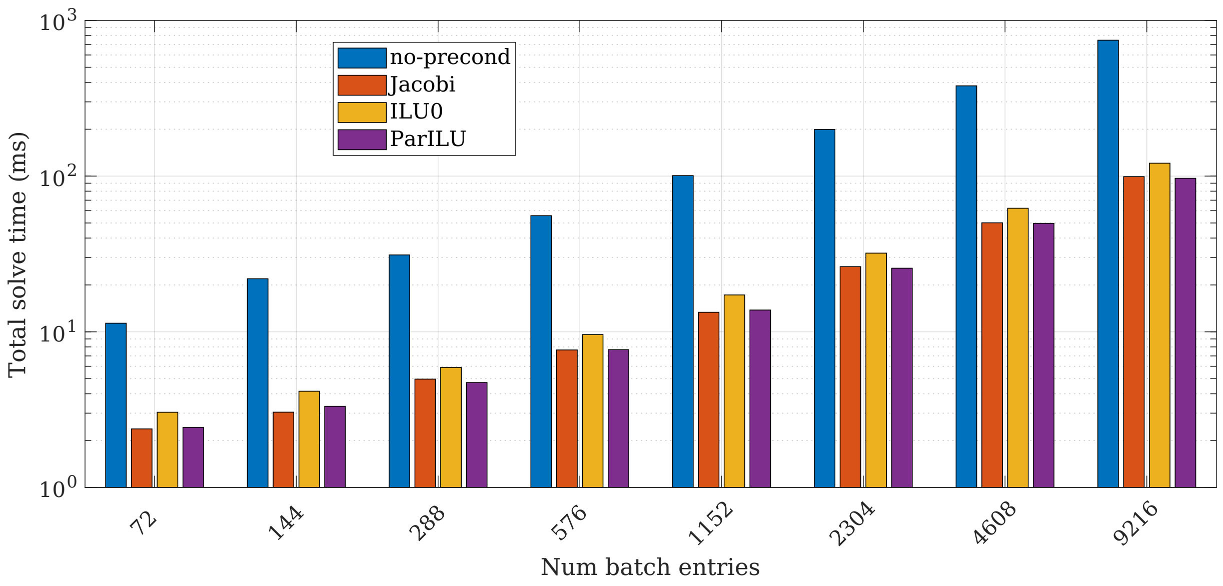


Figure 5.6: Total solve time for increasing batch sizes for the isooctane problem.

5.5 Summary

The efficiency of iterative solvers can be improved with preconditioning. We looked at the different types of state-of-the-art preconditioners, such as Jacobi, ILU, ParILU, and ISAI and studied their properties. We extended these monolithic preconditioners and incorporated them in our batched solvers for cases where all the batch entries share a sparsity pattern.

We implemented these batched preconditioners, namely **`BatchJacobi`**, **`BatchIsai`**, and **`BatchIlu`** for GPUs, along with a tailored batched sparse triangular solver to ensure efficient preconditioner application. Using datasets from SuiteSparse and real-world applications, we benchmarked the performance of these batched preconditioners. We showed that using these preconditioners can reduce the solver iteration count and the total time to solution. Using the batched preconditioners, we solved a more extensive set of matrices, including ones with large condition numbers.

For matrices from real-world applications, batched preconditioners can reduce the imbalance in the iteration counts by reducing the number of iterations required for convergence. This can enable the GPU runtime to optimize its scheduling (mapping linear systems to SMs), thereby reducing the overall time to solution.

6 Asynchronous Iterative methods

The bulk-synchronous nature of our state-of-the-art algorithms brings us a unique challenge: overcoming communication and synchronization bottlenecks to scale our methods to exascale and beyond. This challenge has been identified in the literature [AHL+11; Saa18]. The most popular approaches to overcome the communication and synchronization challenges are:

1. (Task-based approach) Derive or infer a task graph from the data dependencies, maximizing parallelism and minimizing synchronization while balancing the load by assigning data-independent computations to different parallel processing elements.
2. (Data-asynchronous approach) Relax or avoid synchronization needed for the data exchange, allowing computation on possibly stale data while availing no synchronization bottlenecks.

Task-based approaches have been well-explored with many libraries using different techniques to build the task graph, distribute and manage the compute resources, and balance the workload on parallel computing units [ATN+11; BBD+13; JBY+15]. While task-based approaches can alleviate some synchronization-related bottlenecks, they are limited by the critical path in the task graph. They do not modify the inherent algorithm, which may have data dependencies that enforce synchronization and serialization.

The data-asynchronous approach, on the other hand, modifies the algorithm, relaxing the inherent data dependencies, either hiding the synchronization bottlenecks or removing them altogether. This approach also provides advantages in terms of being able to handle workload imbalances compared to the traditional static workload distributions, where synchronization bottlenecks can cause large workload imbalances [BT15].

Iterative methods, in particular, form an attractive domain for the application of these data-asynchronous methods. Due to the successive approximation nature, iterative methods can be resilient to partial data stagnation. The seminal work of Chazan and Miranker [CM69] first explored these asynchronous iterative methods in the context of chaotic relaxations, where they proved convergence for contracting operators. A few studies have since studied asynchronous variants of iterative methods, in particular for domain decomposition methods such as Schwarz methods [Mag13; YCB+19] and also for relaxation-based methods [CFS20].

A particular challenge is the theoretical convergence analysis of asynchronous methods. Due to the non-deterministic nature of these iterative methods, traditional numerical analysis techniques need to be combined with probabilistic analysis methods to infer convergence, consistency, and optimality [Str97; Str02; FS00]. An alternative approach has been to view the dependencies between the components in the asynchronous iteration as a random directed acyclic graph (DAG), and estimate the convergence rate by looking at the statistical properties of the DAG [HD18]. There have also been efforts in the direction of bounding the convergence rate through randomization, in particular for simpler asynchronous variants such as the Asynchronous(Randomized) Gauss-Seidel method [ADG15]. Of particular note is the differentiation between consistent and inconsistent reads (See Avron et.al [ADG15] for more details), which lead to different bounds, with enforcing read consistency leading to better convergence bounds.

In Section 6.1, we define an asynchronous iteration and study its convergence. We develop a probabilistic model in Section 6.2 and perform an empirical analysis of the asynchronous Richardson iteration. We showcase our asynchronous Richardson iteration implementation on the GPU in Section 6.3, including a detailed analysis of the performance and convergence behavior of the method. Finally, we provide a brief summary in Section 6.4.

We note that some contents of this chapter have been published in [TNC+22].

6.1 Convergence and optimality

To analyze the asynchronous iterative methods, we need to develop a framework that enables analysis with stale data. Chazan and Miranker [CM69] and Baudet [Bau78] developed an elegant framework that is general enough to analyze these asynchronous methods. We first define an asynchronous iteration and then look at the conditions necesssary for convergence.

Definition 6.1 (Asynchronous iteration)**.** *Given an operator $\mathcal{F}$ from $\mathbb{R}^n \to \mathbb{R}^n$, an asynchronous iteration from $\mathcal{F}$, given an initial guess vector x_0, is a sequence of successive approximations, $x_k, k = 0, 1, \cdots, n$ of vectors in $\mathbb{R}^n$ defined by*

$$x_{k+1}^l = \begin{cases} x_k^l & \text{if} \quad l \notin \mathcal{I}_k \\ f_l(x_{s_1(k)}^1, \cdots, x_{s_n(k)}^n) & \text{if} \quad l \in \mathcal{I}_k \end{cases} \tag{6.1}$$

where $\mathcal{I}_k \subseteq \{1, ..., J\}$ is the sequence of nonempty subsets, representing the iteration subsets, the data is gathered from, with $\mathcal{S}_k = s_1(k), \cdots, s_n(k)$, the sequence of elements in $\mathbb{N}_0^J$ representing the delays. In addition, the sets $\mathcal{I}$ and $\mathcal{S}$ are subject to the following conditions

$$s_j(k) \leq k - 1 \qquad \forall\ j, k \tag{6.2a}$$

$$lim_{k\to\infty} s_j(k) = \infty \qquad for\ j = 1, ..., J \tag{6.2b}$$

$$\text{the sets } \{k \mid j \in \mathcal{I}_k\} \text{ are unbounded for } j = 1, ..., J \tag{6.2c}$$

An asynchronous iteration corresponding to $\mathcal{F}$ starting with x_0 and defined by the sets $\mathcal{I}$ and $\mathcal{S}$ will be denoted by $(\mathcal{F}, x_0, \mathcal{I}, \mathcal{S})$.

The conditions in eq. (6.2a) to eq. (6.2c) provide the necessary conditions for convergence of an asynchronous iteration. Equation (6.2a) states that only previously computed iterates can be used in the computation. Equation (6.2b) states that the latest update available locally in the processing element has to be used, and Equation (6.2c) guarantees that each component is continuously updated.

Theorem 6.1.1 (Contracting operator convergence)**.** *Given a contracting operator, $\mathcal{L}$ on a closed subset D of $\mathbb{R}^n$ and given that $\mathcal{L}(D) \subset D$, then any asynchronous iteration $(\mathcal{L}, x_0, \mathcal{I}, \mathcal{S})$ converges to a unique fixed point of $\mathcal{L}$ in D.*

Proof. See Baudet [Bau78]. □

As we are mainly concerned with linear operators, restricting $\mathcal{L}$ to be a linear operator instead of the larger class of contracting operators, we have the following result from Chazan and Miranker [CM69].

Theorem 6.1.2 (Linear operator convergence)**.** *An asynchronous iteration, $(\mathcal{L}, x_0, \mathcal{I}, \mathcal{S})$ with the iteration matrix T*

(a) *converges if there exists a positive vector, v and a scalar $\alpha < 1$ such that $|T|v \leq \alpha v$.*

(b) *The spectral radius of the iteration matrix, T, $\rho(|T|) < 1$.*

Proof. See Chazan and Miranker [CM69]. □

An important aspect to note is that the condition for convergence of the asynchronous iteration, $\rho(|T|)$ is stricter than that for the synchronous version $\rho(T)$. As observed from a practical standpoint by others, for example in [CFS20] this seems to be a pessimistic estimate for asynchronous iterative methods, and there have been cases where the asynchronous method converges even when the synchronous version does not converge.

6.1.1 Convergence of the asynchronous Richardson iteration

The asynchronous Richardson method is the asynchronous variant of the well-known Richardson method defined in Chapter 2. We have the following corollary from Chazan and Miranker [CM69] that gives the necessary conditions for convergence for the asynchronous Richardson iteration.

Theorem 6.1.3 (Asynchronous Richardson). *Consider the matrix A with the splitting $A = M - N$. Let $M = A_{ii}$ and $N = A_{i\neq j}$. Then the asynchronous Richardson iteration $(T_\omega, x_0, \mathcal{I}, \mathcal{S})$ with the iteration matrix, $T_\omega = I - \omega M^{-1}A$ converges if*

(a) $\rho(|T_\omega|) = \alpha < 1$

(b) $0 < \omega < \frac{2}{1+\alpha}$

Proof. We aim to show that $|T_\omega|v \leq \beta v, \beta < 1$. From theorem 6.1.2, we know $\exists v > 0$ s.t $|T_1|v \leq \alpha v$. Therefore,

$$|T_\omega|v \leq (I(1-\omega) + \omega|T_1|)v \leq |(1-\omega)|v + \omega\alpha v = (|1-\omega| + \omega\alpha)v$$

Let $\beta_\omega = (|1-\omega| + \omega\alpha)$. Then if $1 \leq \omega \leq \frac{2}{1+\alpha}$,

$$\beta_\omega = \alpha\omega + (\omega - 1) = (1+\alpha)\omega - 1 < 1$$

Also given that $\alpha < 1$, if $0 \leq \omega \leq 1$,

$$\beta = \alpha\omega + (1-\omega) = (-(1-\alpha)\omega + 1) < 1$$

□

Therefore, the asynchronous Richardson iteration converges if the system matrix, A satisfies any one of the following properties:

1. Symmetric and strictly diagonally dominant.
2. Irreducibly diagonally dominant.
3. Symmetric positive definite with non-positive off-diagonal elements.

Table 6.1: Probability distributions

Probability distribution	Probabilistic distribution function	Mean/Expected value ($E[X]$)
Half-Normal	$f(x\|a,b) = \sqrt{\frac{2}{\pi}}\frac{1}{b}e^{-\frac{1}{2}(\frac{x-a}{b})^2}; x \geq a$	$\mu = a + b\sqrt{(\frac{2}{\pi})}$
Exponential	$f(x\|\lambda) = \lambda e^{-\lambda x}$	$\mu = \frac{1}{\lambda}$

6.2 A Probabilistic Model

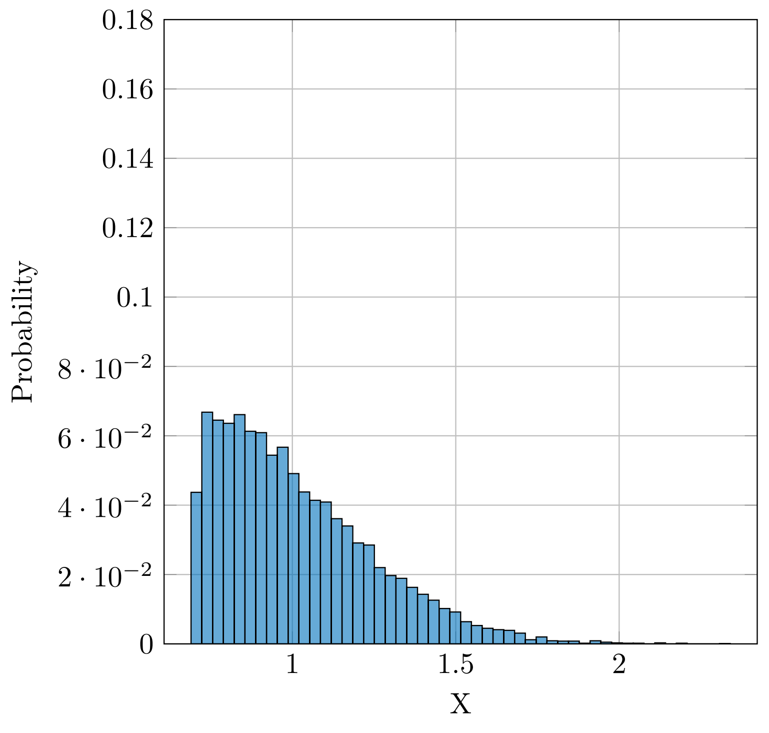
Asynchronous methods are unique in that due to the lack of synchronization, they do not have a pre-determined sequence of operations, even between two separate runs of the same solver. Hence, the usual deterministic techniques cannot analyze these methods, and one must use a stochastic or probabilistic approach. To study these asynchronous methods, we develop a probabilistic model, where we artificially sample the delays from a known probability distribution and study the convergence rate of the asynchronous iteration varying the distribution properties. Table 6.1 shows the different probability distributions, their PDFs, and the expected value we use to sample our delays.

Definition 6.2 (A probabilistic model). *Let $\mathcal{F}$ be an asynchronous iteration, written as $(\mathcal{F}, x_0, \mathcal{I}, \mathcal{S})$, as defined in Definition 6.1. Let $k_{\mathcal{F}}$ represent the latest local update iteration count and let $\mathcal{P}(x)$ be some probability distribution where we sample the set of delays $\mathcal{D}$ from, and $\mathcal{P} : \mathbb{N}_0 \to \mathbb{N}_0$. Therefore, $\mathcal{S}_{\mathcal{P}} = k_{\mathcal{F}} - \mathcal{D}$ gives us the iteration from which we incorporate our information. Then the probabilistic model can be written as the asynchronous iteration $(\mathcal{F}, x_0, \mathcal{I}, \mathcal{S}_{\mathcal{P}})$ where the sampled delays $p(k) \in \mathcal{D} \sim \mathcal{P}$ are subject to the following conditions:*

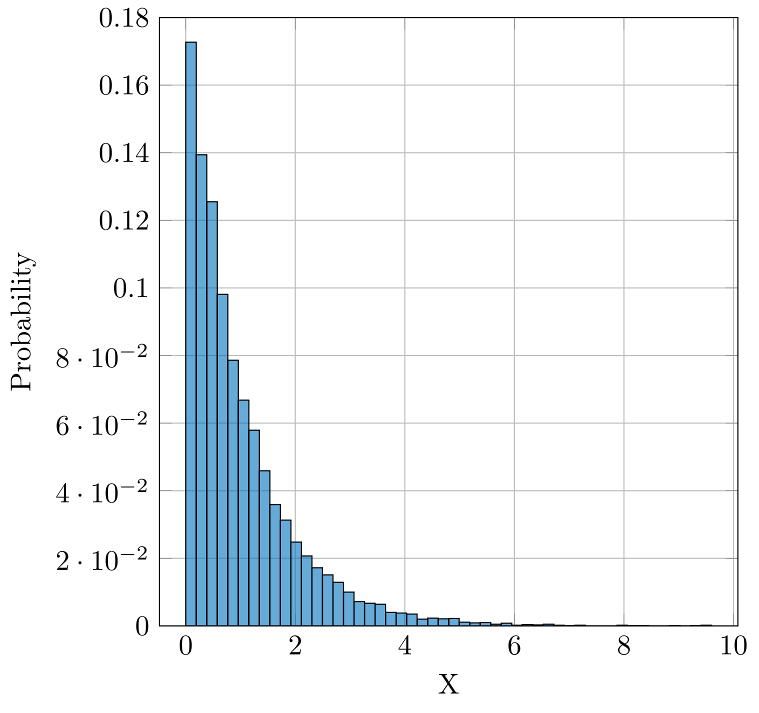
1. *Positivity: $p(k) \geq 0$*
2. *Validity: $p(k) \leq k - 1$, delays cannot be greater than the current iteration.*
3. *Using latest available update: $p(k+1) - p(k) > 0$*
4. *No stagnation: $p(k+1) \nrightarrow \infty$*
5. *Offset:*
$$p(k) \begin{cases} = 0, & \text{if } \; k < o_{\mathcal{F}} \\ \in \mathcal{P}_S, & \text{otherwise} \end{cases} \tag{6.3}$$
where $o_{\mathcal{F}} \geq 0$ is the iteration offset at which the delays are introduced.

6.2.1 Delay and synchronization

A crucial aspect of our probabilistic model is the sampling of the delays, $p(k) \in \mathbb{N}_0^J$. These delays should satisfy the conditions listed in Definition 6.2. We have a few probabilistic distributions from which we can sample the delay. The probability distributions and their distribution functions are shown in Table 6.1, and their sampling is visualized in Figure 6.1, with a total of 10^4 samples being collected and distributed into 50 bins in the histogram, with a mean of 1 for both distributions.



(a) Half Normal, $\mu = 1.01$, $\sigma = 0.05$



(b) Exponential, $\mu = 1$

Figure 6.1: Histograms for different probability distributions collected in 50 bins with certain parameters giving the probability of sampling the delay, X

6.2.1.1 Choice of probability distributions

The probability distribution we sample our delay, $p(k) > 0$, should reflect the properties of the system we run the solver on. In particular, we aim to model the behavior of the solver parameterized on specific system characteristics.

The half-normal distribution is a variant of the normal distribution, which is distributed over only positive values. The mean, μ is $a + \frac{b\sqrt{2}}{\sqrt{\pi}}$, and its variance, σ is $b^2(1-2/\pi)$. The distribution is parameterized by a, the location parameter, and b, the scale parameter.

The exponential distribution in Figure 6.1 is parameterized by λ and the decay is exponential. The mean or expected value $E[X]$ of the distribution with a rate parameter, λ is $\frac{1}{\lambda}$ and its variance is $1/\lambda^2$.

There are some salient differences between the two distributions; in particular, the half-normal distribution has a much smaller spread and a smaller tail compared to the exponential distribution. For two probability distributions of the same mean, one half-normal and one exponential, the exponential distribution has a more considerable variance and, hence, a higher probability of sampling a value far from the mean.

6.2.2 Empirically estimating convergence rates

The central idea of iterative methods is to reduce the residual error, a scaled version of the backward error, $r = b - Ax$. Given that the residual has been minimized in some appropriate norm and is smaller than the prescribed tolerance, we assume the solution has converged.

The quality and effectiveness of an iterative method are characterized by its convergence rate, which defines the number of steps (iteration) taken by the iterative method to converge to a prescribed tolerance, with some measure of the error norm. While this does not necessarily reflect the time to solution, as each iteration's cost may differ for different iterative methods, it is a reasonable estimate of the method's effectiveness given a particular input. In literature, there have been many definitions of convergence rates, some generic, parameterized on the system matrix and the iterative solver algorithm [Saa03].

This process has proven difficult for asynchronous methods as the definition of an iteration is no longer clear: different components may have seen different updates at any point in time. [ADG15; Str02]. Due to the non-deterministic nature, a probabilistic approach is necessary to bound the convergence rates. In addition, there has been no work that estimates convergence rates for multi-level methods, which may add further complexity to the problem but might be necessary to improve convergence, in particular for block-asynchronous and domain decomposition methods, as we will see in Chapter 7.

We aim to estimate the convergence rates with an empirical approach. This approach is valid for specific matrices, and while the convergence rates themselves might not be valid for other matrices, the general convergence behavior of the asynchronous iterative method can be inferred using this approach.

To empirically estimate the convergence rate, we need to estimate the rate at which this residual norm error reduces per iteration. We achieve this by plotting the residual history of the iterative method, performing a linear regression fit on the graph, and considering the slope of the line. A negative slope implies a reduction of the residual norm, a zero slope implies stagnation and a positive slope implies divergence. The larger the absolute value of the negative slope, the larger the convergence rate of the method we consider. Figure 6.2 shows the convergence rate of the **`Cg`**, **`Bicgstab`** and **`Gmres`** (30) methods for a 7 point Laplacian matrix. We plot the residual history, which is expected to be monotonic for this matrix, and do a linear fit. The slope of the line gives us the convergence rate of the method. As the system matrix is symmetric and positive definite (SPD), we see that CG is as efficient as **`Gmres`** and hence has the same estimated convergence rate.

We must note that this estimation process is only a linear approximation of the convergence rate. It has been shown that some iterative methods, such as the Krylov subspace methods, have a super-linear convergence rate [BK01; SST21]. The linear regression fit then provides only a pessimistic lower bound for the convergence rate, which can be a reasonable estimate for smaller matrices but not for larger matrices [BK01].

6.2.3 Modeling systems with different characteristics

With systems becoming increasingly hierarchical, latencies between the memory and compute units can vary widely. Consider one node of Frontier as shown in Figure 2.10. Latencies increase as we go from within one GPU to GPU-GPU to GPU-CPU and observe the largest latencies for communication between off-node CPUs due to the network cards being interfaced through the GPUs.

We use parameterized probability distributions to study the behavior of systems with a variety of communication and latency characteristics. Our model differentiates between computation and communication by assigning processing elements to the computations and the exchange of information between these processing elements as distinct steps.

In a bulk synchronous model, the exchange of information necessitates global synchronization. The latencies between the processing elements within this global synchronization model are subsumed in the communication step.

In a data-dependency model, the exchange of information is required only between processing elements with data dependencies. These data dependencies can be expressed as a task graph. In this case, the pairwise latencies between processing elements become essential.

To accurately model the asynchronous iteration on different systems, the following system properties are important:

1. The number of processing elements or the size of the system,
2. The latencies between the processing elements and
3. The range of the latencies between processing elements which is an indicator of the heterogeneity (communication imbalance) in the system.

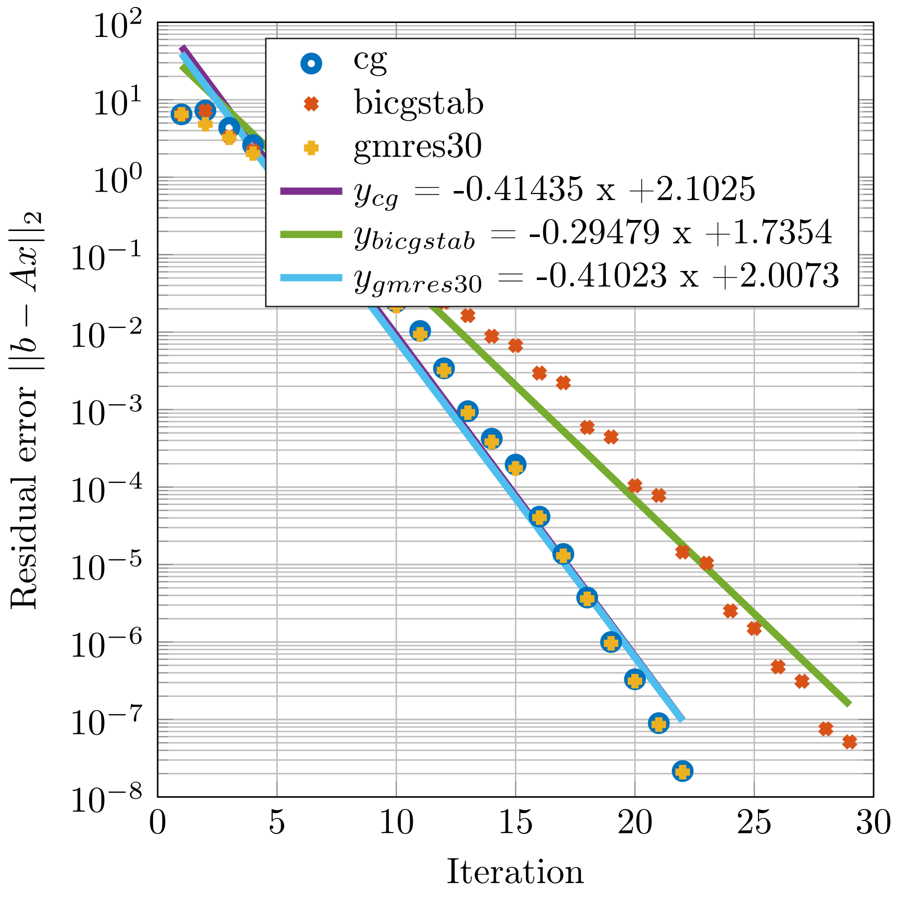


Figure 6.2: Empirically estimating convergence rate of CG, BiCGSTAB and GMRES(30) Krylov methods.

The larger the system, the higher the probability of system noise that can accumulate and propagate through time [HSL10]. In a heterogeneous system consisting of hierarchical compute elements, in addition to the pairwise latencies between processing elements (work-items $\iff$ subgroups $\iff$ work-groups $\iff$ GPUs $\iff$ nodes $\iff$ node-clusters), the difference in latencies between processing elements handling similar computation is also an important factor. Large differences in latency lead to larger synchronization overheads. To this end, we try to model the effects of hierarchy, considering two different types of systems:

1. A hierarchical system with pairwise low relative latencies, but hierarchical memory that can lead to a few relatively large delays due to cache effects.
2. A flat system with larger relative latencies but possibly fewer large delays.

6.2.4 The asynchronous Richardson iteration

As we have seen in Section 6.1.1, the asynchronous Richardson iteration is the asynchronous variant of the well-known relaxation method. Given an iteration matrix, T_ω, the criteria that are necessary for convergence for the asynchronous Richardson is $\rho(|T_\omega|) < 1$, with the relaxation parameter, $\omega < 1$. To empirically study the convergence behavior, we consider the 2D-Laplacian 5-point stencil problem on a grid of size 5×5, leading to a system matrix of size 25×25. The sparsity pattern and the eigenvalue distribution are shown in Figure 6.3.

The asynchronous Richardson iteration algorithm is shown in Algorithm 16, where in iteration k, the vector x takes information from the set of delays s_k. To model this approach, we consider a probabilistic distribution and randomly sample a value from the distribution satisfying the properties of the set $\mathcal{S}_k$ as shown in Algorithm 17.

Algorithm 16 The Asynchronous Richardson method

```
1:  $A, b, x, \omega$
2:  $r \leftarrow b - Ax, e \leftarrow \mathbf{0}$
3:  for $k < N_{iter}$ do
4:      $d \leftarrow s_k^{k-1} \in \mathcal{S}_k$
5:      $r \leftarrow b - Ax(d)$
6:      if $\|r\| < \tau$ then
7:          break
8:      end if
9:      $e \leftarrow$ PRECOND($r$)
10:     $x \leftarrow x + \omega e$
11: end for
```

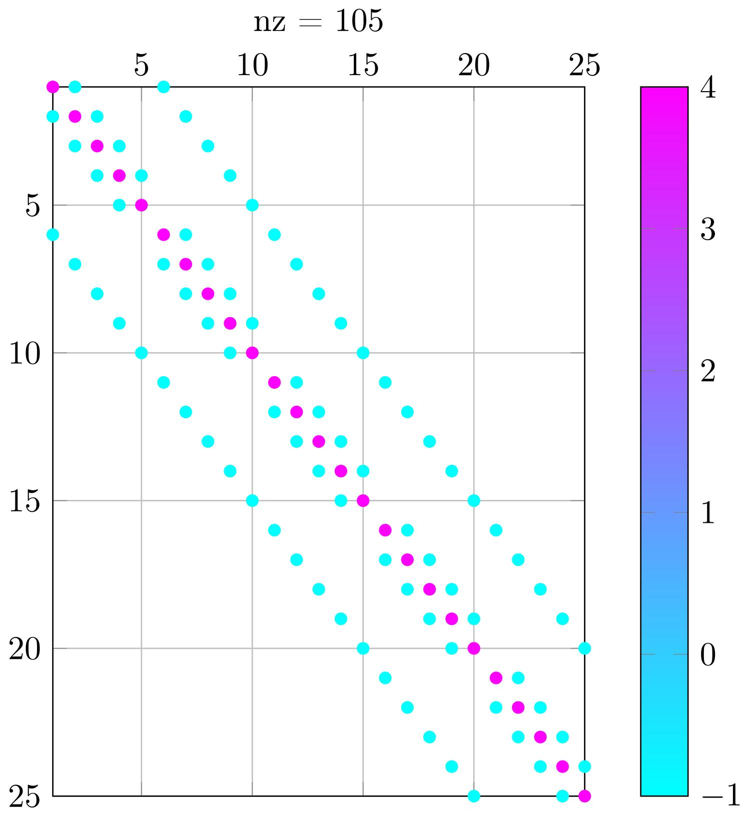
The choice of the probability distribution to sample the delays affects the convergence. The properties of the distribution can be qualitatively parameterized on the system characteristics, and we can hence infer the behavior of the asynchronous iterative method given a particular system.

6.2.4.1 Highly hierarchical compute systems

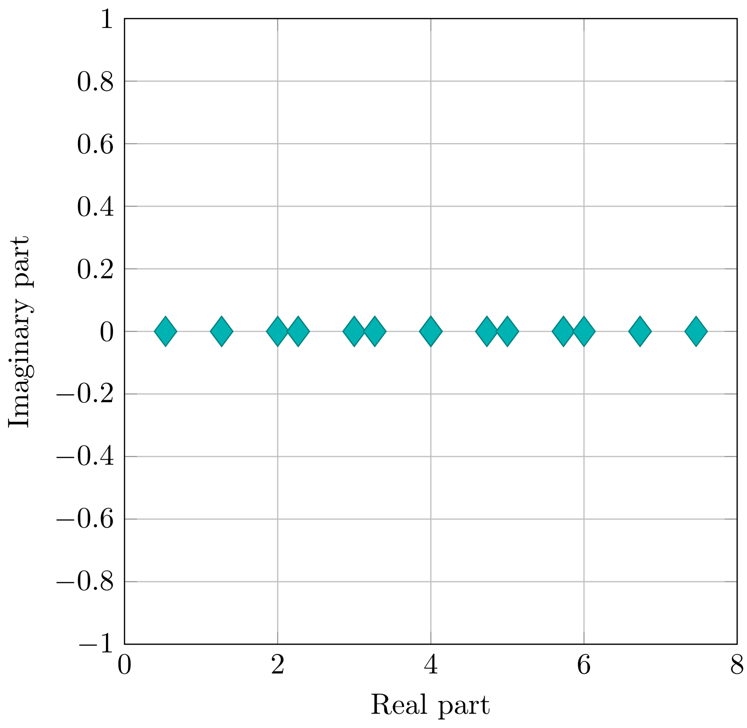
A hierarchical system, such as the GPU, has hierarchical compute units and memories. This hierarchy is generally associated with differing latencies. While the relative pairwise latencies can be small, fetching from the main memory can result in larger delays due to cache misses. This type of behavior with a few outliers of large delays with most iterations being up to date can be modeled with the exponential distribution. The exponential distribution is characterized by λ, which gives us the mean of the distribution. The variance of the distribution is then given by $\frac{1}{\lambda^2}$. Figure 6.4 shows the convergence rate (estimated with linear regression) of the asynchronous Richardson method with a relaxation parameter of 0.8. Each green dot represents one solver run with delays sampled from the Exponential distribution. Increasing the

Algorithm 17 The Asynchronous Richardson method: Random sampling model

```
1: $A, b, x, \omega$
2: $r \leftarrow b - Ax, e \leftarrow \mathbf{0}$
3: for $k < N_{iter}$ do
4:     $d \leftarrow$ RANDOM_SAMPLE$(k, a, b)$
5:     $r \leftarrow b - Ax(d)$
6:     if $\|r\| < \tau$ then
7:         break
8:     end if
9:     $e \leftarrow$ PRECOND$(r)$
10:    $x \leftarrow x + \omega e$
11: end for
```



(a) Sparsity pattern



(b) Eigenvalue distribution

Figure 6.3: The 2D Laplacian problem on a 5×5 square grid with a 5 point stencil discretization.

mean delay (Figure 6.4a) improves convergence, but for larger mean delays (and large variance), it leads to divergence. Figure 6.4b shows the actual sampled delays for one experiment with a mean delay, $\mu = 2.5$. As we set the offset for the delay to 40 iterations, we have no delays for 40 iterations, and after the 40th iteration, delays are introduced into the algorithm.

In Figure 6.4c and Figure 6.4d, we delay the start of asynchronicity. This characterizes architectures where initially there is no system and runtime noise but is only accumulated after a certain amount of time (or iterations in our case) [HSL10]. This is controlled by the `delay_offset_iter` ($o_{\mathscr{I}}$, See Definition 6.2) parameter on the X axis of Figure 6.4c. As expected, increasing the delay offset iteration makes the solver more robust, enabling convergence of all runs. Again, the overall median convergence rate is higher when delays are incorporated early. This is in line with observations where asynchronous methods with delays in some cases can be faster to converge than synchronous methods [CW18].

6.2.4.2 Flat hierarchy compute systems

Compared to highly hierarchical systems such as GPUs, computing systems such as CPU clusters have a relatively flatter hierarchy. In these cases, one can expect the relative latencies to be higher, hence the mean delay is generally higher. This kind of distribution can be captured with a half-normal distribution, which is a truncated normal distribution. The normal distribution has been previously used to model asynchronous iterations [WC19]. It is characterized by two variables, a, the location parameter, and b, the scale parameter. Table 6.1 shows its pdf and mean which can be calculated from a and b.

Figure 6.5 shows the convergence rate (estimated with linear regression) of the asynchronous Richardson method with a relaxation parameter of 0.8. Each green dot represents one solver run with delays sampled from the Half-Normal distribution. Increasing the mean delay (Figure 6.4a, with a constant variance of 0.098), improves convergence until an inflection point, and for substantial mean delays, can lead to divergence. Figure 6.4b shows the actual sampled delays for one experiment with mean, $\mu = 2.39$ and variance, $\sigma = 0.098$. Figure 6.5c also shows similar behavior, where we keep the same location parameter but increase the shape parameter and hence the variance. With very high variances, implying a very imbalanced system, we see some runs failing to converge. Figure 6.5c shows the histogram of the sampled delays on one experiment run with a location parameter of 2.5.

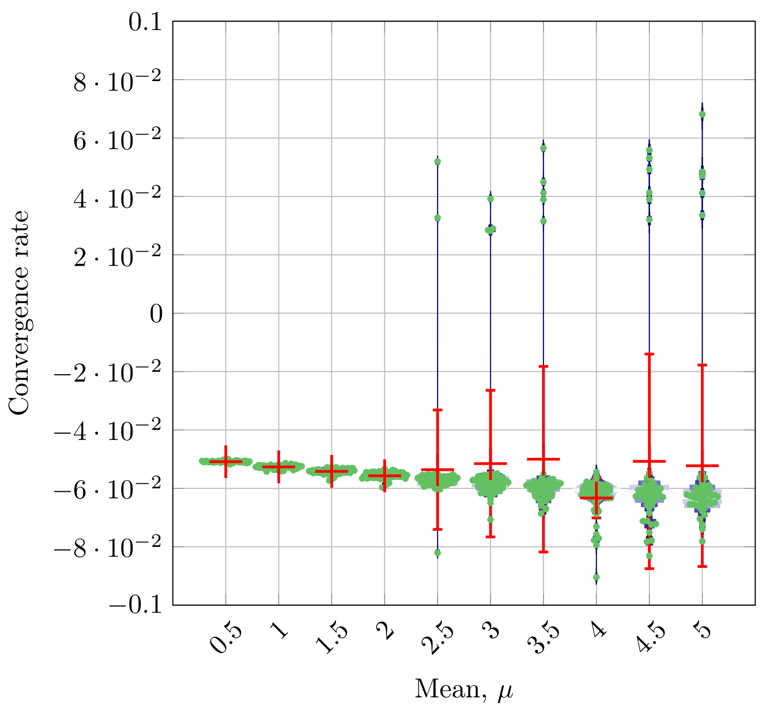
In Figure 6.5e and Figure 6.5f, similar to the previous experiment, we increase the iteration at which we introduce the delays. We see that we observe convergence across all parameters for a mean delay of 0.35 and a variance of 0.014. In contrast to the Exponential distribution, we observe divergence for lower offset iteration counts for larger mean delays. This can be explained by the nature of the Half Normal distribution, where the sampled delay is always greater than the location parameter.

6.2.4.3 Effects of the relaxation factor, ω

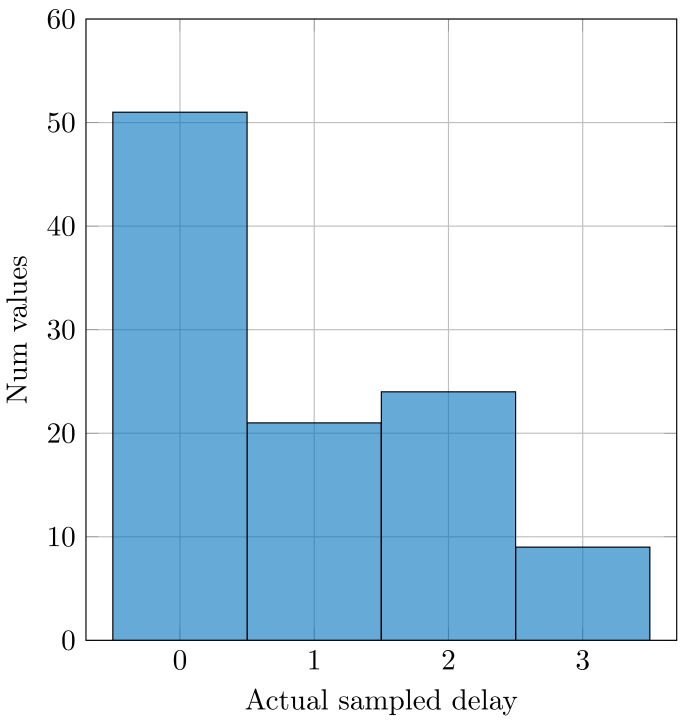
Since we are dealing with contracting operators, relaxation parameters can affect convergence. We have seen in Theorem 6.1.3 that we require the relaxation factor to be < 1 for the Richardson iteration to converge. This under-relaxation can help accelerate convergence by reducing over-correction. This factor is, of course, problem-specific, and the optimal value depends on the problem at hand [CFS20].

In Figure 6.6, we explore varying relaxation parameters' behavior on the Exponential distribution's convergence rate and with increasing mean delays. An interesting observation is that the asynchronous Richardson iteration converges for large mean delays for low relaxation factors, $\omega \leq 0.3$. Increasing the relaxation factor and the mean delays at the same time worsens the convergence. This observation can be used to fine-tune the asynchronous implementation based on the system characteristics, controlling the relaxation parameter for systems depending on the noise and delay characteristics of the system.

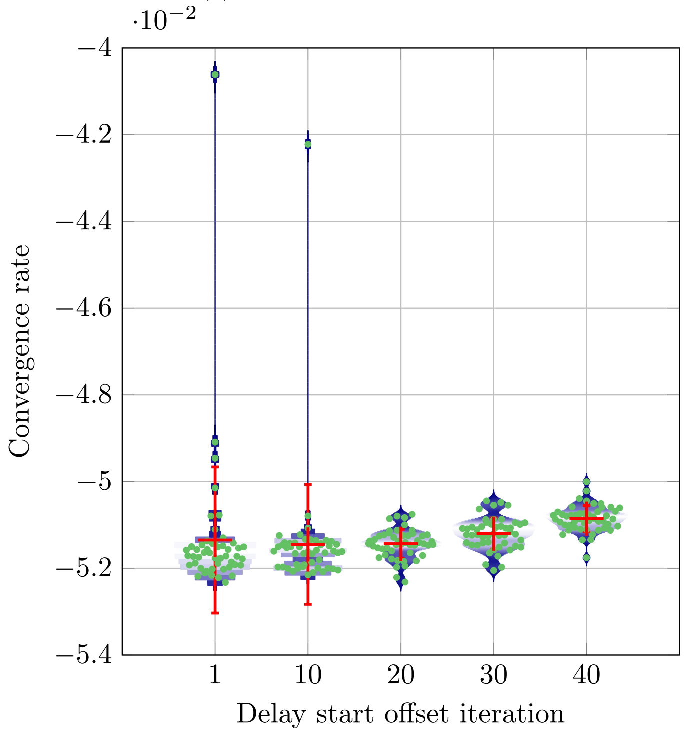
An interesting observation that can be made from Figure 6.6 is that delay in data exchange is equivalent to over-relaxation. Increasing the relaxation parameter and the delays increase the over-correction (sudden incorporation of a



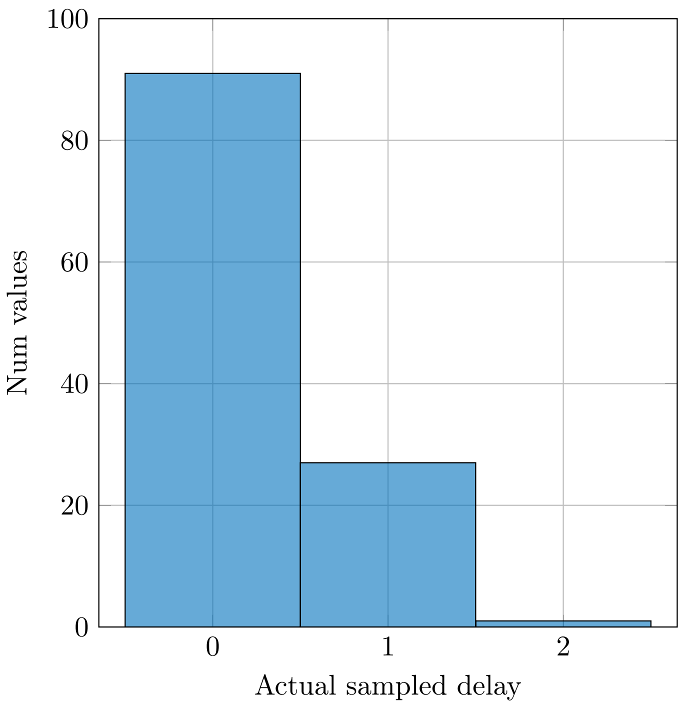
(a) Increasing the mean delays



(b) Empirically sampled delays for $\mu = 2.5$

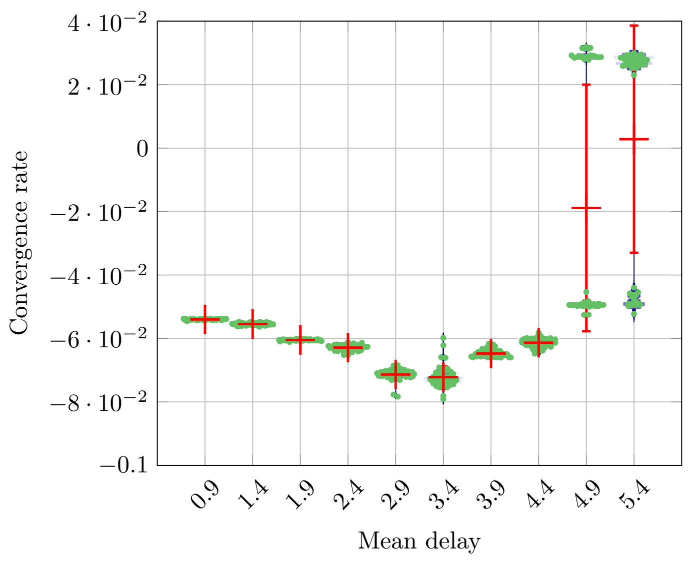


(c) Increasing the offset iteration, $\mu = 0.5$

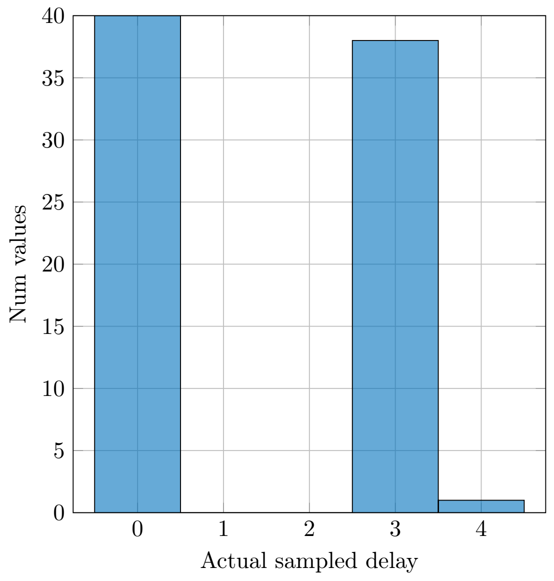


(d) Empirically sampled delays for offset iteration of 40

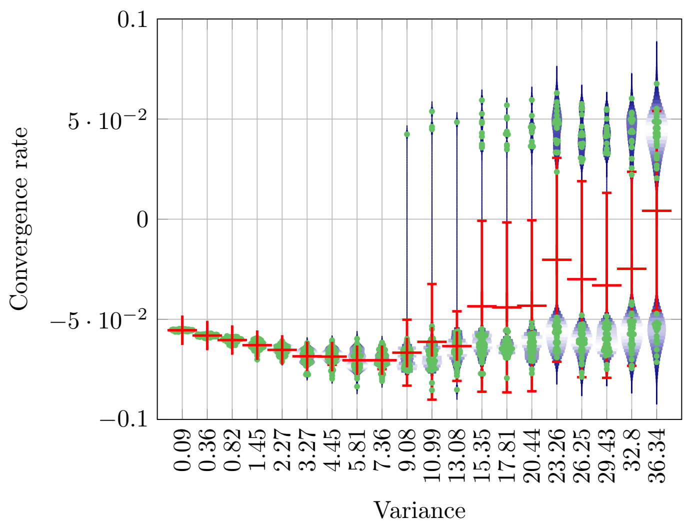
Figure 6.4: Exponential distribution sampling: Correlation between the convergence rate and the delay sampling parameters(left). The histogram showing the empirically sampled delays for one run(right)



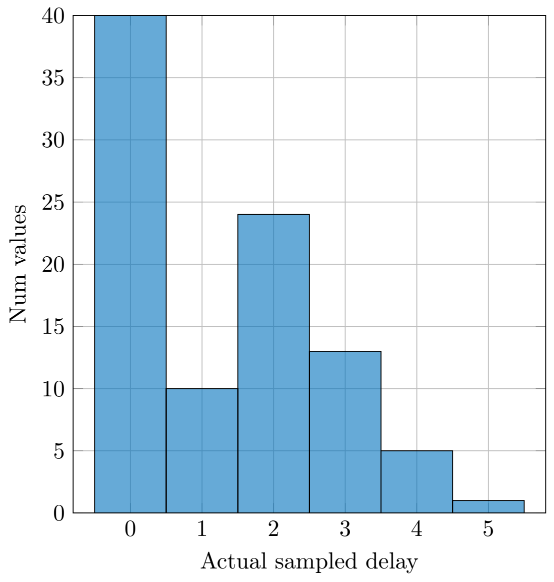
(a) Increasing the mean delays, $\sigma = 0.098$



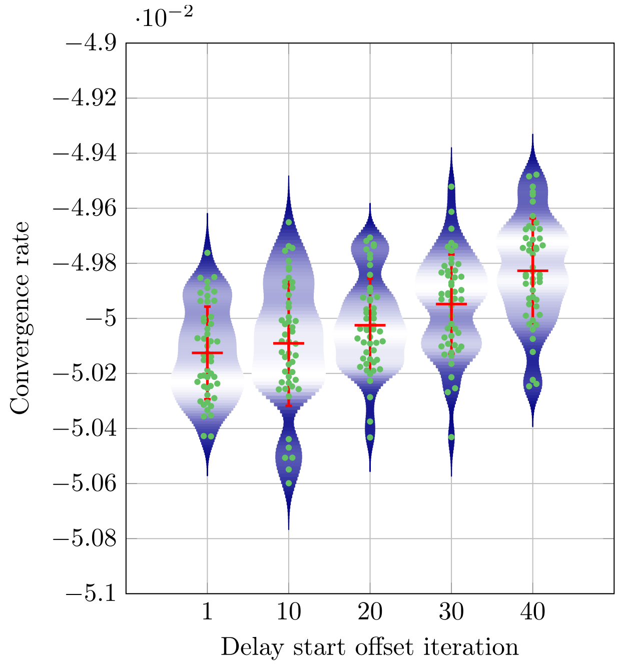
(b) $\mu = 2.39$, $\sigma = 0.098$



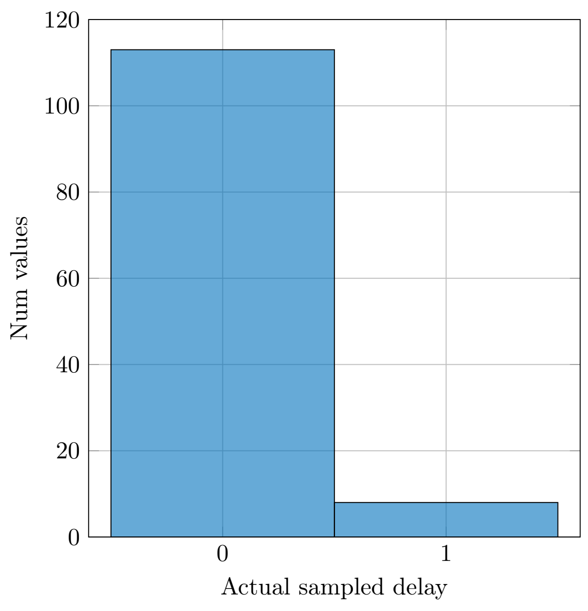
(c) Increasing variance



(d) Location parameter = 2.5



(e) Increasing the offset iteration,
$\mu = 0.35$, $\sigma = 0.014$



(f) Offset iteration = 40

Figure 6.5: Half-Normal distribution sampling: Correlation between the convergence rate and the delay sampling parameters(left). The histogram showing the empirically sampled delays for one run(right)

large amount of information), making the iteration unstable. With a low relaxation parameter, larger delays can be sustained without losing convergence. In fact, for lower relaxation parameters, adding artificial delays, like in Figure 6.6a improves the effective convergence rate.

6.3 GPU capable asynchronous solvers

The hierarchical parallelism available on the GPUs through work groups, subgroups, and work-items allows for asynchronous execution on multiple hierarchy levels. This suggests different strategies for realizing asynchronous algorithms. We evaluate the behavior of these different strategies for the asynchronous Richardson iteration on GPUs.

The target problem in all our experimental evaluations is the Laplace problem in 2D discretized with the 5-point stencil $\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}$. Therefore, we assume $\omega = 1$ for all our experiments, using the optimal value for synchronous Richardson. [Saa03].

We compare the different strategies of the asynchronous Richardson with the synchronous Richardson iteration, which is the traditional bulk-synchronous version of the algorithm. The central idea of the asynchronous Richardson algorithm lies in the asynchronous computation of the x vector elements without explicit synchronization of the parallel compute units assigned to compute the relaxation on these rows. We consider two main strategies: static and dynamic assignment of the parallel compute units to the rows.

6.3.1 Synchronous Richardson iteration

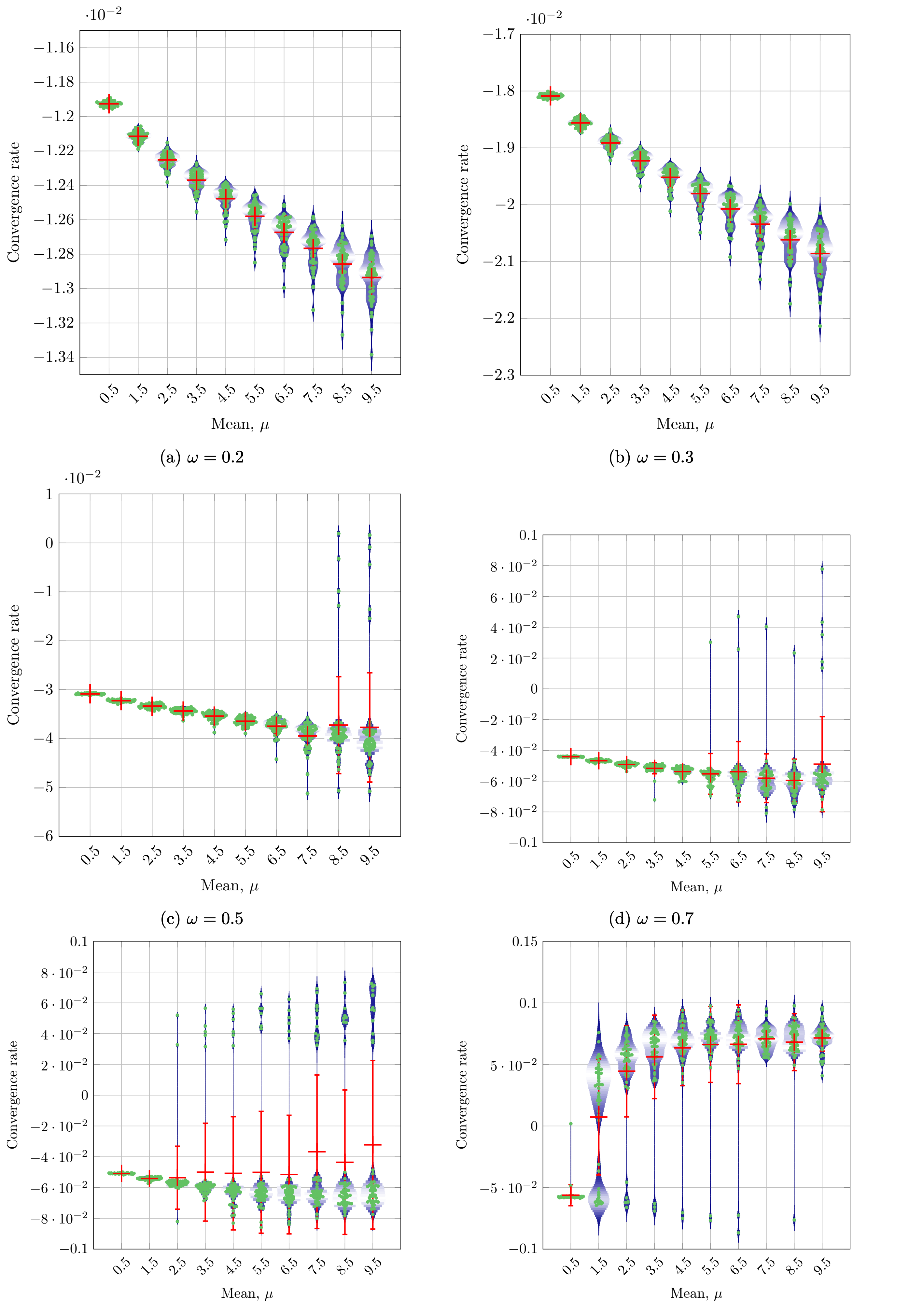
The classical synchronous Richardson iteration first computes the residual and checks for convergence using the computed residual. A preconditioner or inner solver is then applied, if any, and finally, the solution vector is updated with a relaxed residual (residual multiplied by the relaxation factor). This process is repeated for a maximum of 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 $\mathcal{O}(n_{nz})$, where n_{nz} is the number of non-zeros in the matrix. We store our matrix in a Compressed Sparse row (CSR) format, a versatile and efficient storage format for sparse matrices [ACC+20a]. The synchronous Richardson algorithm is shown in Algorithm 18. Steps 2 and 3 involve an explicit device-wide synchronization as each step of the algorithm requires all the elements of the solution vector to be updated before proceeding with the next iteration.

Algorithm 18 Synchronous Richardson iteration

```
 1: INPUT: b, A, $\tau$
 2: OUTPUT: x
 3: for $k < N_{iter}$ do
 4:     $r \leftarrow b - Ax$                    ▷ Residual computation
 5:     if $\|r\| < \tau$ then                    ▷ Convergence check and synchronization
 6:         break
 7:     end if
 8:     $r \leftarrow$ PRECOND($r$)              ▷ Inner solver apply
 9:     $x \leftarrow x + \omega r$              ▷ Update
10: end for
```

6.3.2 Static assignment of subgroups to rows

Subgroup is a concept that allows the splitting of a group of work-items (warps in CUDA, wavefront for AMD) into smaller execution entities [TCR+21]. A subgroup can be as small as a single thread and as large as a complete subgroup



(a) $\omega = 0.2$

(b) $\omega = 0.3$

(c) $\omega = 0.5$

(d) $\omega = 0.7$

(e) $\omega = 0.8$

(f) $\omega = 0.9$

Figure 6.6: Studying the effects of the relaxation parameter, ω on the convergence rate with a mean delay, $\mu = 1$ with an Exponential distribution.

(32 work-items in CUDA, for example). In the static implementation strategy (shown in Algorithm 19), every row of the iteration vector is handled by one subgroup, and the number of subgroups launched matches the number of rows in the solution vector. For example with a subgroup size of 1, we assign one work-item to one row of the linear system. The scheduling of the subgroup on the hardware compute units is handled by the GPU runtime. To ensure the consistency of the memory reads and writes across different work-items, we need to use a fence operation (`__threadfence()` in CUDA). Using the static subgroup-to-row strategy, only one subgroup updates an entry of the iteration vector, so all the updates are incremental.

Algorithm 19 Static assignment: one subgroup to one row

```
INPUT: b, A, $\tau$
OUTPUT: x
for all  row in [0, #rows) in parallel do                 ▷ One subgroup to one row
    temp = A(row, :)  * x                                  ▷ Accumulate
    x[row] += $\omega$ * (b[row]-temp)                     ▷ Update
    fence();                                               ▷ Enforce consistency
end for
```

6.3.3 Dynamic assignment of subgroups to rows

An alternative strategy is to reduce the number of launched workgroups and assign subgroups to handle multiple rows in each iteration, as shown in Algorithm 20. This implies that the subgroup-to-row assignment after the first cycle is handled by the runtime, potentially allowing for better load balancing and allowing different subgroups to update different rows. A GPU generally contains multiple tens of multiprocessing (MP) units, and each unit has multiple schedulers that the runtime uses to schedule one subgroup to a compute unit. One unit can also concurrently execute multiple subgroups at once. Statically scheduling the maximum number of subgroups to the MP unit can prevent the runtime from using context switching to hide latencies, which can be very useful for architectures with deep cache hierarchies. To allow the runtime to automatically switch contexts if necessary, we oversubscribe the MP units by launching more workgroups than the number of MP units available. For example, the NVIDIA V100 GPU has 80 MP units, with each MP unit having four subgroup schedulers. One MP unit can concurrently execute 128 work-items (4 subgroups of size 32). We launch `oscb`$\times$80 work groups of size 128, where `oscb` is the oversubscription parameter that can be controlled at runtime and ranges from 1 to 16. Using this dynamic subgroup-to-row strategy, multiple subgroups may update a row simultaneously.

Algorithm 20 Dynamic assignment: Multiple subgroups may handle one row

```
INPUT: b, A, $\tau$
OUTPUT: x
REQUIRE: num_subgroups $\leq$ num_rows
for all id in [0, #subgroups) parallel do
    iter=0 , row=id
    while iter < max_iterations do
        temp = A(row, :)  * x                              ▷ Accumulate
        Update:  x[row] += $\omega$ * (b[row] - temp)      ▷ Update
        fence()                                            ▷ Enforce consistency
        row += num_subgroups                               ▷ Get next working row
        if  row $\geq$ num_rows then                       ▷ Modulo and increment
            row -= num_rows
            iter += 1
        end if
    end while
end for
```

6.3.4 Visualizing the two asynchronous strategies

Figure 6.7 visualizes the update process of the two implementations. The y-axis represents the distinct rows of the iteration vector; the columns represent the iterations. The different colors represent the updates that occur in the same update cycle. The square represents the first subgroup, and the circle represents the last subgroup out of all the subgroups launched in the same cycle.

The static subgroup to row implementation (left in Figure 6.7) launches enough subgroups to update all rows in the first update cycle. The dynamic subgroup to row assignment (right in Figure 6.7) usually uses fewer subgroups than the number of rows. Therefore, it may require more update cycles, $k > n$, to complete the n updates. However, due to the aforementioned latency hiding during the scheduling, we can achieve better performance for some subgroup sizes.

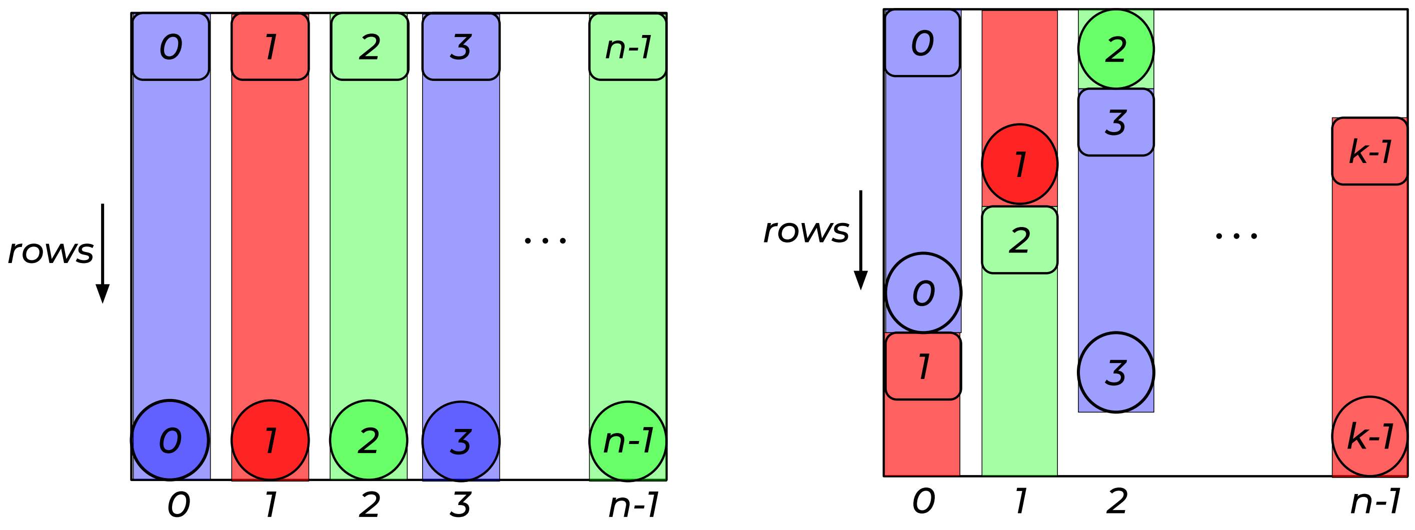


Figure 6.7: Visualization of the update process of the static subgroup-to-row assignment (left) and the dynamic subgroup-to-row assignment (right).

6.3.5 Reproducibility of results and measurement strategies

Reproducibility of experimental results using asynchronous algorithms is challenging, due to its non-deterministic nature. For benchmarking purposes, it is thus necessary to rely on statistical data collected over many runs. As the CUDA runtime scheduling the thread blocks and orchestrating the execution is not reproducible, we report statistical information. To obtain insight into the execution process, we utilize specialized mechanisms to log:

1. *Time*: Recording a work-item's start and completion time (all updates done by this work-item).
2. *Age of the last iteration update*: Recording the age of all values used for the last iteration update of a vector value.
3. *Age of the mid-iteration update*: Recording the age of all values used for the vector value update midway through the iteration process.

It is important that the logging mechanisms do not affect the GPU scheduling or the runtime performance. This isolation is achieved by replacing the output in the iteration vector with the logging information (instead of accessing additional memory locations). Therefore, this logging functionality will not affect the performance as the Richardson iteration (with SpMV as the bottleneck) is known to be memory-bound [ACC+20a][1]. In addition, using the memory

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

space of the iteration vector for logging mechanisms implies that when analyzing the algorithm execution with these logging mechanisms, no actual output is generated.

The iteration vector stores each row's 64-bit (IEEE-754 double precision) value. For the *time logging*, every subgroup writes the timestamp of every row update into its row index position.

To analyze the age of the updates (last and mid-iteration), we encode the age information in a customized 64-bit datatype, which is stored in the IEEE-754 double precision iteration vector, as shown in Figure 6.8. The age of each row value and its four neighbors is encoded with overall 60 bits ($12 \times 5 = 60$). A decoding step can be applied during the post-processing (incurring no solver apply overhead) with an appropriate bit shift and utilizing the mask `0xFFF`. For example, as the current row's 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 done so by replacing the numeric values in the system matrix. Therefore the the 5-point stencil is instead stored as $\begin{bmatrix} & 0 & \\ 1 & 2 & 3 \\ & 4 & \end{bmatrix}$.

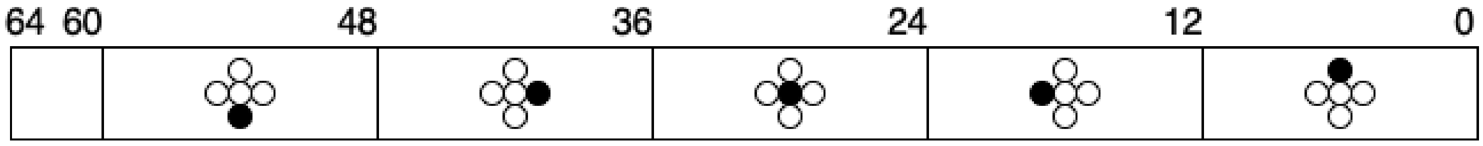


Figure 6.8: Visualization of the customized 64-bit datatype that encodes the age of the row value and the neighboring elements.

6.3.6 Results and analysis

We use the normalized version of the 5-pt stencil for all our experiments 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 has ten 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 $\in \mathbb{R}^{10^3 \times 10^3}, \mathbb{R}^{4 \cdot 10^3 \times 4 \cdot 10^3}$ and $\mathbb{R}^{9 \cdot 10^3 \times 9 \cdot 10^3}$ respectively. The overall time, backward error, and some additional details are computed a posteriori. All experiments use IEEE-754 double precision and are run on the Summit supercomputer with an NVIDIA V100 GPU with CUDA-11.4.2 and gcc-9.3.0.

We first compare the synchronous Richardson iteration with the static assignment strategy of a subgroup-to-row-based asynchronous Richardson iteration (labeled "static-async"), which uses a subgroup size of 1. Figure 6.9a visualizes the median update time which decreases as the number of updates increases. Each point is a separate run, thereby incurring no penalty with the performance measurement. We observe that after the initial ramp-up phase (with initialization overheads), the cost of an update is constant for both synchronous and asynchronous Richardson iterations. The asynchronous iteration is around 3 to 10× faster than the synchronous iteration. Figure 6.9b visualizes the median convergence of the synchronous and asynchronous Richardson iteration over the update counts and indicates that the synchronous version has slightly superior convergence characteristics to the base asynchronous version. Figure 6.9b reveals that the higher update rate of the asynchronous Richardson (more updates per second) results in superior convergence characteristics in the runtime metric. Finally, Figure 6.9c 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, which takes much less time to converge to a similar or lower residual while running for an equal number of iterations.

To show that our approach works for larger and sparser problems, we also consider a 3D grid with a 7-point stencil to solve the Laplacian problem. From Figure 6.10, we can see that the behavior observed on a 3D grid is similar to that on a 2D grid, with the asynchronous versions performing far better.

6.3.7 Comparing different asynchronous implementations

To improve the GPU occupancy with the asynchronous iteration, we can increase the subgroup size. Figure 6.11a visualizes different versions of the asynchronous Richardson iteration (static and dynamic) implementations with different subwarp sizes. Using larger subwarps reduces the update time[2], but with degrading convergence as shown in Figure 6.11b. Figure 6.11c relating the convergence to the runtime indicates that a subwarp size of 1 is the best choice for the static strategy. We also see that a `__threadfence()` is necessary for convergence of the Richardson iteration to enforce consistency between reads and writes of the different threads. Without a `__threadfence()`, the updates are much faster (see Figure 6.11a), but the convergence can not be guaranteed because the updated values may no longer be immediately visible to other threads (the delays accumulate, and condition Equation (6.2c) might not be satisfied anymore). Figure 6.11b shows that without a `__threadfence()`, the asynchronous Richardson iteration fails to converge for this test problem.

Another strategy that we discussed in Algorithm 20 earlier was the dynamic assignment of subwarps to rows. This dynamic assignment enables better load balancing and, hence, higher occupancy. Furthermore, we use an oversubscription parameter, **`oscb`**, which enables the runtime to hide latency through context switching. From Figure 6.11, it is clear that this is crucial in obtaining good performance, and we also see that a moderate oversubscription (**`oscb`** $= \{4, 8\}$) gives the best results considering both the convergence and performance metrics.

An important aspect to note is that with these oversubscription factors, the asynchronous Richardson iteration exhibits convergence characteristics competitive with the synchronous variant while processing the updates significantly faster.

6.3.7.1 Early stagnation with large subgroup sizes

We instantiate more threads than the maximum number of work-items the device can launch simultaneously. On an NVIDIA V100 GPU, the limit is equal to 80×2048 work-items without accounting for other factors, such as the number of registers or shared memory pressure. From Figure 6.12, we see that the second half of the result vector gets updated only after its first half has finished its updates. This forced serialization (at $(80 * 2048)/32 = 5120$) in the updates adversely affects the convergence of the `async-static(subwarp 32)` implementation as seen in Figure 6.11b.

6.3.8 Understanding the behaviour of the asynchronous iteration

For a deeper understanding of the execution of the asynchronous algorithm, we record the history and age (number of updates the result vector index has seen) of the values used for a particular update.

We first need to classify the result vector indices based on their dependencies. This communication pattern is dependent on the grid. In our case of a 2D grid, shown in Figure 6.13a, we have 3 types of grid points depending on whether they have 2, 3, or 4 neighbors. In addition, edges between these grid points can be further classified based on which grid points they connect, and we have six different types of edges, denoted by {#connection of target}_{#connection of source} for a connection, as seen in Figure 6.13a. For example, **2__3** is the edge connecting a corner grid point with two connections, and the face grid point has three connections. The update history captures the age of the source of the grid point providing the information. In the synchronous version, this is well-defined (all neighbors must provide information for the iteration to continue). In the asynchronous case, depending on the asynchronous algorithm, the sources are restricted only to conditions mentioned in Equations (6.2a) to (6.2c), are hence not necessarily well-defined and regular.

Figure 6.13b shows this history and source age distribution for each edge type collected for all the grid points of the 300×300 grid, midway of the asynchronous iteration (500th update for the grid point out of a total of 1000 updates). The three main strategies are shown: 1) Asynchronous iteration with static assignment of the subgroups to rows (in

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

blue), 2) the dynamic assignment of subgroup to rows with an oversubscription parameter of 4 (in magenta) and 3) the asynchronous iteration without a memory fence to enforce consistency (in green).

For all methods, the median is close to 500. The dynamic assignment strategy has a fairly low variance, which may explain its convergence behavior being close to that of the synchronous Richardson iteration. The "no fence" implementation (in green) has a lack of consistency of the reads and writes from the different threads, implying that the values being read are stale, which leads to stagnation and non-convergence of the asynchronous iteration (violating condition in Equation (6.2c)).

Across the different grid points, only the static assignment strategy seems to have differences. In particular, we see that edge connections of type **3__3** and **4__4** have very low variances (blue boxplots are very small). Edge connections of type **4__3** and **2__3** seem to mostly take information from grid points that are further in their updates. On the contrary, connections of type **3__4** and **3__2** mostly take information from grid points lagging in the update count.

This gives us insight into the update process and that different asynchronous implementations and scheduling strategies can lead to different behavior and the implementation needs to be based on the communication pattern of the matrix.

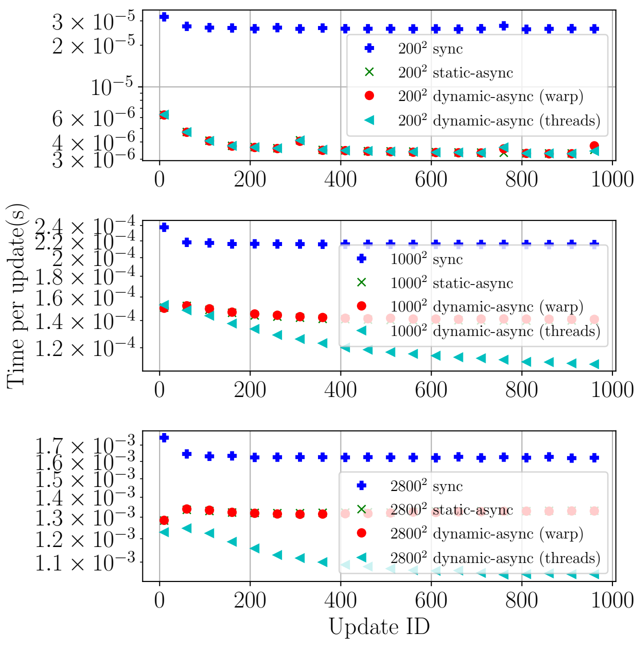
To ensure this history analysis does not incur significant overheads, we perform this analysis on a smaller grid size of 300×300. However, the analysis is extensible to larger grid sizes and matrices. We observed an overall execution overhead of about 3.9% when logging the history midway and about 6.4% when logging the update at the end. In comparison, recording detailed runtime traces of the updates incurs about 20% overhead.

6.4 Summary

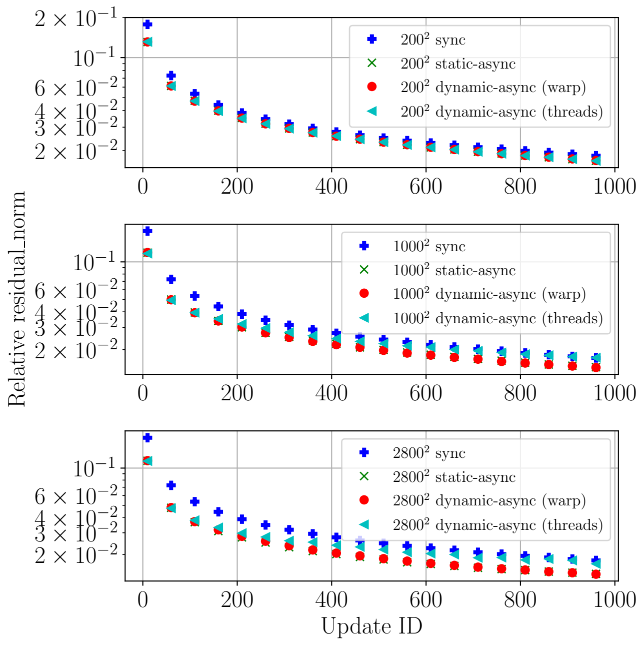
Asynchronous methods help overcome the synchronization and communication bottlenecks using a data asynchronous approach. Relaxing information exchange between parallel processing elements can accelerate the time to solution. We looked at the conditions required for convergence for some asynchronous relaxation methods.

With the help of a probabilistic model, we studied the convergence behavior of the asynchronous Richardson method. We provided some estimates of the convergence rates using an empirical approach. The probabilistic model also enables modeling for systems with different hardware characteristics.

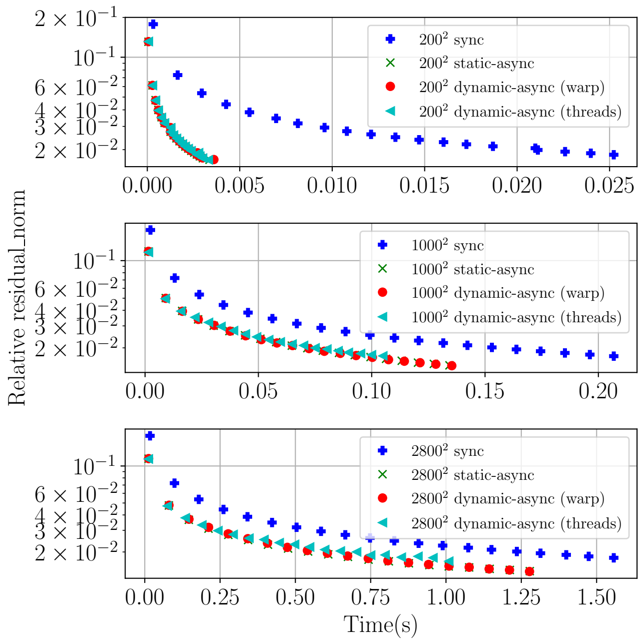
We also implemented a fine-grained asynchronous Richardson method on a GPU, showcasing its performance benefits over the synchronous Richardson method. We provided a detailed analysis of our fine-grained implementations, analyzing the history of the updates and the behavior of the different strategies (static and dynamic). On average, we observed that the asynchronous iteration is $3\times$ to $10\times$ faster than the synchronous iteration. These fine-grained asynchronous methods can make use of the massive parallelism provided by the GPUs, and in the future, we would like to apply these techniques to other iterative methods, such as the Krylov methods.



(a) The median time per update.

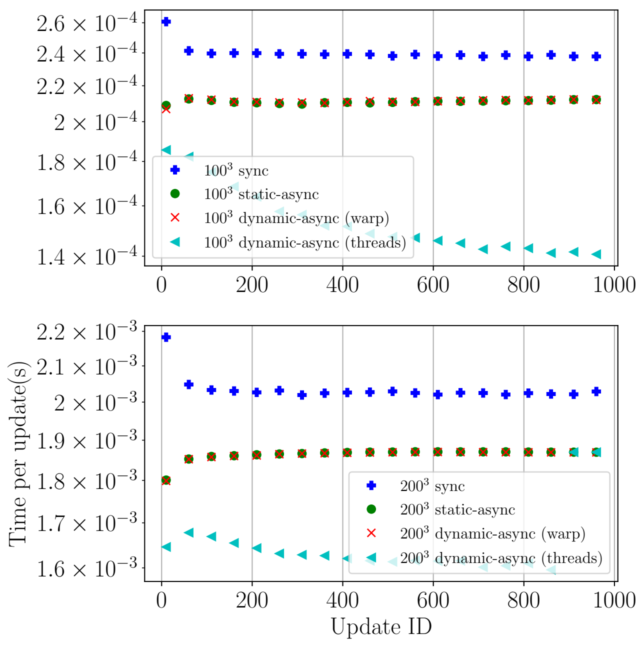


(b) The median relative residual norm related to the update count.

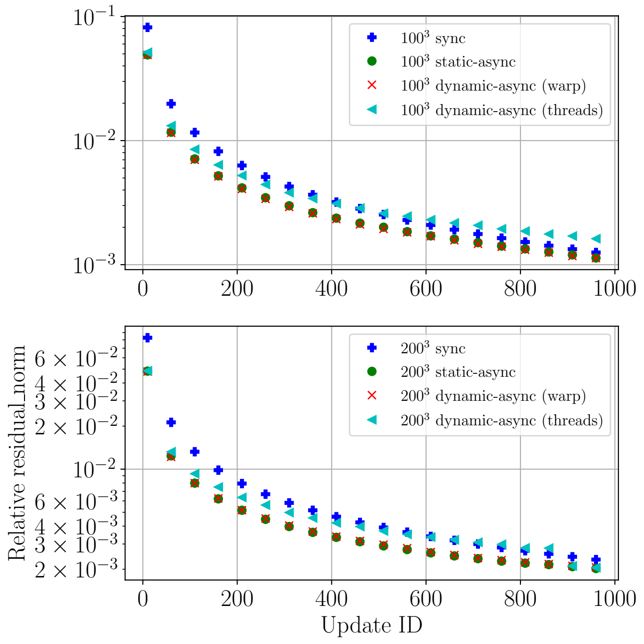


(c) The median relative residual norm over time.

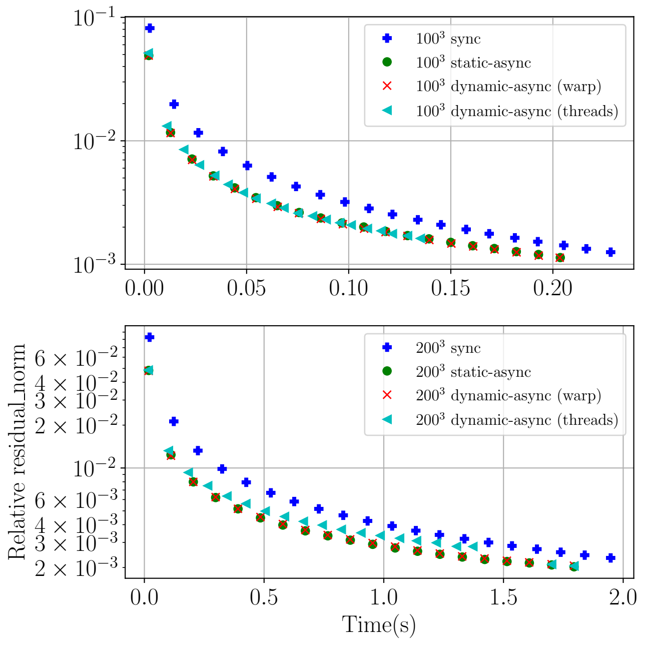
Figure 6.9: Comparing the synchronous iteration with different asynchronous strategies for a 2D grid.



(a) The median time per update.

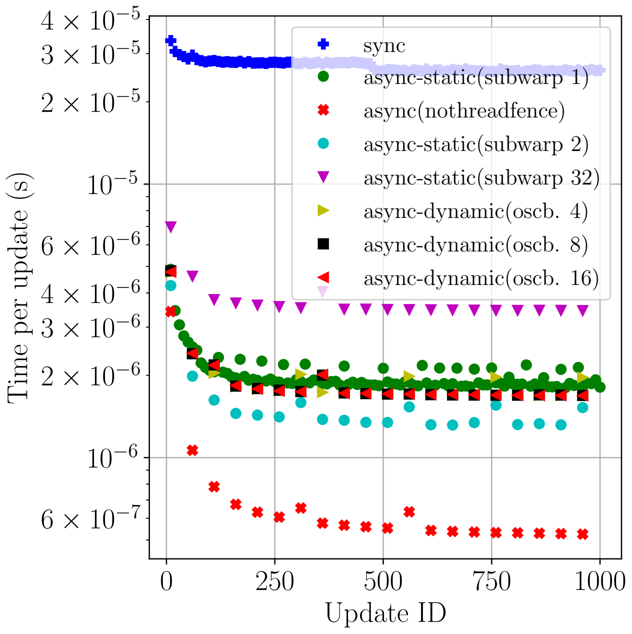


(b) The median relative residual norm related to the update count.

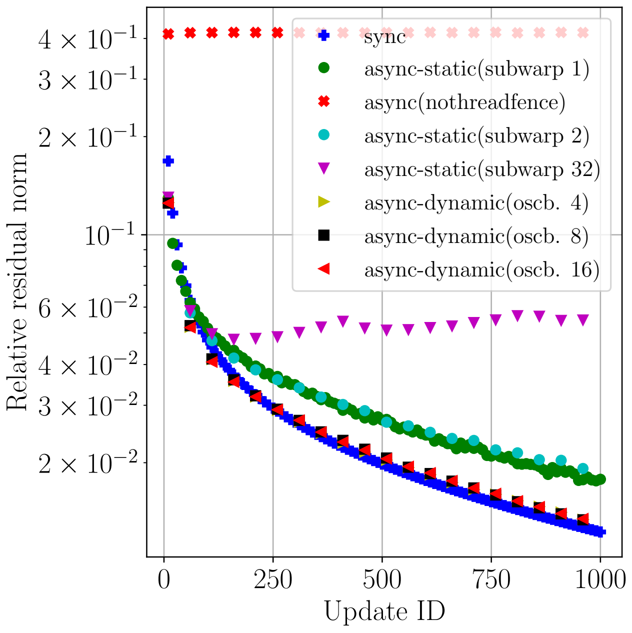


(c) The median relative residual norm over time.

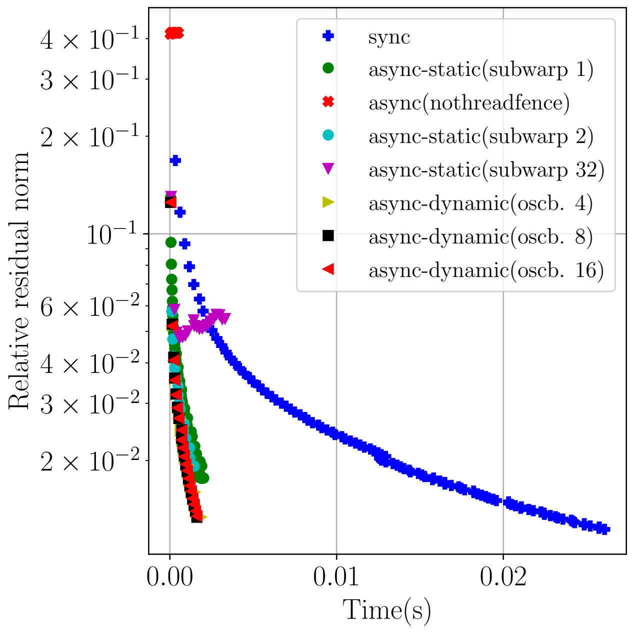
Figure 6.10: Comparing the synchronous iteration with different asynchronous strategies for a 3D grid.



(a) The median time per update.



(b) The median relative residual norm related to the update count.



(c) The median relative residual norm over time.

Figure 6.11: Comparing the different static and dynamic asynchronous iteration strategies.

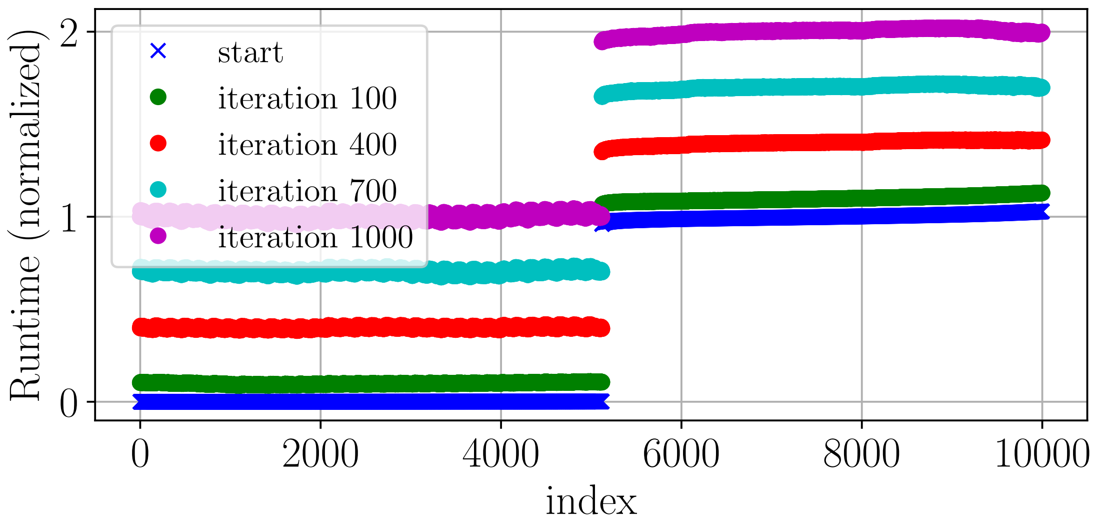
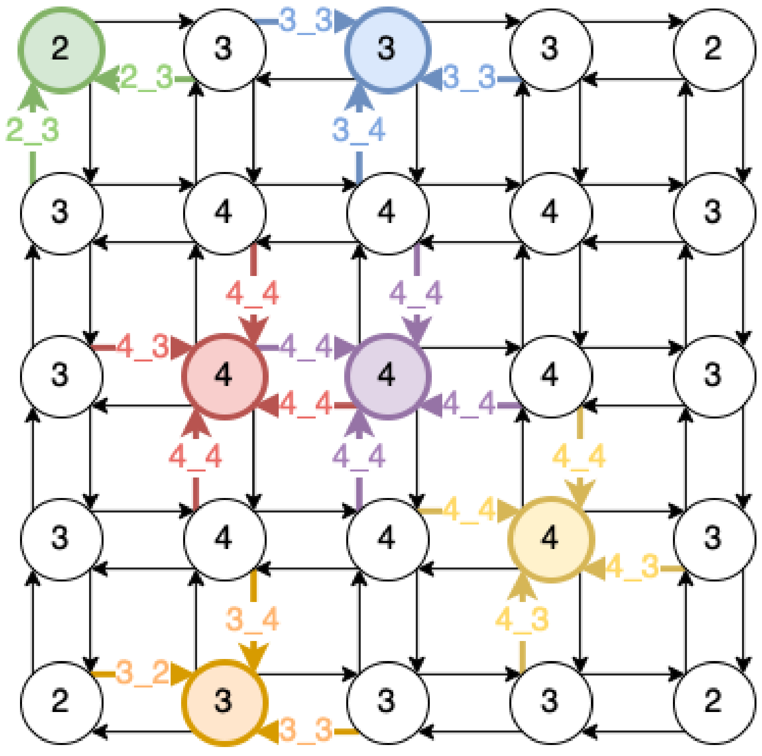
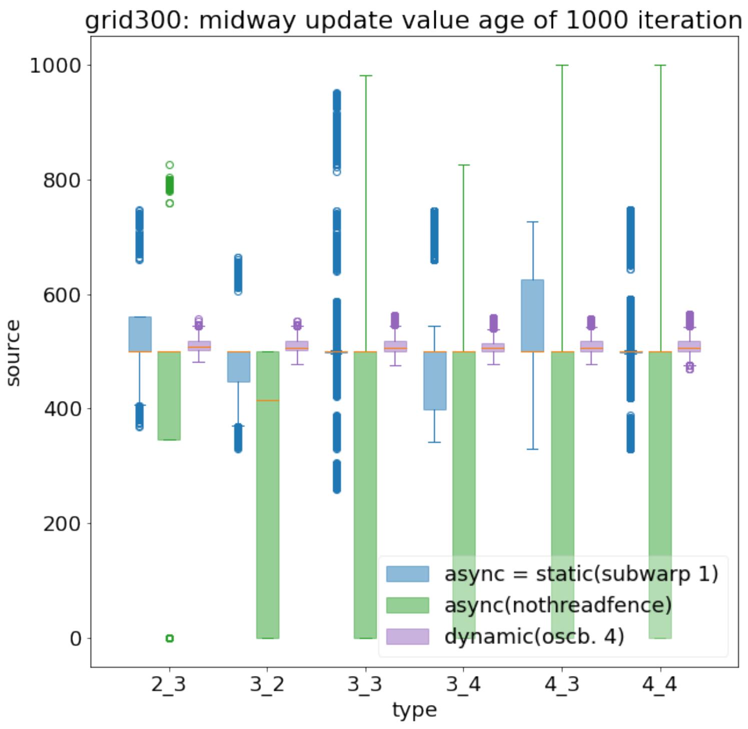


Figure 6.12: Update schedule for each result index, for the static(subwarp 32) on grid size of 100 × 100



(a) Communication schematic for grid points



(b) Update history for a 300^2 grid

Figure 6.13: Communication scheme and update history

7 Distributed Asynchronous iterative methods

To solve large scientific problems, we need to be able to effectively distribute the computing workload across the entire system. This involves distribution across all levels of the hierarchy: multi-core CPUs, GPUs, sockets, nodes, and node clusters. While synchronization within a level is relatively cheap, synchronization between hierarchical levels is expensive.

In the previous chapter, we explored the benefits of using asynchronous iterative methods on a single GPU, removing explicit synchronizations within an iterative solver running on a GPU. The particular method we focused on was the Richardson method and its asynchronous variant. This level of fine-grained parallelism is not suited for large distributed systems, as the overall ratio of communication to computation is high, and the available parallelism at the large distributed scale is left unutilized. To better utilize the hierarchical compute resources on distributed systems, we use a generalized block version of the Richardson method: the Schwarz domain decomposition method.

Schwarz iterative methods were first used to solve elliptic boundary value problems on irregular domains by decomposing them into regular domains, on which the solutions were known [Sch70]. With the advent of parallel computing, these domain decomposition methods became a natural way of distributing work to the compute units to enable parallel processing. The central idea is to partition the domain into smaller sub-domains, managing load-balancing where possible and solving each sub-domain in parallel. This is done iteratively, with the global solution being communicated between the distinct sub-domains every iteration [Saa03]. Each communication between neighbors, therefore, necessitates a synchronization.

Asynchronous variants relax synchronizations within an iteration and operate on existing or stale data with a partially updated global solution. In particular, these methods are beneficial with existing load imbalance [FSS97].

In Section 7.1, we study the convergence behavior of the asynchronous Schwarz methods, elaborating on the necessary conditions for convergence. Section 7.2 looks at a probabilistic model to empirically analyze the convergence behavior of the asynchronous Schwarz method, including multi-level Schwarz methods. We then implement multi-GPU asynchronous Schwarz solvers and compare them with their synchronous counterparts in Section 7.3. For this, we consider both a single-stage solver, in Section 7.4 and a two-stage solver, in Section 7.5. Finally, we provide a brief summary in Section 7.6.

We note that some contents of this chapter have been published in [NCA20a; NCA20b].

7.1 Convergence and optimality

The asynchronous Schwarz methods build upon the Schwarz domain decomposition methods, defined in Definition 2.50. We first look at the general asynchronous Schwarz, formulating a framework to analyze these methods and elaborate the necessary conditions for convergence. We also showcase the properties of an optimization of the standard asynchronous Schwarz method that approximates the inner solver also with an iterative approach.

7.1.1 Asynchronous Schwarz iterative methods

Definition 7.1 (Asynchronous Weighted additive Schwarz)**.** *Given an operator $\mathcal{F}_{schw}$ from $\mathbb{R}^n \to \mathbb{R}^n$, the asynchronous weighted additive Schwarz method from $\mathcal{F}_{schw}$, given an initial guess vector x_0, is a sequence of successive approximations, $x_k, k = 0, 1, \cdots, n$ of vectors in $\mathbb{R}^n$ defined by*

$$x^l_{k+1} = \begin{cases} x^l_k & \text{if} \quad l \notin \mathcal{I}_k \\ \sum_{j=1}^J D^{(k)}_{l,j} \quad y^j_{s_j(k)} & \text{if} \quad l \in \mathcal{I}_k \end{cases} \tag{7.1}$$

with

$$y^j_{s_j(k)} = M_j^{-1}(N_j x^j_{s_j(k)} + f) \tag{7.2}$$

where $\mathcal{I}_k \subseteq \{1, ..., J\}$ is the sequence of nonempty subsets, representing the iteration subsets, the data is gathered from, with $\mathcal{S}_k = s_1(k), \cdots, s_J(k)$, the sequence of elements in $\mathbb{N}_0^J$ representing the delays. In addition, the sets $\mathcal{I}$ and $\mathcal{S}$ are subject to the same conditions as for the asynchronous iteration, Equations (6.2a) to (6.2c).

The Schwarz method, both the synchronous (as in definition 2.50) and the asynchronous one are characterized by the splittings, $A = M_j - N_j \;\; j = 1, \cdots, J$ and the weighting matrices, $D^{(k)}_{l,j}$ such that $\sum_{j=1}^J D^{(k)}_{l,j} = I \;\; \forall l, k$. These weighting matrices define a decomposition of the monolithic matrix into sub-domains, where parallel independent local solves can be performed. The corresponding entries of the weighting matrices therefore define the weighting between the local solution $(y^j_{s_j(k)})$ to compute the global solution.

An asynchronous iteration corresponding to $\mathcal{F}_{schw}$ starting with x_0 and defined by the sets $\mathcal{I}$ and $\mathcal{S}$ will be denoted by $(\mathcal{F}_{schw}, x_0, \mathcal{I}, \mathcal{S})$.

Collecting the iterates from the different sub-domains, we have a vector $\mathbf{x}_k \in \mathbb{R}^{Jn}$

$$\mathbf{x}_k = (x_{k,1}, x_{k,2}, \cdots, x_{k,J}) \tag{7.3}$$

We also define $\mathbf{c}_k \in \mathbb{R}^{Jn}$

$$\mathbf{c}_k = \left(\sum_{j=1}^J D^{(k)}_{1,j} M_j^{-1} f \;, \;\cdots\;, \; \sum_{j=1}^J D^{(k)}_{J,j} M_j^{-1} f \right) \tag{7.4}$$

and denote the l-th block of the $\mathbf{c}_k$ by $[c_k]_l = \sum_{j=1}^J D^{(k)}_{l,j} M_j^{-1} f$. Setting $T_j = M_j^{-1} N_j, \; j = 1, \cdots, J$ and the global block weighting matrix $\mathbf{H}_k \in \mathbb{R}^{Jn \times Jn}$

$$\mathbf{H}_k = \begin{pmatrix} D^{(k)}_{1,1} T_1 & \cdots & D^{(k)}_{1,J} T_J \\ \vdots & \ddots & \vdots \\ D^{(k)}_{J,1} T_1 & \cdots & D^{(k)}_{J,J} T_J \end{pmatrix} \tag{7.5}$$

we can see that Equation (7.1) can be written as

$$[\mathbf{x}_{k+1}]_l = \begin{cases} [\mathbf{x}_k]_l & \text{if} \quad l \notin \mathcal{J}_k \\ \left[\mathbf{H}_k([\mathbf{x}_{s_1(k)}]_1, \cdots, [\mathbf{x}_{s_J(k)}]_J)\right]_l + [\mathbf{c}_k]_l & \text{if} \quad l \in \mathcal{J}_k \end{cases} \tag{7.6}$$

Lemma 7.1.1. *The fixed point, $\mathbf{x}^*$ of equations of the form $\mathbf{x} = \mathbf{H}_k\mathbf{x} = \mathbf{c}_k$, with $\mathbf{H}_k$ of the form eq. (7.5) and $\mathbf{c}_k$ of the form eq. (7.4) is*

$$\mathbf{x}^* = (x^*, \cdots, x^*) \ \in \mathbb{R}^{Jn} \tag{7.7}$$

with x^* being the solution of the linear system, $Ax = f$, $x^* = A^{-1}f$. If in addition, $\rho(\mathbf{H}) < 1$, then $\mathbf{x}^*$ is the unique fixed point of the affine operator, $\mathbf{x} \mapsto \mathbf{H}\mathbf{x} + \mathbf{c}$.

Proof. Since we know $Ax^* = f$, we can write $x^* = M_j^{-1}(N_jx^* + f)$ for $j = 1, \cdots, J$ and therefore

$$\begin{aligned} \sum_{j=1}^{J} D_{l,j}M_j^{-1}N_jx^* + \sum_{j=1}^{J} D_{l,j}M_j^{-1}f &= \sum_{j=1}^{J} D_{l,j}M_j^{-1}(N_jx^* + f) \\ &= \sum_{j=1}^{J} D_{l,j}x^* = x^* \quad \text{for} \ \ l = \ 1, \cdots, J \end{aligned} \tag{7.8}$$

If $\rho(\mathbf{H}) < 1$, then we have that the matrix $(\mathbf{I} - \mathbf{H})$ is non-singular, therefore $\mathbf{x} = \mathbf{H}\mathbf{x} + \mathbf{c} \leftrightarrow (\mathbf{I} - \mathbf{H})\mathbf{x} = \mathbf{c}$ has a unique solution, which is $\mathbf{x}^*$. □

Theorem 7.1.2 (Asynchronous Schwarz fixed point)**.** *Let $\mathbf{x}^0$ be some initial guess, and let $\|.\|_w$ be a weighted max norm, on $\mathbb{R}^{Jn}$. Additionally, let $\mathbf{H}_k$, $\ \ k = 1, 2, \ldots$ be a sequence of operators having a common fixed point x^**

$$\|\mathbf{x}\|_w := \max_{l=1,\cdots,J}\{\frac{1}{w_l}\|[\mathbf{x}]_l\|_l\} \tag{7.9}$$

with $w \in \mathbb{R}^J$, $\ w > 0$ and some constant $\theta \in [0, 1)$, let

$$\|\mathbf{H}_k\mathbf{x}\|_w \leq \theta \ \|\mathbf{x}\|_w \quad \forall \mathbb{R}^{Jn} \ \ k = 1, 2, \cdots, \tag{7.10}$$

Then the asynchronous Schwarz iteration defined in Equation (7.6) and consequently Equation (7.1) converges to $\mathbf{x}^*$ for every starting vector $\mathbf{x}^0$.

Proof. See Theorem 3.2 in [FS94] and Theorem 3.4 in [ET82]. □

Given an iteration matrix, T_j we also have the following asymptotic convergence result for the global solution

Theorem 7.1.3 (Asynchronous Schwarz asymptotic convergence)**.** *Given some vector $u > 0$, $\in \mathbb{R}^n$ such that*

$$|T_j|u \leq \theta u \quad \theta \in [0, 1) \quad \textit{for } j = 1, \cdots, J. \tag{7.11}$$

we have $\lim_{k\to\infty} \mathbf{x}_k = x^*$ for the asynchronous iterates defined in Equation (7.6)

Proof. Denoting $\mathbf{u} = (u, \cdots, u)$, we can write

$$[|\mathbf{H}_k|\mathbf{u}]_l = \sum_{j=1}^{J} |D_{l,j}^{(k)}T_j|u = \sum_{j=1}^{J} D_{l,j}^{(k)}|T_j|u \leq \theta \sum_{j=1}^{J} |D_{l,j}^{(k)}|u = \theta u \tag{7.12}$$

Therefore, $|\mathbf{H}_k|\mathbf{u} \leq \theta\mathbf{u}$. Therefore using Lemma 2.1.16 we get

$$\|\mathbf{H}_k\mathbf{x}\|_\mathbf{u} \leq \theta\|\mathbf{x}\|_\mathbf{u} \ \ \forall \ \mathbf{x} \in \mathbb{R}^{Jn} \tag{7.13}$$

But $\|\mathbf{x}\|_\mathbf{u} = \max_{l=1,\cdots,J}\|[\mathbf{x}]_l\|_u$, therefore the condition Equation (7.10) is satisfied with $a = (1, \cdots, 1)$. □

7.1.2 Two stage Asynchronous Schwarz iterative methods

The above discussion specifies that a solution needs to be computed on each subdomain, which is then weighted in some fashion. In general, when the subdomains are small, giving us small linear systems, we can use direct methods to obtain machine precision accuracy. For cases with larger subdomains, a two-stage approach, where the inner solution is also computed with an iterative solver, can be beneficial.

Definition 7.2 (Two stage asynchronous Schwarz). *Given the global splitting $A_j = M_j - N_j$, and an outer asynchronous iteration matrix defined by $T_j = M_j^{-1}N_j$, we see that each subdomain, can use a splitting $M_j = F_j - G_j$, $j = 1, \cdots, J$ for the inner solver. Let the subdomain solution $y_{s_j(k),j}$ be defined as in Equation (7.2). Assuming $q(j)$ iterations for the subdomain j, with an initial vector $x_{s_j(k),j}$, we have the resulting vector to be*

$$\tilde{y}_{s_j(k),j} = (F_j^{-1}G_j)^{q(j)}x_{s_j(k),j} + \sum_{t=0}^{q(j)-1}(F_j^{-1}G_j)^tF_j^{-1}(N_jx_{s_j(k),j} + f) \tag{7.14}$$

hence giving us the iteration matrix

$$\tilde{T}_j = (F_j^{-1}G_j)^{q(j)} + \sum_{t=0}^{q(j)-1}(F_j^{-1}G_j)^tF_j^{-1} \tag{7.15}$$

Setting $\tilde{M}_j = M_j(I - (F_j^{-1}G_j)^{q(j)-1})^{-1}$ and $\tilde{N}_j = A - \tilde{M}_j$, we have $\tilde{T}_j = \tilde{M}_j^{-1}\tilde{N}_j$. Therefore, we see that the two-stage variants can be represented in an equivalent form as for the single stage asynchronous iterative methods as shown in Equations (7.4) to (7.6). And hence the asymptotic convergence and the fixed points that have been shown for the standard single stage asynchronous Schwarz hold for the two-stage variants as well. Additionally, we have the following results for certain matrix classes that are common in many scientific applications.

Theorem 7.1.4 (Two stage asynchronous Schwarz convergence). *Given asynchronous iterates, x_{k+1}^l as shown in Equation (7.1) with a set of initial vectors x_0^l $l = 1, \cdots, J$ and a choice of inner iterations, $q(j) \geq 1$ $j = 1, \cdots, J$, the two-stage asynchronous weighted Schwarz method converges to x^* with the iteration matrices as given by Equation (7.15) in the following cases:*

(a) $A^{-1} \geq O$ and the splittings $A = M_j - N_j$ and $M_j = F_j - G_j$ are weak regular, with $F_j^{-1}N_j \geq 0$, $j = 1, \cdots, J$.

(b) A is a H-matrix, i.e $\langle A\rangle \leq \langle M_j\rangle - |N_j|$, and the splittings $M_j = F_j - G_j$ are H-compatible.

Proof. Denote $e = (1, \cdots, 1)$ as the unit vector in $\mathbb{R}^n$. We need to verify that the Equation (7.11) is satisfied for (a) and (b), which we see can be satisfied with $u = A^{-1}e$ for (a) and $u = \langle A\rangle^{-1}e$ for (b). More details can be found in [FS97]. □

7.1.3 Enhancing convergence with multi-level methods

While the single-level domain decomposition methods, such as the Schwarz methods, are highly parallel, they suffer from slow convergence due to the lack of information exchange between subdomains that are "far" from each other (subdomains that do not explicitly exchange information but are d-distance from each other, with d being the maximum depth/width in the dependency graph, considering the subdomains as nodes of the graph).

In Definition 2.50, we have seen the benefits of using a coarse space to enhance convergence. There are two methods of incorporating the information from the coarse space: additive and multiplicative, as shown in Algorithm 21. The multiplicative coarse correction requires re-computing the residual, while the additive coarse correction operates on the previously computed residual. As such, we expect the multiplicative coarse correction to be more effective than the additive version. However, due to its composition nature, it requires synchronization, making it difficult to use in asynchronous methods.

7.2 A probabilistic model

In a fashion similar to the plain asynchronous iterative methods, the lack of determinism in the asynchronous Schwarz methods means that we must resort to a probabilistic analysis. Our probabilistic model introduces delays sampled from a probabilistic distribution, and we analyze the convergence of the asynchronous Schwarz method based on the properties of this sampled probability distribution. The formal definition of the probabilistic model and its constraints are given in Definition 6.2.

Algorithm 21 The two level Asynchronous Schwarz method: probabilistic model

```
1: $A, b, x, \omega$
2: $r \leftarrow b - Ax, e \leftarrow \mathbf{0}$
3: for $k < N_{iter}$ do
4:     if $\|r\| < \tau$ then
5:         break
6:     end if
7:     for $j < N_{domains}$ do
8:         $d_j \leftarrow s_j(k) \in \mathcal{S}_k$                      ▷ Get the sampled delay
9:         $r_j \leftarrow b_j - A_j x_j^{d_j}$                          ▷ Communicate and update global solution with delay
10:        $e_j = A_{j,j} \backslash r_j$                                 ▷ Solve locally
11:        $x_j^{k+1} := x_j^k + e_j$                                    ▷ Update local solution
12:    end for
13:    if coarse_corr == additive then
14:        if mod($f$, $N_{iter}$)==0 then                              ▷ Update only for coarse correction frequency, $f$
15:            $e_{coarse} = A_{coarse} \backslash Rr$                   ▷ Compute coarse correction
16:            $x^{k+1} := x^{k+1} + \alpha_{coarse} P e_{coarse}$       ▷ Weighted additive coarse correction
17:        end if
18:    else if coarse_corr == multiplicative then
19:        $r \leftarrow b - Ax^{k+1}$                                   ▷ Compute updated residual
20:        $e_{coarse} = A_{coarse} \backslash Rr$                       ▷ Compute coarse correction
21:        $x^{k+1} := x^{k+1} + P e_{coarse}$                           ▷ Multiplicative coarse correction
22:    end if
23: end for
```

7.2.1 Choice of probability distributions

Table 7.1 shows the properties of the Half Normal distribution that we consider. We restrict ourselves to the Half-normal distribution for the analysis due to the coarse nature of the iteration compared to the asynchronous Richardson iteration presented in the previous chapter. Figure 7.1 shows the histogram for the probability distribution collected into 50 bins with 10^4 samples for a mean of 1.01 and variance of 0.05.

Given that the asynchronous Schwarz methods has independent delays for each subdomain, we need to sample $p_j(k) \in \mathbb{N}_0$ for each subdomain j and iteration k. As before, these sampled delays must fulfill the conditions elaborated in Definition 6.2.

7.2.2 Analyzing the convergence behaviour

We consider a matrix derived from discretizing a 2D Laplacian problem to study the convergence behavior using a 5-point stencil. A grid of size 8×8 leads to a matrix $A \in \mathbb{R}^{64\times 64}$ with the sparsity pattern and eigenvalue distribution as shown in Figure 7.2. To estimate the convergence rate, we follow a similar procedure as elaborated in Section 6.2.2

Table 7.1: Probability distributions

Probability distribution	Probabilistic distribution function	Mean/Expected value ($E[X]$)
Half-Normal	$f(x\|a,b) = \sqrt{\frac{2}{\pi}}\frac{1}{b}e^{-\frac{1}{2}(\frac{x-a}{b})^2}; x \geq a$	$\mu = a + b\sqrt{(\frac{2}{\pi})}$

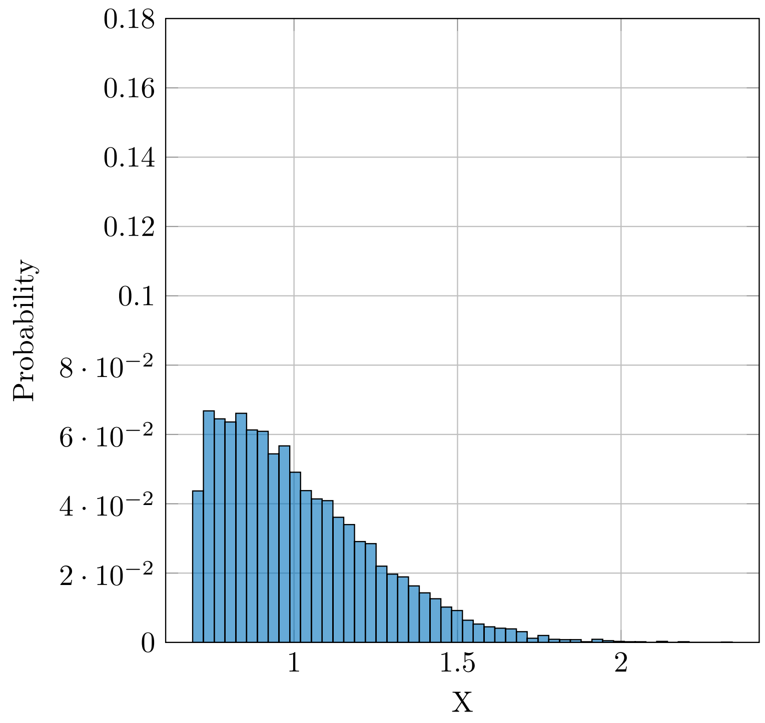
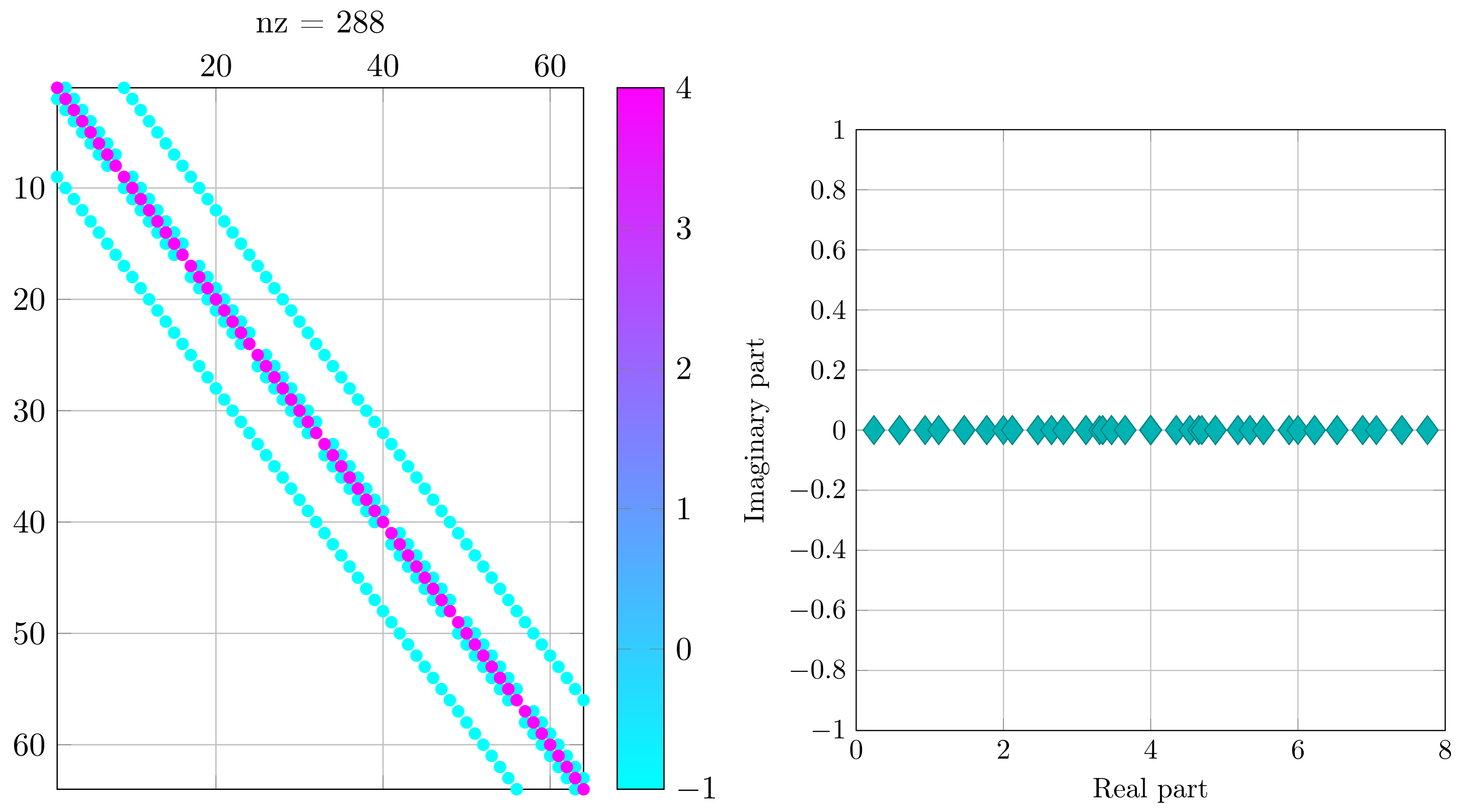


Figure 7.1: Histograms for the Half Normal probability distribution, $\mu = 1.01$, $\sigma = 0.05$ collected in 50 bins



(a) Sparsity pattern (b) Eigenvalue distribution

Figure 7.2: The 2D Laplacian problem on a 8 × 8 square grid with a 5 point stencil discretization.

by logging the residual norm history through the iteration process and performing a linear regression on the log of the residual norm. A negative slope implies convergence, a positive slope implies divergence and a zero slope implies stagnation. Each experiment is run 20 times with sampling the delay from a probability distribution.

Algorithm 21 shows the asynchronous Schwarz iteration used for the probabilistic modeling. The main idea is to sample the delay, d_j from a probabilistic distribution for each domain, solve locally and compute the global residual with this sampled delay.

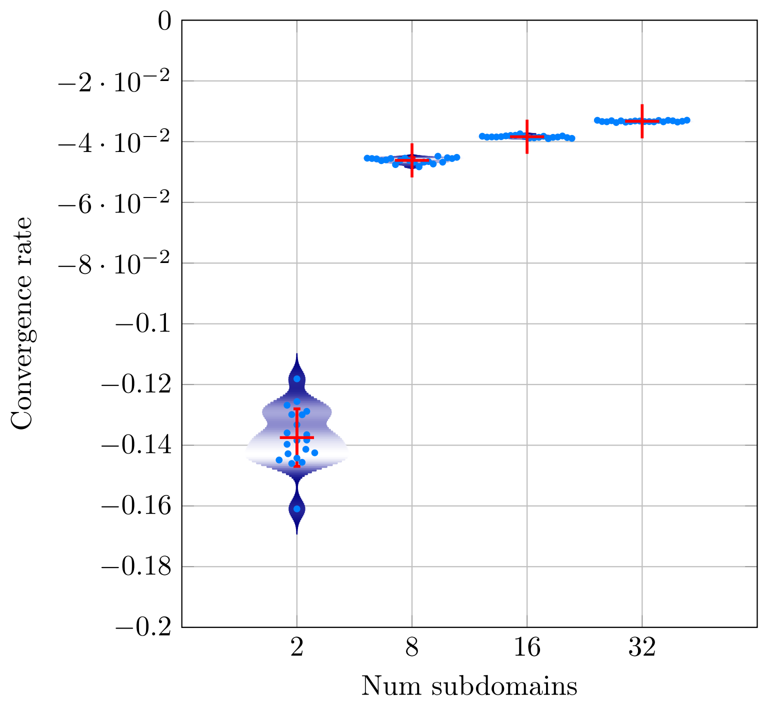
7.2.2.1 Incorporating coarse correction information

Figure 7.3 shows the need to incorporate coarse correction information, particularly when utilizing many subdomains. The convergence is slow without a coarse correction method. As shown in Figure 7.3b and Figure 7.3c, using a weighted additive coarse correction with weights $\alpha = \{0.2, 0.5\}$ enables better convergence behaviour. We observe that $\alpha = 0.5$ gives us the fastest convergence. For this choice, the local subdomain and global coarse-level corrections are weighted equally. We note that these coarse corrections are incorporated at a frequency of $f = 4$, which seem to be necessary to avoid over-correction for the asynchronous variants.

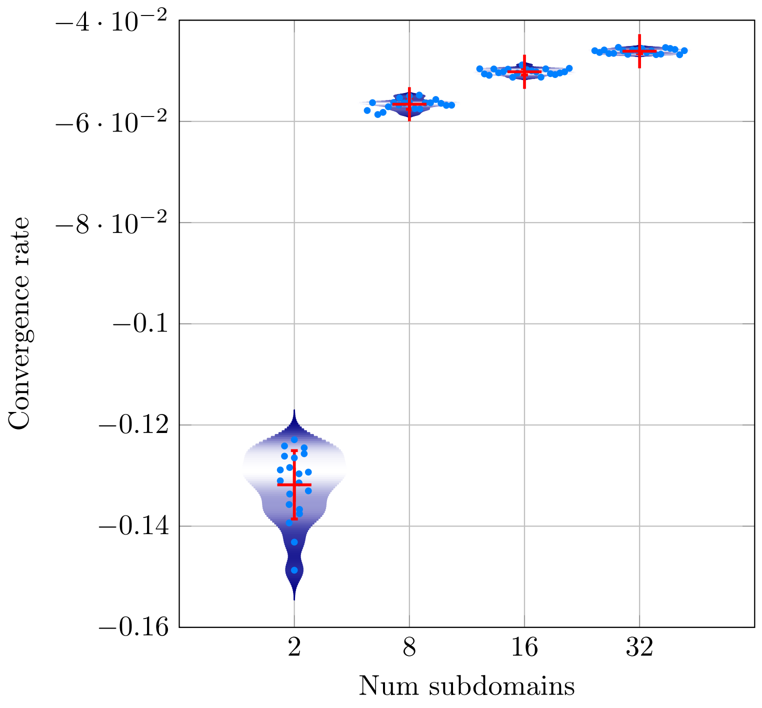
From Figure 7.3d, we see that the multiplicative coarse correction is effective and can significantly accelerate convergence, but due to its sequential application nature, it is unsuitable for the asynchronous Schwarz method.

7.2.2.2 Increasing the mean delay and variance

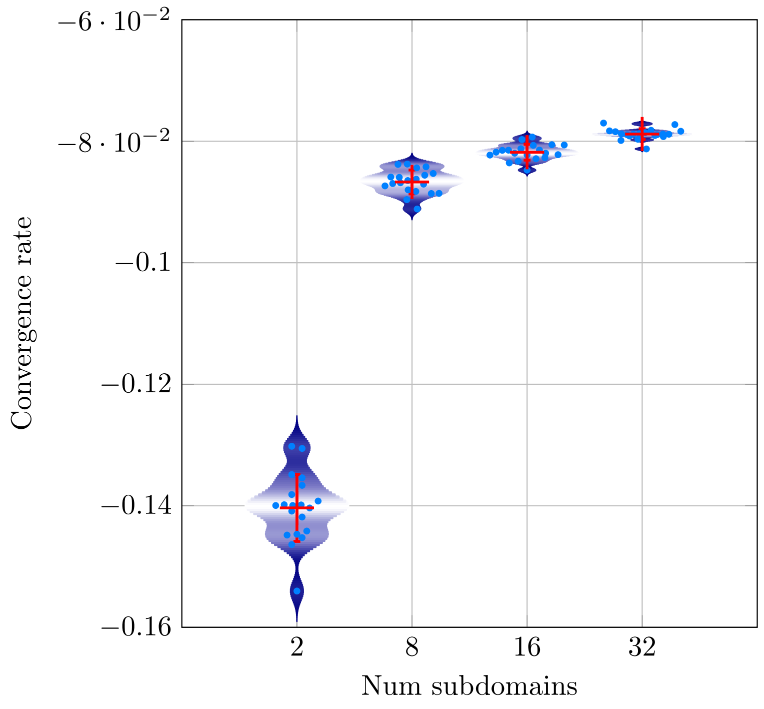
In Figure 7.4, we increase the mean delay by increasing the half-normal probability distribution's shape parameter, a. Increasing the mean delays from 0.32 to 1.32, we observe that a larger mean delay results in slower convergence.



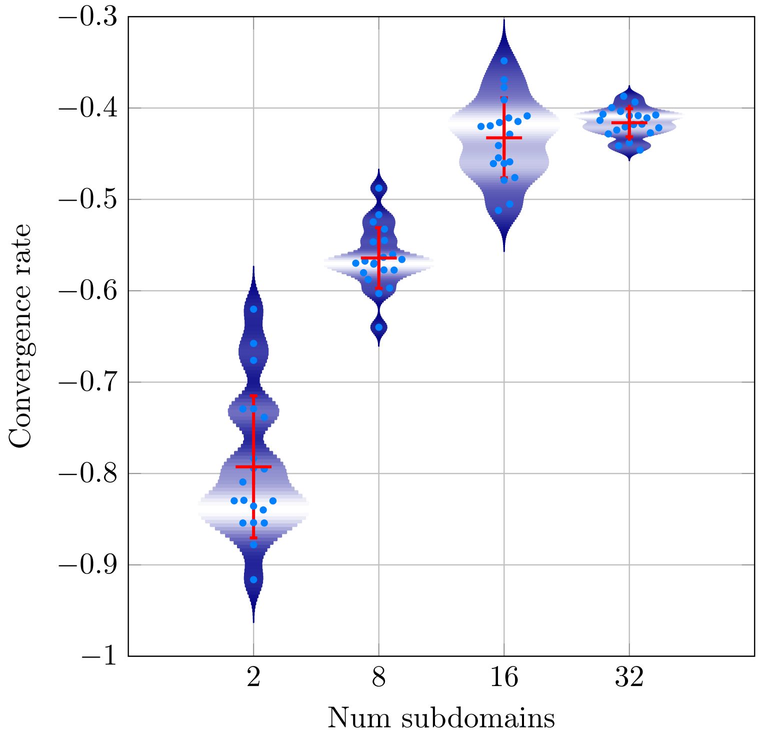
(a) No coarse correction



(b) Additive coarse correction, $\alpha = 0.2$



(c) Additive coarse correction, $\alpha = 0.5$



(d) Multiplicative coarse correction

Figure 7.3: With additive, multiplicative and no coarse correction for a Laplacian matrix of size 64×64

Increasing the number of subdomains from 2 to 64, we observe a plateauing of the convergence rates with increasing mean delays.

An important aspect to note is that for 64 subdomains, the method is identical to the asynchronous Richardson method (with diagonal preconditioner) as the problem size we consider here is 64×64, and we have one row (diagonal element) per subdomain.

Figure 7.5 shows the convergence rates with increasing the variance with a shape parameter, $a = 0.4$. Very high variances, > 0.4 imply an uncharacteristically large imbalance in the system, and is therefore quite rare [HSL10]. We do observe that the convergence suffers for these types of systems. For typical systems, with a variance around 0.1, we observe behavior similar to Figure 7.4b, though with a larger spread among the different runs due to the larger variance in the sampled delay.

7.2.2.3 Incorporating coarse correction information with different frequencies

A peculiar observation for the two-level additive coarse correction of the asynchronous variants is the divergence of the solution when the coarse correction is added in every iteration. This behavior has also been observed by other authors [GRB+18], but the concrete reason is yet unknown. To study this behavior, we vary the additive coarse correction frequency, f, from 1 to 8. With $f = k$, we add the weighted coarse correction to the solution only every k iteration.

From Figure 7.6, we clearly see that for $f = 1$ the solution diverges. For $f = 2$, while the solution converges for a smaller number of subdomains, it diverges when the number of subdomains increases. For $f = \{4, 8\}$, the solution converges. Furthermore, experiments reveal that the convergence rate reduces for a large f. This is because the information from the coarse correction is not sufficiently incorporated into the solution.

The above observations lead us to postulate that similar to the effect of the relaxation factor observed in the previous chapter for the asynchronous Richardson iteration, reducing the frequency of the coarse correction reduces the over-correction and stabilizes the asynchronous Schwarz method. In general, the optimal frequency appears to be dependent on the problem size, condition number and the number of subdomains.

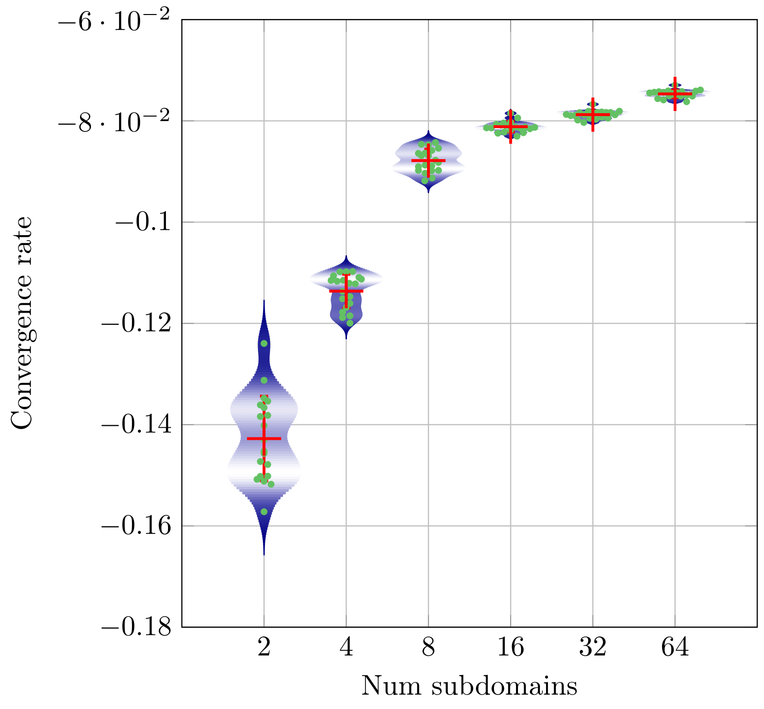
7.2.2.4 Multi-level coarse correction

Figure 7.7 shows the convergence behavior using a single coarse level and two coarse levels, both being added in an additive fashion. We observe that the two-level method (one coarse level + one fine level) is competitive with the three-level (two coarse levels + one fine level) method, particularly for a larger number of subdomains. Additionally, the three-level method requires an additional linear system solution and is more expensive than the two-level method.

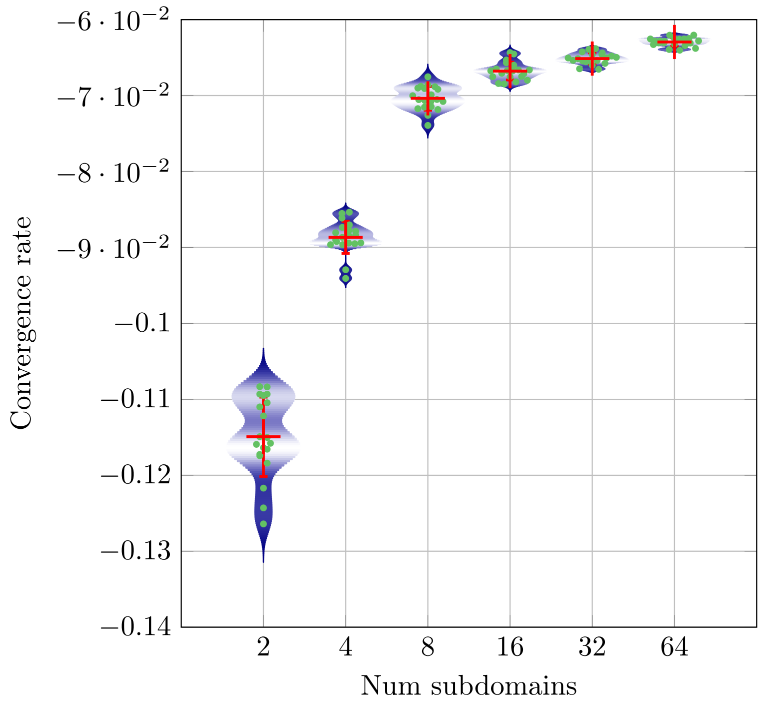
7.3 An asynchronous Schwarz multi-GPU solver

The framework we develop interacts with multiple external tools to provide specific functionalities, among which are METIS ([KK98]) for partitioning, DEAL.II [ABF+22] for discretization of PDEs, CHOLMOD [Dav13] for local direct solvers and GINKGO [ACC+20b; ACF+22] for the iterative solvers, generic data management and execution management features. See Chapter 3 for more details.

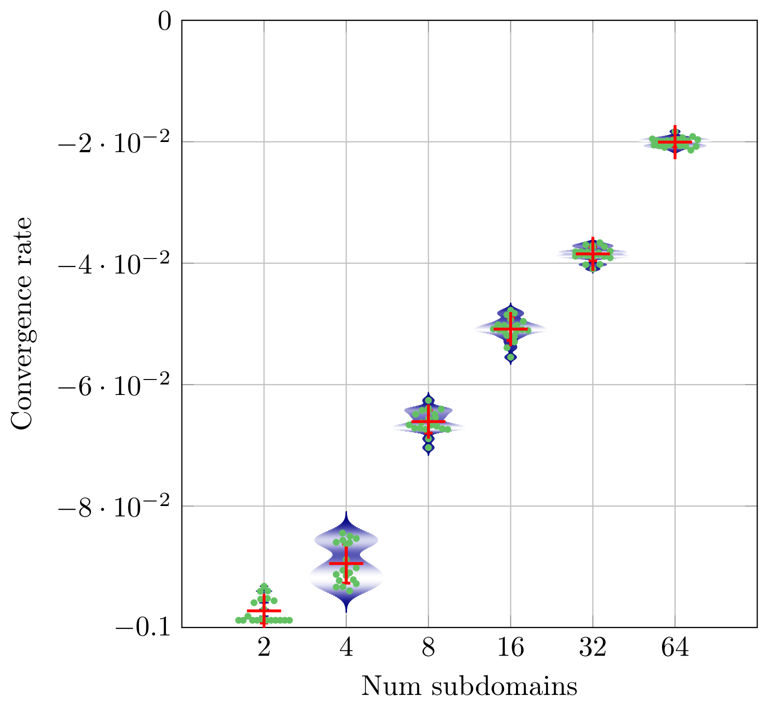
The core algorithm we explore is the Schwarz algorithm. We focus on the overlapping Schwarz method to enable convergence without a coarse solver. In particular, the restricted additive Schwarz (RAS) method is generally superior to the weighted additive Schwarz methods [CS99]. Algorithm 22 shows the steps involved in the RAS solve. The initialization step handles the setup and initialization of the solver, while the solve step handles the local solves, communication, convergence detection, and the termination of the solver. The timings and performance data we report cover only the solve step, as the initialization and setup is a constant, non-repeating part of the RAS solver.



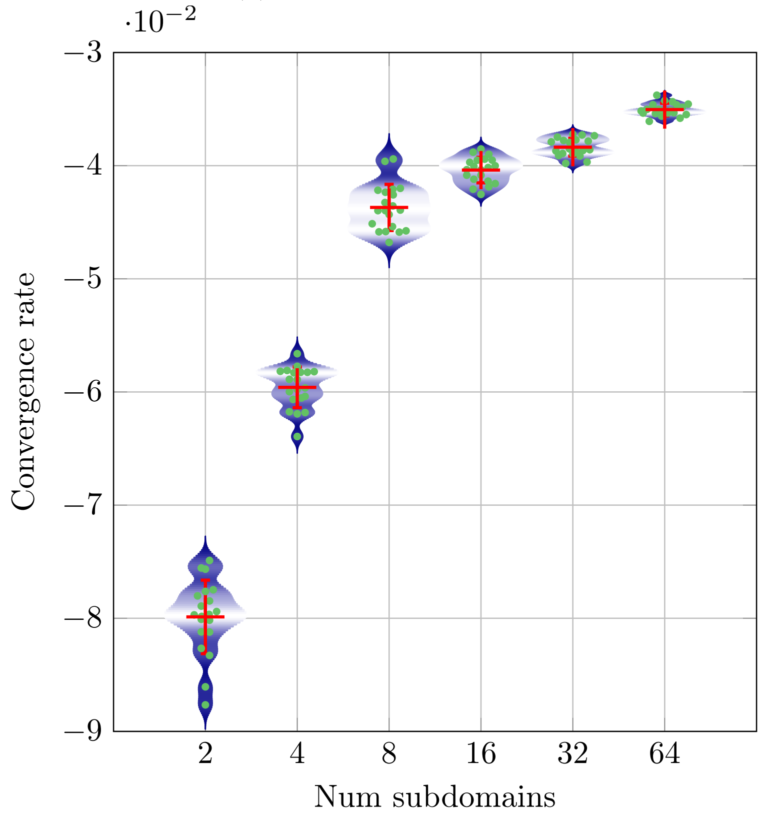
(a) Mean delay, $\mu = 0.32$



(b) Mean delay, $\mu = 0.56$

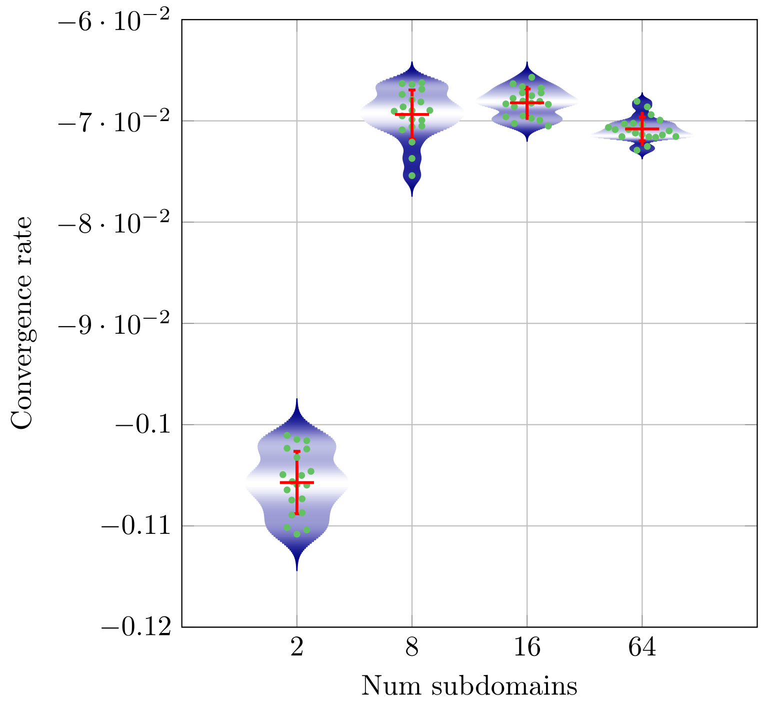


(c) Mean delay, $\mu = 0.82$

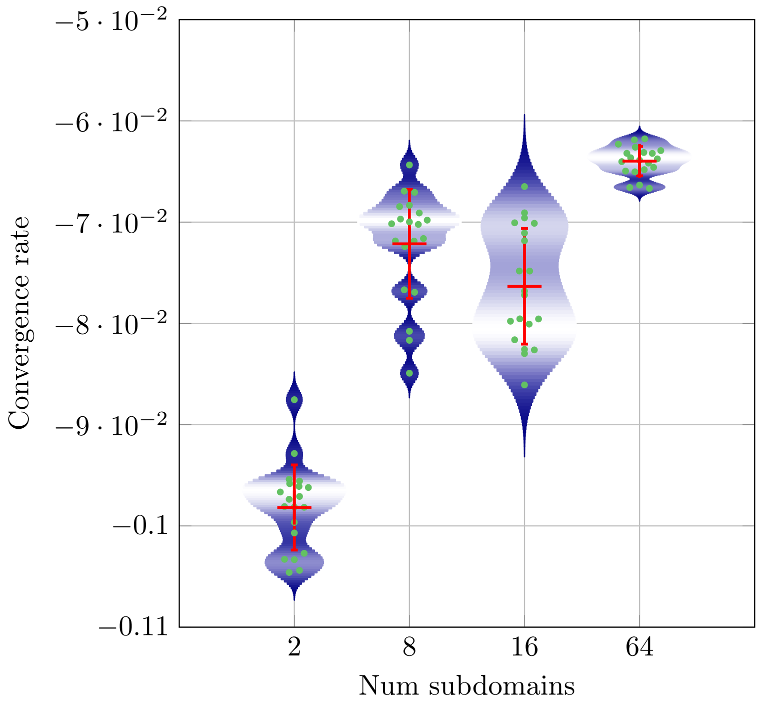


(d) Mean delay, $\mu = 1.32$

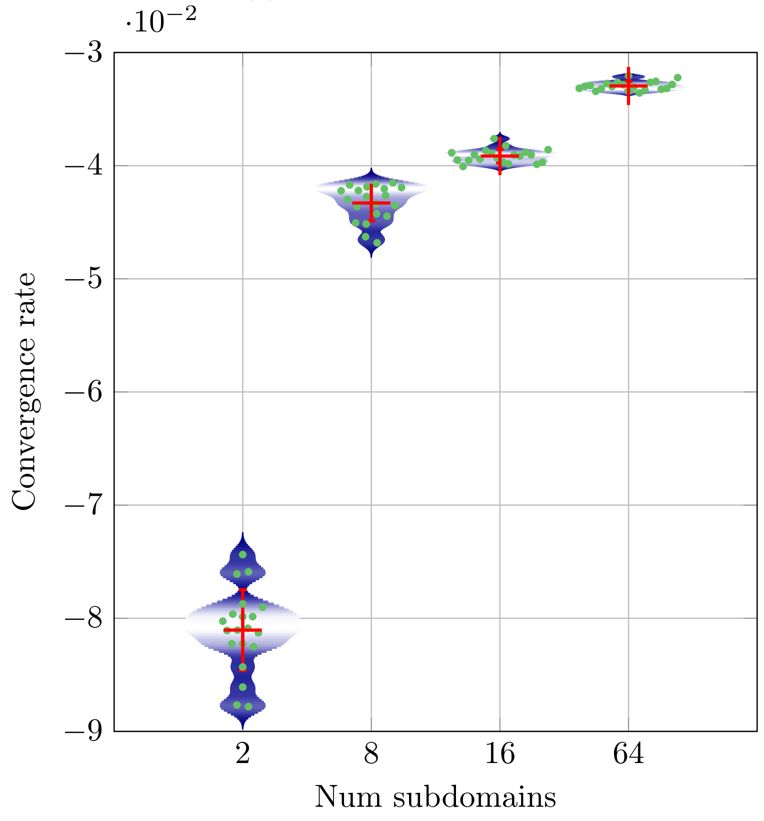
Figure 7.4: Increasing the mean delays with coarse correction, for Laplacian matrix of size 64×64



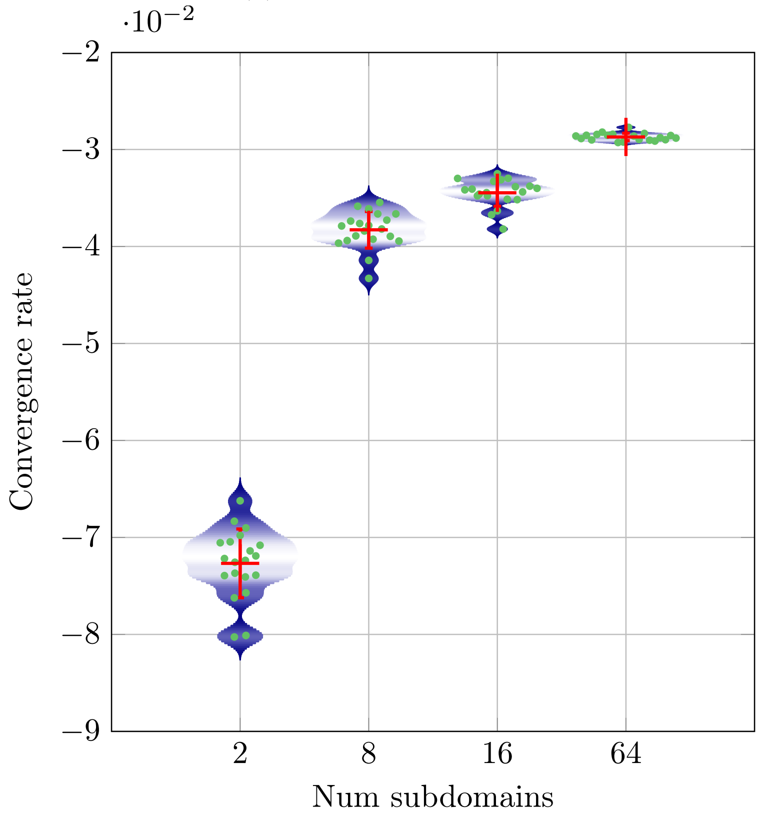
(a) Variance, $\sigma = 0.023$



(b) Variance, $\sigma = 0.098$

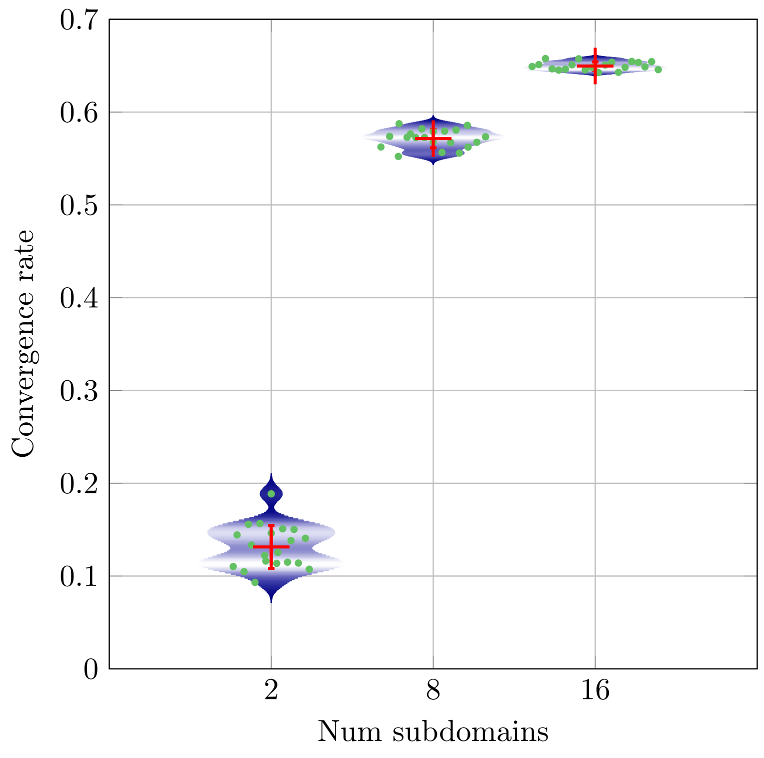


(c) Variance, $\sigma = 0.36$

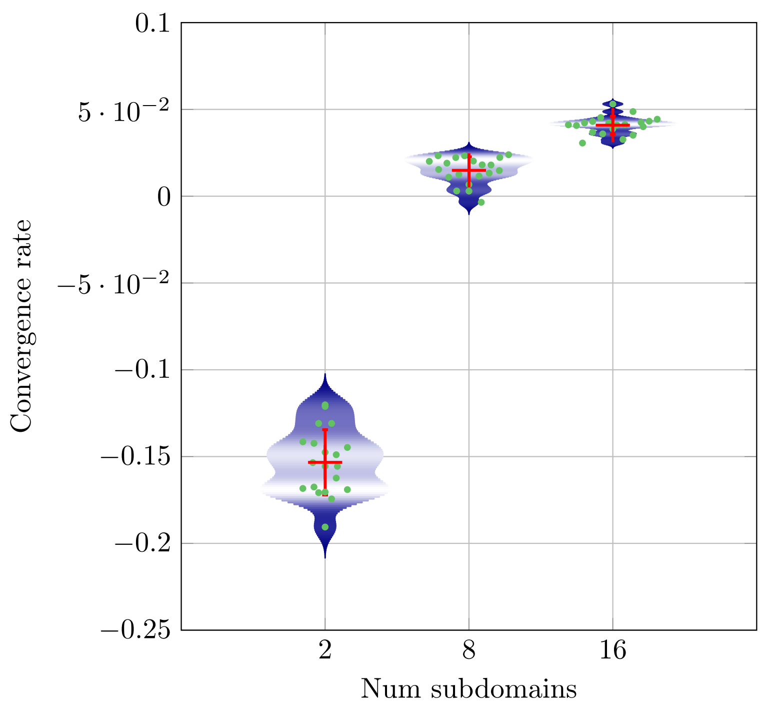


(d) Variance, $\sigma = 0.82$

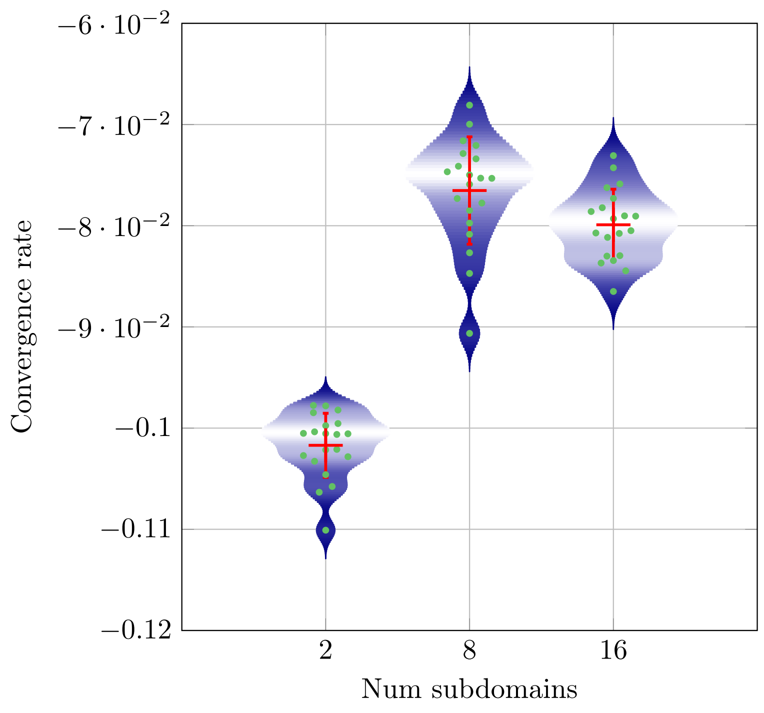
Figure 7.5: Increasing the variance with coarse correction, with a $= 0.4$, for Laplacian matrix of size 64×64



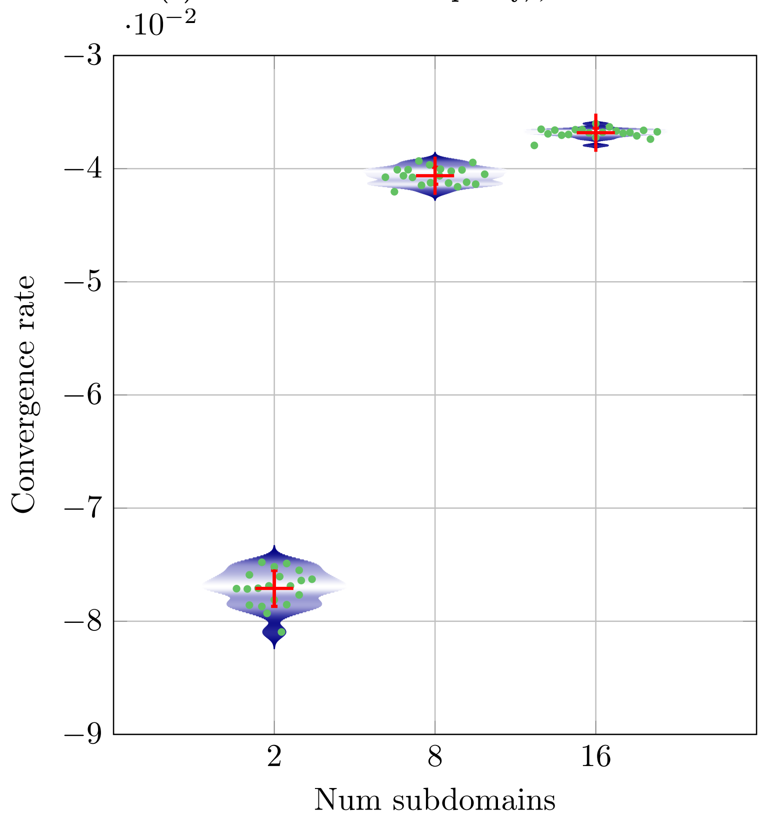
(a) Coarse correction frequency, $f = 1$



(b) Coarse correction frequency, $f = 2$

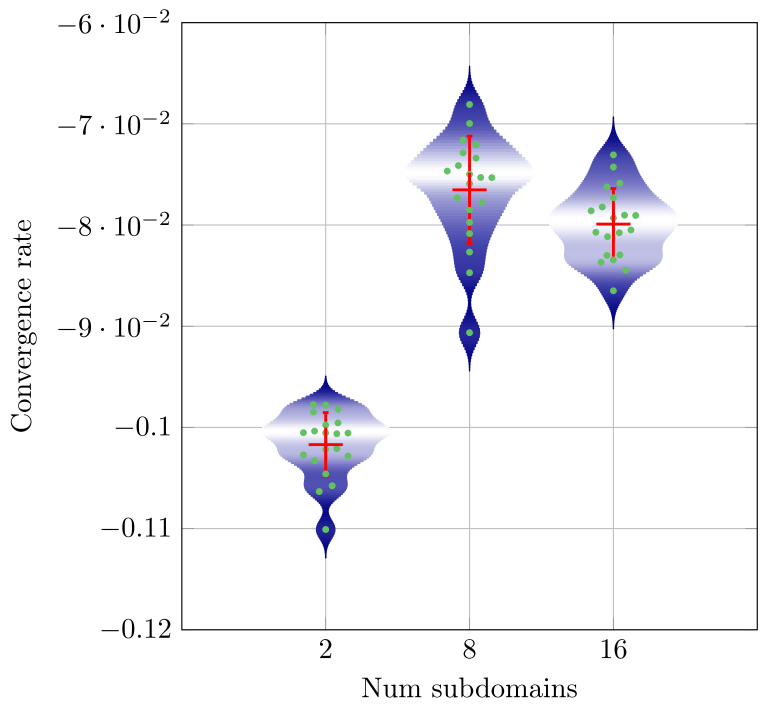


(c) Coarse correction frequency, $f = 4$

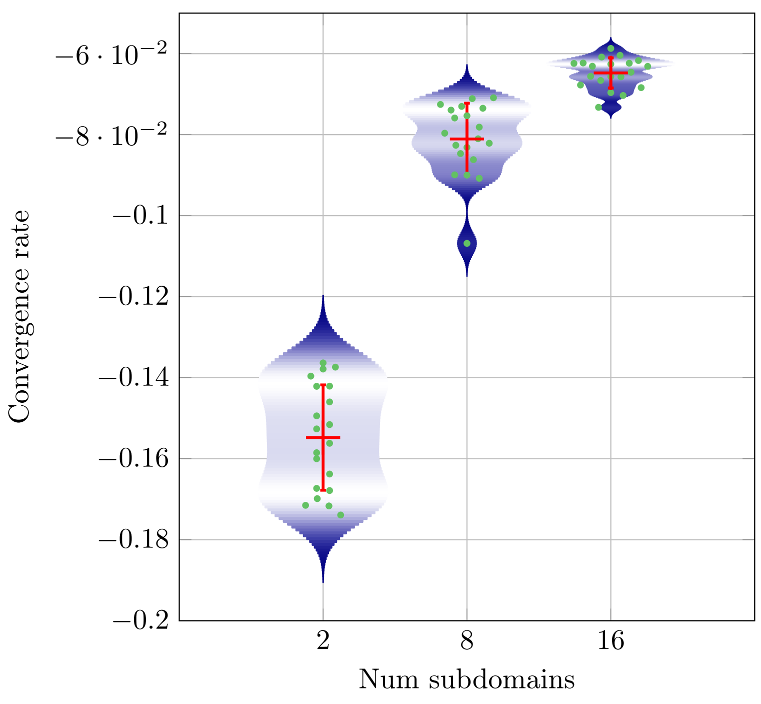


(d) Coarse correction frequency, $f = 8$

Figure 7.6: Varying the frequency of the coarse correction, a = 0.4, b=0.4, for Laplacian matrix of size 64 × 64



(a) Number of coarse correction levels, $\ell = 1$



(b) Number of coarse correction levels, $\ell = 2$

Figure 7.7: Varying the number of the coarse correction levels, a = 0.4, b=0.4, for Laplacian matrix of size 64×64

Algorithm 22 Restricted Additive Schwarz Iterative solver

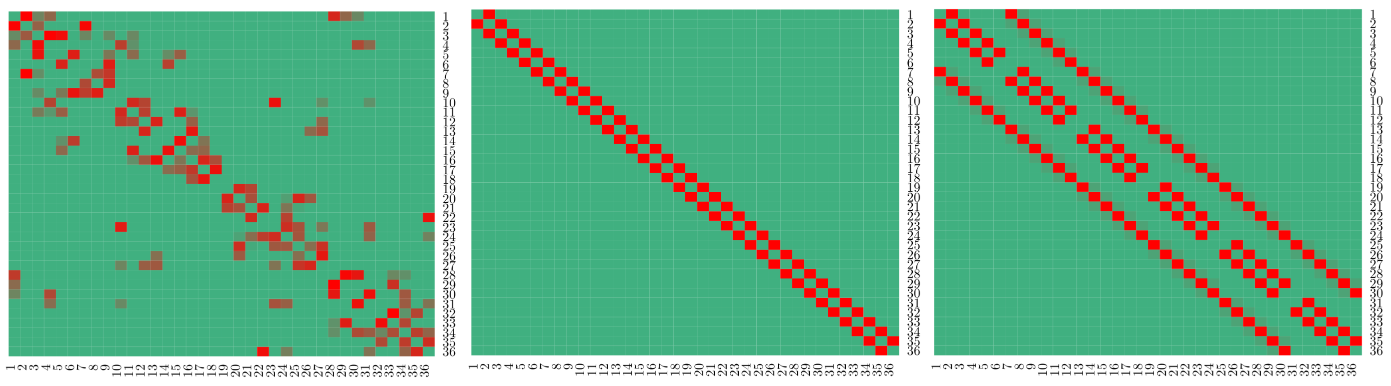
```
1: INPUT: $A, b, x$
2: $r \leftarrow b - Ax$
3: for $k < N_{iter}$ do
4:     $e_j = A_{j,j} \setminus r_j \quad \forall j \in \{1, \cdots, J\}$    ▷ Local solve: Direct, $\tau_{local} = \epsilon_{machine}$ accuracy
5:     $x = x + e$    ▷ Update local solution
6:     $r = b - Ax$    ▷ Exchange boundary info
7:     if $\|r\| < \tau$ then    ▷ Check for Global convergence: Centralized or Decentralized
8:         break
9:     end if
10: end for
```

7.3.1 Setting up local matrices

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 7.3.3 as it is a separate component impacting both the initialization and the solve characteristics.

Our Schwarz framework [NCA20a] supports three partitioning schemes as shown in Figure 7.8. We see the communication patterns that arise for the three partitioning schemes as heatmaps for a 2D Laplace problem discretized using a 5 point stencil. The partition strategy and the sparsity of the problem govern the communication pattern. The filled-in colors indicate the cells (and the corresponding subdomains) that communicate. Different shades of red reflect the communication volume for each cell and a green cell indicates no communication for that cell.

A simple **regular-1d** partition has at most two neighbors and a regular communication pattern. However, the information propagation is inefficient due to the maximum "distance" between the subdomains. For N subdomains, the rate



(a) Metis (b) 1D partition (c) 2D partition

Figure 7.8: Communication patterns resulting from different partitioning schemes for 36 subdomains

of information propagation, defined here as the amount of solver iterations needed for data exchange between the two farthest subdomains, is $N-1$ [SGT07].

In a two-dimensional `regular-2d` partitioning, each subdomain can have at most four neighbors. This allows for faster information propagation at the cost of increased communication compared to the `regular-1d` scheme.

On the other hand, using the `metis` partitioning scheme, each subdomain can have multiple neighbors, and therefore, the effective maximum "distance" for a subdomain is, in general, significantly smaller.

The partitioning scheme assigns each grid point to a subdomain (or multiple if the grid point is part of the overlap). This information is used to assemble the local subdomain matrix, and an interface matrix is used to communicate between the different subdomains using a 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.

7.3.2 Solving the local problems

Once the local matrices have been assembled and given a certain overlap, we can solve each local problem in parallel. Using the partition information, we map the global to local vectors and vice-versa, which is necessary for the communication step's residual and solution vectors.

For the solution of the local problems, as we have seen, we can: 1) Use a direct solver (such as a LU/Cholesky factorization) and subsequently two triangular solves, or 2) an iterative solver such as a Krylov method (CG, GMRES, BiCGSTAB, etc.).

We first focus on direct local solutions and use the Cholesky factorization available in the CHOLMOD library (part of the SuiteSparse package) [Dav13]. The arising triangular systems are handled with Ginkgo's triangular solve – which in turn interfaces to NVIDIA's cuSPARSE library. Using CUDA version 10.1, the `cusparse_csrsm2` sparse triangular solve routine is based on the level-set strategy and can handle multiple right-hand sides.

7.3.3 Communication

The tailored `Communication` class handles the communication between the subdomains. It sets up the required buffers, allocates and initializes the MPI windows necessary for the one-sided communication. Additionally, the `Communication` class also provides functionalities utilized in the convergence detection and boundary exchange (used by the `Solver` class). The synchronous and asynchronous Schwarz algorithms differ mainly in their communication strategies:

Synchronous communication: Each step in the solve procedure in Algorithm 22 is performed in a lock-step fashion. Each subdomain is associated with one MPI rank, further mapped to one GPU. This computational unit performs a local solve, and in each iteration, it exchanges information with its neighbors as necessitated and defined by the partition.

For efficient synchronous communication and to enable the overlapping of communication and computation, we use the non-blocking MPI point-to-point functions, namely `MPI_Isend` and `MPI_Irecv`. Additionally, given that all our data is resident on the GPU, it is vital to use a GPU-aware MPI implementation for efficient communication between GPUs on the same node and between nodes. The data is packed before sending and unpacked at the receiver. An `MPI_Wait` enforces synchronization between the subdomains in each iteration.

Asynchronous communication: Each step in the solve procedure in Algorithm 22 is executed without synchronizing with neighbors. The mapping of subdomains to MPI ranks and GPUs remains the same as in the synchronous version. To enable asynchronous communication between the subdomains without deadlocks, we use the Remote Memory Access (RMA) functions introduced in the MPI-2 standard [Mpi]. These functions enable remote MPI ranks to access dedicated buffers without explicit synchronization. Each process "put"s its data into the buffer of its neighbor through a "window" using the MPI RMA function `MPI_Put`. To ensure the operations are completed, the `MPI_Win_flush` (or other similar sync operations) "flush"es the data on the windows.

7.3.4 Convergence detection

As the asynchronous method does not need global synchronizations, a synchronizing convergence detection such as a global reduction (using an `MPI_AllReduce`) is highly inefficient. As each subdomain computes a local solution and iterates without waiting for data from another subdomain, we must adopt asynchronous termination detection strategies. We showcase two different types of termination strategies:

Centralized, tree-based : With a centralized termination detection strategy, the subdomains are arranged in a tree fashion, with one root and recursively spawning leaves each of its neighbors. In this strategy, the leaves detect local convergence and propagate this to their root, and global convergence is achieved once the main root has detected convergence from all its leaves [YCB+19]. Once the main root has detected convergence, termination flags are recursively sent to all the leaves.

The local convergence criterion for each of the subdomains is

$$||\tilde{r}_p||_2^2 < \tau^2 ||\tilde{b}_p||_2^2, \tag{7.16}$$

where $\tilde{r}_p$ is the local residual vector with the values of the subdomain p (including those in the overlap), τ is the global solver tolerance, and $\tilde{b}_p$ is the local right hand side. The global convergence criterion is

$$||r|| < \tau ||b||$$

where r is the global residual vector and b is the global right-hand side. Once all the subdomains have satisfied their local convergence criteria, we additionally verify that the global convergence criterion is satisfied post-termination.

Decentralized, leader based: With the centralized termination strategy, we have a single root process, which can be a bottleneck when many MPI ranks are being used. Using a decentralized termination algorithm, we can remove this bottleneck. In this strategy, each subdomain considers the number of neighbors that have sent a local convergence message. Once all neighbors have confirmed convergence, the subdomain broadcasts the global convergence flag to all its neighbors, which in turn broadcasts it to their neighbors, and so on. The local convergence criterion is the same as in Equation (7.16) and, as in the previous algorithm, the processes first detect their local convergence status and propagate this to their neighbors [BCC+05].

7.4 Single stage multi-GPU Schwarz solvers

We first explore the single-stage asynchronous and synchronous Schwarz solvers and analyze their performance for various parameters. The experimental analysis focusses on the 2D Laplace problem discretized using a 5 point stencil, $\mathcal{S} = \{-1, -1, 4, -1, -1\}$ with offsets $\{-N, -1, 0, 1, N\}$ and N are the grid points in each discretization direction. The resulting linear system is regular; therefore, when partitioned, the linear systems to be solved are balanced between different subdomains. These types of well-balanced problems have been known to be efficient for synchronous methods [Bau78]. We use a random right-hand side and define the tolerance for the global RAS solver as $1e-7$.

The benchmark runs are conducted on the Summit supercomputer at the Oak Ridge National Lab. Each node is composed of two IBM Power 9 CPUs and 6 NVIDIA Tesla V100 GPUs. The 3 GPUs connected to the same socket are directly connected via an NVLINK brick with a 50GB/s bi-directional bandwidth; see Figure 2.9. Each V100 features 16GB of High Bandwidth Memory (HBM2). The Summit supercomputer contains more than 4,600 nodes of this kind. The nodes are connected via a dual-rail EDR InfiniBand network (non-blocking fat tree) providing a bandwidth of 23GB/s. Hence, in the scaling analysis, we need to consider the different bandwidth capabilities. The fastest communication between two GPUs is through the direct NVLINK (50GB/s) that connects GPU's of the same node. The second fastest communication is the off-socket on-node, but still on node: NVLINK (50GB/s) – Inter-socket (64GB/s) – NVLINK (50GB/s) path. The slowest communication is between GPUs attached to different nodes, which requires going through the Interconnect (23GB/s).

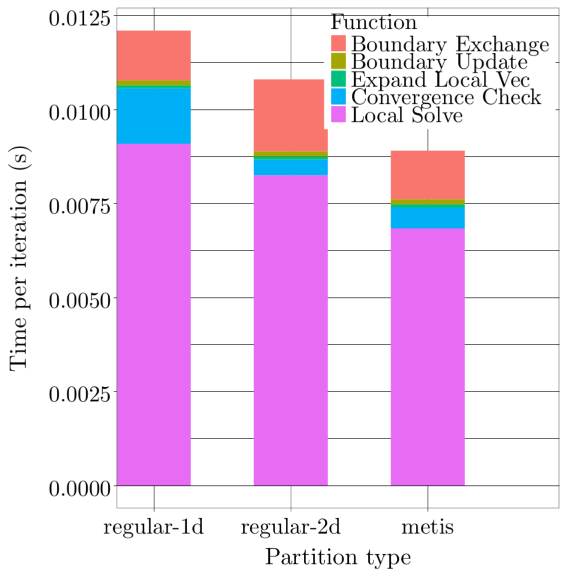
We use GINKGO as the central building block for the RAS solver. The local on-node (on-GPU) operations use GINKGO functionality, which calls CUDA functions internally, while the explicit communication between the subdomains (mapped to different GPUs) is handled via MPI. To maximize efficiency, we use the IBM Spectrum MPI (a specific implementation of OpenMPI), a vendor-optimized MPI implementation for the Summit system. The behavior of different MPI implementations, including the performance of the one-sided RMA functions have been studied by other authors [YCB+19].

Due to the non-deterministic nature of asynchronous methods, it is necessary to run all experiments multiple times and collect the average runtime across these runs for benchmarking purposes. In our case, we ran each experiment 10 times. We note that all runs converge to the same solution and tolerance, and the solution is verified to be correct. Multiple runs are necessary only to ensure runtime reproducibility and are not necessary for production runs when integrated into applications.

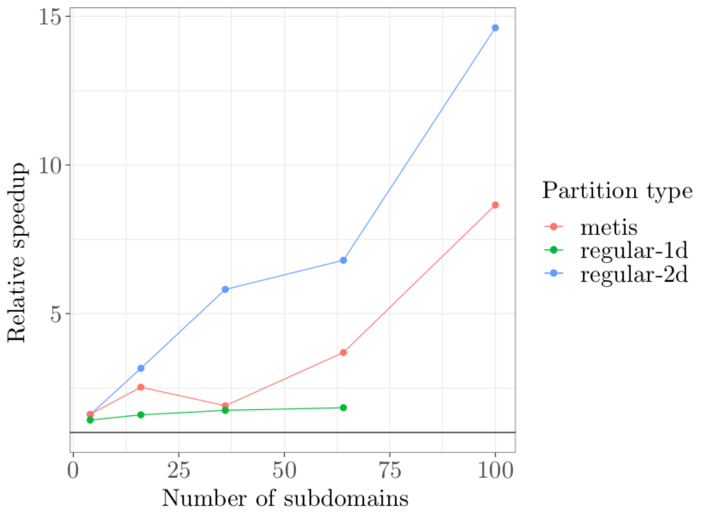
7.4.1 Comparing partitioning strategies

Figure 7.9a shows the time required for one synchronous RAS iteration using non-blocking two-sided MPI communication averaged over all the subdomains (the global problem has 262144 unknowns divided into four subdomains) for different partitioning strategies. Partitioning affects the solution in two main ways: 1) It controls the fill-in governed by the factorization of the local subdomain matrices (as we use a direct solver here). `regular-2d` and `metis` strategies can better minimize the fill-in of the resulting local matrices when distributing the rows to the subdomains. Hence, the local solution is cheaper for these strategies compared to `regular-1d`. 2) It dictates the communication volume between the subdomains. `metis`, being an objective-based partition designed to minimize the communication volume, provides a more efficient partition, as can be seen from Figure 7.9a.

We perform a weak scaling analysis to analyze performance on a larger number of GPUs. Fixing the local problem size to 4096 elements per subdomain, we look at the speedup of the asynchronous solver over the synchronous version for the three partition strategies in Figure 7.9b. We increase the number of subdomains (GPUs) from 4 to 100. The solid black line indicates the reference time for the synchronous method. We see that the `regular-1d` partition does not converge in a reasonable amount of time for a large number of subdomains (>64) for the synchronous and the asynchronous versions due to the slow information exchange. While the `metis` partition is the most efficient overall (it is about 1.3× faster than `regular-2d` partition), we see significant speedups (approximately 15×) for the `regular-2d` partition for 100 GPUs when compared to the synchronous method with a `regular-2d` partition. This can be attributed to the larger communication volume for the `regular-2d` partition, which cannot be well-hidden in the synchronous method. For all further experiments, we consider only the `metis` partitioner.



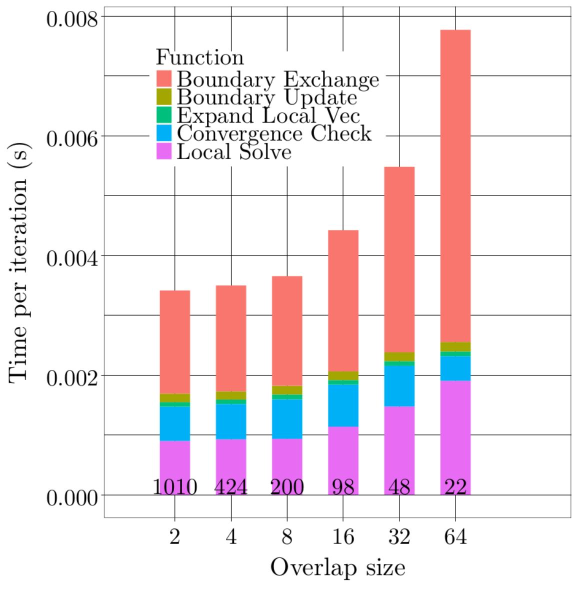
(a) Averaged subdomain update breakdown



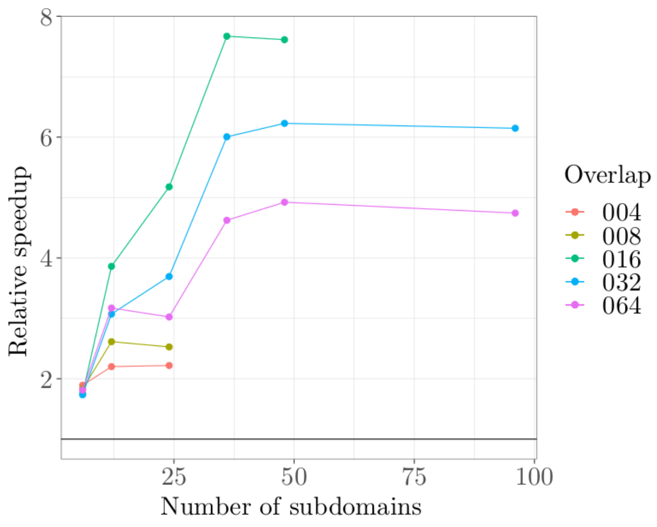
(b) Asynchronous vs Synchronous

Figure 7.9: Comparing the different partitioning strategies.

7.4.2 Comparing different overlaps



(a) Runtime breakdown for the averaged subdomain update & subdomain update count



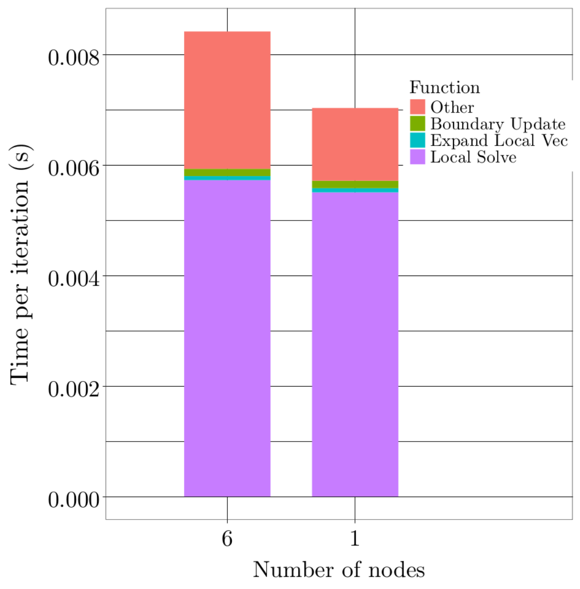
(b) Speedup - Asynchronous vs Synchronous

Figure 7.10: Comparing performance for different overlaps between the subdomains.

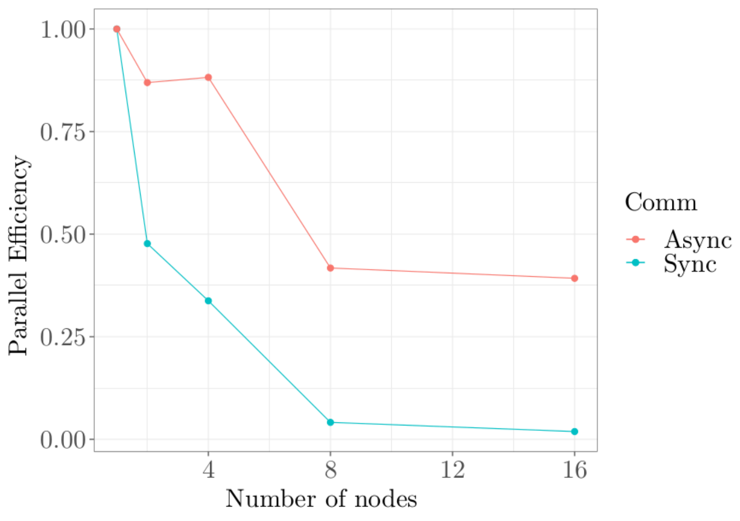
In Figure 7.10, we study the performance of the Schwarz methods with varying overlap between the subdomains. Unlike the traditional Schwarz methods, the overlap we implement is algebraic and applicable to any matrix, even ones not derived from a grid-based discretization. In Figure 7.10a, we study the breakdown for different functions for the synchronous Schwarz method for a global problem size of 16384 with six subdomains and increase the overlaps between the subdomains from 2 to 64. Increasing the overlap increases the time spent for the local solution and the boundary exchange due to the larger local matrix size and the amount of information to be communicated. Nevertheless, we also see that larger overlaps significantly reduce the number of global iterations due to the improved information exchange between the subdomains.

The optimal overlap has to balance these aspects. It depends on the problem characteristics, the partitioning scheme, and the used architecture's ratio between compute power and memory bandwidth. The asynchronous method benefits from a large overlap because larger overlaps imply higher communication costs, which are relatively lower for the asynchronous variant due to no explicit synchronizations. This is also seen from Figure 7.10b, where we compare the speedup for the same overlap size of the asynchronous version against the synchronous version with increasing subdomains. The time to converge for the lower overlap values, namely "4" and "8" for the synchronous version for the larger subdomains (above 24), is high and is excluded from this graph. The asynchronous version performs much better than the synchronous version when using an overlap of size "16", as most of the communication time can be hidden. Above a certain number of subdomains (48 subdomains here), the synchronous version takes too long to converge due to the larger local problem size.

7.4.3 Comparing scalability of asynchronous and sychronous methods



(a) Averaged runtime breakdown (Asynchronous)



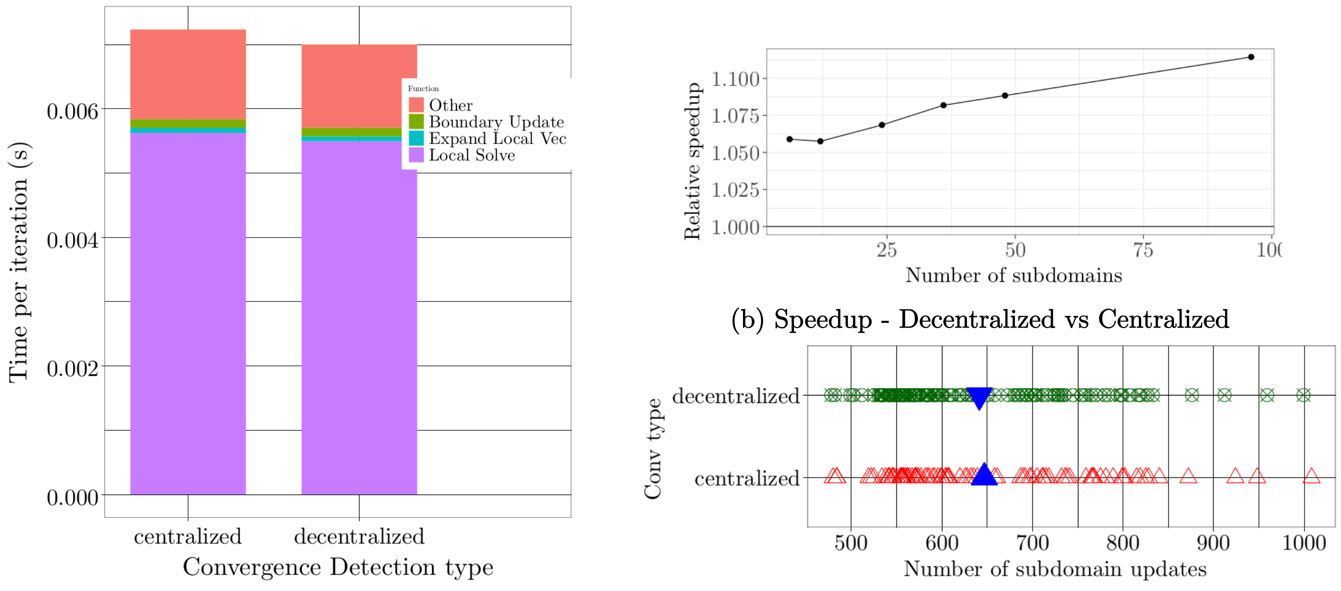
(b) Scaling Efficiency

Figure 7.11: Performance scaling for asynchronous and synchronous Schwarz methods.

As shown in the node diagram in Figure 2.9, a larger imbalance exists between GPUs on one node and GPUs on different nodes. This is attributed to the varying bandwidths: intra-node GPU bandwidth is 50GB/s through NVLINK, while inter-node bandwidth is 23GB/s through the Infiniband network. In Figure 7.11a, we see the time per iteration with the breakdown for different components for the asynchronous method with all subdomains (6, one subdomain per GPU) on one single node v/s using six nodes (one GPU per node). We see that the local solver cost remains unaffected while the communication cost decreases by about 20%. We note that for the asynchronous methods, isolating the communication cost component is difficult, and it is bunched with the "Other" part of the bar graph.

In Figure 7.11b, we compare the synchronous and asynchronous methods' weak scaling efficiency. Using a fixed problem size of 4096 elements per subdomain and with six subdomains per node (with one subdomain mapped to 1 GPU), we increase the number of nodes and compare the time to solution with that of 1 node. The effect of global synchronization with the synchronous Schwarz method means that we struggle to scale beyond four nodes. The asynchronous method shows a better scaling behavior, and for 96 GPUs, we have a parallel efficiency of 37%. We note that the scaling behavior of the Schwarz iterative methods without a coarse correction is known to be sub-optimal. Our objective is to show that the asynchronous methods scale better than the synchronous method, and using a coarse correction, these scaling behaviors can be further improved.

7.4.4 Comparing convergence detection algorithms



(a) Averaged runtime breakdown for the subdomain update

(b) Speedup - Decentralized vs Centralized

(c) Subdomain update spread

Figure 7.12: Comparing the centralized and decentralized convergence detection strategies.

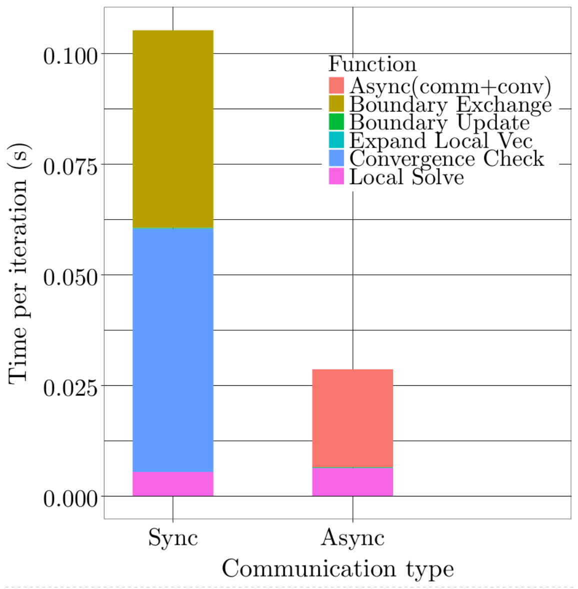
To study the effects of the convergence detection strategies, we compare the performance of the termination detection algorithms. Figure 7.12 compares the centralized, tree-based strategy with the decentralized leader election-based strategy. Both variants use a `metis` partitioner, an overlap of 16 elements and a problem size of 262144. From Figure 7.12a, we see that from a time-per-iteration, the strategies are mostly equivalent, with decentralized being slightly faster. Additionally, Figure 7.12b shows that for a small number of subdomains, the decentralized does not have any benefits. With increasing subdomain count, we see more concrete benefits of using a decentralized termination strategy, where the root process does not form the bottleneck. Figure 7.12c shows that the update spread, the number of local solves performed by a subdomain, is similar between both the strategies, with the decentralized strategy having a slight advantage (being able to detect and communicate termination earlier).

7.4.5 Comparing two-sided and one-sided communication

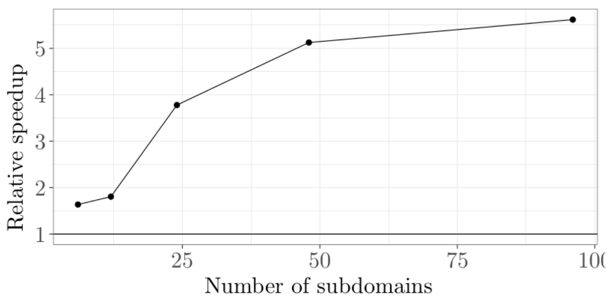
Finally, we compare the synchronous and the asynchronous variants of the Schwarz method in Figure 7.13. A runtime breakdown of the time per iteration (averaged, for asynchronous), shows that both the methods spend a similar amount of time in the local solve, but the synchronous variant spends approximately 4× more time in the convergence check and boundary exchange than the synchronous variant. We see this translate to an overall speedup of over 4× as we increase the number of subdomains of the asynchronous method over the synchronous method, reaching a peak of 5.5× for 96 subdomains as shown in Figure 7.13b.

To better understand the convergence, we plot the update spread of the asynchronous method in Figure 7.13c. The synchronous method needs 690 global iterations for this particular case, meaning that all subdomains update and perform 690 local solves. We have a few subdomains updating more than 690 times for the asynchronous method, for a maximum of 820, and some updating only 150 times due to its asynchronous nature. The median of these subdomain update counts is around 590, showing that, on average, most subdomains need to perform fewer local solves and communicate less than the synchronous variant.

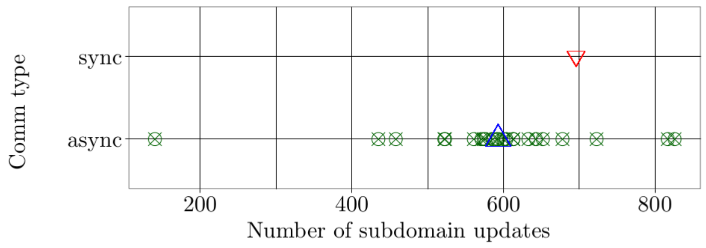
Despite this problem being well-balanced, asynchronous methods are pretty advantageous. Given more sophisticated problems with load imbalances, we can expect the asynchronous method to perform better.



(a) Runtime breakdown for the averaged subdomain update



(b) Speedup - Asynchronous vs Synchronous



(c) Subdomain update spread

Figure 7.13: Comparing the synchronous and the asynchronous Schwarz methods.

7.5 Two stage multi-GPU Schwarz solvers

Each subdomain needs to compute a local solution and solve a linear system for every update it performs. In the previous section, we used a direct solver with Cholesky factorization (pre-computed on the CPU) and a GPU-based triangular solver on the GPU. Given that each update is an approximation and is iteratively refined, we can benefit by using a preconditioned iterative solver instead. Using an iterative solver, we reduce the computational cost of the local solve, solving the local problems to a lower accuracy. A tolerance for the inner local iterative solver allows balancing between the local and global iterations, allowing problem-specific tuning. This is shown in Algorithm 23.

Algorithm 23 Two stage restricted Additive Schwarz Iterative solver

```
1: INPUT: $A, b, x$
2: $r \leftarrow b - Ax$
3: for $k < N_{iter}$ do
4:     $e_j = A_{j,j} \setminus r_j \quad \forall j \in \{1, \cdots, J\}$      ▷ Local solve: Iterative ($\tau_{local} > \epsilon_{machine}$ accuracy)
5:     $x = x + e$      ▷ Update local solution
6:     $r = b - Ax$      ▷ Exchange boundary info
7:     if $\|r\| < \tau$ then      ▷ Check for Global convergence: Decentralized termination
8:         break
9:     end if
10: end for
```

As before, we use `metis` partitioning, but allow the overlaps to be varied. We again store all matrices in the CSR format.

7.5.1 Iterative local solutions

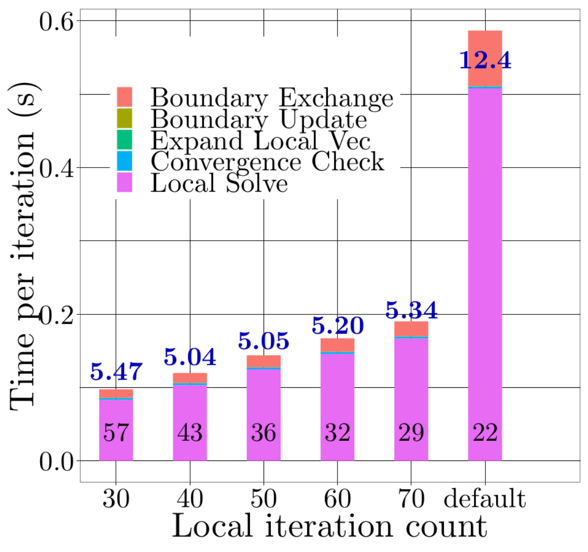
After assembling the local problems, to compute the local solution, we use the conjugate gradient (CG) method for symmetric positive definite (SPD) matrices and the Generalized Minimal Residual method(GMRES) for non-SPD matrices. This is determined beforehand and set at runtime for all the subdomains. As the condition numbers can be quite high, we use preconditioners to improve the effectiveness of the local iterative solvers. We experiment between two types of preconditioners: A block-Jacobi preconditioner, that is based on (block-) diagonal scaling, and a ILU preconditioner with ILU factors generated through the ParILU algorithm [CP15].

The local subdomain update count refers to the RAS solver's traditional "global iteration" count. Local iteration counts refer to the number of iterations the local iterative solver performs in the subdomain. The local subdomain solver criterion in this work refers to the relative norm *reduction*, where we aim to reduce the current residual norm of the current local solution and not the absolute residual norm.

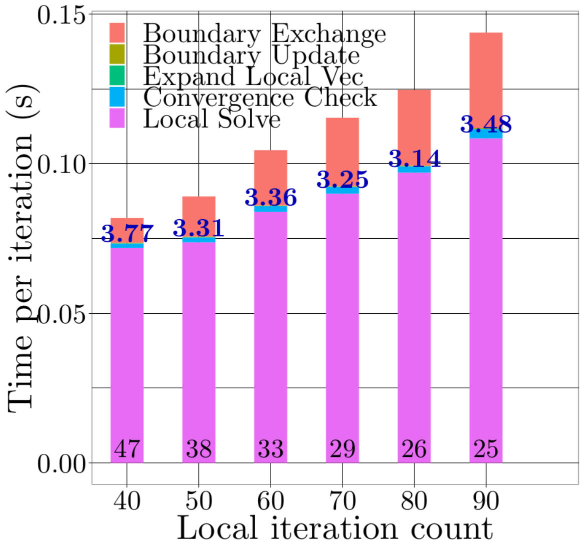
7.5.2 Tuning the local solver accuracy

We experiment with linear systems derived from two types of problems. First, a Laplacian problem, with second-order finite element discretization, where we use a preconditioned CG method as the local solver. Second, an advection problem, with fifth-order finite element discretization, 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.

Figure 7.14 shows the runtime breakdown for one synchronous Schwarz iteration using a CG local solver preconditioned with a block-Jacobi preconditioner (averaged over all subdomains, 2×10^6 unknowns with 6 subdomains). We see that the local solve dominates the overall runtime. With increasing local solver iterations, the local solve is more accurate, and its corresponding cost is higher. Consequently, we need to perform fewer global RAS iterations, as shown in the inset under the bars in Figure 7.14. This increased accuracy in the local solution does not always translate to better runtime but is dependent on the problem at hand. For the Laplacian problem, the optimal number of iterations is around 40, while for the advection problem, the optimal number of iterations is 80, reflected in the total runtime, shown in blue above the bars. The default case for both problems has a local solver tolerance of $1e-3$ and is omitted for the advection problem, as it takes 729s to converge and obscures the graph readability. We also see that the advection problem is harder to solve, as it is more dense and, being non-SPD, requires the more sophisticated GMRES solver.



(a) Laplacian problem

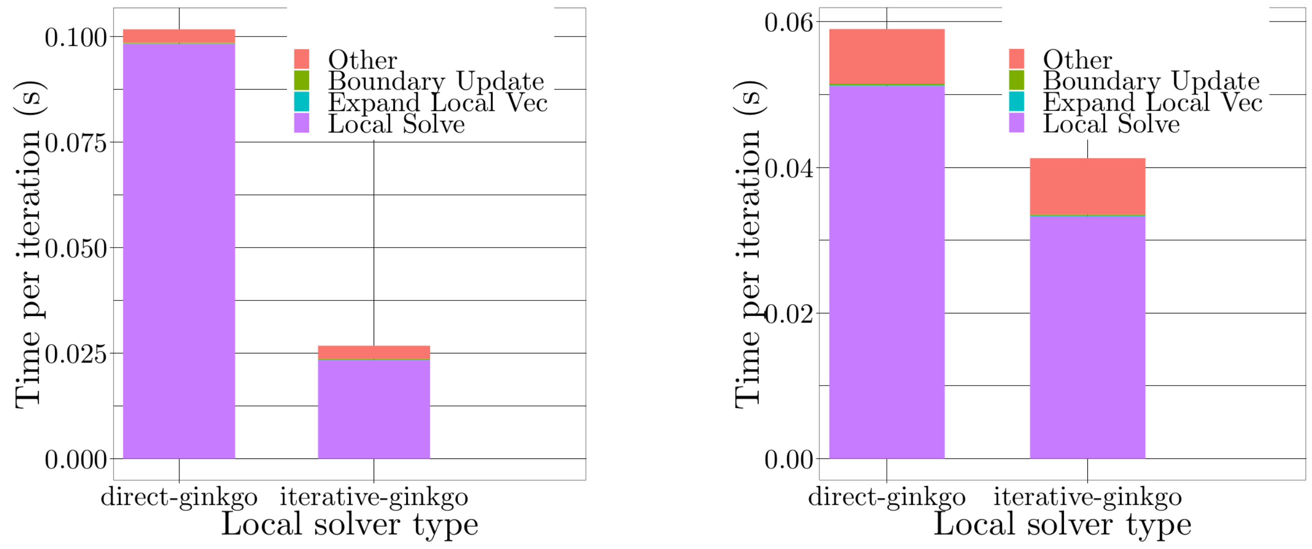


(b) Advection problem

Figure 7.14: The runtime breakdown in the different functions for different local solver maximum iteration criteria.

7.5.3 Comparing with local direct solvers

To showcase the effectiveness of the iterative solvers for the local systems, we compare them with the direct solvers. For the Laplacian problem, we pre-compute the Cholesky factorization on the CPU using the CHOLMOD library. For the advection problem, we use the UMFPACK library to pre-compute the LU factorization. We note that we do not include the solver factorization time in the performance analysis. In each RAS iteration, we then only need to perform sparse triangular solves (forward and backward pass), for which we use cuSPARSE's triangular solver on the GPU. Figure 7.15 shows the time per iteration for both problems using the asynchronous RAS iteration. For the Laplacian problem, we see that the iterative solver is, on average, 4× faster. For the advection problem, the GMRES iterative solver is about 1.5× faster. The asynchronous RAS for the Laplacian problem, with a direct solver, needs, on average, 21 subdomain updates and takes 2.05s, while the two-stage variant with PCG iterative solver needs 27 subdomain updates and takes 0.6s, which is about 3.5× faster than the single-stage asynchronous RAS solver. Similar behavior is seen for the advection problem, using a GMRES local solver, which converges in 0.73s compared to 1.05s for the direct solver, giving a speedup of 1.4×.



(a) Laplacian problem

(b) Advection problem

Figure 7.15: Comparing direct and iterative local solvers

7.5.4 Analysis of the residual reduction.

In Figure 7.16, we plot the residual and iteration history for two subdomains (0 and 31) through the whole RAS iteration process for the advection problem with overall 36 subdomains. Subdomain 31 (subd-31) has relatively more expensive local solutions and hence performs fewer overall updates. Subdomain 0, on the other hand, has cheaper local solutions and performs significantly more updates. The asynchronous solver allows the faster subdomains to optimize their local solves by reducing the iteration limit if they are yet to receive data from their neighbors and are computing on old data. The jumps in the residual denote this incorporation of new information. From a runtime perspective, we see that when subdomain 0 is performing its 85th global update, subdomain 31 is on its 50th update. This autonomy in iteration update, as a result of the asynchronous nature of the algorithm, enables the asynchronous RAS solver to outperform the synchronous version.

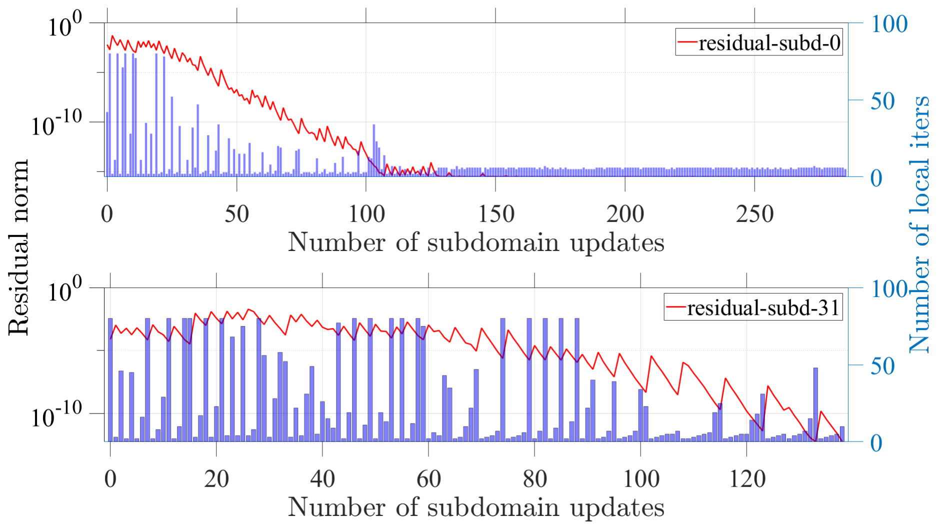


Figure 7.16: Iteration and residual norm history for two subdomains.

7.6 Summary

This chapter looked at coarse-grained asynchronous solvers that can harness parallelism at the node and multi-node levels. We first defined the asynchronous single-stage and two-stage Schwarz methods and established the conditions that were necessary for their convergence.

Using a probabilistic model, we empirically studied the convergence rates of asynchronous Schwarz methods and parameterized them on the probability distribution. We also explored multi-level asynchronous Schwarz methods, particularly the need to avoid over-correction. Using the probabilistic model, we also looked at the behavior of the method on systems with differing latency and bandwidth characteristics.

We implemented the asynchronous Schwarz method for multi-GPU clusters and showed that the asynchronous variants outperform the synchronous variants by about 5× on up to 100 GPUs. Our breakdown analysis showed that removing the synchronization bottlenecks enabled the acceleration of the asynchronous solver. In addition, we explored different partitioning schemes, overlaps between subdomains, and convergence detection methods. Using a two-stage asynchronous Schwarz method, we showed that we could optimize the local solution costs by tuning the local solver accuracy. We observed average speedups of 1.4× to 4× for realistic test cases from finite element simulations.

8 Summary and Future Work

In this work, we tackled the challenges of synchronization in parallel computing algorithms. Synchronization and information exchange lies at the center of parallel computing, enabling computational methods to be consistent and robust. In the exascale era, with hundreds of thousands of parallel computing units, each with massive amounts of parallelism, it is crucial to prevent large-scale synchronization to maxmize computational efficiency.

We looked at two techniques that enable this synchronization-free computing. Batched methods harness the perfect parallelism on massively parallel devices such as GPUs. They consist of independent computations (such as linear system solutions) that aim to occupy the available compute resources, maximizing their utilization fully. We looked at batched iterative solvers, a novel alternative to the state-of-the-art batched direct solvers. Our high-performance implementation, which involved carefully designing data structures and computational kernels, significantly outperforms existing state-of-the-art implementations. In addition to tailored data structures, we implemented Krylov-based solvers and preconditioners, enabling linear system solutions for various problems. We also provided a detailed analysis of the performance on a diverse data set, showing the benefits of our implementation. To accelerate applications, we integrated the batched solvers into two real-world applications, combustion simulations and plasma physics. We showed that utilizing our batched iterative solvers significantly improves their time to solution, enabling them to accelerate their scientific workflows.

For problems with explicit global data dependencies, we explored data asynchronous methods, which remove explicit synchronizations and compute on possibly stale data. These data-asynchronous methods have been shown to converge asymptotically, but their performance studies have been lacking. In this work, we implemented a fine-grained asynchronous Richardson iteration on a single GPU that outperforms the synchronous version. We also performed a detailed analysis of the performance and behavior of the method, showcasing its benefits. We implemented a coarse-grained asynchronous Schwarz method to tackle synchronization bottlenecks at the multi-GPU scale. We demonstrated that the developed algorithm outperforms the synchronous counterpart and can scale to around 100 GPUs. Using realistic test cases from Finite element simulations, we showcased the advantages of these asynchronous methods.

In the future, we plan to expand the applicability of these techniques to other methods. To this end, we list the following possibilities that can build upon the existing work explored here:

1. Extending the batched methods to the solution of linear systems with non-uniform batch entries.
2. Utilizing batched methods in monolithic methods such as block-Jacobi and block-based relaxation methods.
3. Extending the probabilistic analysis of asynchronous methods for fine-grained analysis, modeling the delays for components within one iteration.
4. Extending our implementation to asynchronous Krylov methods, whose synchronized versions have provably better convergence properties than relaxation methods.
5. Utilizing the synchronization-free batched methods in the local solutions for the coarse-grained asynchronous Schwarz methods.
6. Implementing a multi-level asynchronous Schwarz method for multi-GPU systems, which incorporate global information and are known to have faster convergence.

7. Exploring optimized asynchronous Schwarz methods, which incorporate higher order boundary information between subdomains and have better convergence properties.

We have shown how applications can benefit from utilizing synchronization-free methods, enabling efficient computing by maximizing the available compute usage. We hope that applications will adopt these synchronization-free techniques that can help accelerate their scientific simulations, enabling scientific discovery in a wide range of fields.

Relevant publications of the author

† denotes publications with alphabetical author ordering
* denotes authors who contributed equally to the work

[ACC+20a] † H. Anzt, T. Cojean, Y.-C. Chen, J. Dongarra, G. Flegar, **P. Nayak**, S. Tomov, Y. M. Tsai, and W. Wang. "Load-Balancing Sparse Matrix Vector Product Kernels on GPUs". In: *ACM Transactions on Parallel Computing* (2020). DOI: 10.1145/3380930.

[ACC+20b] † H. Anzt, T. Cojean, Y.-C. Chen, G. Flegar, F. Göbel, T. Grützmacher, **P. Nayak**, T. Ribizel, and Y.-H. Tsai. "Ginkgo: A High Performance Numerical Linear Algebra Library". In: *Journal of Open Source Software* (Aug. 2020). DOI: 10.21105/joss.02260.

[ACF+22] † 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". In: *ACM Transactions on Mathematical Software* 48.1 (Feb. 2022), 2:1–2:33. ISSN: 0098-3500. DOI: 10/gphfsq.

[AKN+21] I. Aggarwal*****, A. Kashi*****, **P. Nayak***, C. J. Balos, C. S. Woodward, and H. Anzt. "Batched Sparse Iterative Solvers for Computational Chemistry Simulations on GPUs". In: *2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. Nov. 2021, pp. 35–43. DOI: 10/gn3xcg.

[ANK+22] I. Aggarwal*****, **P. Nayak***, A. Kashi, and H. Anzt. "Preconditioners for Batched Iterative Linear Solvers on GPUs". In: *Accelerating Science and Engineering Discoveries Through Integrated Research Infrastructure for Experiment, Big Data, Modeling and Simulation*. Ed. by K. Doug, G. Al, S. Pophale, H. Liu, and S. Parete-Koon. Communications in Computer and Information Science. Cham: Springer Nature Switzerland, 2022, pp. 38–53. ISBN: 978-3-031-23606-8. DOI: 10.1007/978-3-031-23606-8_3.

[KNK+22] A. Kashi*****, **P. Nayak***, D. Kulkarni, A. Scheinberg, P. Lin, and H. Anzt. "Batched Sparse Iterative Solvers on GPU for the Collision Operator for Fusion Plasma Simulations". In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2022, pp. 157–167. DOI: 10.1109/IPDPS53621.2022.00024.

[KNK+23] A. Kashi*****, **P. Nayak***, D. Kulkarni, A. Scheinberg, P. Lin, and H. Anzt. "Integrating Batched Sparse Iterative Solvers for the Collision Operator in Fusion Plasma Simulations on GPUs". In: *Journal of Parallel and Distributed Computing* (Apr. 2023). ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2023.03.012.

[NA23] **P. Nayak** and H. Anzt. "Utilizing Batched Solver Ideas for Efficient Solution of Non-Batched Linear Systems". In: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2023, pp. 662–665. DOI: 10.1109/IPDPSW59300.2023.00113.

[NCA20a] **P. Nayak**, T. Cojean, and H. Anzt. "Evaluating Asynchronous Schwarz Solvers on GPUs". In: *The International Journal of High Performance Computing Applications* (Aug. 2020). DOI: 10.1177/1094342020946814.

[NCA20b] **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. DOI: 10/gphfs2.

[NNA23] P. Nguyen, **P. Nayak**, and H. Anzt. "Porting Batched Iterative Solvers onto Intel GPUs with SYCL". In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1048–1058. ISBN: 9798400707858. DOI: 10.1145/3624062.3624181.

[TNC+22] Y.-H. M. Tsai, **P. Nayak**, E. Chow, and H. Anzt. "Implementing Asynchronous Jacobi Iteration on GPUs". In: *2022 IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Heterogeneous Systems (ScalAH)*. Nov. 2022, pp. 1–9. DOI: 10.1109/ScalAH56622.2022.00006.

Bibliography

[AAB+20] A. Abdelfattah et al. “A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic”. In: *arXiv:2007.06674 [cs, math]* (July 2020). arXiv: 2007.06674 [cs, math].

[AAG+21] J. I. Aliaga, H. Anzt, T. Grützmacher, E. S. Quintana-Ortí, and A. E. Tomás. “Compression and Load Balancing for Efficient Sparse Matrix-Vector Product on Multicore Processors and Graphics Processing Units”. In: *Concurrency and Computation: Practice and Experience* (July 2021). ISSN: 1532-0634. DOI: 10/gnsv46.

[AAG+22] J. I. Aliaga, H. Anzt, T. Grützmacher, E. S. Quintana-Ortí, and A. E. Tomás. “Compressed Basis GMRES on High-Performance Graphics Processing Units”. In: *The International Journal of High Performance Computing Applications* (Aug. 2022), p. 10943420221115140. ISSN: 1094-3420. DOI: 10.1177/10943420221115140.

[ABF+22] D. Arndt et al. “The Deal.II Library, Version 9.4”. In: *Journal of Numerical Mathematics* 30.3 (Sept. 2022), pp. 231–246. ISSN: 1569-3953. DOI: 10.1515/jnma-2022-0054.

[ACD+17] H. Anzt, G. Collins, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. “Flexible Batched Sparse Matrix-Vector Product on GPUs”. In: *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ScalA ’17. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 1–8. ISBN: 978-1-4503-5125-6. DOI: 10.1145/3148226.3148230.

[ADF+17a] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. “Batched Gauss-Jordan Elimination for Block-Jacobi Preconditioner Generation on GPUs”. In: *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM’17. New York, NY, USA: Association for Computing Machinery, Feb. 2017, pp. 1–10. ISBN: 978-1-4503-4883-6. DOI: 10/gnn4fz.

[ADF+17b] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. “Variable-Size Batched LU for Small Matrices and Its Integration into Block-Jacobi Preconditioning”. In: *2017 46th International Conference on Parallel Processing (ICPP)*. Aug. 2017, pp. 91–100. DOI: 10.1109/ICPP.2017.18.

[ADG15] H. Avron, A. Druinsky, and A. Gupta. “Revisiting Asynchronous Linear Solvers: Provable Convergence Rate through Randomization”. In: *Journal of the ACM* 62.6 (Dec. 2015), 51:1–51:27. ISSN: 0004-5411. DOI: 10.1145/2814566.

[AH74] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1974. ISBN: 978-0-201-00029-0.

[AHL+11] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shaif, R. Lucas, and K. Yellick. *Exascale Programming Challenges: Report of the 2011 Workshop on Exascale Programming Challenges*. Technical Report. Marina del Rey, July 2011, p. 69.

[AHT+16] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. “Performance, Design, and Autotuning of Batched GEMM for GPUs”. In: *High Performance Computing*. Ed. by J. M. Kunkel, P. Balaji, and J. Dongarra. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 21–38. ISBN: 978-3-319-41321-1. DOI: 10.1007/978-3-319-41321-1_2.

[AMD20] AMD. *Introducing AMD CDNA Architecture*. Tech. rep. Advanced Micro Devices Corporation, 2020.

[AMD23] AMD. *ROCm-Developer-Tools/HIP*. ROCm Developer Tools. July 2023.

[Amd67] G. M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). New York, NY, USA: Association for Computing Machinery, Apr. 1967, pp. 483–485. ISBN: 978-1-4503-7895-6. DOI: 10.1145/1465482.1465560.

[And99] E. Anderson, ed. *LAPACK Users’ Guide*. 3rd ed. Software, Environments, Tools. Philadelphia: Society for Industrial and Applied Mathematics, 1999. ISBN: 978-0-89871-447-0.

[ARF+19] H. Anzt, T. Ribizel, G. Flegar, E. Chow, and J. Dongarra. “ParILUT - A Parallel Threshold ILU for GPUs”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2019, pp. 231–241. DOI: 10.1109/IPDPS.2019.00033.

[ATN+11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), p. 187. DOI: 10.1002/cpe.1631.

[BAT+20] N. Beams, A. Abdelfattah, S. Tomov, J. Dongarra, T. Kolev, and Y. Dudouit. “High-Order Finite Element Method Using Standard and Device-Level Batch GEMM on GPUs”. In: *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. Nov. 2020, pp. 53–60. DOI: 10.1109/ScalA51936.2020.00012.

[Bau78] G. M. Baudet. “Asynchronous Iterative Methods for Multiprocessors”. In: *Journal of the ACM (JACM)* (1978). ISSN: 1557735X. DOI: 10.1145/322063.322067.

[BBD+13] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. “PaRSEC: Exploiting Heterogeneity to Enhance Scalability”. In: *Computing in Science & Engineering* 15.6 (Nov. 2013), pp. 36–45. ISSN: 1558-366X. DOI: 10.1109/MCSE.2013.98.

[BCC+05] J. M. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier. “A Decentralized Convergence Detection Algorithm for Asynchronous Parallel Iterative Algorithms”. In: *IEEE Transactions on Parallel and Distributed Systems* (2005). ISSN: 10459219. DOI: 10.1109/TPDS.2005.2.

[BCR16] E. Blayo, D. Cherel, and A. Rousseau. “Towards Optimized Schwarz Methods for the Navier–Stokes Equations”. In: *Journal of Scientific Computing* 66.1 (2016), pp. 275–295. ISSN: 08857474. DOI: 10.1007/s10915-015-0020-9.

[BDG+17] M. Bonazzoli, V. Dolean, I. Graham, E. Spence, and P.-H. Tournier. “Two-Level Preconditioners for the Helmholtz Equation”. In: (2017), pp. 1–8. arXiv: 1705.08139.

[BGW+21] C. J. Balos, D. J. Gardner, C. S. Woodward, and D. R. Reynolds. “Enabling GPU Accelerated Computing in the SUNDIALS Time Integration Library”. In: *Parallel Computing* 108 (Dec. 2021), p. 102836. ISSN: 0167-8191. DOI: 10/gpghnv.

[BK01] B. Beckermann and A. B. J. Kuijlaars. “Superlinear Convergence of Conjugate Gradients”. In: *SIAM Journal on Numerical Analysis* 39.1 (Jan. 2001), pp. 300–329. ISSN: 0036-1429. DOI: 10.1137/S0036142999363188.

[BMT96] M. Benzi, C. D. Meyer, and M. Tůma. “A Sparse Approximate Inverse Preconditioner for the Conjugate Gradient Method”. In: *SIAM Journal on Scientific Computing* 17.5 (Sept. 1996), pp. 1135–1149. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/S1064827594271421.

[BSB+19] D. A. Beckingsale, T. R. Scogland, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, and B. S. Ryujin. “RAJA: Portable Performance for Large-Scale Scientific Applications”. In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Denver, CO, USA: IEEE, Nov. 2019, pp. 71–81. ISBN: 978-1-72816-003-0. DOI: 10.1109/P3HPC49587.2019.00012.

[BT15] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, Mar. 2015. ISBN: 978-1-886529-15-1.

[BTL+18] W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes. “Batched QR and SVD Algorithms on GPUs with Applications in Hierarchical Matrix Compression”. In: *Parallel Computing*. Parallel Matrix Algorithms and Applications (PMAA’16) 74 (May 2018), pp. 19–33. ISSN: 0167-8191. DOI: 10.1016/j.parco.2017.09.001.

[CDH+08] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. “Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate”. In: *ACM Transactions on Mathematical Software* (2008). ISSN: 00983500. DOI: 10.1145/1391989.1391995.

[CETS14] H. Carter Edwards, C. R. Trott, and D. Sunderland. “Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns”. In: *Journal of Parallel and Distributed Computing* (2014). ISSN: 07437315. DOI: 10.1016/j.jpdc.2014.07.003.

[CFS20] E. Chow, A. Frommer, and D. B. Szyld. “Asynchronous Richardson Iterations: Theory and Practice”. In: *Numerical Algorithms* (Oct. 2020). ISSN: 1017-1398, 1572-9265. DOI: 10/gjqg5w.

[CFW+21] J. Choi, Z. Fink, S. White, N. Bhat, D. F. Richards, and L. V. Kale. “GPU-aware Communication with UCX in Parallel Programming Models: Charm++, MPI, and Python”. In: *arXiv:2102.12416 [cs]* (Mar. 2021). arXiv: 2102.12416 [cs].

[CM69] D. Chazan and W. Miranker. “Chaotic Relaxation”. In: *Linear Algebra and Its Applications* (1969). ISSN: 00243795. DOI: 10.1016/0024-3795(69)90028-7.

[CM94] T. Chan and T. Mathew. “Domain Decomposition Algorithms”. In: *Acta Numerica 3* (1994), pp. 61–143. DOI: http://dx.doi.org/10.1017/S0962492900002427.

[CP15] E. Chow and A. Patel. “Fine-Grained Parallel Incomplete LU Factorization”. In: *SIAM Journal on Scientific Computing* 37.2 (Jan. 2015), pp. C169–C193. ISSN: 1064-8275. DOI: 10.1137/140968896.

[CS98] E. Chow and Y. Saad. “Approximate Inverse Preconditioners via Sparse-Sparse Iterations”. In: *SIAM Journal on Scientific Computing* 19.3 (May 1998), pp. 995–1023. ISSN: 1064-8275. DOI: 10.1137/S1064827594270415.

[CS99] X. C. Cai and M. Sarkis. “Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems”. In: *SIAM Journal of Scientific Computing* (1999). ISSN: 10648275. DOI: 10.1137/S106482759732678X.

[CW18] E. Chow and J. Wolfson-Pou. “Convergence Models and Surprising Results for the Asynchronous Jacobi Method”. In: *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium, IPDPS 2018* 2 (2018), pp. 940–949. DOI: 10.1109/IPDPS.2018.00103.

[Dav13] T. A. Davis. “User Guide for CHOLMOD : A Sparse Cholesky Factorization and Modification Package”. In: *Department of Computer and Information Science, University of Florida* (2013), pp. 1–140. DOI: 10.1.1.220.238.

[DB99] M. S. Day and J. B. Bell. “Numerical Simulation of Laminar Reacting Flows with Complex Chemistry”. In: *Combustion Theory and Modelling* 4.LBNL-44682 (Dec. 1999). DOI: 10.1088/1364-7830/4/4/309.

[DBM+11] J. Dongarra et al. “The International Exascale Software Project Roadmap”. In: *The International Journal of High Performance Computing Applications* 25.1 (Feb. 2011), pp. 3–60. ISSN: 1094-3420. DOI: 10.1177/1094342010391989.

[DDG+16] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. Higham, J. Hogg, P. Valero-Lara, S. D. Relton, S. Tomov, and M. Zounon. *A Proposed API for Batched Basic Linear Algebra Subprograms*. Tech. rep. 2016.25. The University of Manchester, Apr. 2016.

[de 99] E. de Sturler. “Truncation Strategies for Optimal Krylov Subspace Methods”. In: *SIAM Journal on Numerical Analysis* 36.3 (Jan. 1999), pp. 864–889. ISSN: 0036-1429. DOI: 10/fsvxf7.

[DER17] I. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Second. Oxford University Press, Jan. 2017. ISBN: 978-0-19-850838-0.

[DGG10] V. Dolean, M. J. Gander, and L Gerardo-Giorda. “Optimized Schwarz Methods for Maxwell’s Equations”. In: *SIAM Journal on Scientific Computing* 20.6 (2010), pp. 2994–3013. DOI: 10.1137/080728536.

[DH11] T. A. Davis and Y. Hu. “The University of Florida Sparse Matrix Collection”. In: *ACM Transactions on Mathematical Software* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

[Doi91] S. Doi. “On Parallelism and Convergence of Incomplete LU Factorizations”. In: *Applied Numerical Mathematics* 7.5 (June 1991), pp. 417–436. ISSN: 0168-9274. DOI: 10.1016/0168-9274(91)90011-N.

[DW87] M Dryja and O. B. Widlund. *An Additive Variant of the Schwarz Alternating Method for the Case of Many Subregions*. Tech. rep. 339. 1987.

[DW90] M Dryja and O Widlund. “Chapter 16 - Some Domain Decomposition Algorithms for Elliptic Problems”. In: *Iterative Methods for Large Linear Systems*. Ed. by D Kincaid and L Hayes. Academic Press, Jan. 1990, pp. 273–291. ISBN: 978-0-12-407475-0. DOI: 10.1016/B978-0-12-407475-0.50022-X.

[ET82] M. N. El Tarazi. “Some convergence results for asynchronous algorithms”. In: *Numerische Mathematik* 39.3 (Oct. 1982), pp. 325–340. ISSN: 0945-3245. DOI: 10.1007/BF01407866.

[EVM+20] J. Eichstädt, M. Vymazal, D. Moxey, and J. Peiró. “A Comparison of the Shared-Memory Parallel Programming Models OpenMP, OpenACC and Kokkos in the Context of Implicit Solvers for High-Order FEM”. In: *Computer Physics Communications* 255 (Oct. 2020), p. 107245. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2020.107245.

[FAC+21] G. Flegar, H. Anzt, T. Cojean, and E. S. Quintana-Ortí. “Adaptive Precision Block-Jacobi for High Performance Preconditioning in the Ginkgo Linear Algebra Software”. In: *ACM Transactions on Mathematical Software* 47.2 (Apr. 2021), 14:1–14:28. ISSN: 0098-3500. DOI: 10.1145/3441850.

[FLP+12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. “Cache-Oblivious Algorithms”. In: *ACM Transactions on Algorithms* 8.1 (Jan. 2012), pp. 1–22. DOI: 10.1145/2071379.2071383.

[Fro] *Frontier User Guide*. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#id2. Sept. 2023.

[FS00] A. Frommer and D. B. Szyld. “On Asynchronous Iterations”. In: *Journal of Computational and Applied Mathematics* 123.1-2 (2000), pp. 201–216. ISSN: 03770427. DOI: 10.1016/S0377-0427(00)00409-X.

[FS12] D. C.-L. Fong and M. Saunders. “CG Versus MINRES: An Empirical Comparison”. In: *Sultan Qaboos University Journal for Science [SQUJS]* 17.1 (Apr. 2012), pp. 44–62. ISSN: 2414-536X. DOI: 10/ggt54v.

[FS94] A. Frommer and D. B. Szyld. “Asynchronous Two-Stage Iterative Methods”. In: *Numerische Mathematik* (1994), pp. 1–18. ISSN: 09453245. DOI: 10.1007/s002110050085.

[FS97] A. Frommer and H. Schwandt. “A Unified Representation and Theory of Algebraic Additive Schwarz and Multisplitting Methods”. In: *SIAM Journal on Matrix Analysis and Applications* 18.4 (Oct. 1997), pp. 893–912. ISSN: 0895-4798, 1095-7162. DOI: 10.1137/S0895479896301212.

[FSS97] A. Frommer, H. Schwandt, and D. B. Szyld. “Asynchronous Weighted Additive Schwarz Methods”. In: *Electronic Transactions on Numerical Analysis* 5.June (1997), pp. 48–61. ISSN: 10689613.

[GAQ23] T. Grützmacher, H. Anzt, and E. S. Quintana-Ortí. “Using Ginkgo’s Memory Accessor for Improving the Accuracy of Memory-Bound Low Precision BLAS”. In: *Software: Practice and Experience* 53.1 (2023), pp. 81–98. ISSN: 1097-024X. DOI: 10.1002/spe.3041.

[GGR+21] F. Göbel, T. Grützmacher, T. Ribizel, and H. Anzt. “Mixed Precision Incomplete and Factorized Sparse Approximate Inverse Preconditioning on GPUs”. In: *Euro-Par 2021: Parallel Processing*. Ed. by L. Sousa, N. Roma, and P. Tomás. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 550–564. ISBN: 978-3-030-85665-6. DOI: 10.1007/978-3-030-85665-6_34.

[GÓNP19] A. Gloster, L. Ó Náraigh, and K. E. Pang. “cuPentBatch—A Batched Pentadiagonal Solver for NVIDIA GPUs”. In: *Computer Physics Communications* 241 (Aug. 2019), pp. 113–121. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2019.03.016.

[GRB+18] C. Glusa, P. Ramanan, E. G. Boman, E. Chow, and S. Rajamanickam. “Asynchronous One-Level and Two-Level Domain Decomposition Solvers”. In: (2018). arXiv: 1808.08172.

[Gre97] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. Frontiers in Applied Mathematics 17. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997. ISBN: 978-0-89871-396-1.

[Gus88] J. L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Communications of the ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415.

[GVD+23] W. F. Godoy, P. Valero-Lara, T. E. Dettling, C. Trefftz, I. Jorquera, T. Sheehy, R. G. Miller, M. Gonzalez-Tallada, J. S. Vetter, and V. Churavy. *Evaluating Performance and Portability of High-Level Programming Models: Julia, Python/Numba, and Kokkos on Exascale Nodes*. Mar. 2023. DOI: 10.48550/arXiv.2303.06195. arXiv: 2303.06195 [cs].

[GVL13] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Fourth edition. Johns Hopkins Studies in the Mathematical Sciences. Baltimore: The Johns Hopkins University Press, 2013. ISBN: 978-1-4214-0794-4.

[Hac16] W. Hackbusch. *Iterative Solution of Large Sparse Linear Systems.* Second edition. Vol. 95. Leipzig: Springer, 2016. ISBN: 978-3-319-28481-1.

[HD18] J. Hook and N. Dingle. “Performance Analysis of Asynchronous Parallel Jacobi”. In: *Numerical Algorithms* 77.3 (Mar. 2018), pp. 831–866. ISSN: 1572-9265. DOI: 10.1007/s11075-017-0342-9.

[HDT+15] A. Haidar, T. T. Dong, S. Tomov, P. Luszczek, and J. Dongarra. “A Framework for Batched and GPU-Resident Factorization Algorithms Applied to Block Householder Transformations”. In: *High Performance Computing*. Ed. by J. M. Kunkel and T. Ludwig. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 31–47. ISBN: 978-3-319-20119-1. DOI: 10.1007/978-3-319-20119-1_3.

[Hin02] A. C. Hindmarsh. *SUNDIALS: Suite of Nonlinear/Differential/Algebraic Equation Solvers*. Tech. rep. UCRL-JC-149711. Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Aug. 2002. DOI: 10.1145/1089014.1089020.

[HJ85] R. A. Horn and C. R. Johnson. *Matrix Analysis*. Cambridge: Cambridge University Press, 1985. DOI: 10.1017/CBO9780511810817.

[HM21] N. J. Higham and T. Mary. *Mixed Precision Algorithms in Numerical Linear Algebra*. Tech. rep. 2021.20. The University of Manchester, Dec. 2021, p. 67.

[Hor] *HoreKa Supercomputer Documentation*. https://www.nhr.kit.edu/userdocs/horeka/. Sept. 2023.

[HP12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Fifth. Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[HSL10] T. Hoefler, T. Schneider, and A. Lumsdaine. “Characterizing the Influence of System Noise on Large-Scale Applications by Simulation”. In: *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010* November (2010). DOI: 10.1109/SC.2010.12.

[Int23a] Intel. *Intel oneAPI Math Kernel Library*. Intel Corporation. 2023.

[Int23b] Intel. *Introduction to the Xe-HPG Architecture*. Tech. rep. Intel Corporation, 2023.

[JBY+15] N. Jain, A. Bhatele, J.-S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kale. “Charm++ and MPI: Combining the Best of Both Worlds”. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. May 2015, pp. 655–664. DOI: 10.1109/IPDPS.2015.102.

[KB22] A. Küchler and C. Banse. “Representing LLVM-IR in a Code Property Graph”. In: vol. 13640. 2022, pp. 360–380. DOI: 10.1007/978-3-031-22390-7_21. arXiv: 2211.05627 [cs].

[Khr14] Khronos. *SYCL - C++ Single-source Heterogeneous Programming for Acceleration Offload*. Jan. 2014.

[KK98] G. Karypis and V. Kumar. *METIS* A Software Package for Partitioning Unstructured Graphs , Partitioning Meshes , and Computing Fill-Reducing Orderings of Sparse Matrices*. Manual. 1998.

[Koc23] M. Koch. *Exploring the Exascale Frontier with Ginkgo*. ICIAM 2023. Tokyo, Aug. 2023.

[LBP+20] K. Liegeois, R. Boman, E. T. Phipps, T. A. Wiesner, and M. Arnst. “GMRES with Embedded Ensemble Propagation for the Efficient Solution of Parametric Linear Systems in Uncertainty Quantification of Computational Models”. In: *Computer Methods in Applied Mechanics and Engineering* 369 (Sept. 2020), p. 113188. ISSN: 0045-7825. DOI: 10.1016/j.cma.2020.113188.

[LH16] Z. Liu and Y. He. “Restricted Additive Schwarz Preconditioner for Elliptic Equations with Jump Coefficients”. In: *Advances in Applied Mathematics and Mechanics* 8.6 (2016), pp. 1072–1083. ISSN: 20751354. DOI: 10.4208/aamm.2014.m669.

[LLH+16] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter. “A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves”. In: *Euro-Par 2016: Parallel Processing*. Ed. by P.-F. Dutot and D. Trystram. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 617–630. ISBN: 978-3-319-43659-3. DOI: 10.1007/978-3-319-43659-3_45.

[Mag13] F. Magoulès. “Asynchronous Schwarz Methods for Peta and Exascale Computing”. In: 2013. DOI: 10.4203/csets.31.10.

[Mpi] *MPI 3.1: A Message-Passing Interface Standard*. June 2015.

[NAC+15] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka. “AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods”. In: *SIAM Journal on Scientific Computing* 37.5 (Jan. 2015), S602–S626. ISSN: 1064-8275. DOI: 10/f7xxgd.

[NDB18] A. Nonaka, M. S. Day, and J. B. Bell. “A Conservative, Thermodynamically Consistent Numerical Approach for Low Mach Number Combustion. Part I: Single-level Integration”. In: *Combustion Theory and Modelling* 22.1 (Jan. 2018), pp. 156–184. ISSN: 1364-7830. DOI: 10.1080/13647830.2017.1390610.

[NER23] NERSC. *Perlmutter Supercomputer Architecture*. Sept. 2023.

[NVI17] NVIDIA. *NVIDIA Tesla V100 GPU Architecture*. Tech. rep. WP-08608-001_v1.1. NVIDIA Corporation, 2017.

[NVI20] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. Tech. rep. NVIDIA Corporation, 2020.

[NVI22] NVIDIA. *NVIDIA Open-Gpu-Doc Repository*. NVIDIA Corporation. July 2022.

[NVI23a] NVIDIA. *cuBLAS - Basic Linear Algebra on NVIDIA GPUs*. NVIDIA Corporation. Sept. 2023.

[NVI23b] NVIDIA. *CUDA C++ Programming Guide*. Sept. 2023.

[OA23] G. Olenik and H. Anzt. “Improving Heterogenous Efficient for GPU Solvers”. In: *18th OpenFOAM Workshop*. Genoa, Italy, July 2023.

[OR00] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Classics in Applied Mathematics 30. Philadelphia: Society for Industrial and Applied Mathematics, 2000. ISBN: 978-0-89871-461-6.

[OYI+19] S. Ohshima, I. Yamazaki, A. Ida, and R. Yokota. “Optimization of Numerous Small Dense-Matrix–Vector Multiplications in H-Matrix Arithmetic on GPU”. In: *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. Oct. 2019, pp. 9–16. DOI: 10.1109/MCSoC.2019.00009.

[Pela] *PeleC: An Adaptive Mesh Refinement Solver for Compressible Reacting Flows*. AMReX-Combustion. Apr. 2023.

[Pelb] *PeleLMeX: An Adaptive Mesh Hydrodynamics Simulation Code for Low Mach Number Reacting Flows without Level Sub-Cycling*. AMReX-Combustion. Apr. 2023.

[Pelc] *PelePhysics: A Collection of Physics Databases and Implementation Code for Use with the Pele Suite of of Codes*. AMReX-Combustion. Apr. 2023.

[PHP03] N. Park, B. Hong, and V. Prasanna. “Tiling, Block Data Layout, and Memory Hierarchy Performance”. In: *IEEE Transactions on Parallel and Distributed Systems* 14.7 (July 2003), pp. 640–654. ISSN: 1558-2183. DOI: 10.1109/TPDS.2003.1214317.

[PS07] E. Polizzi and A. Sameh. “SPIKE: A Parallel Environment for Solving Banded Linear Systems”. In: *Computers and Fluids* 36.1 (2007), pp. 113–120. ISSN: 00457930. DOI: 10.1016/j.compfluid.2005.07.005.

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Second. Society for Industrial and Applied Mathematics, Jan. 2003. ISBN: 978-0-89871-534-7 978-0-89871-800-3. DOI: 10.1137/1.9780898718003.

[Saa18] Y. Saad. *A Brief Journey into the Past of Iterative Methods for Solving Sparse Linear Systems*. Nov. 2018.

[Sch70] H. A. Schwarz. *II. Ueber Einen Grenzübergang Durch Alternirendes Verfahren*. 1870.

[SGT07] A. St-Cyr, M. J. Gander, and S. J. Thomas. “Optimized Restricted Additive Schwarz Methodss”. In: *Lecture Notes in Computational Science and Engineering* 55.January (2007), pp. 213–220. ISSN: 14397358. DOI: 10.1007/978-3-540-34469-8_22.

[Son89] P. Sonneveld. “CGS, A Fast Lanczos-Type Solver for Nonsymmetric Linear Systems”. In: *SIAM Journal on Scientific and Statistical Computing* 10.1 (Jan. 1989), pp. 36–52. ISSN: 0196-5204. DOI: 10.1137/0910004.

[SS86] Y. Saad and M. Schultz. “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems”. In: (1986). DOI: 10.1137/0907058.

[SST21] C. E. Schaerer, D. B. Szyld, and P. J. Torres. “A Posteriori Superlinear Convergence Bounds for Block Conjugate Gradient”. In: *arXiv:2107.10320 [cs, math]* (July 2021). arXiv: 2107.10320 [cs, math].

[Str02] J. C. Strikwerda. “A Probabilistic Analysis of Asynchronous Iteration”. In: *Linear Algebra and Its Applications* (2002). ISSN: 00243795. DOI: 10.1016/S0024-3795(02)00258-6.

[Str23] G. Strang. *Introduction to Linear Algebra*. Sixth edition. Wellesley, MA: Wellesley-Cambridge Press, 2023. ISBN: 978-1-73314-667-8.

[Str97] J. C. Strikwerda. “A Convergence Theorem for Chaotic Asynchronous Relaxation”. In: *Linear Algebra and its Applications* 253.1 (Mar. 1997), pp. 15–24. ISSN: 0024-3795. DOI: 10.1016/0024-3795(95)00698-2.

[Sum] *Summit User Guide*. https://docs.olcf.ornl.gov/systems/summit_user_guide.html. Sept. 2023.

[Sv08] P. Sonneveld and M. B. van Gijzen. “IDR(s): A Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Systems of Linear Equations”. In: *SIAM Journal on Scientific Computing* 31.2 (Nov. 2008), pp. 1035–1062. ISSN: 1064-8275. DOI: 10.1137/070685804.

[TBA23] Y.-H. M. Tsai, N. Beams, and H. Anzt. “Mixed Precision Algebraic Multigrid on GPUs”. In: *Parallel Processing and Applied Mathematics*. Ed. by R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2023, pp. 113–125. ISBN: 978-3-031-30442-2. DOI: 10.1007/978-3-031-30442-2_9.

[TCA22] Y. M. Tsai, T. Cojean, and H. Anzt. “Porting Sparse Linear Algebra to Intel GPUs”. In: *Euro-Par 2021: Parallel Processing Workshops*. Ed. by R. Chaves, D. B. Heras, A. Ilic, D. Unat, R. M. Badia, A. Bracciali, P. Diehl, A. Dubey, O. Sangyoon, S. L. Scott, and L. Ricci. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 57–68. ISBN: 978-3-031-06156-1. DOI: 10.1007/978-3-031-06156-1_5.

[TCR+21] 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*. Ed. by B. Balis, D. B. Heras, L. Antonelli, A. Bracciali, T. Gruber, J. Hyun-Wook, M. Kuhn, S. L. Scott, D. Unat, and R. Wyrzykowski. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 109–121. ISBN: 978-3-030-71593-9. DOI: 10.1007/978-3-030-71593-9_9.

[Top] *TOP500 Supercomputer List*. https://www.top500.org/.

[van69] A. van der Sluis. “Condition Numbers and Equilibration of Matrices”. In: *Numerische Mathematik* 14.1 (Dec. 1969), pp. 14–23. ISSN: 0945-3245. DOI: 10.1007/BF02165096.

[van92] H. A. van der Vorst. “Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems”. In: *SIAM Journal on Scientific and Statistical Computing* 13.2 (Mar. 1992), pp. 631–644. ISSN: 0196-5204. DOI: 10.1137/0913035.

[VMS+18] P. Valero-Lara, I. Martínez-Pérez, R. Sirvent, X. Martorell, and A. J. Peña. “cuThomasBatch and cuThomasVBatch, CUDA Routines to Compute Batch of Tridiagonal Systems on NVIDIA GPUs”. In: *Concurrency and Computation: Practice and Experience* 30.24 (2018), e4909. ISSN: 1532-0634. DOI: 10.1002/cpe.4909.

[VS94] J. S. Vitter and E. A. M. Shriver. “Algorithms for Parallel Memory, II: Hierarchical Multilevel Memories”. In: *Algorithmica* 12.2 (Sept. 1994), pp. 148–169. ISSN: 1432-0541. DOI: 10.1007/BF01185208.

[WC19] J. Wolfson-Pou and E. Chow. “Modeling the Asynchronous Jacobi Method without Communication Delays”. In: *Journal of Parallel and Distributed Computing* 128 (2019), pp. 84–98. ISSN: 07437315. DOI: 10.1016/j.jpdc.2019.02.002.

[YCB+19] I. Yamazaki, E. Chow, A. Bouteiller, and J. Dongarra. “Performance of Asynchronous Optimized Schwarz with One-Sided Communication”. In: *Parallel Computing* 86 (2019), pp. 66–81. ISSN: 01678191. DOI: 10.1016/j.parco.2019.05.004.

[ZSL+21] F. Zhang, J. Su, W. Liu, B. He, R. Wu, X. Du, and R. Wang. “YuenyeungSpTRSV: A Thread-Level and Warp-Level Fusion Synchronization-Free Sparse Triangular Solve”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.9 (Sept. 2021), pp. 2321–2337. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3066635.

Keywords