Asynchronous fine-grain parallel iterative solvers for computational fluid dynamics

Aditya Kashi
Department of Mechanical Engineering
McGill University, Montreal
August, 2020

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

Abstract

Computational fluid dynamics (CFD) is an important tool for analyzing fluid flow around objects of interest such as aircraft and automobiles as well as for optimizing their properties to improve performance. However, in many cases, industrially relevant problems are very large and computing their solutions, even on high-performance computers, takes time. New massively parallel hardware, such as general-purpose graphics processing units, promise greater levels of performance for CFD codes. In order to take advantage of such hardware, fine-grain parallel algorithms are needed. This thesis presents novel fine-grain parallel iterations suitable for the solution of large sparse systems of equations arising from discretization of the compressible Navier-Stokes equations.

The concept of chaotic relaxation, or asynchronous iteration, is applied to develop fine-grain parallel iterations suitable for compressible fluid dynamics. Theoretical proofs of some properties of these asynchronous iterations are presented. Suitable grid orderings for improving their effectiveness for CFD are proposed. Ideas from sparse approximate inverse preconditioners are also incorporated to provide a reliable solver that scales in parallel. Numerical experiments have been performed to demonstrate the smoothing property and parallel scalability of the proposed iterations when used as smoothers in multigrid solvers, as well as to show that they can be adapted to build preconditioners for Krylov subspace solvers. Several cases of external aerodynamics, differing in computational grid size and flow complexity, are used for the studies. For this class of problems, it is found that the proposed solvers are not only effective in terms of number of iterations, they can also be implemented to perform well on modern parallel hardware including many-core processors and graphics processing units.

Keywords: parallel preconditioner, multigrid smoothers, asynchronous iterations, point-block preconditioner, node-level parallelism, many-core processor, graphics processing unit, incomplete LU factorization, parallel triangular solver, compressible flow

Résumé

La mécanique des fluides numérique (MFN) est un outil important pour analyser l'écoulement de fluide autour d'objets d'intérêt tels que les avions et les automobiles, ainsi que pour optimiser leurs propriétés afin d'améliorer les performances. Cependant, dans de nombreux cas, les problèmes industriels sont de très grande taille. Le calcul de leurs solutions, même sur des ordinateurs à hautes performances, prend du temps. Les nouvelles architectures informatiques massivement parallèles, tels que les unités de traitement graphique à usage général, promettent de meilleurs niveaux de performances pour les codes de MFN. Afin d'exploiter ces architectures, des algorithmes parallèles à granularités fines sont nécessaires. Cette thèse présente de nouvelles itérations parallèles à granularités fines adaptées à la solution des systèmes d'équations creux à grande taille résultant de la discrétisation des équations de Navier-Stokes compressibles.

Le concept de relaxation chaotique, ou itération asynchrone, est appliqué par la présente pour développer des itérations parallèles à granularités fines adaptées à la dynamique des fluides compressibles. Des preuves théoriques de certaines propriétés de ces itérations asynchrones sont présentées. Des énumérations des éléments du maillage de calcul, appropriées pour améliorer leur efficacité pour MFN, sont proposées. Des idées de préconditionneurs d'inverse approximatif creux sont également incorporées pour fournir un solveur fiable et évolutif en parallèle. Des expériences numériques ont été réalisées pour démontrer la propriété de lissage d'erreur et l'évolutivité parallèle des itérations proposées lorsqu'elles sont utilisées comme lisseurs dans les solveurs multigrilles, ainsi que pour montrer qu'elles peuvent être adaptées pour construire des préconditionneurs pour les solveurs de sous-espace Krylov. Plusieurs cas d'aérodynamique externe, différant par la taille du maillage et la complexité de l'écoulement, sont utilisés pour les études. Pour cette classe de problèmes, on constate que les solveurs proposés sont non seulement efficaces en termes de nombre d'itérations, mais ils peuvent également être mis en œuvre pour obtenir de bonnes performances sur des architectures informatiques parallèles modernes, y compris des processeurs à plusieurs cœurs et des unités de traitement graphique.

Mots clés: préconditionneur parallèle, lisseur pour solveurs multigrilles, itération asynchrone, préconditionneur aux point-blocs, parallélisme au niveau des nœuds, processeur à plusieurs cœurs, unité de traitement graphique, factorisation inférieure-supérieure incomplète, solveur des matrices triangulaires, écoulement compressible

Acknowledgements

Firstly, I express my heartfelt thanks to my advisor and mentor Siva Nadarajah for guiding me in my research, while allowing a lot of freedom. I am very grateful to him for always being available and extending his help and support in many different ways.

I am grateful to my committee members Dr. Xiao-Wen Chang and Dr. Jovan Nedic for providing helpful feedback.

I want to thank my co-workers and friends Syam Vangara and Doug Shi-Dong for the long helpful discussions on various topics, and their help and support in general. Sections 2.5 and 3.4.1 were completed with the help of Syam's inputs and work.

I especially want to thank the late Philip Zwanenburg for being a great colleague and friend, and inspiring me to do better. I am lucky to have had the opportunity to discuss math, software development and even philosophy with him, along with a myriad other topics.

I thank Patrice Castonguay formerly of Bombardier Aviation for helpful discussions and suggestions on the work in chapter 3. I am also grateful to Hong Yang of Bombardier Aviation for some discussions and help with some of the contour plots used here.

Thanks are due to Bart Oldeman of Calcul-Québec for his help with various aspects of the compute clusters that were used for this work.

I am extremely thankful to my parents for being so understanding and supportive of my doctoral work. None of this would have been possible without their support. I apologize to my sister for having become "boring" over the course of my PhD, and thank her for being supportive of my work as well.

Finally, I gratefully acknowledge funding from the Natural Sciences and Engineering Research Council (NSERC) of Canada, the Consortium for Aerospace Research and Innovation in Canada (CARIC), the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ), Bombardier Aerospace, Cray Inc. and the McGill University Faculty of Engineering.

Contents

List of Figures

List of Tables

List of Symbols

v_j	The jth entry of vector $\boldsymbol{v}$
$\boldsymbol{v}_j$	The jth sub-vector (of some specified size) of vector $\boldsymbol{v}$
A_{ij}	The (i, j)th (scalar) entry of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{ij}$	The block at the (i, j)th block-index of matrix $\boldsymbol{A}$ according to some specified blocking
$\boldsymbol{A}$	Arbitrary nonsingular matrix
$\boldsymbol{D}$	(Block-) Diagonal part of a matrix
$\boldsymbol{E}$	Strictly (block) lower triangular matrix
$\boldsymbol{F}$	Strictly (block) upper triangular matrix
$\boldsymbol{I}_d$	$d \times d$ identity matrix
$\boldsymbol{I}_H^h$	Prolongation (interpolation) matrix in a multigrid solver
$\boldsymbol{I}_h^H, \tilde{\boldsymbol{I}}_h^H$	Restriction matrices in a multigrid solver
$\boldsymbol{I}_m$	$m \times m$ identity matrix
$\boldsymbol{L}$	Unit lower (block-) triangular part of a matrix
$\boldsymbol{M}$	Preconditioning matrix
$\boldsymbol{r}$	vector of residuals of fluxes at cell-centres
$\boldsymbol{U}$	Upper (block-) triangular part of a matrix
$\boldsymbol{u}$	vector of conserved variables
$\boldsymbol{w}$	vector of discretized (cell-centred) conserved variables

$\boldsymbol{x}$	Arbitrary vector
μ	Molecular viscosity
μ_t	Turbulent eddy viscosity
$\rho(\boldsymbol{M})$	The spectral radius of matrix $\boldsymbol{M}$
τ	Pseudo-time
$\lvert\boldsymbol{M}\rvert$	The matrix of the same size as $\boldsymbol{M}$ and having as its entries the absolute values of the corresponding entries of $\boldsymbol{M}$
S	Sparsity pattern made up of indices (i, j) of non-zero entries
S_B	Block sparsity pattern made up of indices (i, j) of non-zero blocks

Chapter 1

Introduction

Computation has become the third pillar of analysis, complementing theory and experiments. Computational fluid dynamics (CFD) has come into its own as an integral part of the design process in mechanical and aerospace engineering. This involves simulation of turbulent flows that have a chaotic nature with features spanning a wide range of length and time scales. Since a direct simulation of all length and time scales poses numerous challenges, the state of practice currently consists of "Reynolds-averaged" simulation of turbulent flows. In this process, averaged quantities describing the flow are computed, and the effect of turbulent flow features are modelled. The Reynolds-averaged Navier-Stokes (RANS) equations are a system of partial differential equations (PDEs) that need to be discretized in space, typically, to obtain a system of nonlinear algebraic equations. In many of the scalable ways of solving such a system, the most computationally intensive task is the solution of large sparse linear systems of algebraic equations. Speeding up the linear system solver is an ever-present theme in CFD research.

Many industries are tasked with developing new product lines with improved performance and lower environmental impact in a highly competitive market. This requires simulations with ever greater geometric detail and physical fidelity, solved for a wider range of operating conditions. This necessitates ever larger computational grids with an increasing number of variables to compute. In this scenario, the idea of scalable solvers becomes essential. Noting that the size of the problem refers to the number of variables N whose values are to be found, the holy grail is a solver with $\mathcal{O}(N)$ cost - a solver for which the cost asymptotically scales only linearly with problem size. Ideally, if one doubles the problem size, it should take no more than twice as many resources to compute the solution.

Industrially-relevant simulations may have as many as billions of variables to be computed. Running these simulations on a single node with one or two central processing units (CPUs) is not feasible because of memory limitations and time considerations. Therefore such simulations are carried out on a networked collection of CPUs, referred to as a ‘cluster’, which may have hundreds or even thousands of CPUs communicating with each other over a network. The domain of simulation is divided up among the CPUs and they compute on their assigned subdomains concurrently. Communication between processors is handled by passing messages over the network. This level of parallel computing is referred to as exploiting ‘coarse-grained’ parallelism, and is the state of practice in established CFD codes. The computation on each subdomain is mostly serial.

Today’s new high-performance computers have multiple levels of parallelism, from distributed memory parallelism of clusters to vector operations in individual processor cores. To further speed up computational fluid dynamics (CFD) codes and enable solutions with greater detail, these levels of parallelism need to be fully harnessed. In CFD, while parallelism at the level of a cluster is fairly mature, parallelism within each node of a cluster is a work-in-progress, especially for sparse linear solvers. With the development and commercialization of fine-grain parallel processors such as general-purpose graphics processing units (GPGPUs, or simply, GPUs) [1] and many-core central processing units (CPUs) [2], node-level parallelism has become increasingly important. It has been found that many of the established techniques are incapable of fully exploiting the increased level of available parallelism [3]. In future, we may have even more massively parallel devices than GPUs. One intriguing example is the wafer-scale processor with *hundreds of thousands* of cores [4].

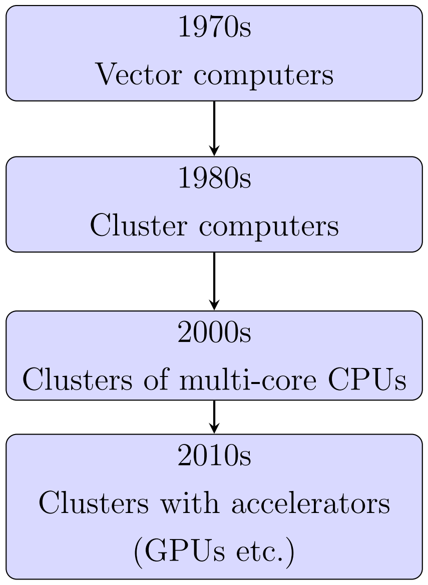

1970s
Vector computers
1980s
Cluster computers
2000s
Clusters of multi-core CPUs
2010s
Clusters with accelerators (GPUs etc.)

Figure 1.1: Evolution of hardware used for parallel CFD

1.1 Linear systems of equations in CFD

We consider the industrial workhorse for turbulent flows - the Reynolds-averaged Navier-Stokes equations. In this work, we only consider the compressible RANS equations, also referred to as Favre-averaged equations. The various spatial discretization schemes used in computational fluid dynamics include finite difference, finite volume, continuous finite elements, discontinuous finite elements, and spectral methods. There are also other methods which are very different from those mentioned above, such as smooth particle hydrodynamics and lattice Boltzmann methods, but we shall not consider them here. In many industries, such as aerospace engineering, the most common technique is the finite volume method.

Two broad classes of discretization in time are explicit methods and implicit methods. Explicit time-stepping in the context of RANS equations comes with restrictive time step limitations imposed by linear stability considerations of the scheme. For steady-state problems, implicit time-stepping is more suitable. While perhaps requiring more memory for the storage of the Jacobian matrix, they allow usage of much larger time steps than for explicit schemes and lead to convergence in less iterations. This has been shown by several researchers, such as Swanson et al. [5] in the context of a multigrid solver and Luo et al. [6] in the context of a Newton-Krylov type solver. Even for many types of unsteady problems, the time step limit of explicit schemes is much smaller than what is required for accuracy [7].

Implicit methods require the solution of one or more systems of linear equations at every time step, and thus it is of interest to ensure that the linear solver is highly optimized and can run as fast as possible. In the broader field of applied mathematics and industrial computing, solving systems of linear equations is one of the most widely used operations. In fluid dynamics, matrices are mostly non-symmetric (owing to convective terms in the PDEs) and may be indefinite. Sometimes, as in the case of the pressure equation in pressure projection methods for incompressible flow [8, 9], symmetric positive-definite matrices may also arise. In most 'density-based' formulations for compressible flow, including the one used in this work, the matrices are exclusively non-symmetric.

Methods of solving systems of linear equations are broadly divided into direct methods and iterative methods [10]. Direct methods are commonly regarded as too expensive for very large sparse linear systems [11], for which iterative methods are better suited [12]. Iterative methods may be stationary iterative methods, incomplete factorization, multigrid schemes, domain decomposition methods and Krylov subspace methods, among others. Stationary iterative methods include classical solvers such as Jacobi and Gauss-Seidel, and their many variations. The incomplete factorization methods include the different incomplete LU (ILU) and incomplete Cholesky (IC) methods. Another class of solvers is collectively referred to as domain decomposition. It is a class of techniques for solving a problem defined on some domain by subdividing it into smaller problems defined on subdomains [13]. Among other uses, domain decomposition can be used to develop parallel solvers.

A class of iterative solvers used in CFD are the Krylov subspace methods. These methods search for an optimal solution in a certain affine space. The definition of optimality depends on the specific method chosen. The generalized minimal residual (GMRES) method, the stabilized biconjugate gradient method, and the standard conjugate gradient method (when symmetric linear systems need to be solved) are popular Krylov subspace methods in CFD. 'Preconditioning' is crucial to the success of Krylov subspace methods. It refers to finding an equivalent problem which is easier to solve - one that is ‘better conditioned’. Preconditioned Krylov methods are usually much faster than stationary iterative methods. However, stationary iterative methods such as symmetric Gauss-Seidel (SGS) and ILU factorization are popular as preconditioners for Krylov subspace solvers. Most operations in Krylov solvers are relatively easy to parallelize. The most notable exception today is the preconditioning operation, as it tends to depend on traditional serial methods. Thus, fine-grain parallel preconditioning is one of the thrusts of this work. A class of techniques that has arisen to address this issue is that of sparse approximate inverses [11], though it has its issues as well (subsection 2.7.2). It is also noteworthy that some inherently parallel operations in Krylov methods, such as the sparse matrix vector product, have other issues when it comes to massively parallel hardware such as difficult vectorization or thread divergence arising mainly from unfavourable memory access patterns. Further, for very large problem sizes on massively parallel systems, there are latency issues associated with the numerous inner products that are required to be computed. There have been efforts to address these issues (see section 2.6).

An important class of solvers is the class of multigrid methods. They aim to solve the system of algebraic equations that arise from the discretization of a PDE optimally (with $\mathcal{O}(N)$ cost) by utilizing a hierarchy of grids. Multigrid solvers depend on single-grid iterations referred to as ‘multigrid smoothers’, or simply smoothers. Currently popular choices are weighted Jacobi and Gauss-Seidel [14, 15] smoothers. Also popular are Runge-Kutta type smoothers [16], usually for nonlinear multigrid schemes, which are popular in industrial CFD [17]. While the parallelization of multigrid schemes is in itself an open question [18], this thesis is concerned with the effectiveness and parallelization of the smoother.

1.2 Parallel hardware platforms

Figure 1.2 shows a legacy Sandy Bridge processor with 8 cores and 4 parallel double-precision lanes per core for a total of 32 processing elements. In 2016, Intel released the Xeon Phi Knights Landing (figure 1.3), the top-bin part of which has 576 processing elements. This is already more than an order-of-magnitude increase in the amount of parallelism available per chip. Also released in 2016 was NVIDIA’s P100 data centre GPU (figure 1.4), which has 1792 double-precision processing elements. At the time of writing, the latest NVIDIA data centre GPU code-named Ampere contains more than 3456 double-precision processing elements. As mentioned earlier, cutting edge experimental devices can now even have hundreds of thousands of processing elements. In the coming years, it will be essential for computational fluid dynamics software, and scientific computing software in general, to utilize the newer

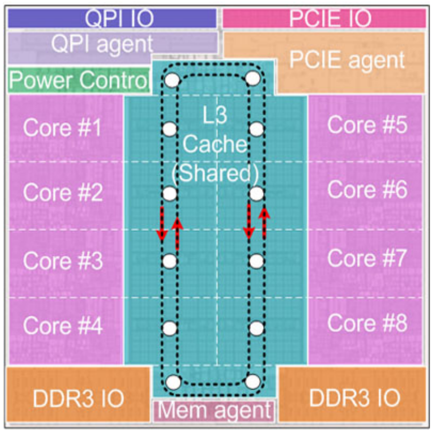


Figure 1.2: 8-core Intel Sandy-Bridge, 32 processing elements

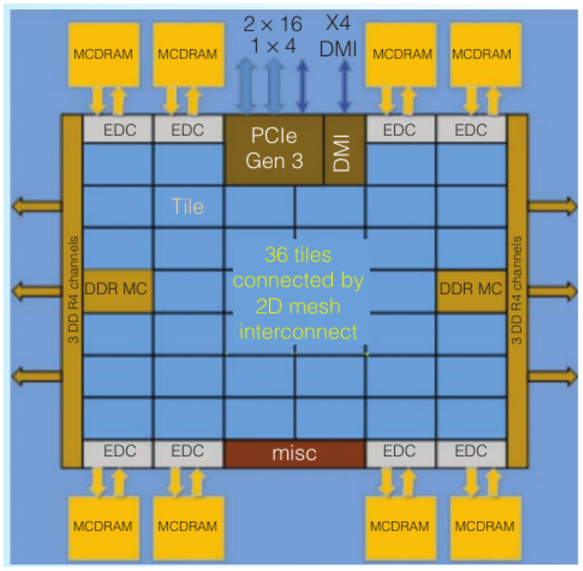


Figure 1.3: 72-core Intel Xeon-Phi “Knights Landing”, 576 processing elements [2]

massively parallel hardware platforms well. There are at least two factors pointing in this direction - speed of execution and energy efficiency.

GPUs have been shown to provide large jumps in speed of execution. Karantasis et al.[20] report a speedup of 24x over multi-threaded CPU execution for an explicit large eddy simulation (LES) code. Several researchers have found GPUs to be more energy-efficient than CPUs for certain high-performance computing workloads [21]. For a specific use case involving a biology-related simulation code, GPUs have been observed to be 9 times more energy efficient than some multi-core CPUs and a soaring 237 times better than the multi-core CPUs when energy and time to completion are simultaneously considered [22].

However, there are some challenges involved in programming these highly parallel devices. The expression of parallelism has to take into account the various levels of parallelism

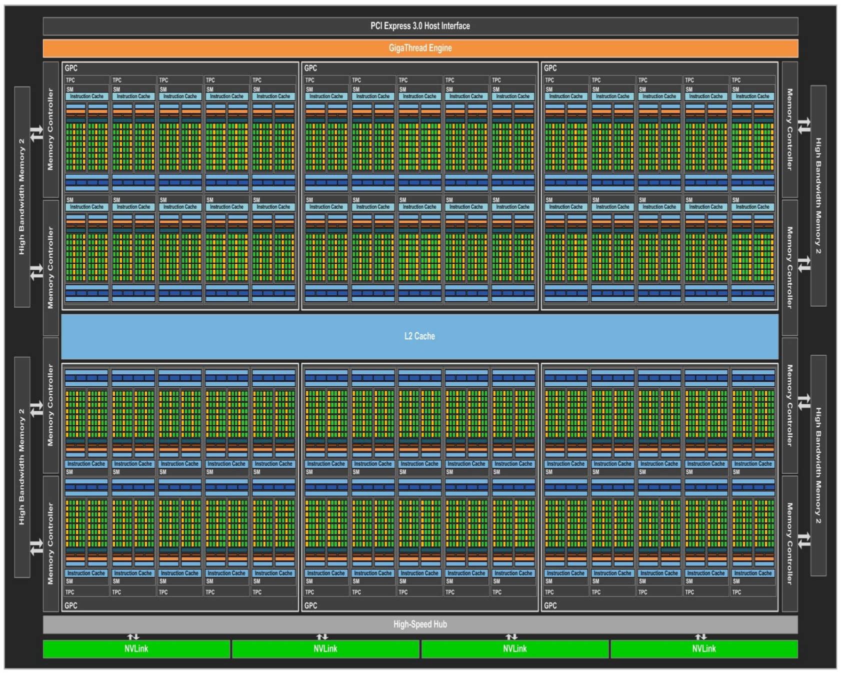


Figure 1.4: NVIDIA Tesla “Pascal”, 1792 processing elements [19]

available. These may include device level, the ‘warp’, ‘wavefront’, or ‘vector’ level and finally thread or scalar level. Vectorization, that is the use of Single Instruction Multiple Data (SIMD) type instructions, is crucial to getting benefits. Code has to be written such that vectorization opportunities are detected by the compiler, or explicit ‘pragmas’ (commands giving extra information to the compiler) or intrinsics (device-specific low-level functions) might have to be used and which may not be portable. Corresponding to the levels of parallelism, there are levels of memory as well. Since memory available for a team of threads is limited, care has to be taken not to exceed local memory limits, especially for GPUs. Different compute architectures may have different memory access requirements for optimal performance (section 2.9.4). Most parallel accelerators also have their own high-speed device memory, so unless the whole program runs on the accelerator, care has to be taken about slow transfers of data between the device and the host CPU. Finally, in some cases, the latency of launching a parallel execution on a device may also be significant. In such cases the number of these launches must be minimized. The design of fine-grain parallel iterations has to take all of these under consideration.

1.3 Research objectives

The objectives of this thesis are twofold.

1. Develop solvers that, while achieving reliable solution of compressible turbulent flow cases, take full advantage of the massive parallelism available in modern computing platforms. We are interested in the development of iterations that have good convergence rates for our problems, and also implementations informed by hardware characteristics.

2. Develop an understanding of the convergence properties of these solvers from both experimental and theoretical points of view. This is important for establishing trust in these methods and helps to determine their range of applicability. In this thesis, the properties of interest include multigrid smoothing property, convergence bounds of parallel sub-iterations, and the parallel scalability.

1.4 Thesis contributions

This thesis presents novel asynchronous iterations and their application to solving industrially-relevant problems in compressible turbulent fluid flow. Individual novel contributions are listed below.

1. Demonstration of smoothing property of asynchronous block symmetric Gauss-Seidel (SGS) for compressible flows on multi-core CPUs. A conference presentation was given at the American Institute of Aeronautics and Astronautics (AIAA) conference AVIATION 2018 [23] and a paper published in Computers and Fluids [24] on this subject.

2. Extension of convergence theory of asynchronous iterations to the point-block case suitable for compressible flows. A conference presentation was given on this topic at the Society of Industrial and Applied Mathematics (SIAM) Copper Mountain Conference on Iterative Methods 2018, and a paper is under review with the SIAM Journal on Scientific Computing [25].

3. Investigation of grid-ordering schemes for asynchronous point-block ILU preconditioning on unstructured grids on many-core CPUs. This work was presented at the SIAM Conference on Computational Science and Engineering 2019 and is included in the article submitted to SIAM [25].

4. Investigation of properties of the asynchronous block ILU preconditioner applied to the compressible Euler and Navier-Stokes equations under different conditions on unstructured grids. These properties include convergence of the asynchronous iteration,

the linear solver and the nonlinear solver, diagonal dominance of triangular factors and strong scaling on a many-core CPU. This is also part of the SIAM submission [25].

5. Development of specialized asynchronous smoothers for structured grids on many-core CPUs with wide vector units and demonstration of their competence in comparison to currently available smoothers. A conference paper at the AIAA Scitech Forum 2020 is based on this work [26].

6. Demonstration of an incomplete sparse approximate inverse method to solve triangular linear systems in compressible turbulent flow problems; included in the conference paper for AIAA Scitech Forum 2020 [26]

7. Extension of the specialized asynchronous smoothers and incomplete sparse approximate inverse triangular solver mentioned above to graphics processing units (GPUs) for turbulent compressible flow problems. This was presented at the SIAM Conference on Parallel Processing for Scientific Computing, 2020.

1.5 Thesis overview

In the next chapter (2), we provide an overview of some concepts important to the development of solvers as well as currently popular solvers. A literature review of the current state of parallel solvers is also presented, along with hardware and implementation considerations on modern architectures.

Chapter 3 details the development of an asynchronous block symmetric Gauss-Seidel relaxation for Reynolds-averaged Navier-Stokes (RANS) flow cases on structured grids, running on multi-core and many-core CPU architectures. Numerical results with multigrid solvers demonstrate good smoothing properties independent of core count on a number of cases of external aerodynamics.

Next, chapter 4 develops point-block extensions of asynchronous ILU factorization and asynchronous triangular solves for general unstructured grids. Convergence proofs of the block-asynchronous iterations are given. A grid ordering for the effective use of asynchronous block ILU preconditioning for viscous flows is developed. Numerical results on inviscid and viscous compressible flows on mixed unstructured grids demonstrate the good performance of the proposed methods as preconditioners in Krylov subspace methods.

Chapters 5 and 6 pivot to efficient multigrid smoothers specialized to structured grids on platforms with SIMD-type vector parallel processing, including many-core processors with wide vector units (chapter 5) and graphics processing units (chapter 6). The focus is again on compressible RANS cases, but using more efficiently vectorizable algorithms. This involves moving to different data storage layouts and faster algorithms specialized to structured grids, as demonstrated with an asynchronous block ILU algorithm. Further, incomplete sparse approximate inverse iterations are found to be effective in applying triangular solves in this vectorization-friendly implementation. A comparison with parallel smoothing by simple domain decomposition is also made on many-core CPUs, while a red-black Gauss-Seidel smoother is used for comparison on GPUs. Results indicate the great potential of the proposed iterations.

Finally, chapter 7 concludes the thesis with a summary of the potential of asynchronous iterations for CFD problems, as well as points out limitations and avenues of future work.

Chapter 2

Preliminaries

We first provide an overview of the governing equations and the discretization schemes in the CFD codes used in this work. One of the codes used is a version of the "Full Aircraft Navier-Stokes Code" [17, 27], a 3D multi-block structured grid compressible Reynolds-averaged Navier-Stokes (RANS) code developed at Bombardier Aviation, called FANSC-Lite. The other is FVENS [1], an in-house 2D unstructured grid compressible viscous flow code.

The governing equations of compressible fluid flow, the compressible Navier-Stokes equations, are given in integral form as follows.

$$\frac{d}{dt}\int_{\Omega} \rho d\omega + \int_{\Gamma} \rho \boldsymbol{v} \cdot \hat{\boldsymbol{n}} d\gamma = 0 \tag{2.1}$$

$$\frac{d}{dt}\int_{\Omega} \rho \boldsymbol{v} d\omega + \int_{\Gamma} \rho \boldsymbol{v}(\boldsymbol{v} \cdot \hat{\boldsymbol{n}}) d\gamma = -\int_{\Gamma} p \hat{\boldsymbol{n}} d\gamma + \int_{\Gamma} \underline{\boldsymbol{\sigma}} \hat{\boldsymbol{n}} d\gamma \tag{2.2}$$

$$\frac{d}{dt}\int_{\Omega} \rho e d\omega + \int_{\Gamma} \rho e \boldsymbol{v} \cdot \hat{\boldsymbol{n}} d\gamma = -\int_{\Gamma} p \hat{\boldsymbol{n}} \cdot \boldsymbol{v} d\gamma + \int_{\Gamma} \underline{\boldsymbol{\sigma}} \hat{\boldsymbol{n}} \cdot \boldsymbol{v} d\gamma - \int_{\Gamma} \boldsymbol{q} \cdot \hat{\boldsymbol{n}} d\gamma \tag{2.3}$$

where the spatial domain $\Omega \subset \mathbb{R}^d$, $d (= 1, 2, 3)$ is the number of spatial dimensions of relevance to the problem, $\rho(\mathbf{x}, t), \rho\boldsymbol{v}(\mathbf{x}, t)$ and $\rho e(\mathbf{x}, t)$ are the fluid's instantaneous density, momentum and energy, $\underline{\boldsymbol{\sigma}}$ is the viscous stress tensor and $\boldsymbol{q}$ is the heat flux vector. In the context of Reynolds-averaged (or, more precisely, Favre-averaged) equations, the flow variables $\rho, \rho\boldsymbol{v}, \rho e$ are regarded as ensemble averages, and thus referred to as the 'mean flow variables'.

These equations can be converted to the conservative differential form by using the divergence theorem.

$$\frac{\partial \boldsymbol{u}}{\partial t} + \nabla \cdot \boldsymbol{F}^i(\boldsymbol{u}) + \nabla \cdot \boldsymbol{F}^v(\boldsymbol{u}, \nabla \boldsymbol{u}, \tilde{\nu}) = 0, \tag{2.4}$$

[1]FVENS: https://github.com/Slaedr/FVENS

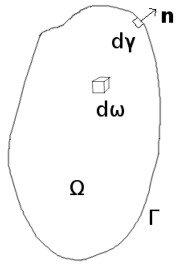
where $\boldsymbol{u} \in [L^2(\Omega)]^{d+2}$ is the vector of conserved variables, $\boldsymbol{F}^i \in [\mathcal{C}^1(\mathbb{R})]^{(d+2)\times d}$ is the matrix of inviscid fluxes and $\boldsymbol{F}^v \in [\mathcal{C}^1(\mathbb{R})]^{(d+2)\times d}$ is the matrix of viscous fluxes. These are

$$\boldsymbol{u} = \begin{bmatrix} \rho \\ \rho \boldsymbol{v} \\ \rho e \end{bmatrix}, \quad \boldsymbol{F}^i(\boldsymbol{u}) = \begin{bmatrix} \rho \boldsymbol{v}^T \\ \rho \boldsymbol{v}\boldsymbol{v}^T + p\boldsymbol{I}_d \\ (\rho e + p)\boldsymbol{v}^T \end{bmatrix}, \text{and } \boldsymbol{F}^v(\boldsymbol{u}, \nabla \boldsymbol{u}, \tilde{\nu}) = -\begin{bmatrix} 0 \\ \underline{\boldsymbol{\sigma}} \\ \boldsymbol{v}^T \underline{\boldsymbol{\sigma}} - \boldsymbol{q}^T \end{bmatrix}. \tag{2.5}$$

Note that if $\boldsymbol{n}$ is some vector, then the matrix-vector products $\boldsymbol{F}^i(\boldsymbol{u})\boldsymbol{n}$ and $\boldsymbol{F}^v(\boldsymbol{u}, \nabla \boldsymbol{u})\boldsymbol{n}$ would give us the fluxes in that direction. Finally, $\tilde{\nu} : \Omega \to \mathbb{R}$ is a 'turbulence working variable' that is responsible for modelling the effects of turbulence, to be defined later in this section.

The equations are closed by constitutive relations. The stress - strain rate relation for a Newtonian fluid is given by

$$\underline{\boldsymbol{\sigma}} = -\frac{2}{3}(\mu + \mu_t)\nabla \cdot \boldsymbol{v}\boldsymbol{I}_d + (\mu + \mu_t)(\nabla \boldsymbol{v} + \nabla \boldsymbol{v}^T), \tag{2.6}$$



where $\mu(\boldsymbol{u})$ and $\mu_t(\tilde{\nu}, \boldsymbol{u})$ are the molecular and eddy viscosity coefficients. The ideal gas equation of state $p = \rho RT$ is used, where R is the characteristic gas constant and T is the temperature. Fourier's law of heat conduction $\boldsymbol{q} = -k\nabla T$ and Sutherland's law for relating molecular viscosity to temperature,

$$\mu = \frac{1.45T^{3/2}}{T + 110}10^{-6}, \tag{2.7}$$

are also employed.

The eddy viscosity μ_t, applicable in case of the RANS equations, is computed using one of the eddy viscosity models. The Spalart-Allmaras (SA) model is widely used in aerodynamics and is the one adopted for this work. This one-equation PDE model is given in integral form as

$$\frac{d}{dt}\int_\Omega \tilde{\nu}\, d\omega + \int_\Gamma (\boldsymbol{F}_T^i - \boldsymbol{F}_T^v)\cdot \hat{\boldsymbol{n}}\, d\gamma = \int_\Omega Q_T\, d\omega \tag{2.8}$$

where the convective flux $\boldsymbol{F}_T^i(\tilde{\nu}, \boldsymbol{u}) = \boldsymbol{v}\tilde{\nu}$, the viscous flux

$$\boldsymbol{F}_T^v(\tilde{\nu}, \boldsymbol{u}) = \frac{1}{\sigma}(\nu + \tilde{\nu})\nabla\tilde{\nu}, \tag{2.9}$$

where $\nu := \mu/\rho$ is the kinematic molecular viscosity, and the source term

$$Q_T(\tilde{\nu}, \boldsymbol{u}) = C_{b1}(1 - f_{t2})\tilde{S}\tilde{\nu} + \frac{C_{b2}}{\sigma}|\nabla\tilde{\nu}|^2 - \left(C_{w1}f_w - \frac{C_{b1}}{\kappa^2}f_{t2}\right)\left(\frac{\tilde{\nu}}{d}\right)^2. \tag{2.10}$$

The SA model PDE can be written in differential form as

$$\frac{\partial \tilde{\nu}}{\partial t} + \nabla \cdot \boldsymbol{F}_T^i = \nabla \cdot \boldsymbol{F}_T^v + Q_T. \tag{2.11}$$

The eddy viscosity is then given by $\mu_t = f_{v1}\rho\tilde{\nu}$. The definitions of the various parameters in these equations may be found in [28, chapter 7].

If we drop the viscous terms by ignoring $\boldsymbol{F}^v$ in equation (2.4), we get the compressible Euler equations of inviscid flow. If we drop the SA equation and the dependence on $\tilde{\nu}$ in equation (2.4), we obtain equations that govern compressible laminar flow.

2.1 Discretization

In this work, cell-centred finite volume methods are used to discretize the integral governing equations (2.1),(2.2),(2.3),(2.8).

In FANSC-Lite, the spatial discretization is a cell-centred finite volume scheme using a central flux with matrix-dissipation [29] for the inviscid terms. The viscous flux is computed using averaged Green-Gauss gradients [27]. The code applies this discretization on multi-block structured grids. Such a grid is made up of several 'logically cubical' blocks indexed by three indices (typically i, j and k). The different blocks interface with neighbouring blocks in an unstructured manner, requiring detailed specification of block interfaces.

In the case of the unstructured grid code FVENS, grids with both quadrilaterals and triangles are supported. Upwind numerical fluxes are used for the inviscid terms; the Roe [28, section 4.3.3] [30] and HLLC [31, 32] approximate numerical fluxes have been used in this work. Gradients at cell-centres are estimated by a least-squares approach using data from face-neighbouring cells and reconstruction to faces is done by linear interpolation. Limiters are available, but were not used for the results shown in this work. Gradients at faces, required for viscous fluxes, are computed as a modified average of the left and right cell-centred gradients [28, section 5.4.2]. Second-order accuracy is achieved for smooth flows.

We drop the time derivative terms in case of a steady-state problem. The steady-state flow equations are discretized in space to obtain a nonlinear system of equations.

$$\boldsymbol{r}(\boldsymbol{w}) = \boldsymbol{0}. \tag{2.12}$$

where $\boldsymbol{w} \in W \subseteq \mathbb{R}^n$ with n being the number of cells times the number of conserved variables. $\boldsymbol{r}$ is the vector of the discretized residual functions on all cells stacked in the same order as $\boldsymbol{w}$.

2.2 Nonlinear solvers

To solve equation (2.12), it is common in CFD to use a ‘pseudo-time stepping approach by adding an artificial time term to the equation, thus replacing it with a system of nonlinear ordinary differential equations (ODEs).

$$\boldsymbol{V}\frac{d\boldsymbol{w}}{d\tau} + \boldsymbol{r}(\boldsymbol{w}) = \boldsymbol{0} \tag{2.13}$$

where $\boldsymbol{V}$ is some mass matrix depending on the spatial discretization. In case of the cell-centred finite volume method, this is a diagonal matrix with the volumes of the respective cells as the entries. Under appropriate regularity assumptions, we can iterate to the steady-state solution after starting from a trivial initial condition such as uniform flow. As $\frac{d\boldsymbol{w}}{d\tau} \to 0$, $\boldsymbol{w}$ approaches the solution of the original equation (2.12). One of the simplest ways of solving equation (2.13) is to discretize in pseudo-time using the explicit forward Euler method:

$$\frac{\boldsymbol{V}}{\Delta\tau}(\boldsymbol{w}^{n+1} - \boldsymbol{w}^n) + \boldsymbol{r}(\boldsymbol{w}^n) = \boldsymbol{0}. \tag{2.14}$$

Since we do not require accuracy in time, each cell can have a different time step $\Delta\tau$. Linear stability, in case of finite volume methods, requires $\Delta\tau \leq \frac{h}{|\boldsymbol{v}+c|}$, where h is the characteristic length dimension of the cell, $\boldsymbol{v}$ is a characteristic flow velocity in the cell and c is a characteristic speed of sound in the cell. If we use the maximum time step, the update can then be written as

$$\boldsymbol{w}^{n+1} = \boldsymbol{w}^n - \frac{h}{|\boldsymbol{v}+c|}\boldsymbol{V}^{-1}\boldsymbol{r}(\boldsymbol{w}^n). \tag{2.15}$$

A different approach to solve equation (2.13) is an implicit pseudo-time discretization, such as the backward Euler method. This is otherwise known as a Newton-Raphson method with a pseudo-transient continuation.

$$\frac{\boldsymbol{V}}{\Delta\tau}(\boldsymbol{w}^{n+1} - \boldsymbol{w}^n) + \boldsymbol{r}(\boldsymbol{w}^{n+1}) = \boldsymbol{0}, \tag{2.16}$$

and linearizing about the current time step, we get

$$\left(\frac{\boldsymbol{V}}{\Delta\tau} + \frac{\partial\tilde{\boldsymbol{r}}}{\partial\boldsymbol{w}}(\boldsymbol{w}^n)\right)\Delta\boldsymbol{w}^n = -\boldsymbol{r}(\boldsymbol{w}^n), \tag{2.17}$$

$$\boldsymbol{w}^{n+1} = \boldsymbol{w}^n + \omega\Delta\boldsymbol{w}^n, \tag{2.18}$$

where ω is a relaxation factor which is ideally close to 1.0, but can be less for robustness reasons.

Thus at every time step, we compute the right hand side and Jacobian matrix, solve the linear system of equations and update the solution. We can ramp up the time step progressively, and in the case of an exact Jacobian matrix ($\tilde{\boldsymbol{r}} = \boldsymbol{r}$), the Newton method can be recovered as $\Delta\tau \to \infty$. In the finite volume context, however, it is typical to use an approximate linearization of the residual as the Jacobian matrix, usually the linearization of a first-order accurate discretization, while the residual used on the right hand side is a second-order accurate discretization. In the case of the unstructured grid code FVENS, the Jacobian matrix used in (2.18) is computed ignoring the reconstruction, while higher order artificial dissipation terms are ignored in FANSC-Lite. This is done to drop dependence on neighbours of neighbours of a cell and thus keep the linearized discretization 'compact'. This is also done to the viscous flux linearization: gradients of flow variables are computed using the 'thin-layer' approximation while computing the derivative of the viscous flux.

Designing an effective and efficient linear solver is an important part of this approach. One good option that is potentially optimally scalable (having $\mathcal{O}(n)$ complexity) is the multigrid method [33].

On the other hand, the nonlinear system (2.12) can be solved using an efficient nonlinear solver, such as the full approximation storage multigrid method [33]. Both this and the linear multigrid method used in this work are described in section 2.5.

2.3 Classical iterations

Some concepts necessary for analyzing solvers are reviewed in this section. Suppose we need to solve

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{2.19}$$

$\boldsymbol{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{b} \in \mathbb{R}^n$. Stationary iterative methods are generally based on an additive splitting of the matrix $\boldsymbol{A} = \boldsymbol{M} + \boldsymbol{N}$, such that $\boldsymbol{M}$ is easy to invert. The original linear system can the be expressed as $(\boldsymbol{M} + \boldsymbol{N})\boldsymbol{x} = \boldsymbol{b}$. In this case, a stationary linear iteration can be written as

$$\boldsymbol{x}^{k+1} = \boldsymbol{M}^{-1}(\boldsymbol{b} - \boldsymbol{N}\boldsymbol{x}^k), \quad k \in \{1, 2, ...\}. \tag{2.20}$$

Such an iteration will henceforth be referred to as a linear 'relaxation'. It has been shown [34, 35] that this iteration converges from any initial guess if and only if $\rho(\boldsymbol{M}^{-1}\boldsymbol{N}) < 1$, where ρ denotes the spectral radius of a matrix.

An alternate way of thinking about linear iterations is provided by the concept of a preconditioner. Preconditioning refers to replacing the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ with an equivalent system that is easier to solve, by finding an operator to apply to both sides of the equation.

The operator chosen is called a preconditioner.

Let $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ be a matrix such that $\boldsymbol{M}^{-1}\boldsymbol{A}$ or $\boldsymbol{A}\boldsymbol{M}^{-1}$ is easier to solve than $\boldsymbol{A}$ itself. If we adopt left-preconditioning, we will solve the preconditioned problem

$$\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b}. \tag{2.21}$$

If we choose right-preconditioning instead, we replace equation (2.19) with the equivalent system

$$\boldsymbol{A}\boldsymbol{M}^{-1}\boldsymbol{M}\boldsymbol{x} = \boldsymbol{b} \tag{2.22}$$

and choose the right-preconditioning matrix $\boldsymbol{M}$. 'Easy to solve' can be defined in a number of ways with the intention of quantifying well-posedness of the linear problem or faster convergence of an iterative method. One such measure is the condition number $\kappa(\boldsymbol{A}) := \|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\|$ in some norm $\|\cdot\|$, which determines the sensitivity of the solution $\boldsymbol{x}$ with respect to perturbations in the data, the coefficients of $\boldsymbol{A}$ and $\boldsymbol{b}$ [36]. If $\kappa(\boldsymbol{M}^{-1}\boldsymbol{A}) < \kappa(\boldsymbol{A})$, then $\boldsymbol{M}^{-1}\boldsymbol{A}$ is said to be better conditioned than $\boldsymbol{A}$, i.e. its solution is less sensitive to perturbations in the data. The condition number also appears in convergence estimates of some linear solvers, such as the conjugate gradient method for symmetric positive definite systems [36].

The concept of preconditioning is very useful in improving the efficacy of linear solvers, most commonly Krylov subspace solvers discussed later (section 2.6). We begin with a simple iterative solver - the preconditioned Richardson iteration - that also allows us to express stationary linear iterations in an alternative form. The preconditioned Richardson iteration for this problem is given by algorithm 1, where $\|\cdot\|$ is any vector norm.

Algorithm 1 Preconditioned Richardson algorithm

```
Require: An initial guess $\boldsymbol{x}_0 \in \mathbb{R}^n$, tolerance $\tau \in \mathbb{R}$, maximum number of iterations $N_{iter}$.
  $\boldsymbol{r}_0 \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
  $\Delta\boldsymbol{x} \leftarrow \boldsymbol{0}$
  $k = 1$                                                   ▷ Iteration number
  while $\|\boldsymbol{r}^{k-1}\|/\|\boldsymbol{b}\| > \tau$ and $k < N_{iter}$ do
      $\boldsymbol{r}^k \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^k$                        ▷ Residual computation
      $\boldsymbol{z} \leftarrow \boldsymbol{M}^{-1}\boldsymbol{r}^k$                              ▷ Preconditioning operation
      $\boldsymbol{x}^{k+1} \leftarrow \boldsymbol{x}^k + \boldsymbol{z}$                            ▷ Update
      $k \leftarrow k + 1$
  end while
```

This iteration can also be seen as an inexact Newton method from optimization, with the preconditioning matrix $\boldsymbol{M}$ playing the role of an approximate Hessian matrix, providing second-order information for increasing the rate of convergence. The vector $\boldsymbol{r} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$, commonly referred to as the linear residual or the defect, serves as the 'gradient' to be driven to zero.

Any stationary linear iteration, or relaxation, can be expressed as a preconditioned Richardson iteration and vice versa. Suppose $\boldsymbol{A} = \boldsymbol{M} + \boldsymbol{N}$ defines a relaxation as in equation (2.20). Then algorithm 1 is equivalent to, going backwards from $\boldsymbol{x}^{k+1}$,

$$\begin{aligned}
&\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \boldsymbol{M}^{-1}\boldsymbol{r}^k \\
\iff\ &\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \boldsymbol{M}^{-1}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^k) \\
\iff\ &\boldsymbol{x}^{k+1} = (\boldsymbol{I} - \boldsymbol{M}^{-1}(\boldsymbol{M}+\boldsymbol{N}))\boldsymbol{x}^k + \boldsymbol{M}^{-1}\boldsymbol{b} \\
\iff\ &\boldsymbol{x}^{k+1} = \boldsymbol{M}^{-1}\boldsymbol{b} - \boldsymbol{M}^{-1}\boldsymbol{N}\boldsymbol{x}^k.
\end{aligned}$$

Thus, it should be remembered that the classical iterations discussed here can be written in both relaxation form and preconditioned Richardson form. The latter is also sometimes referred to as a defect correction form. For classical linear iterations, the two are clearly equivalent.

Classical linear iterations such as Jacobi, Gauss-Seidel, incomplete LU (ILU) factorization and others are sometimes called 'approximate factorization' methods, and the difference $\boldsymbol{A} - \boldsymbol{M}$ is called the factorization error. This terminology is useful for understanding such methods because $\boldsymbol{M}$ is the only part of $\boldsymbol{A}$ that is actually being inverted in every iteration. If $\boldsymbol{M} = \boldsymbol{A}$, the solve would be over in one very expensive iteration. Thus, the factorization error is an indication of how many iterations would be required for convergence to some accuracy.

2.3.1 Jacobi and Gauss-Seidel iterations

Suppose $\boldsymbol{D}$ is the diagonal matrix consisting of diagonal entries of $\boldsymbol{A}$. This will be referred to as the 'diagonal part' of $\boldsymbol{A}$. Further, let $\boldsymbol{E}$ be the strictly lower triangular part and $\boldsymbol{F}$ be the strictly upper triangular part of $\boldsymbol{A}$. That is,

$$E_{ij} = \begin{cases} A_{ij} & \text{if } i > j \\ 0 & \text{otherwise} \end{cases}$$

and

$$F_{ij} = \begin{cases} A_{ij} & \text{if } i < j \\ 0 & \text{otherwise} \end{cases} .$$

Then, $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{E} + \boldsymbol{F}$. The Jacobi iteration is given by the splitting $\boldsymbol{M} = \boldsymbol{D}, \boldsymbol{N} = \boldsymbol{E} + \boldsymbol{F}$ in either the relaxation form (2.20) or the defect correction form (algorithm 1). Often, a weighted Jacobi iteration is used, given by $\boldsymbol{M} = \frac{1}{\omega}\boldsymbol{D}$, with ω typically tuned for different cases and purposes. To illustrate the Jacobi iteration, suppose we have a one-dimensional grid of cells indexed by i. Then the unweighted Jacobi iteration for a scalar PDE (with only one physical variable and one physical equation) is given as

$$x_i^{k+1} = x_i^k - \frac{1}{D_i}(\boldsymbol{A}\boldsymbol{x}^k - \boldsymbol{b})_i = x_i^k - \frac{1}{D_i}(E_i x_{i-1}^k + D_i x_i^k + F_i x_{i+1}^k - b_i) \tag{2.23}$$

where D_i, E_i and F_i are the (only) diagonal, lower and upper entries in the matrix $\boldsymbol{A}$.

It can be seen that the Jacobi iteration has the advantage that it is trivially parallel. As a preconditioner, its application only needs division of each vector entry by the corresponding diagonal entry. In relaxation form, it additionally needs multiplication by the upper and lower triangular matrices, which is also a parallel operation. However, it is typically has poor convergence rate [35], and is unreliable as a multigrid smoother [17]. The weight ω drastically affects the smoothing property [37, chapter 2].

One theme in computational fluid dynamics of compressible flows is the domain of dependence and the range of influence. Depending on the flow physics and conditions, flow variables (density, velocity, pressure) at a given grid location can get influenced by, and therefore depend on, a certain number of other locations. It can also influence the flow variables at certain other grid locations. For example, in subsonic flow, every cell in the domain ultimately depends on, and influences, every other cell. In supersonic flow, only cells that are upstream of a given cell influence it, and it only influences cells that are downstream from it. Intuitively, a solver will converge fast if it is able to quickly and accurately spread the influence of changes in flow variables to all the grid cells that physically need to be influenced. In case of the Jacobi iteration, each update of a cell's flow values only influence at most its immediate neighbours. Thus if there are N cells, the Jacobi solver needs $\mathcal{O}(N)$ iterations for every cell to receive any information from all other cells. In fact, this is similar to an explicit time stepping scheme for a linear problem, which (2.23) reminds us of.

The forward Gauss-Seidel method, otherwise known simply as Gauss-Seidel, is given by $\boldsymbol{M} = \boldsymbol{D} + \boldsymbol{E}$. Note that in this case, a lower triangular matrix needs to actually be inverted at every iteration. A recursion is typically employed to perform this inversion, and the iteration at one cell is

$$x_i^{k+1} = x_i^k - \frac{1}{D_i}(E_i x_{i-1}^{k+1} + D_i x_i^k + F_i x_{i+1}^k - b_i). \tag{2.24}$$

Thus, there is a data dependency and therefore it is difficult to parallelize the Gauss-Seidel iteration. Further, the iteration now depends on the ordering of the grid cells, which can also be seen from the fact that the definition of the strictly lower triangular matrix $\boldsymbol{E}$ depends on the ordering of the matrix $\boldsymbol{A}$. However, because of that dependency, information now propagates across the entire domain in each iteration, unlike Jacobi, albeit in only one direction. In the one-dimensional grid example, cell 1 is able to influence cell N in a single iteration, though the latter still needs N iterations to influence the former. Thus, one can see that for a supersonic flow in the direction of consecutive grid ordering, just one iteration of forward Gauss-Seidel would be enough to compute the solution of the linearized problem. Note that the actual number of mathematical operations needed to compute the update is equal to that of the Jacobi iteration, and thus this method is typically more efficient for a given number of memory loads and floating point operations ('flops'). It is observed that this method is a good multigrid smoother [37] and its convergence rate is generally better than Jacobi [35]. A backward Gauss-Seidel can similarly be defined by $\boldsymbol{M} = \boldsymbol{D} + \boldsymbol{F}$. The direction of the recursion is now reversed; in the 1D grid example, cell N can now influence cell 1 in a single iteration, though not vice-versa.

As usual, the preconditioned Richardson form is mathematically equivalent to the relaxation form in case of the Gauss-Seidel iteration. However, there is a difference in the cost. The forward Gauss-Seidel update can be written in relaxation form as

$$\boldsymbol{x}^{k+1} = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{E}\boldsymbol{x}^{k+1} - \boldsymbol{F}\boldsymbol{x}^{k}) \tag{2.25}$$

while the defect correction (preconditioned Richardson) form is

$$\begin{aligned} \boldsymbol{r} &= \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{k} \\ \boldsymbol{z} &= \boldsymbol{D}^{-1}(\boldsymbol{r} - \boldsymbol{E}\boldsymbol{z}). \\ \boldsymbol{x}^{k+1} &= \boldsymbol{x}^{k} + \boldsymbol{z} \end{aligned} \tag{2.26}$$

The cost of relaxation is roughly equivalent to one residual computation, which is just the first step of the defect correction iteration. The latter needs one extra triangular solve in the preconditioner application step.

One of the most popular iterations in CFD is the symmetric Gauss-Seidel method. It consists of a forward Gauss-Seidel sweep, followed by a backward Gauss-Seidel sweep in each iteration. It is given by $\boldsymbol{M} = (\boldsymbol{D} + \boldsymbol{E})\boldsymbol{D}^{-1}(\boldsymbol{D} + \boldsymbol{F})$. This iteration enables propagation of information in all directions. Since it is the product of two iterations that are difficult to parallelize, it has the same issue.

Diagonal dominance

For future reference, we describe here the concept of diagonal dominance, important in the convergence analysis of linear iterations. Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ be a matrix of interest. We define the degree of diagonal dominance for a row of the matrix as

$$\beta_i := \frac{|a_{ii}| - \sum_{j\neq i} |a_{ij}|}{|a_{ii}|} = 1 - \frac{\sum_{j\neq i} |a_{ij}|}{|a_{ii}|}. \tag{2.27}$$

In general, $\beta_i \in (-\infty, 1]$. If $\min_i \beta_i > 0$, the matrix is diagonal dominant.

Now, noting that the infinity norm (which is the maximum row sum of a matrix) of the Jacobi iterative matrix for $\boldsymbol{A}$ is

$$\|\boldsymbol{S}_J\|_\infty = \|\boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{A}\|_\infty = \max_i \frac{\sum_{j\neq i} |a_{ij}|}{|a_{ii}|}, \tag{2.28}$$

it is seen that

$$\|\boldsymbol{S}_J\|_\infty = \max_i (1 - \beta_i) = 1 - \min_i \beta_i. \tag{2.29}$$

Thus, the minimum diagonal dominance of a matrix gives us a bound on the convergence (in the max norm) of a Jacobi iteration applied to the matrix.

2.3.2 Incomplete LU factorization

LU factorization is commonly used as a robust direct solver. It refers to computing triangular factors as $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$, where $\boldsymbol{L}$ is a unit lower triangular matrix and $\boldsymbol{U}$ is an upper triangular matrix. It is commonly computed using Gaussian elimination. However, the cost scales in a super-linear manner, eg. $\mathcal{O}(N^2)$ for elliptic PDEs discretized on two-dimensional structured grids [37, chapter 1]. One reason for this is the fact that additional non-zeros are introduced in the lower and upper triangular factors in locations where $\boldsymbol{A}$ itself does not have any zeros, making the factorization much more dense compared to the original matrix. One way to adapt LU factorization to iterative methods is the class of incomplete LU (ILU) factorizations, which are commonly used as (sequential) preconditioners. ILU preconditioners and iterations are given by the splitting $\boldsymbol{A} = \boldsymbol{M} + \boldsymbol{N}$ with the preconditioning matrix

$$\boldsymbol{M} = \boldsymbol{L}\boldsymbol{U}, \tag{2.30}$$

where $\boldsymbol{L}$, a unit lower triangular matrix and $\boldsymbol{U}$, an upper triangular matrix which approximate the LU factorization of $\boldsymbol{A}$. It is common to simply use Gaussian elimination, coupled with a methodology for dropping excess non-zero entries.

ILU factorization is one of the most commonly used preconditioners for Krylov subspace solvers in CFD [38, 39]. They are sometimes used simply as iterations [40]. Finally, they are used as multigrid smoothers for anisotropic problems and convection-diffusion problems [37, chapter 7] [41, chapter 10] [42].

Several variants are available depending on parameters such as level-of-fill, threshold for a drop-tolerance and weighting [34, 35]. Depending on these parameters, the factorization error $\boldsymbol{N} = \boldsymbol{A} - \boldsymbol{L}\boldsymbol{U}$, and thus convergence rate, can be controlled. Level of fill refers to allowing non-zeros in the lower and upper triangular factors at locations determined exclusively by the sparsity pattern of the original matrix. ILU(0) corresponds to allowing non-zeros only in those locations that are non-zero in the original matrix; higher levels add more non-zeros based only on the sparsity pattern of the previous level. On the other hand, sparsity of the factors can be controlled depending on the magnitude of the introduced non-zeros and a drop tolerance. These two approaches can also be combined. Similar to Gauss-Seidel type iterations, ILU factorizations are significantly affected by the ordering of the original matrix. This has been studied in the context of CFD [38] and it is found that the ordering can be changed to attempt to improve convergence.

The traditional algorithms for computing ILU preconditioners are sequential and difficult to parallelize. A typical algorithm for fixed-pattern ILU factorization, which progressively replaces $\boldsymbol{A}$ with $\boldsymbol{L}$ and $\boldsymbol{U}$, is shown in algorithm 2.

Algorithm 2 Fixed-pattern ILU $\boldsymbol{A} \approx \boldsymbol{L}\boldsymbol{U}$

```
Require: Non-singular $\boldsymbol{A}$ with non-zero diagonal, a set $\mathcal{S}$ of non-zero locations including
    the diagonal
1: for $i$ from 1 to $n-1$ do
2:     for $j$ from $i+1$ to $n$ s.t. $(i,j) \in \mathcal{S}$ do
3:         $a_{ji} \leftarrow a_{ji}/a_{ii}$                              ▷ Data dependency
4:         for $k$ from $i+1$ to $n$ s.t. $(j,k) \in \mathcal{S}$ do
5:             $a_{jk} \leftarrow a_{jk} - a_{ji}a_{ik}$                  ▷ Data dependency
6:         end for
7:     end for
8: end for
```

2.4 Matrix ordering

At the outset, we note that while working with the mathematics and implementation of preconditioners and solvers, we need to deal with two kinds of orderings:

1. the logical or mathematical ordering, which affects the properties of the iteration matrix and therefore the number of iterations required for convergence,
2. the layout of the vectors and matrices in computer memory, which typically only affects the wall-clock time taken by each iteration but not the number of iterations.

In this section, we only deal with the logical ordering. Memory layouts are discussed separately (see section 2.9).

It has been found that the order in which the variables and equations in a linear system are arranged affects the performance of some iterations, such as Gauss-Seidel and ILU [43] [38]. The ordering of variables corresponds to the column ordering of the matrix while the ordering of the equations corresponds to the row ordering, though we assume symmetric orderings, in the sense that the rows and columns are ordered in the same way. For scalar PDEs, the matrix ordering depends only on the ordering of the relevant grid entities (cells, for the purpose of this thesis). For systems of PDEs, the ordering of variables in the matrix can be thought of as arising from two considerations - the ordering of the mesh entities, and the ordering of the different physical variables (mass, x-momentum, y-momentum, z-momentum, energy etc.).

2.4.1 Grid orderings

With structured grids, there is a 'natural' ordering corresponding to the ijk ordering of the structured block, typically with cells numbered along i first, then j-lines and finally k-planes. However, this ordering is not always the best suited for linear iterations. Cuthill-McKee, reverse Cuthill-McKee [44], nested dissection and minimum degree are some of the common topological grid orderings [34]. These are topological orderings as they depend only on the connectivity of the grid (more generally, the non-zero pattern of the matrix), rather than the values of matrix entries or physical locations of grid points. These orderings lead to different properties of the matrix. For example, reverse Cuthill-McKee is typically favoured for ILU preconditioning, as it reduces the bandwidth (the number of nonzero sub-diagonals and super-diagonals) of the matrix, thereby reducing the number of non-zeros that have to be dropped and thus decreasing the factorization error. Other orderings may be advantageous for different types of solvers; direct solvers frequently make use of such reordering [45].

In addition, some topological orderings can be used for improving parallelism of Gauss-Seidel and ILU-type preconditioners. One example is multi-coloured ordering. These are discussed in section 2.7.

There are also reorderings that are used to great effect which are not topological. An example of this are line orderings that depend on the spatial location of grid entities. These are discussed in section 4.5.

2.4.2 Point-block iterations

In systems of PDEs, as in the Navier-Stokes equations, there are several physical quantities to be solved for, and an equal number of coupled PDEs to solve. For the Navier-Stokes equations, the variables are density, x-, y- and z-momentum, and energy, while the (non-linear) residuals of the equations correspond to mass flux, momentum flux and energy flux in the integral form. The choice of arrangement of degrees of freedom (discrete variables) corresponding to different physical variables, in the logical ordering, is important.

One can order all the degrees of freedom corresponding to density first, followed by those corresponding to x-momentum, and so on. This leads to a fixed number of large sparse blocks in the Jacobian matrix; 5×5 in case of the three-dimensional Navier-Stokes equations. We refer to this as physics-blocking, as each block corresponds to the dependence of one physical flux on one physical quantity across the entire grid. Different block preconditioners may be used to couple all the blocks, thus coupling the different physical quantities. These include block Gauss-Seidel and approximate block LU, also referred to as Schur-complement preconditioners. Such block preconditioners depend on inner preconditioners to approximately invert the diagonal blocks. This framework has its strengths, especially for saddle-point problems such as some formulations of the incompressible Navier-Stokes equations. It is also flexible in that it lets us use different inner preconditioners for different physics blocks, if necessary.

On the other hand, one can use point-block ordering, in which the degrees of freedom of different physical quantities at the same grid location are placed consecutively. This leads to a large number of small dense blocks in the Jacobian matrix. For cell-centred discretizations, ‘cell-block’ ordering might be more appropriate, but we still use the more common term point-block ordering. The matrix $\boldsymbol{A}$ can now be viewed as being made up of these blocks and each block can be given an (i, j) block-index depending on its location in the matrix with respect to other blocks. A $b \times b$ block is referred to as a ‘non-zero block’ if there is at least one non-zero entry in the block. One can now derive point-block variants of stationary linear iterations, where the small diagonal blocks are inverted exactly. Intuitively, for systems like the compressible Navier-Stokes equations where there is a high degree of coupling between the five variables (in 3D), this is an attractive proposition. The most common such iteration is the point-block symmetric Gauss-Seidel solver, commonly referred to as ‘LU-SGS’ [46, 47, 17]. While there are a few variants of LU-SGS that differ in the faithfulness of the linearization used for the Jacobian blocks, in this thesis we do not make a distinction between point-block SGS and LU-SGS. We use an accurate linearization of the first-order fluxes, as described in section 2.2.

In block SGS, the matrices in $\boldsymbol{D}, \boldsymbol{E}$ and $\boldsymbol{F}$ are modified as follows. $\boldsymbol{D}$ is now a block-diagonal matrix

$$\boldsymbol{D} = \begin{bmatrix} \boldsymbol{A}_{11} & & & & \\ & \boldsymbol{A}_{22} & & & \\ & & \ddots & & \\ & & & \boldsymbol{A}_{N-1,N-1} & \\ & & & & \boldsymbol{A}_{N,N} \end{bmatrix} \tag{2.31}$$

while $\boldsymbol{E}$ and $\boldsymbol{F}$ are strictly lower block triangular and strictly upper block triangular matrices

$$\boldsymbol{E} = \begin{bmatrix} \mathbf{0}_{b\times b} & & & & \\ \boldsymbol{A}_{21} & \mathbf{0}_{b\times b} & & & \\ \vdots & & \ddots & & \\ \boldsymbol{A}_{N-1,1} & \dots & \boldsymbol{A}_{N-1,N-2} & \mathbf{0}_{b\times b} & \\ \boldsymbol{A}_{N,1} & \dots & \boldsymbol{A}_{N,N-2} & \boldsymbol{A}_{N,N-1} & \mathbf{0}_{b\times b} \end{bmatrix}, \quad \boldsymbol{F} = \begin{bmatrix} \mathbf{0}_{b\times b} & \boldsymbol{A}_{12} & \boldsymbol{A}_{13} & \dots & \boldsymbol{A}_{1,N} \\ & \mathbf{0}_{b\times b} & \boldsymbol{A}_{23} & \dots & \boldsymbol{A}_{2,N} \\ & & \ddots & & \vdots \\ & & & \mathbf{0}_{b\times b} & \boldsymbol{A}_{N-1,N} \\ & & & & \mathbf{0}_{b\times b} \end{bmatrix} \tag{2.32}$$

where $\mathbf{0}_{b\times b}$ denotes the $b \times b$ zero matrix respectively and $\boldsymbol{A}_{ij}$ denotes the $b \times b$ block of the matrix $\boldsymbol{A}$ at the (i, j) block position. A blank in any location also indicates a zero block .

Such point-block iterations allow us to extend the classical iterations to systems of PDEs while accurately resolving the coupling between the different physical quantities at each grid location. The author has observed that point-block SGS is a much more robust and fast iteration than scalar SGS for some simple compressible flow cases.

Point-block ILU factorization is also used in the solution of compressible flows by finite volume methods. Block factorization is defined here with respect to dense square blocks of size $b \times b$, such that n (the total number of rows in the square matrix) is divisible by b. While the results presented here can be extended to more general and variable block sizes, we adopt fixed-size square blocks for this work. For our problems, $b = d + 2$ and each block represents the coupling between the $d+2$ fluxes and flow variables at two grid locations. Our finite volume discretization of the compressible Navier-Stokes equations gives dense, disjoint blocks.

For the block ILU factorization, $\boldsymbol{L}$ is a lower block triangular matrix with identity matrices for the diagonal blocks and $\boldsymbol{U}$ is an upper block triangular matrix with non-singular diagonal blocks.

$$
\boldsymbol{L} = \begin{bmatrix} \boldsymbol{I}_{b\times b} & & & & \\ \boldsymbol{L}_{21} & \boldsymbol{I}_{b\times b} & & & \\ \vdots & & \ddots & & \\ \boldsymbol{L}_{N-1,1} & \dots & \boldsymbol{L}_{N-1,N-2} & \boldsymbol{I}_{b\times b} & \\ \boldsymbol{L}_{N,1} & \dots & \boldsymbol{L}_{N,N-2} & \boldsymbol{L}_{N,N-1} & \boldsymbol{I}_{b\times b} \end{bmatrix}, \quad \boldsymbol{U} = \begin{bmatrix} \boldsymbol{U}_{11} & \boldsymbol{U}_{12} & \boldsymbol{U}_{13} & \dots & \boldsymbol{U}_{1,N} \\ & \boldsymbol{U}_{22} & \boldsymbol{U}_{23} & \dots & \boldsymbol{U}_{2,N} \\ & & \ddots & & \vdots \\ & & & \boldsymbol{U}_{N-1,N-1} & \boldsymbol{U}_{N-1,N} \\ & & & & \boldsymbol{U}_{N,N} \end{bmatrix} \tag{2.33}
$$

where $\boldsymbol{I}_{b\times b}$ denotes the $b \times b$ identity matrix.

2.5 Multigrid solvers

Multigrid methods are based on the ideas of smoothing and coarse grid correction [33, 14]. Scalable ($\mathcal{O}(N)$) single-grid relaxation methods (‘iterations’) can be designed that quickly reduce ‘high-frequency’ error components on a given grid irrespective of the mesh size (number of cells) N. The remaining ‘low-frequency’ components are dealt with recursively via corrections computed on coarser grids. On each coarse grid, some of these low-frequency error components can quickly be reduced by single-grid iterations. FANSC-Lite has nonlinear full approximation storage (FAS) multigrid for 2D and 3D problems and linear multigrid capability for 3D problems. In the following, the example of a two-grid method is frequently used to explain multigrid solvers, involving a fine grid with characteristic cell size h and a coarser grid with characteristic cell size H, where $h < H$. Recursive application of the two-grid method to solve the coarse grid problem leads to different multigrid methods. On the fine mesh with characteristic edge length h, we write the discretized system of nonlinear equations as

$$
\boldsymbol{r}_h(\boldsymbol{w}_h) = \boldsymbol{0}. \tag{2.34}
$$

These concepts are discussed in great detail in books by Trottenberg, Oosterlee and Schuller [37] and Hackbusch [41], among others. Multigrid methods have been observed to be effective as linear or nonlinear solvers for CFD problems [16, 48].

2.5.1 Nonlinear multigrid using full approximation storage

The nonlinear FAS multigrid method [33] is employed to solve the nonlinear equation (2.34). The sequence of operations in a two-grid FAS method is shown in figure 2.1. The multigrid method is obtained by solving the coarse-grid equation using FAS recursively. As shown,

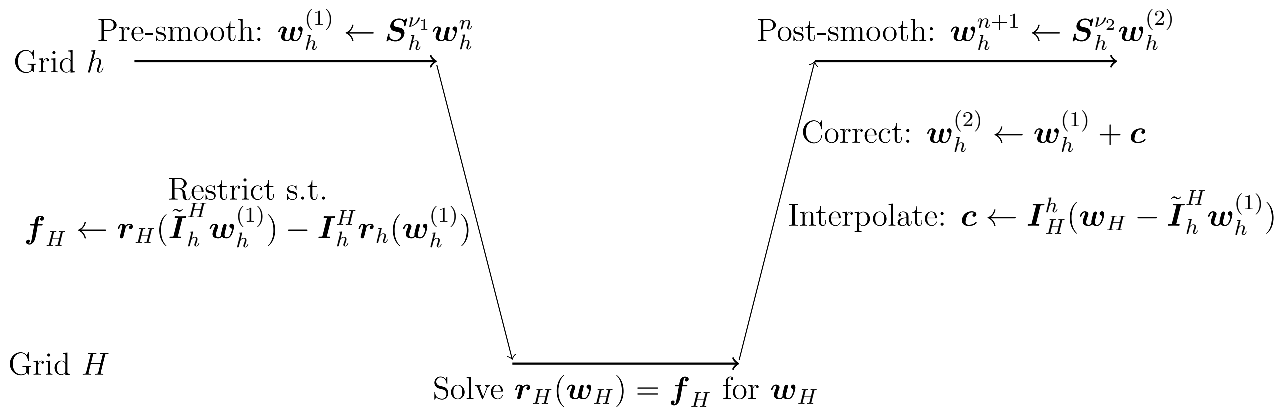

Pre-smooth: $\boldsymbol{w}_h^{(1)} \leftarrow \boldsymbol{S}_h^{\nu_1}\boldsymbol{w}_h^n$
Post-smooth: $\boldsymbol{w}_h^{n+1} \leftarrow \boldsymbol{S}_h^{\nu_2}\boldsymbol{w}_h^{(2)}$
Grid h
Correct: $\boldsymbol{w}_h^{(2)} \leftarrow \boldsymbol{w}_h^{(1)} + \boldsymbol{c}$
Restrict s.t.
$\boldsymbol{f}_H \leftarrow \boldsymbol{r}_H(\tilde{\boldsymbol{I}}_h^H\boldsymbol{w}_h^{(1)}) - \boldsymbol{I}_h^H\boldsymbol{r}_h(\boldsymbol{w}_h^{(1)})$
Interpolate: $\boldsymbol{c} \leftarrow \boldsymbol{I}_H^h(\boldsymbol{w}_H - \tilde{\boldsymbol{I}}_h^H\boldsymbol{w}_h^{(1)})$
Grid H
Solve $\boldsymbol{r}_H(\boldsymbol{w}_H) = \boldsymbol{f}_H$ for $\boldsymbol{w}_H$

Figure 2.1: Full approximation storage (FAS) two-grid method for solving $\boldsymbol{r}_h(\boldsymbol{w}_h) = \boldsymbol{0}$

the coarse grid equation is given by

$$\boldsymbol{r}_H(\boldsymbol{w}_H) = \boldsymbol{f}_H, \tag{2.35}$$

$$\boldsymbol{f}_H = \boldsymbol{r}_H\big(\tilde{\boldsymbol{I}}_h^H\boldsymbol{w}_h\big) - \boldsymbol{I}_h^H\boldsymbol{r}_h(\boldsymbol{w}_h). \tag{2.36}$$

Here $\tilde{\boldsymbol{I}}_h^H$ and $\boldsymbol{I}_h^H$ are different restriction operators. This is solved for $\boldsymbol{w}_H$ and the correction is given by

$$\boldsymbol{w}_h \leftarrow \boldsymbol{w}_h + \boldsymbol{I}_H^h\big(\boldsymbol{w}_H - \tilde{\boldsymbol{I}}_h^H\boldsymbol{w}_h\big). \tag{2.37}$$

as shown in figure 2.1, where $\boldsymbol{I}_H^h$ is called a prolongation or interpolation matrix. This process of coarse grid correction is carried out recursively down to some coarsest level, where the nonlinear problem is solved by a single-grid nonlinear iteration. This iteration on the coarsest grid could be the same as that used for smoothing or it could be a stronger solver approaching an exact coarse grid solve.

The smoother $\boldsymbol{S}_h$ may be any nonlinear iteration capable of damping high-frequency modes of the error. It is applied on equation (2.34) ν_1 times for pre-smoothing and ν_2 more times after coarse-grid correction. A simple option is to use explicit smoothers, such as forward Euler (equation (2.15)). However, explicit smoothers are slow and may lead to the solver stalling for complex cases [17]. A more desirable option is to use an implicit Newton-like iteration for smoothing. Since it may be useful to add pseudo-transient continuation, we show below one step of a backward Euler smoother.

$$\left(\frac{\boldsymbol{V}_h}{\Delta\tau_h} + \frac{\partial \boldsymbol{r}_h}{\partial \boldsymbol{w}_h}(\boldsymbol{w}_h^n)\right)\Delta\boldsymbol{w}_h^n = -\boldsymbol{r}_h(\boldsymbol{w}_h^n), \tag{2.38}$$

$$\boldsymbol{w}_h^{n+1} = \boldsymbol{w}_h^n + \omega\Delta\boldsymbol{w}_h^n, \tag{2.39}$$

where $\boldsymbol{V}_h$ is the diagonal matrix of cell volumes and ω is a relaxation parameter. One may also choose to drop the pseudo-time term $\frac{\boldsymbol{V}_h}{\Delta\tau_h}$ and just perform an inexact Newton iteration; whether this is desirable depends on the case and the quality of the linearization $\frac{\partial \boldsymbol{r}_h}{\partial \boldsymbol{w}_h}$. In any case, note that we are not interested in iterating to convergence; typically we only perform one nonlinear iteration ($\nu_1 = \nu_2 = 1$ in figure 2.1) and continue with the multigrid cycle.

In this work, the relaxation factor ω in equation (2.39) is kept between 0.2 and 1.0 and is computed as a function of the relative change in the density and energy in the update $\Delta\boldsymbol{w}$. Further, in our implementation, a spatially first-order discretization is used to generate the nonlinear smoother on the coarser grids. Only the top (finest) grid uses the second-order discretization. Such a scheme typically leads to better robustness on the coarser levels. This entire method is referred to as 'FAS' henceforth.

2.5.2 Linear multigrid within a Newton-like nonlinear solver

On the other hand, we can also apply a Newton-like method directly to the original system (2.34) on the fine grid. The linear multigrid method [49, 33] is then used to solve the linear equation arising at each nonlinear iteration. The case of a linear two-grid method used to solve a Newton-Raphson step is shown in figure 2.2. As with FAS, the linear two-grid method is used to solve the coarse grid equation recursively for a multigrid method.

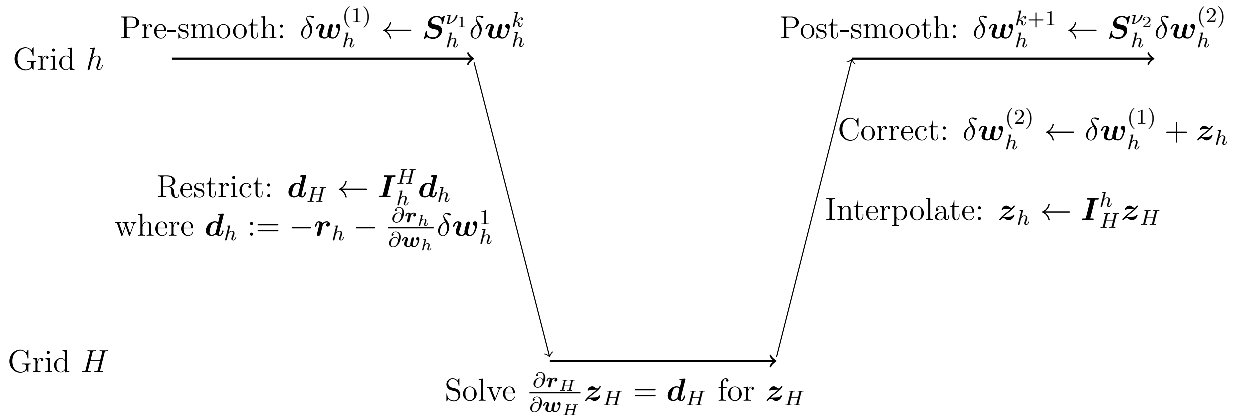

Grid h
Pre-smooth: $\delta\boldsymbol{w}_h^{(1)} \leftarrow \boldsymbol{S}_h^{\nu_1}\delta\boldsymbol{w}_h^k$
Post-smooth: $\delta\boldsymbol{w}_h^{k+1} \leftarrow \boldsymbol{S}_h^{\nu_2}\delta\boldsymbol{w}_h^{(2)}$
Correct: $\delta\boldsymbol{w}_h^{(2)} \leftarrow \delta\boldsymbol{w}_h^{(1)} + \boldsymbol{z}_h$
Restrict: $\boldsymbol{d}_H \leftarrow \boldsymbol{I}_h^H\boldsymbol{d}_h$
where $\boldsymbol{d}_h := -\boldsymbol{r}_h - \frac{\partial \boldsymbol{r}_h}{\partial \boldsymbol{w}_h}\delta\boldsymbol{w}_h^1$
Interpolate: $\boldsymbol{z}_h \leftarrow \boldsymbol{I}_H^h\boldsymbol{z}_H$
Grid H
Solve $\frac{\partial \boldsymbol{r}_H}{\partial \boldsymbol{w}_H}\boldsymbol{z}_H = \boldsymbol{d}_H$ for $\boldsymbol{z}_H$

Figure 2.2: kth cycle of the linear two-grid method for solving the Newton step $\frac{\partial \boldsymbol{r}_h}{\partial \boldsymbol{w}_h}(\boldsymbol{w}^n)\delta\boldsymbol{w}_h = -\boldsymbol{r}_h(\boldsymbol{w}^n)$

We may also use backward Euler pseudo-time stepping (Newton method with pseudo-transient continuation) instead of the full Newton method by adding a time term to the

Jacobian matrices as follows.

$$\left(\frac{\boldsymbol{V}_h}{\Delta\tau_h} + \frac{\partial \boldsymbol{r}_h}{\partial \boldsymbol{w}_h}(\boldsymbol{w}_h^n)\right)\Delta\boldsymbol{w}_h^n = -\boldsymbol{r}_h(\boldsymbol{w}_h^n), \tag{2.40}$$

$$\boldsymbol{w}_h^{n+1} = \boldsymbol{w}_h^n + \omega\Delta\boldsymbol{w}_h^n, \tag{2.41}$$

where ω is a relaxation factor computed similarly to the one described for pseudo-time step (equation (2.39)) in FAS. Linear multigrid is used to solve equation (2.40).

The smoothing operator $\boldsymbol{S}_h$ in the case of linear multigrid is any linear iteration capable of damping high-frequency error modes of the linear problems (2.40) within the Newton-like process. Variants of symmetric Gauss-Seidel iterations are quite popular in the multigrid and CFD communities. Finally, the coarse grid equation is given by

$$\left(\frac{\boldsymbol{V}_H}{\Delta\tau_H} + \frac{\partial \boldsymbol{r}_H}{\partial \boldsymbol{w}_H}(\tilde{\boldsymbol{I}}_h^H\boldsymbol{w}_h^n)\right)\boldsymbol{z} = \boldsymbol{d}_H, \tag{2.42}$$

where

$$\boldsymbol{d}_H = \boldsymbol{I}_h^H\left(-\left(\frac{\boldsymbol{V}_h}{\Delta\tau_h} + \frac{\partial \boldsymbol{r}_h}{\partial \boldsymbol{w}_h}(\boldsymbol{w}_h^n)\right)\Delta\boldsymbol{w}_h^{(1)} - \boldsymbol{r}_h(\boldsymbol{w}_h^n)\right). \tag{2.43}$$

(2.42) is approximately or exactly solved for $\boldsymbol{z}$ and the correction is given by $\Delta\boldsymbol{w}_h^n \leftarrow \Delta\boldsymbol{w}_h^{(1)} + \boldsymbol{I}_H^h\boldsymbol{z}$. This is then further used to correct the flow variables $\boldsymbol{w}$ by equation (2.41).

2.5.3 Multigrid cycles and components

Both nonlinear and linear two-grid methods (figures 2.1 and 2.2), are extended to multigrid methods by recursively solving the coarse-grid problem using a 2-grid method. Depending on the number of cycles γ used to solve the coarse grid problem, different multigrid cycles are obtained. When only one two-grid cycle is used to solve each coarse-grid problem, $\gamma = 1$ and the cycle is called a V-cycle. When two cycles are used, we get a W-cycle. Since the objective of this thesis is not to study cycling strategies, only the V-cycle is used throughout the work for uniformity.

Coarse grids for both linear and nonlinear multigrid methods are formed by agglomerating 2×2 (or $2\times 2\times 2$) fine grid cells for 2 (or 3) dimensional structured grids. Solution variables $\boldsymbol{w}$ are restricted using volume weighted restriction $\tilde{\boldsymbol{I}}_h^H$ and defects $\boldsymbol{r}$ or $\boldsymbol{d}$ are restricted ($\boldsymbol{I}_h^H$) through summation of defects in all fine grid cells corresponding to a coarse grid cell. Corrections are prolongated ($\boldsymbol{I}_H^h$) using bilinear (2D) or trilinear (3D) interpolation. Jacobian matrices on the coarse grids are computed from the restricted solution variables. As mentioned before, all Jacobian matrices are computed ignoring the terms required for second-order accuracy; that is, the first-order residual is linearized to compute the Jacobian.

On each grid level, linear fixed-point iterations are used to approximately solve equation (2.42) in linear multigrid and equation (2.38) in FAS.

In FANSC-Lite, multigrid is used only to solve the mean flow equations. The turbulence model variable(s) is updated in a segregated manner only on the fine grid.

Smoothing

An efficient and effective smoothing iteration is necessary for good convergence of multigrid methods; it must have a good smoothing property such that it can quickly damp high-frequency components of the error. Many iterative methods can be used as smoothers depending on the problem, such as polynomial iterations (explicit Runge-Kutta iterations), weighted Jacobi, Gauss-Seidel etc. However, effective smoothers such as symmetric Gauss-Seidel tend not to be amenable to fine-grain parallelism due to inherent data-dependence or unfavourable memory access patterns.

The weighted Jacobi smoother can be effective with a weight that is close to optimal, and is sometimes used for solving the Poisson equation [37]. They have the advantage of being trivially parallel. However, without a good weight, it is usually not a good smoother [14]. Finding a good weight requires knowledge of the spectrum of the operator. For more complex and nonlinear problems such as the Navier-Stokes equations, it is difficult to know good weighting values. The optimal weighting value could also change over the course of a nonlinear solve. For CFD problems, even point-block versions of Jacobi smoothers tend to have low convergence rates and are not robust [17] in case of non-optimal weights. This can be seen in the results of chapter 6 as well.

Polynomial smoothers involve application of a polynomial of the discrete PDE operator as a smoother. These are known as multi-stage Runge-Kutta smoothers in the CFD community. They are explicit iterations in the sense that no part of the operator (matrix) is actually inverted in each iteration. For general nonlinear problems, such multi-stage smoothers have the form

$$\boldsymbol{w}^{(1)} = \boldsymbol{w}^n - \alpha_1 \Delta\tau\, \boldsymbol{r}(\boldsymbol{w}^n) \tag{2.44}$$

$$\boldsymbol{w}^{(2)} = \boldsymbol{w}^n - \alpha_2 \Delta\tau\, \boldsymbol{r}(\boldsymbol{w}^{(1)}) \tag{2.45}$$

$$\vdots \tag{2.46}$$

$$\boldsymbol{w}^{(s-1)} = \boldsymbol{w}^n - \alpha_{s-1} \Delta\tau\, \boldsymbol{r}(\boldsymbol{w}^{(s-2)}) \tag{2.47}$$

$$\boldsymbol{w}^{n+1} = \boldsymbol{w}^n - \alpha_s \Delta\tau\, \boldsymbol{r}(\boldsymbol{w}^{(s-1)}) \tag{2.48}$$

where s is the number of stages and α_k are coefficients to be determined for good smoothing.

It was shown that with good coefficients, these smoothers can be very effective in solving the compressible Euler equations for inviscid flow problems [16]. However, several researchers have shown that implicit iterations, especially point-block symmetric Gauss-Seidel ('LU-SGS'), are required for fast and robust solution of Navier-Stokes and RANS calculations by a multigrid solver [50, 5, 17]. This is typically attributed to the inability of explicit Runge-Kutta smoothers to provide good smoothing in presence of highly stretched grids in boundary layers. Some researchers have combined implicit smoothing by SGS and implicit Runge-Kutta iterations [50, 5], which can be seen as polynomial smoothing applied to the SGS-preconditioned nonlinear operator:

$$\boldsymbol{w}^{(1)} = \boldsymbol{w}^n - \alpha_1 \boldsymbol{M}(\boldsymbol{w}_{(0)})^{-1}\,\boldsymbol{r}(\boldsymbol{w}^n) \tag{2.49}$$

$$\boldsymbol{w}^{(2)} = \boldsymbol{w}^n - \alpha_2 \boldsymbol{M}(\boldsymbol{w}_{(1)})^{-1}\,\boldsymbol{r}(\boldsymbol{w}^{(1)}) \tag{2.50}$$

$$\vdots \tag{2.51}$$

$$\boldsymbol{w}^{(s-1)} = \boldsymbol{w}^n - \alpha_{s-1} \boldsymbol{M}(\boldsymbol{w}_{(s-2)})^{-1}\,\boldsymbol{r}(\boldsymbol{w}^{(s-2)}) \tag{2.52}$$

$$\boldsymbol{w}^{n+1} = \boldsymbol{w}^n - \alpha_s \boldsymbol{M}(\boldsymbol{w}_{(s-1)})^{-1}\,\boldsymbol{r}(\boldsymbol{w}^{(s-1)}). \tag{2.53}$$

Here, $\boldsymbol{M}(\boldsymbol{w})$ refers ot the SGS operator constructed from an approximate Jacobian matrix at state $\boldsymbol{w}$. However, it is unclear whether this is advantageous compared to simply using a backward Euler nonlinear smoother with SGS iteration to approximately solve the resulting linear system. There is some evidence that smaller number of Runge-Kutta stages (smaller degree of the polynomial) yields faster convergence in CPU time [5]. Some optimizations may be possible where the implicit operator $\boldsymbol{M}$ is only updated at the first stage, or only some inexpensive parts are updated every stage. Such implicit Runge-Kutta smoothers have not been explored in this work, though they may be of interest in future.

A few researchers have considered polynomial smoothing based on Chebyshev polynomials [51]. However, they need estimates of at least the maximum eigenvalue. In a nonlinear solver like ours, estimating the eigenvalues at every pseudo-time step is an expensive proposition. In the general non-symmetric case, a separate iterative method must first be used to estimate the eigenvalue(s).

2.6 Krylov subspace methods

A k-dimensional Krylov subspace of $\mathbb{R}^n$ with respect to a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and vector $\boldsymbol{v} \in \mathbb{R}^n$ is defined as

$$\mathcal{K}(\boldsymbol{A}, \boldsymbol{v}) := \text{span}\{\boldsymbol{v}, \boldsymbol{A}\boldsymbol{v}, \boldsymbol{A}^2\boldsymbol{v}, ..., \boldsymbol{A}^{k-1}\boldsymbol{v}\}. \tag{2.54}$$

To solve $\boldsymbol{Ax} = \boldsymbol{b}$, Krylov subspace methods search for a solution in the affine space $\boldsymbol{x}_0 + \mathcal{K}(\boldsymbol{A}, \boldsymbol{r}_0)$, where $\boldsymbol{x}_0$ is an initial guess and $\boldsymbol{r}_0 := \boldsymbol{b} - \boldsymbol{Ax}_0$, such that the computed solution is optimal in some sense. This is accomplished using an orthogonality condition on the residual with respect to another 'test' subspace $\mathcal{L}$. The choice of this latter subspace determines the sense in which the solution is optimal and differentiates the different Krylov subspace methods [34].

Generalized minimum residual (GMRES) [52] is a very popular Krylov subspace method used by the CFD community. In this method, the test space is chosen such that the 2-norm of the residual is minimized in the affine subspace $\boldsymbol{x}_0 + \mathcal{K}$. This space is $\mathcal{L} = \boldsymbol{A}\mathcal{K}$, that is, the space spanned by $\{\boldsymbol{Ax} \,|\, \boldsymbol{x} \in \mathcal{K}(\boldsymbol{A}, \boldsymbol{r}_0)\}$. Another popular Krylov subspace solver is the stabilized bi-conjugate gradient (BiCGStab) method of Van der Vorst [53]. The advantage of this method (and others in the BiCG family) is that it does not require storage for Krylov subspace basis vectors.

Preconditioning, as defined in section 2.3, is essential for good performance of Krylov subspace methods [11]. As we will survey in the next section, parallel preconditioning is still an open question, though several avenues are being investigated by researchers. In this thesis, we investigate one class of techniques for parallel preconditioning, rooted in the theory of asynchronous iterations described in section 2.8.

Nevertheless, we briefly mention the many efforts by different communities to parallelize Krylov accelerators on GPUs and other parallel platforms. For example, preconditioned conjugate gradient (CG) was parallelized on a Cray vector computer in the 1980's [54]. More recently, [55] gives details of implementation of Krylov subspace methods with sparse approximate inverse preconditioners on GPUs.

Yang et al. [56] implemented preconditioned GMRES on multiple GPUs and reported a 20 times speedup over 'traditional CPU'. However, the preconditioner computation was carried out on the CPU. Also, they report GPU occupancy to be about 67% and that it is difficult to properly design sparse matrix vector product (SpMV) kernels. He et al. [57] report a 3 to 12 times speedup for dynamic circuit analysis in case of GMRES on multi-GPUs. In their implementation as well, ILU factorization is done on the CPU. The preconditioner is applied in a level-scheduled manner using CuSparse [58]. They implement a new SpMV kernel to achieve fully coalesced memory access.

Anzt et al. [59] have developed highly optimized kernels for operations in BiCGStab. Dahnavi [55] reports between 5 and 40 times speedup for sparse approximate inverse preconditioned BiCGStab executed on a GPU compared to serial CPU execution for certain kinds of matrices.

However, the large number of global reductions required by Krylov subspace methods may result in bottlenecks to parallel scaling to very large number of processors [60]. Some of the techniques under development by some researchers that aim to mitigate this problem fall under the categories of pipelined methods [61] and communication-avoiding methods [62, 63].

2.7 Current state of parallel linear iterations

Since the advent of vector computers and later, multi-core central processing units (CPUs) (figure 1.1), there have been attempts to employ parallel iterations. In case of implicit schemes too, a few approaches have been developed, some of which are tailored to CFD problems with anisotropic boundary layers. Vectorized line smoothers, as implemented by Mavriplis [64] for example, are quite effective for compressible turbulent flow problems. However, each line is processed sequentially, leading to reduced parallelism.

2.7.1 Stationary iterations and incomplete factorization

Attempts have been made by several authors to parallelize incomplete LU factorization and triangular solves since the 1980s. Some newer work has been done for GPUs and related hardware [40, 12, 65]. These techniques include level-scheduling, multi-colour re-orderings and the asynchronous iterations.

Level-scheduling

Level-scheduling consists of identifying rows of the matrix that can be eliminated simultaneously due to the sparsity of the matrix. Such rows are grouped into 'levels'. On structured grids with cells ordered lexicographically (by i, j, k), this approach gives rise to wavefront algorithms, where planes of cells having the same value of $i + j + k$ are processed in parallel [40]. This method has the advantage of being mathematically equivalent to the corresponding sequential iteration while enabling reasonably good memory access patterns. However, the parallelism available is limited. On a 2D square structured grid with N cells, the maximum parallelism available is $\sqrt{N}$, which is the length (in cells) of the diagonal. For a 3D cubic structured grid with N cells, the maximum number of cells that can be processed in parallel is $\mathcal{O}(N^{2/3})$. Thus even the maximum available parallelism is sub-optimal, and not even this much is available during much of each iteration. At the beginning and end of each iteration, there is barely any parallelism as the initial and final levels typically have very few rows.

Level-scheduled ILU factorization has been attemped on GPUs [12, 40], though Aissa et al.[66] report unsatisfactory speedup in a structured-grid finite volume CFD code. Luo et al. [40] also use a wavefront block-ILU(0) method on GPUs. Their case was an unsteady incompressible RANS simulation on a structured grid using finite volumes. Level-scheduled triangular solves on GPUs have been attempted [67], but it is unclear whether this is significantly better than multi-threaded CPU execution because of the limited parallelism available.

Multi-colouring

A typical way of formulating a parallel ILU algorithm (and parallel triangular solves) is to re-order the unknowns and equations such that in the new ordering, many rows or columns can be eliminated in parallel. This is possible because of the sparsity of the system matrix. One such ordering is the multi-colour ordering [68, 69]. This involves labelling every unknown with a colour such that unknowns labelled with the same colour do not have a direct dependence on each other. The unknowns are then re-ordered such that those assigned the same colour are numbered consecutively, which leads to a block structure with (usually large) diagonal blocks. The advantage of this approach is that the degree of parallelism is proportional to the problem size for common discretizations. When two-coloured ('red-black') Gauss-Seidel is used as a multigrid smoother for the Poisson problem on structured grids, high convergence rates are observed [37, chapter 2]. However, when multi-coloured ILU is used as a preconditioner researchers have observed [68, 69] significant increases in the number of linear solver iterations compared to other non-parallel orderings. Furthermore, a multi-colour ordering may lead to bad memory access patterns, especially on GPUs, due to a non-unit stride [70]. This could hurt performance.

2.7.2 Sparse Approximate Inverses

This class of methods aims to construct an explicit sparse approximate inverse (SAI) $\tilde{\boldsymbol{M}}$ of the system matrix $\boldsymbol{A}$. Thus, the preconditioner application involves only a sparse matrix vector product, for which parallel implementations exist. There are two broad classes of sparse approximate inverse methods - monolithic and factored.

'Monolithic' methods compute a single matrix as the approximate inverse. One of the popular examples is the SpAI algorithm of Grote and Huckle [71]. This is based on the a sequence of Frobenius norm minimizations of the form given below.

$$\|\boldsymbol{A}\tilde{\boldsymbol{M}} - \boldsymbol{I}_n\|_F \tag{2.55}$$

is minimized over all possible $\tilde{\boldsymbol{M}}$ obeying some sparsity constraint. Here, $\boldsymbol{I}_n$ is the $n \times n$ identity matrix. $\|\cdot\|_F$ denotes the Frobenius norm. The use of this norm allows us to decouple the minimization problem for each column of $\tilde{\boldsymbol{M}}$, which enables parallel computation. If $\mathcal{J}$ is the set of indices on which the jth column of $\tilde{\boldsymbol{M}}$ is non-zero, and $\mathcal{I}$ is the set of row-indices of $\boldsymbol{A}$ which have non-zeros in columns having indices $\mathcal{J}$, the minimization problem for the jth column is given by

$$\min_{\boldsymbol{m}_j} \|\boldsymbol{A}(\mathcal{I},\mathcal{J})\tilde{\boldsymbol{m}}_j(\mathcal{J}) - \boldsymbol{e}_j(\mathcal{I})\|_2, \tag{2.56}$$

where $\boldsymbol{e}_j$ is the jth column of the identity matrix, and index sets in parentheses indicate restrictions of the corresponding matrices to those indices. Thus if the number of indices in $\mathcal{I}$ is $n_{\mathcal{I}}$ and that in $\mathcal{J}$ is $n_{\mathcal{J}}$, $\boldsymbol{A}(\mathcal{I},\mathcal{J})$ is the $n_{\mathcal{I}} \times n_{\mathcal{J}}$ matrix containing those entries of $\boldsymbol{A}$ corresponding to the rows in $\mathcal{I}$ and the columns in $\mathcal{J}$. $\mathcal{I}$, which depends on $\mathcal{J}$, is sometimes referred to as the 'shadow' of $\mathcal{J}$ with respect to the matrix $\boldsymbol{A}$. $\mathcal{I}$ consists of the indices of all those rows which have a non-zero in any of the columns in $\mathcal{J}$.

Typically, the matrix $\tilde{\boldsymbol{M}}$ is used as the preconditioner in a Krylov subspace solver or in a multigrid smoother. Solution of the equations (2.56) requires n small $n_{\mathcal{I}} \times n_{\mathcal{J}}$ dense least-squares solves. The size of these small problems depends on the sparsity pattern chosen, which can be static (pre-computed and fixed) or dynamic (where both the SAI and its pattern are iteratively updated). Static patterns can be based on the sparsity pattern of A^k for some small k such as 1, 2 or 3. The SpAI algorithm of Grote and Huckle is an iterative scheme with dynamically updated sparsity pattern. However, dynamic procedures may be complicated to implement for parallel hardware and may have a large set-up time to compute the preconditioner [3].

Typically, computation and application of SAI is more expensive in terms of memory loads and flops than SGS or ILU-type iterations. For an intuitive reason for this, one can go back to the concepts of domain of dependence and range of influence in the fluid domain. A sparse explicit inverse provides only a small range of influence for every cell depending on the sparsity pattern chosen. One may attempt to fix this issue by dynamically computing a sparsity pattern that captures the most important non-zeros and thus the most important signal propagations, but this tends to significantly increase preconditioner computation costs because several solutions of equation (2.55) are now needed for different candidate sparsity patterns. Nevertheless, it may be possible to combine ideas from sparse approximate inverses with other algorithms to generate more efficient parallel iterations, as already done by Anzt et al. [72], for example. This is further discussed in section 5.4.2 in the context of multigrid smoothing.

Other SAI methods that this thesis does not explore are AINV and FSAI. AINV [73] is based on computing $\tilde{\boldsymbol{M}}$ in factored form by inverting $A = LDV$. There is usually no selection of a sparsity pattern; rather, terms are dropped based on a tolerance. Numerical stability issues may arise, especially for very non-symmetric matrices [3]. Further, the bi-conjugation process used for building the approximate inverse is difficult to parallelize. A related implementation for NVidia GPUs in the CUSP library [74] only works for symmetric positive definite matrices.

The factored SAI (FSAI) method is a hybrid method that computes the inverse in factored form but uses Frobenius-norm minimization in the process. It is reported to work well, but only for symmetric positive-definite matrices [11]. The problem of sparsity pattern determination mentioned for monolithic methods holds for FSAI as well.

Sparse approximate inverse smoothers have been used with geometric multigrid [75]; while their simpler fixed-pattern variants are highly parallel, they did not show good performance for convection-dominated linear problems, and have not been demonstrated for compressible turbulent flow problems. In fact, to our knowledge, none of the SAI-based methods have been applied to compressible turbulent flows.

2.7.3 Domain Decomposition

Domain decomposition is a class of iterative techniques in the spirit of divide-and-conquer. For our purpose, we consider overlapping Schwarz methods, such as additive and multiplicative Schwarz [13]. In these methods, the grid is divided into subdomains which may overlap. The full problem $\boldsymbol{Ax} = \boldsymbol{b}$ is restricted, using some restriction operator, into each of the subdomains. A subdomain (local) solver is applied to each restricted sub-problem to compute an approximate update. These subdomain updates are used to compute a global update to the original problem.

Additive Schwarz iterations are very natural for distributing a large problem among multiple processors. The problem is divided up (possibly with some overlap) and each subdomain is assigned to a processor. The processors solve their subdomains concurrently using some subdomain iteration and the components of the iterates corresponding to overlap regions are exchanged via message passing between domain decomposition iterations. If there are p subdomains, the additive Schwarz preconditioning step can be stated as

$$\boldsymbol{z} = \sum_{i=1}^{p} \boldsymbol{P}_i \boldsymbol{M}_i^{-1} \boldsymbol{R}_i \boldsymbol{r}, \tag{2.57}$$

where $\boldsymbol{R}_i$ is the restriction matrix that acts on the global residual vector $\boldsymbol{r}$ to return its restriction to the ith subdomain. $\boldsymbol{P}_i$ is a prolongation matrix that computes the contribution of the local update to the global update vector $\boldsymbol{z}$. $\boldsymbol{M}_i$ is the preconditioning matrix corresponding to some iteration on the ith subdomain.

In the simplest case with no overlap, the iteration is an approximate subdomain-block-Jacobi. In this case each $\boldsymbol{R}_i$ simply selects values from the global vector that correspond to the ith subdomain and returns them in a vector of appropriate length. The prolongation operator simply plugs the local update values into the corresponding indices of the global update vector. The local update is computed using an approximate solve represented by $\boldsymbol{M}_i^{-1}$. Adding an overlap layer usually leads to some improvement in convergence, at the cost of some redundant computations in the overlap regions.

Such methods as additive Schwarz are often used as preconditioners for Krylov subspace methods (eg. [76]). However, the quality of one-level additive Schwarz preconditioners deteriorates as the number of subdomains increases, which means an increasing number of outer Krylov iterations are needed as more subdomains are added [77, 3]. To improve performance, a global coarse space may be added as one of the subdomains [78], and solved in parallel along with the rest of the subdomains. However, in this thesis we are primarily interested in using domain decomposition for the global smoother in multigrid. Since we already have a hierarchy of coarser spaces which are used in a multiplicative cycle, the scalability issues of subdomain block Jacobi and additive Schwarz should be mitigated. ('Multiplicative cycle' implies that the grids are used one at a time, with the output of one forming the input of the next.) In the FANSC-Lite code, the workhorse global smoother for FAS multigrid is subdomain block Jacobi, used on a CPU cluster. Since the focus of the thesis is on fine-grain parallel iterations within a node, we consistently use subdomain block Jacobi as the global smoother for simplicity.

In the literature, additive Schwarz methods are used for coarse-grain parallelism at the level of nodes in a cluster [79, 80]. Baker et al. [81] used subdomain-block Jacobi smoothers with sequential subdomain iterations on clusters made up of shared-memory multi-core nodes. Their work is notable in having used hybrid parallelism involving both the Message Passing Interface (MPI) and OpenMP [82] threading. However, application to fine-grain parallelism, such as on GPUs or SIMD units of CPUs, is practically non-existent. One reason for this is perhaps the redundant computation needed at subdomain boundaries, depending on the overlap, which can get excessive at the high levels of parallelism needed on GPUs. We do not undertake an investigation of additive Schwarz methods for fine-grain parallelism; instead we attempt to derive fine-grain parallel variants of SGS and ILU methods to use as subdomain iterations. We use subdomain block Jacobi only for coarse-grained parallelism at the level of whole compute nodes in a cluster, and sometimes at the core level in multi-core CPUs for comparison with fine-grain parallel iterations.

2.8 Asynchronous fixed-point iterations

The basis of the work presented in this thesis is a relatively old concept that has seen some resurgence recently. In their 1969 paper, Chazan and Miranker [83] gave a theory of 'chaotic relaxation'. Their iteration scheme can eliminate synchronization and waiting when a linear iteration is executed by multiple processors. Consider a linear algebraic system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ in n equations and n variables, and a splitting $\boldsymbol{A} = \boldsymbol{M} + \boldsymbol{N}$ defining the relaxation

$$\boldsymbol{x}^{j+1} = \boldsymbol{M}^{-1}\boldsymbol{b} - \boldsymbol{M}^{-1}\boldsymbol{N}\boldsymbol{x}^{j}. \tag{2.58}$$

Let $\boldsymbol{B} := -\boldsymbol{M}^{-1}\boldsymbol{N}$ and $\boldsymbol{C} := \boldsymbol{M}^{-1}$ with rows $\boldsymbol{c}_i^T$. In chaotic relaxation, one thinks about linear solvers not in terms of iterations over the entire solution vector, but in terms of a set of 'steps' where each step reads the solution vector $\boldsymbol{x}$ once. In this framework, only one solution vector is stored in memory, not two ($\boldsymbol{x}^j$ and $\boldsymbol{x}^{j+1}$) as in Jacobi-type iterations. Chazan and Miranker define a chaotic relaxation as the following fixed-point iteration scheme.

$$x_i^{j+1} = \begin{cases} x_i^j & i \neq u(j) \\ \sum_{\alpha=1}^n B_{i\alpha} x_\alpha^{j-s_\alpha(j)} + \boldsymbol{c}_i^T \boldsymbol{b} & i = u(j) \end{cases}. \tag{2.59}$$

As an example, Chazan and Miranker frequently used $\boldsymbol{B} := \boldsymbol{I} - \omega \boldsymbol{D}^{-1}\boldsymbol{A}$ and $\boldsymbol{C} := \omega \boldsymbol{D}^{-1}$, thus deriving a chaotic relaxation from the weighted Jacobi iteration. Following Strikwerda [84], we refer to $s_\alpha : \mathbb{N} \to \mathbb{N}$, $\alpha \in \{1, 2, ..., n\}$ as the 'shift' or 'delay' functions and $u : \mathbb{N} \to \{1, 2, ..., n\}$ as the 'update function'. Note that j here does not refer to the jth iterate as in (2.58), but a 'step' at which, in the original formulation, only *one* of the components of $\boldsymbol{x}$ is updated. The following conditions are imposed:

- The shifts are bounded above uniformly:

$$\exists\, \hat{s} \in \mathbb{N} \text{ s.t. } 0 \leq s_i(j) \leq \min\{j-1, \hat{s}\} \quad \forall i \in \{1, 2, ...n\},\ j \in \mathbb{N}. \tag{2.60}$$

- There is no step in the iteration beyond which one of the components of $\boldsymbol{x}$ stops getting updated, which we state precisely as follows.

$$\text{Given } i \in \{1, 2, ..., n\} \text{ and } j \in \mathbb{N}, \quad \exists\, l > j \text{ s.t. } u(l) = i. \tag{2.61}$$

In such a case the chaotic scheme is identified by the tuple $(\boldsymbol{B}, \boldsymbol{C}, \mathcal{S})$ where

$$\mathcal{S} := \{s_1, s_2, ..., s_n, u\}.$$

The main result proved by Chazan and Miranker [83] is the following very general theorem. We state it below for completeness. Note that $\boldsymbol{v} > \boldsymbol{w}$ for two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ of the same size means that each component of $\boldsymbol{v}$ is greater than the corresponding component of $\boldsymbol{w}$, $|\boldsymbol{M}|$ is the matrix of the same size as $\boldsymbol{M}$ but having as its entries the absolute values of the corresponding entries of $\boldsymbol{M}$, and $\rho(\boldsymbol{M})$ denotes the spectral radius of matrix $\boldsymbol{M}$.

Theorem 1. *(a) The scheme $(\boldsymbol{B}, \boldsymbol{C}, \mathcal{S})$ converges if $\exists\, \boldsymbol{v} \in \mathbb{R}^n$ and $\alpha < 1$ such that $\boldsymbol{v} > \boldsymbol{0}$ and $|\boldsymbol{B}|\boldsymbol{v} \leq \alpha \boldsymbol{v}$. (b) This happens if $\rho(|\boldsymbol{B}|) < 1$. (c) If no such $\boldsymbol{v}$ exists, there exists a sequence $\mathcal{S}_0$ depending on $\boldsymbol{B}$ such that the scheme $(\boldsymbol{B}, \boldsymbol{C}, \mathcal{S}_0)$ does not converge.*

These ideas have been extended to nonlinear iterations by Baudet [85] and Frommer and Szyld [86]. These authors use the more modern term 'asynchronous iterations'.

$$\boldsymbol{x} = \boldsymbol{F}(\boldsymbol{x}), \tag{2.62}$$

where $\boldsymbol{F} : D_F \subseteq \mathbb{R}^N \to \mathbb{R}^N$, then the update at step j can be expressed as

$$x_k^{j+1} = \begin{cases} F_k(\tilde{\boldsymbol{x}}), & k = u(j) \\ x_k^j, & k \neq u(j) \end{cases}, \tag{2.63}$$

where $j : \mathbb{N} \to \{1, 2, ..., N\}$. The data $\tilde{\boldsymbol{x}}$ used to compute the update at step j is, again, given by shift functions s. For each step j, the shift function returns an N-tuple $s(j)$ of indices. In this N-tuple, the kth element determines the step from which the kth entry of the vector $\boldsymbol{x}$ is taken to compute the updates for step j. This can be expressed as

$$x_k^{j+1} = \begin{cases} F_k(x_1^{j-s_1(j)}, x_2^{j-s_2(j)}, ..., x_N^{j-s_N(j)}), & k = u(j) \\ x_k^j, & k \neq u(j) \end{cases}. \tag{2.64}$$

Note that subscripts denote entries in vectors and superscripts denote steps. Under these assumptions, the following theorem is stated by Frommer and Szyld [86, theorem 4.4], proved by El Tarazi [87].

Theorem 2. *Let $\boldsymbol{x}^*$ be the unique solution of* (2.62)*. Assume that $\boldsymbol{x}^*$ lies in the interior of D_F and that $\boldsymbol{F}$ is Fréchet differentiable at $\boldsymbol{x}^*$. If*

$$\rho(|\boldsymbol{F}'(\boldsymbol{x}^*)|) < 1, \tag{2.65}$$

then there exists a neighbourhood $\mathcal{N}$ of $\boldsymbol{x}^*$ such that the asynchronous iteration (2.64) *starting at $\boldsymbol{x}_0 \in \mathcal{N}$ converges to $\boldsymbol{x}^*$.*

We generally do not explicitly prescribe the shift and update functions in practice, but rather let them be governed by the hardware and the non-zero pattern of the matrix $\boldsymbol{A}$. The shift and update functions could vary depending on the number of processing elements, memory access latency from different processing elements, cache hierarchy, scheduling etc. as well as the load imbalance among the updates brought about by the sparsity pattern, resulting in chaotic behaviour. The nature of the iteration could range between Jacobi and Gauss-Seidel. If shift functions are such that progressively older components are used for updates until all components are updated once, the iteration becomes Jacobi, while if the shift functions are always zero, the iteration becomes Gauss-Seidel. In practice, it would be somewhere in between. This framework allows a situation in which, for the update of any component, the latest available values of all other components are used. The other components may or may not have been updated yet in the 'current iteration'. This is useful when multiple processors are available. Asynchronous iterations are suited to massively parallel processing not only because work-items are independent, but also because load imbalance among work items is less of a problem. As soon as one update is completed by a processing element, it can immediately request the next work-item.

It should be noted that even though we do not explicitly specify $\mathcal{S}$, we can influence it in several ways. One of the ways used throughout this work is to order the parallel loop in a specific manner. If all the loop iterations (work items) were actually executed in parallel, the ordering would not matter. But in general, the number of work items is much greater than the number of available processing elements and the (partial) order of execution of the work items can be influenced by the ordering of the loop. $\mathcal{S}$ can also be influenced by other features of the programming model, such as thread scheduling and chunk sizes in OpenMP and the size of thread-blocks in CUDA, for example.

Even though asynchronous iterations are defined in terms of 'steps' rather than 'sweeps' or 'iterations', in this work, we will use the term 'iteration' of the asynchronous process to mean an update of all the entries of the solution vector. This is so that we can more naturally compare with traditional relaxations such as Gauss-Seidel.

The potential for utility of asynchronous methods is high in CFD. Firstly, as the number of threads required to achieve good performance on modern chips increases, fine-grained parallel algorithms will be required to replace coarse-grained parallel algorithms. Asynchronous iterations such as the one proposed in this work are fine-grained parallel. Second, as we move to massive parallelism, synchronization between processes or threads will get more and more expensive. Asynchronous methods are better in this regard. Thus it is of great interest to see whether these methods can be adapted for use in CFD for implicit solvers. There have been some efforts to apply asynchronous iterations to the solution of scalar partial differential equations (PDEs) including nonlinear ones. In the 1980's, Anwar and El Tarazi used a non-linear asynchronous iteration to solve a Poisson problem with non-linear boundary conditions [88]. In fact, chaotic relaxation has been used to solve some incompressible flow problems in marine engineering [60]. For those problems, it was found to be more scalable in parallel than Krylov subspace solvers. The mean flow equations for momentum and pressure were solved in a segregated manner using a pressure-based method.

In this work, 'asynchronous iterations' refers to the general broad class of methods described in this section. We refer to Chazan and Miranker's [83] original example, given by $\boldsymbol{B} := \boldsymbol{I} - \omega \boldsymbol{D}^{-1}\boldsymbol{A}$ in equation (2.59), by their chosen term 'chaotic relaxation'.

2.8.1 Asynchronous ILU of Chow and Patel

Chow and Patel [65] proposed a highly parallel ILU factorization algorithm based on comparing the left and right sides of the equation $(LU)_{ij} = A_{ij}$. These lead to a fixed point iteration of the form $\boldsymbol{x}^{n+1} = \boldsymbol{g}(\boldsymbol{x}^n)$ where $\boldsymbol{x}$ is the vector of unknowns consisting of the L_{ij}s and U_{ij}s. This is solved by parallel iterations, shown in algorithm 3. As an initial guess, one can use the entries of A, among other options. For later matrices in a nonlinear solution sequence, the approximate LU factorization from the previous nonlinear iteration can be used as the initial guess.

Algorithm 3 Asynchronous ILU factorization

```
Require: Assign initial values to $L_{ij}$ and $U_{ij}$. Let $S$ be the desired nonzero index-set.
1: for $i_{swp}$ in $1..n_{swp}$ do
2:     for $(i,j) \in S$ do in parallel:
3:         if $i > j$ then
4:             $L_{ij} \leftarrow \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik}U_{kj}\right)/U_{jj}$
5:         else
6:             $U_{ij} \leftarrow A_{ij} - \sum_{k=1}^{i-1} L_{ik}U_{kj}$
7:         end if
8:     end for
9: end for
```

Since the equations for each (i, j) in the sparsity pattern may involve a different amount of computation, the load is inherently unbalanced. To resolve this, the algorithm is executed as an asynchronous parallel iteration [83, 86]. Depending on the local imbalance and number of processors, the behaviour could range from nonlinear Jacobi to nonlinear Gauss-Seidel. This asynchronous fixed-point iteration provably converges irrespective of the initial guess because of the structure of $\boldsymbol{g}$ [65]. The algorithm was also implemented on GPUs [89] taking advantage of low-level thread-block ordering and cache properties of a particular GPU device. For symmetric positive-definite matrices solved using conjugate gradients, Chow et al. observed between 2x and 30x improvement over level-scheduled ILU(0) in NVIDIA's cuSparse library.

Chow and Patel demonstrated the performance of their method on scalar elliptic problems, including the Poisson equation and a convection-diffusion equation. They also used some symmetric positive-definite (SPD) matrices from the University of Florida sparse matrix collection [90]. It is not clear whether the technique can be used for more complex problems, such as systems of tightly-coupled equations having hyperbolic and elliptic character in different parts of the domain, as is the case with the compressible Navier-Stokes equations we are interested in. This is explored in detail in chapter 4.

Once the L and U factors are computed, they need to be applied, possibly several times. The traditional exact ways of accomplishing this, forward- and back-substitution, have inherent dependencies that generally prevent fine-grain parallelism. Multi-colouring and level-scheduling can be employed to alleviate this issue to some extent. However, it has been argued [91] that since the factors are themselves approximate in the case of ILU preconditioners, it is unnecessary to solve the factors exactly. Thus we want to solve the LU factors approximately, but using an algorithm that enables fine-grained parallelism. Anzt et al. [91] favour an asynchronous Jacobi-type iterative process to apply ILU preconditioners. A similar technique can used for triangular solves in Gauss-Seidel preconditioning as well. Some approximate inverse techniques have also been explored [92, 72] and are claimed to be better than level-scheduled triangular solves. Some of these techniques are described and their effectiveness tested in chapters 4, 5 and 6.

2.9 Implementation and performance on parallel hardware

In this section, we describe hardware-related concerns associated with implementation of iterative methods.

2.9.1 Multi-core CPU architecture

Today, multi-core CPUs are the most popular hardware for CFD codes, and indeed, most scientific computing codes. It is common for multi-core CPUs to have had 8 to 16 cores per chip (eg. Intel Sandy Bridge CPUs, figure 1.2), and each node in a cluster has two such chips in two sockets on the system motherboard. CPU cores are optimized for latency and complex logic. Performance is traditionally attained by complex circuits implementing a hierarchy of caches of different sizes and access latency, as well as elaborate features such as branch prediction. Because the number of cores per chip is typically small, it is sufficient to treat each core as a separate processor and thus treat on-node parallelism in the same way as cluster parallelism - using distributed computing via the Message Passing Interface (MPI). Every core is associated with one process, and each process has an associated partition of system memory that other processes cannot directly access. This is commonly used with domain decomposition solvers. Some researchers have exploited on-node multi-core parallelism separately using multiple operating system threads within each MPI process [93, 64, 81], though the benefits were questionable. OpenMP [82] is a framework that enables multi-threaded programming without too many changes to a program made up of tasks that can execute in parallel. Unlike MPI processes, threads represent shared memory parallel computing. This means different threads within a process see the same view of memory and thus require no explicit communication.

An important feature of CPUs, and indeed most processors, is caching of data loaded from system memory. Data from memory is not read into cache one address at a time; rather, a certain number (typically 64 bytes on Intel CPUs) of consecutive locations are read even when only one is requested. This whole *cache line* of data is stored in the cache. Thus, if a (serial) algorithm successively requires memory accesses close to one another, the number of fetches from system memory is reduced and performance improves significantly. Such a memory access pattern is said to be cache-friendly. In multi-threaded execution, cache-friendly memory access by each thread is essential.

However, since the past several years, CPU hardware has evolved. This is exemplified by the second generation Xeon Phi processors from Intel, called Knights Landing (KNL) [2]. Among the many interesting features of this device, we mention four here. The number of cores on the chip is between 64 and 72, there are two 512-bit vector processing units in each core, 4-way simultaneous multi-threading (SMT) is available in each core, and there is a large amount of high-bandwidth on-chip memory. Simultaneous multi-threading, or SMT, refers to the capability of each processing element (core, in this case) to store the states of multiple independent tasks at the same time. This enables quick switching from one task to the other and hides delays arising from waiting tasks. The large number of cores per node and SMT increase the degree of parallelism needed to approach peak performance. They also raise the possibility of gaining performance by using shared-memory multi-threading. Vector processing units perform data parallel execution under the Single Instruction Multiple Data (SIMD) paradigm, which means that a program instruction can be applied to several data entries concurrently under certain conditions. While SIMD units have been available for many years, they have been quite short, usually 128 bits. These were meant primarily for multimedia applications. The vector width was progressively increased to 256 bits and the units were made more flexible in the Haswell and Broadwell generations. With the KNL's 512-bit wide vector units, we are at a point where fine-grain parallelism is essential. Newer Intel Xeon CPUs from the Skylake and Cascade Lake generations retain many of the properties of the KNL.

2.9.2 GPU architecture

Graphics processing units (GPUs) are optimized for high throughput using a large number of small lightweight processing elements called stream processors [1]. When GPUs are available on a node, they are present in addition to the CPU and mounted as a separate card. Communication between the host CPU and the GPU device usually takes place over a standardized interface called PCI-Express, though there are vendor-specific solutions as well. Any part of the program meant for offloading to the GPU is written as a kernel, a set of commands to be executed by one processing element; eg., performing a Jacobi update on one cell in the grid.

The salient features that are relevant to us are the hierarchy of stream processors and the memory hierarchy. We mainly describe NVIDIA GPUs, though the fundamental concerns remain the same for GPUs from Advanced Micro Devices (AMD) as well. Each work item is packaged into a GPU 'thread' for execution on one stream processor. Stream processors are organized into 'streaming multiprocessors' (SMs), as can be seen in figure 1.4. Each SM dispatches a group of threads at a time to its stream processors. Such a group of threads is referred to, in NVIDIA parlance, as a 'warp'. On most NVIDIA GPUs, a warp consists of 32 threads. Each warp normally executes on an SM independently of other warps, though threads within a warp can be made to synchronize. Threads within a warp execute in Single Instruction Multiple Thread fashion which is similar in some ways to SIMD parallelism on vector CPUs. It means that threads in a warp normally execute the same instruction in lockstep on different data entries. Threads are, however, allowed to 'diverge' by executing different instructions because of the presence of branching in the kernel code. However, divergence is to be minimized because all the threads in a warp must take both paths sequentially if even one thread diverges from the common path of the others. Finally, a GPU handles latency by SMT, which means that the states of several warps can be held by the SM at any given time. When one warp stalls for any reason such as a fetch from memory, another warp can immediately take over at practically no cost [1].

On the memory side, each SM has some very fast local shared memory that is shared by threads executing on it. This shared memory can be controlled by the application programmer. There is also a fast texture cache. Finally, the whole GPU has device memory or global memory, also referred to as VRAM (video random access memory) or the frame buffer. Any data to be processed on the GPU must first be transferred to its global memory, from where it can be accessed by the SMs. Transfers between system RAM and GPU global memory must be minimized, as the PCI Express interconnect has much lower bandwidth and higher latency than memory reads and writes. The GPU global memory is typically much faster than system RAM, though it is slower than shared memory in the SMs. The key point to note about memory access by the GPU is that it attempts to coalesce memory accesses made at the same time by threads in a warp into few memory accesses. However, this can be done effectively only when consecutive threads in a warp access consecutive locations in memory [94]. When this is not the case, performance is expected to suffer drastically.

2.9.3 Memory layout of linear algebra objects

For numerical computations, the important factors for achieving high performance are memory bandwidth and memory access latency. These are heavily dependent on the storage layout of the required data in memory.

We discussed in section 2.4 that the mathematical ordering of the grid cells and physical variables affects the convergence of the solver. We now turn to a related but orthogonal issue - the layout of storage of vectors and matrices in computer memory. This is captured by the choice between so-called 'structure-of-arrays' (SoA) layout and the 'array-of-structures' (AoS) layout. This is mainly relevant for systems of PDEs having, suppose, b physical unknowns and coupled PDEs.

Storage of vectors

We noted in section 2.4 that variables can be logically ordered by physical variables first into small blocks, which are in turn ordered by grid cells ('point blocking'), or this can be reversed ('physics blocking'). The same choice holds for memory layout as well.

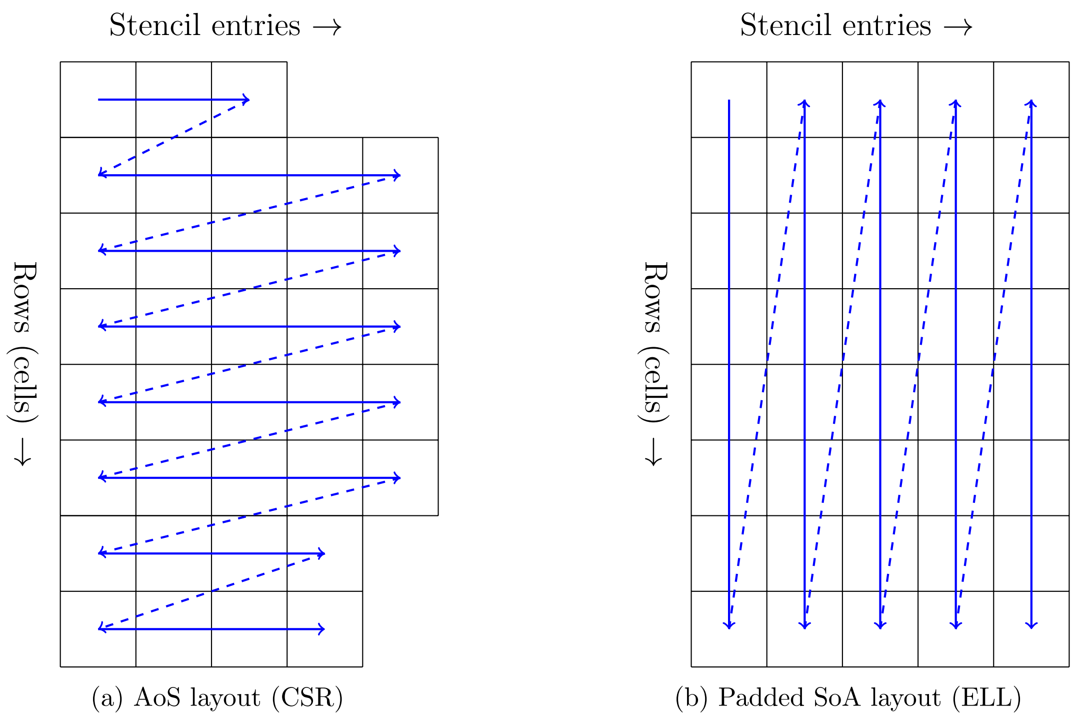
Irrespective of the logical ordering, one can order the storage of flow variables at a grid cell consecutively, followed by those at the next grid cell and so on in a point-block layout. In programming, a structure is a collection of software objects. If we consider a structure object to be the collection of b values at a grid cell, an array of these objects can be used to store a vector, such as $\boldsymbol{w}$ in equation (2.34). Note that by 'array' we mean a set of objects stored contiguously in memory. This kind of storage is referred to as an array of structures.

On the other hand, one can order the density variables of all the cells first, followed by x-momentum of all the cells, and so on. We can have b different arrays containing the values of the respective physical variables at all the grid cells, and a structure containing those b arrays can represent a vector. Such a storage is referred to as having a structure-of-arrays (SoA) layout. Note that either layout can support either point-block solvers or physics-block solvers - the memory layout is a concern orthogonal to the logical or mathematical numbering. This is very useful because one can possibly select a solver well-suited for the problem and a memory layout well-suited for the hardware.

Storage of sparse matrices

The question of storage of sparse matrices is more involved. Different sparse matrix storage formats use different storage layouts. For general matrix computations, the most common format is the compressed sparse row (CSR) storage format. It is organized as a block of memory, which we will call `val`, where non-zero entries of rows are stored contiguously, one row after another. Another block of memory `colind`, with the exact same layout, holds integers denoting the column index of the corresponding non-zero entry in `val`. A separate integer array `rowptr` holds pointers into the blocks denoting the beginning of each row. In the context of matrices arising from stencil operations on grids, this can be thought of as an AoS storage, where the (variable) structure represents the matrix entries in the stencil of each grid cell. A different format is the Ellpack-Itpack format [95], abbreviated to ELL. This format can be described as a zero-padded SoA version of the CSR format. The first non-zero entry of every row is stored contiguously first irrespective of the column index, followed by the second non-zero entry of all rows, and so on. Whenever a row runs out of non-zero entries before other rows, an explicit zero is stored. Column indices are stored using the same layout. No row pointers are needed because the block of memory is logically rectangular, with the width determined by the longest row. The layout of CSR is contrasted with that of ELL in figure 2.3 for a hypothetical scalar PDE (for every cell, there is only one coefficient per neighbouring cell) on a grid with at most four neighbours for every cell. For structured grids, we can use a simplification of the ELL format in which storage of column indices is omitted because they can be inferred from the ijk structure of the grid.

In case of systems of PDEs, a decision needs to be made about how the matrix entries related to the different physical variables are stored. One could store a single matrix with a point-block layout in which the all b^2 matrix entries corresponding to one grid cell adjacency are stored consecutively. In any $b \times b$ block, if even one entry is non-zero, all entries are stored using explicit zeros. This enables us to store just one pointer array (such as `rowptr`) and one index array (such as `colind`) of the same length as for a scalar PDE, even though the matrix is now b^2 times larger. On the other hand, one could store b^2 different coefficient arrays for each of the physical interactions. Again, if each of these matrices is constrained to



(a) AoS layout (CSR) (b) Padded SoA layout (ELL)

Figure 2.3: Two possible layouts of the non-zero coefficients' array of a matrix

the largest required non-zero pattern, separate pointer and indexing arrays are not needed. For our compressible flow solvers, the sparsity patterns are the same for each of the physical interactions, so this does not lead to storage of extra non-zeros. Thus, either storing a flat sparse matrix ignoring the point-block structure, or storing b^2 separate sparse matrix objects leads to unneeded memory bandwidth pressure created by the redundant integer arrays.

2.9.4 Memory access criteria

Different hardware architectures require certain memory access patterns to work efficiently. As discussed above, GPUs require *coalesced* memory access, which means that consecutive work items in a group called a warp, should access consecutive memory locations.

Traditional CPUs require cache-friendly memory access - a key factor in achieving good performance is fetching more useful data in each cache-line fetched. At the multi-core level of parallelism, each thread, over consecutive instruction executions, should ideally access memory locations close to each other. However, this consideration for vector-level parallelism is slightly different. A cache-line (or cache block) is the size of the smallest chunk of memory copied to a cache when memory is read. When the data required by a number of work items scheduled on adjacent SIMD lanes exists in the same cache-line, efficient vectorization is possible. Saule et al. observed [96] that the performance of AVX-512 vectorization on an early Intel Xeon Phi depends significantly on 'ULCD': useful cache-line density, which can be defined as the number of memory locations required by a work item divided by the number of memory locations actually fetched to cache in the process of accessing all of them. A vectorized gather instruction, available in AVX 2.0 and AVX-512 vector instruction sets, can only load those values that are present in the same cache-line. For example, in sparse matrix vector product (SpMV) kernels $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, much higher performance is observed for matrices having non-zeroes close enough to fetch more of the required components from $\boldsymbol{x}$ in a cache-line. In case of AVX 2.0, the presence of at least four required (double-precision) entries of $\boldsymbol{x}$ in the same cache line would lead to full utilization of a vector unit after just one gather instruction. Thus, at the SIMD level of parallelism, consecutive processing elements ideally access consecutive memory locations. This phenomenon is very similar to coalesced memory access important for GPUs.

It has been noted that the consideration of logical ordering is orthogonal to that of memory layout. However, there is also a third decision to be made: the order of storage in memory. In the SoA layout, one still needs to decide the order of entries in the individual arrays; in AoS layout, one needs to decide the order of the structure objects that make up the array. In both cases, what really needs to be decided is the ordering of grid cells in memory. One could choose to use a different ordering compared to the one selected for the mathematical ordering, but this would lead to a bad memory access pattern. For example, consider a structured grid. If we want to perform ILU factorization in wavefront ordering but the matrix is stored in lexicographic (ijk) ordering, memory access will need to jump around in memory to visit the rows and columns in wavefront ordering. Accesses will not be cache-friendly and cannot be coalesced. To avoid this issue, the mathematical ordering of the grid cells typically dictates their ordering in memory for all vectors and matrices. However, this also means that the mathematical ordering now has to address two issues - the convergence properties of the iteration, as well as achieving good memory access pattern. There is sometimes a trade-off involved here.

2.9.5 Measuring parallel performance: strong and weak scaling

Two studies commonly used to demonstrate parallel performance are strong and weak scaling. Strong scaling refers to the capability of a solver to run increasingly faster in terms of wall-clock time as we commit more parallel processors for a fixed problem size. Let $t(n,p)$ denote the wall-clock time required to solve a problem of size n using p processing elements. For our purpose, the problem size can be the number of cells in the grid being solved on or (equivalently) the number of entries in the solution vector. Strong scaling efficiency is defined as

$$\mathrm{Eff}_s(n,p,k) = \frac{t(n,p)}{t(n,kp)}\frac{1}{k} \times 100, \tag{2.66}$$

where k is the factor by which we increase the number of available processing elements. Typically, it is seen [64, 97] that when k is small, one observes good strong scaling close to 100%, but as k increases, the efficiency falls off. This is expected because as we use more processing elements for the same problem size, the time spent doing parallel work decreases, but the time spent doing the serial work, no matter how small, remains constant. Thus the proportion of time spent doing serial work goes on increasing, and addition of more parallel processors yields less benefit. To attain perfect strong scaling, there must be no serial part in the algorithm or program. In our results in the following chapters, we show strong scaling graphically with the speedup $\frac{t(n,p)}{t(n,kp)}$ plotted as a function of the number of processing elements kp. A straight 1:1 reference line is also included to see the deviation from the ideal scaling.

The holy grail of scalable computing is weak scaling. This measures the ability of a solver to scale to larger problems when more parallel processing elements are added. Weak scaling efficiency can be defined as

$$\mathrm{Eff}_w(n,p,k) = \frac{t(n,p)}{t(kn,kp)} \times 100. \tag{2.67}$$

An ideal solver is one that takes the same amount of time to compute the solution as we increase both the problem size and the number of processing elements proportionally; it would have $\mathrm{Eff}_w(n,p,k) \approx 100\%$ for a wide range of k. In other words, $t(kn,kp)$ would ideally be a constant independent of k for some fixed n and p.

We call this the holy grail because if a solver has ideal weak scaling, it means not only that it is ideally parallel but also that the cost scales only linearly (in the asymptotic sense) with problem size, denoted as $\mathcal{O}(n)$. Such 'computationally scalable' algorithms are a big challenge in themselves even without considering parallel computing. For non-trivial problems such as solving PDEs to a physically relevant tolerance, only impossibly perfect algorithms would require exactly twice the number of arithmetic operations (and twice the memory space) when the problem size is doubled. As mentioned in section 2.5, multigrid methods approach this property for several kinds of PDEs and discretizations.

As far as ideal parallelism is concerned, weak scaling has weaker requirements compared to strong scaling (thus the name): the cost of the serial part of the work needs to stay constant as the problem size increases. The serial portion of the program need not be zero (as required for ideal strong scaling), but needs to be asymptotically constant with problem size. While perfect satisfaction of this weaker requirement is also impossible in practice, a closer approximation can be attained than what is required for ideal strong scaling.

2.9.6 Programming models

Various programming models are now available to implement suitable algorithms for massively parallel hardware. There is no industry-wide standard as of writing, though progress is being made. The programming models that are open standards and potentially supported by several hardware vendors include OpenCL, SyCL, OpenMP and OpenACC. But the first and most popular framework, CUDA, is supported only on NVIDIA hardware. There are also other non-standardized but open-source parallel programming frameworks such as Kokkos from Sandia National Laboratory and ROCM from Advanced Micro Devices (AMD).

- OpenMP [82] is a directives-based model for expressing parallelism, suitable for programming multi-core CPUs and Intel Xeon Phi in C, C++ and Fortran. Its advantage is that a code which uses OpenMP can run on traditional CPUs as well. OpenMP 4 introduced support for offloading to accelerators, and until recently the Cray compiler suite was the only one that implemented it. However, implementations for GPUs are now being developed by several teams in industry.

- OpenACC [98] is also a directives-based methodology for C, C++ and Fortran, but has GPU computing as the primary focus. Currently, implementations only support offloading to NVidia GPUs. Support for multi-core CPUs is available from one compiler vendor, PGI, but performance in this case is a question.

- CUDA is an NVidia-specific relatively lower-level programming language and API (application programming interface). It is primarily based on C++ and adds non-standard language features for specifying kernel launches, memory spaces etc. It represents an 'explicit' parallel programming model, as opposed to directive-based models. As the first framework to really support GPGPU computing, it has an advantage in terms of tooling and libraries.

- OpenCL is an open standards specifying a C-based language for kernel implementation as well as a C API. OpenCL is also a relatively low-level explicit parallel programming model. It is supported by NVIDIA and AMD on GPUs and by Intel on its CPUs and integrated GPUs.

- SyCL is also an open-standard. It is a pure standard C++ API for explicit parallel programming, being developed as a cross-platform framework for programming GPUs, Field-Programmable Gate Arrays (FPGAs) as well as CPUs.

For the work presented here, we have adopted OpenMP for shared-memory parallelism on CPUs including the Xeon Phi Knights Landing, and CUDA for NVIDIA GPUs. We will delve into details of these frameworks as and when necessary in later chapters.

To conclude this chapter, it is evident from a survey of existing parallel solvers, that there is a need to develop linear solvers to run on fine-grain parallel hardware for CFD problems, where we encounter large non-symmetric matrices. While there have been some efforts in this direction, a majority of the implicit solvers used today in CFD employ either sequential iterations or their derivatives with limited parallelism. Asynchronous iterations may provide a framework needed to develop fine-grain parallel smoothers and preconditioners on modern architectures.

Chapter 3

Asynchronous point-block symmetric Gauss-Seidel smoother in multigrid solvers for multi-core CPUs

We first develop and study a simple asynchronous iteration suitable for parallel multigrid smoothing for systems of equations arising from pseudo-time implicit discretization of the compressible Reynolds-averaged Navier-Stokes (RANS) equations on multi-core central processing units (CPUs). We perform numerical experiments to demonstrate the smoothing property and parallel scalability of the developed algorithm as a smoother in nonlinear and linear multigrid solvers. Several cases of external aerodynamics, differing in computational grid size and flow complexity, are used for the study. We also show the scalability of the proposed smoother on Intel's Xeon Phi Knights Landing many-core processor, and investigate the impact of some features of this processor on the parallel scalability of the asynchronous smoother. The work in this chapter is the subject of a journal article [24].

3.1 Introduction

As discussed in section 2.5, multigrid methods have been observed to be effective as linear or nonlinear solvers for CFD problems [16, 48]. An efficient and effective smoothing iteration is necessary for good convergence of multigrid methods. However, effective smoothers such as lexicographic symmetric Gauss-Seidel tend not to be amenable to fine-grain parallelism due to inherent data-dependence between work-items in each iteration. Their parallel variants have various issues, as noted in section 2.7. Asynchronous iterations, however, have not been widely explored for CFD. In one example [60], chaotic relaxation was used to solve the incompressible Navier-Stokes equation in a pressure-based solver, where the momentum equations are decoupled from the mass equation using a fractional step approach or multiple inner nonlinear iterations. However, such a scalar algorithm is insufficient for the tightly coupled systems of nonlinear partial differential equations arising in compressible flows. In this chapter, we define an asynchronous point-block symmetric Gauss-Seidel (SGS) relaxation. We demonstrate the effectiveness of this method when used as a multigrid smoother for solving the RANS equations on some benchmark cases. To our knowledge, neither 'asynchronous symmetric Gauss-Seidel' nor point-block variants of asynchronous iterations have yet been studied. Here, an asynchronous point-block SGS method is proposed as a thread-parallel iteration for achieving parallel scaling across multiple compute cores within a node.

In the context of multigrid solvers, we need iterations that have a good smoothing property such that they can damp high-frequency components of the error quickly, as well as iterations that can solve the coarse grid problem efficiently to a reasonable tolerance. In this chapter, we show that asynchronous block SGS relaxation can serve as a good smoother in RANS simulations on multi-core CPUs. We are primarily interested in how well the performance *scales* with increasing amount of available on-node parallelism. For this, we consider two important questions: (1) how is the multigrid convergence rate affected as we use increasing numbers of parallel processors, and (2) what is the reduction in wall-clock time that our implementation attains, for a given flow residual tolerance, as we use increasingly more parallel processors? It is mainly the answers to these two questions that are explored experimentally. We show results on a regular multi-core server CPU, followed by a study of the iteration's performance on Intel's many-core CPU, the Xeon Phi Knights Landing (KNL).

3.2 The solver setup

In this section, we provide an overview of the CFD solver used in this work, FANSC-Lite. The code has nonlinear full approximation storage (FAS) multigrid for 2D and 3D problems and linear multigrid capability for 3D problems. These have been described in section 2.5. As mentioned there, on each grid level, linear fixed-point iterations are used to approximately solve equation (2.40) in linear multigrid and equation (2.38) in FAS.

The relaxation factor ω in equation (2.39) is kept between 0.2 and 1.0 and is computed as a function of the relative change in the density and energy in the update $\Delta \boldsymbol{w}$.

In all the results reported in this paper, for linear multigrid, we carry out one multigrid cycle per backward-Euler step, and equal number of pre- and post-smoothing iterations are carried out by sequential or asynchronous point-block SGS iterations. For FAS, we use 1 pre-smoothing and 1 post-smoothing iteration, where each smoothing iteration corresponds to 1 backward Euler step. Smoothing is accomplished by the combination of backward Euler and the linear iterations used within it. However, since we always use one backward Euler step for smoothing operations, any variation in smoothing effectiveness is due to the iterations that solve the backward Euler step (2.38). Hence, in what follows, we refer to the linear iteration as the 'smoother' even in case of FAS.

Parallelism across nodes in the compute cluster is achieved by simple domain decomposition via the Message Passing Interface (MPI). In particular, the global smoother is simply a subdomain-block Jacobi iteration. In this work, PETSc [99] is used as a framework for distributed (MPI-parallel) linear solvers.

The turbulence closure model is solved separately from the mean-flow equations. The Spalart-Allmaras model equation is solved and the eddy viscosity is updated only on the finest multigrid level once before every multigrid cycle. An ADI (alternating direction implicit) iteration [35, section 8.5] is used for the implicit solve of the Spalart-Allmaras equation. In pure MPI runs, it was observed that the ADI solver for the turbulence model took 2 to 5 percent of the overall solve time, depending on the case. Thus, it was decided to investigate fine-grain parallelism for the mean-flow solver first. This work is not concerned with the parallel scaling of the turbulence model solver.

3.2.1 Ordering of the unknowns

It is important to note the order of variables in $\boldsymbol{w}$ and of residuals in $\boldsymbol{r}$. In our implementation, they are ordered such that all conserved variables in one cell are placed consecutively, followed by the conserved variables of the next cell. For example in two dimensions,

$$\boldsymbol{w}^T = [\rho_1, \rho v_{x1}, \rho v_{y1}, \rho v_{z1}, \rho E_1,\ \rho_2, \rho v_{x2}, \rho v_{y2}, \rho v_{z2}, \rho E_2, ...] = [\boldsymbol{u}_1^T\, \boldsymbol{u}_2^T \ldots \boldsymbol{u}_N^T], \tag{3.1}$$

where subscripts denote cell indices. The residuals are ordered in the same way, with the mass, momentum and energy fluxes for a particular cell placed consecutively. This ordering leads to a block structure of the Jacobian matrix with small dense blocks. It is customary in compressible aerodynamics codes (and sometimes in incompressible flow codes) to take advantage of this 'point-block' structure using point-block variants of common relaxations and preconditioners [6, 40]. Such a treatment deals well with the highly coupled equations relating the state variables in compressible flow. This is our main motivation for using point-block variants of asynchronous iterations.

Thus, the mathematical ordering of vectors and matrices corresponds to the point-block ordering discussed in section 2.4. Further, for the work presented in this chapter, the memory layout is also a point-block array-of-structures (AoS) layout, as discussed in section 2.9.3.

3.3 Asynchronous block symmetric Gauss-Seidel relaxation

The scalar chaotic relaxation algorithm (equation (2.59)) is not suitable for the equations of compressible flow. We define a point-block variant instead, where the density, momenta and energy at a given point (cell-centre) are updated simultaneously and the diagonal blocks are inverted exactly. Note that for our problems, a block will be a point-block formed by the coupling between different physical variables at a given point, as explained in section 3.2.1. For a 3D compressible flow simulation, the blocks are 5×5 (not considering turbulence model equations). In what follows, we assume matrix $\boldsymbol{A}$ has a splitting $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{E} + \boldsymbol{F}$ where $\boldsymbol{D}$ is block-diagonal with non-singular blocks, $\boldsymbol{E}$ is strictly block lower-triangular and $\boldsymbol{F}$ is strictly block upper-triangular. The question of how such a point-block relaxation fits in the definition of asynchronous iterations is addressed in section 4.4.

Algorithm 4 Chaotic block relaxation for $\boldsymbol{Ax} = \boldsymbol{b}$

```
Require: $\boldsymbol{x}$ is an initial guess for the solution, $n_{iter}$ is the number of asynchronous iterations
1: Launch parallel threads in thread-redundant mode
2: for integer $i_{iter}$ in {1,2,... $n_{iter}$} do
3:     for integer $i$ from 1 to $n$ do in parallel dynamically:
4:         $\boldsymbol{x}_i \leftarrow \boldsymbol{D}_{ii}^{-1}(\boldsymbol{b}_i - \sum_{j=1}^{i-1} \boldsymbol{E}_{ij}\boldsymbol{x}_j - \sum_{j=i+1}^{n} \boldsymbol{F}_{ij}\boldsymbol{x}_j)$
5:     end for(No thread synchronization)
6: end for
7: End parallel region
```

Here, each $\boldsymbol{D}_{ii}$ is inverted exactly using dense Gaussian elimination. Notice how we begin the parallel region before the outer iterations start, and do not synchronize at the end of the inner loop, in order to achieve asynchronous iteration. In OpenMP [82], we separate the `parallel` and `for` pragmas and use the `nowait` clause to achieve this.

Depending on the ability to impose a partial ordering on the update of components, we can define the following 'asynchronous block symmetric Gauss-Seidel' (ABSGS) relaxation. Because of the need to propagate disturbances quickly in all directions in subsonic regions of flow, a symmetric Gauss-Seidel approach would be preferable to the regular chaotic relaxation, which approximates a forward Gauss-Seidel approach.

Algorithm 5 Asynchronous block-SGS relaxation for $\boldsymbol{Ax} = \boldsymbol{b}$

```
Require: $\boldsymbol{x}$ is an initial guess for the solution, $n_{iter}$ is the number of asynchronous iterations
 1: Launch parallel threads in thread-redundant mode
 2: for integer $i_{iter}$ in $\{1,2,... n_{iter}\}$ do
 3:     for integer $i$ from 1 to $n$ do in parallel dynamically:
 4:         $\boldsymbol{x}_i \leftarrow \boldsymbol{D}_{ii}^{-1}(\boldsymbol{b}_i - \sum_{j=1}^{i-1} \boldsymbol{E}_{ij}\boldsymbol{x}_j - \sum_{j=i+1}^{n} \boldsymbol{F}_{ij}\boldsymbol{x}_j)$
 5:     end for(No thread synchronization)
 6:     for integer $i$ from $n$ to 1 backward do in parallel dynamically:
 7:         $\boldsymbol{x}_i \leftarrow \boldsymbol{D}_{ii}^{-1}(\boldsymbol{b}_i - \sum_{j=1}^{i-1} \boldsymbol{E}_{ij}\boldsymbol{x}_j - \sum_{j=i+1}^{n} \boldsymbol{F}_{ij}\boldsymbol{x}_j)$
 8:     end for(No thread synchronization)
 9: end for
10: End parallel region
```

Here, we depend on OpenMP's property of preserving the ordering of the work-items within a 'chunk', the set of work-items assigned to a thread at a time ([82, section 2.7.1]).

While we only use asynchronous iterations in a shared-memory framework for achieving parallelism within a compute node, there may still be communications to consider. Even though we intend for the iteration to be completely asynchronous, depending on the hardware, the implementation may be forced to wait for updates of components. The components of the solution vector updated by one core are initially only updated in its local cache. Depending on the cache hardware setup, it is possible that when another core tries to read this value to compute an update, the component is first copied to a shared lower level memory (L3 cache or system RAM) and then read by the other core. The latency in this operation would be much higher than just reading from the local cache. It would be closer to true asynchronous iteration if the other core just reads stale values until the first core writes back to the shared level of memory. Such a write back could be triggered by the relevant cache line being eventually deleted to make space for new data. Thus the nature of the iteration depends heavily on the hardware setup. The effects of this are considered in the weak-scaling study (section 3.4.2) presented in the results.

We expect this asynchronous block SGS iteration to have the smoothing property required for use in multigrid methods because its behaviour approximates that of a lexicographic block SGS relaxation, which has good smoothing properties in many cases.

3.4 Results and discussion

In this section we experimentally investigate the effectiveness of asynchronous point-block SGS iterations in a shared-memory thread-parallel context. While certain large cases are solved on multiple nodes in the compute cluster using domain decomposition via MPI, asynchronous iterations are used only to achieve parallelism within each node independently. Only one MPI rank is used per node. Note that for this study we are not interested in the efficiency of scaling across MPI ranks. In our solver framework, individual MPI ranks are always associated with individual compute nodes and the method for achieving MPI scalability (domain decomposition) is different from the method used to achieve thread-scalability within the node (asynchronous iterations). Furthermore, parallel scalability of finite volume CFD codes across MPI ranks (using some form of domain decomposition) has been studied by many researchers (eg., [100, 101, 102]), and hence we made the decision to investigate only fine-grain parallelism.

The implementation uses OpenMP. In the asynchronous iterations, thread scheduling is dynamic so that load imbalance between the threads is not a problem. We use a chunk size ([82], section 2.7.1) of 400 work items, where each work-item corresponds to performing one relaxation on one cell of the mesh. The OpenMP affinity setting is chosen as 'compact', which means consecutive thread IDs are assigned to the closest core or hyper-thread context in the on-node processor topology [103].

As mentioned in section 3.2, we use PETSc [99] as our sparse linear algebra framework. In order to exploit the block nature of the Jacobian matrix, in this work, we use PETSc's 'BAIJ' storage format. This is a compressed block-sparse row format, which is similar to the compressed sparse row (CSR) format (section 2.9.3, figure 2.3), except that every entry of the values array is a 5×5 dense column-major block. Each block-row is associated with only one block-row pointer and each dense block is associated with only one block-column index - the matrix entries are not indexed individually in the row pointer and column index arrays. Vectors are stored in a corresponding block ordering. The Eigen library [104] has been used for vectorized small dense matrix operations in our block SGS smoothers.

We show strong scaling plots and a weak scaling plot. These plots are based on the total time taken by the asynchronous linear solver iterations. At each backward Euler step, the time taken by the asynchronous iterations consists of one factorization and several applications on each multigrid level. Factorization refers to inverting each of the 5×5 dense diagonal blocks. An application refers to one relaxation sweep across the grid. This terminology is borrowed from preconditioners in linear solvers, where there is a factorization or setup phase to build the preconditioner and there are several applications in each linear solve. The scaling is based on the time taken by both factorization and application over all smoothing operations in the flow solve.

3.4.1 Smoothing property of asynchronous block SGS

We investigate the smoothing property and parallel strong scaling of asynchronous point-block SGS iterations for six RANS cases. The 3D cases are tested with both the FAS multigrid solver as well as the linear multigrid solver (explained in section 2.5). This is to show that asynchronous point-block SGS iterations lead to good smoothing for both types of multigrid solvers. The 2D cases are run using only the FAS multigrid solver, as our linear multigrid scheme has only been tested on 3D cases. A 3-grid V-cycle is used in all cases, except for the NLR 7301 airfoil which uses four multigrid levels.

As explained earlier, whenever we report a certain number of smoothing sweeps in the case of FAS, we mean the number of linear iterations used to solve one backward Euler step. We always carry out one backward Euler step each for pre- and post-smoothing on all multigrid levels except the coarsest. The number of linear iterations used to solve the pre-smoothing backward Euler step is always the same as that used for post-smoothing on a given level. The coarse-grid solver is one backward-Euler step, but solved by sequential ILU(0) (incomplete LU factorization with same sparsity pattern for the factors as for the Jacobian matrix). The linear multigrid case is analogous - smoothing is carried out by multiple asynchronous point-block SGS iterations, pre- and post-smoothing use the same number of sweeps, and the coarse grid is solved by sequential ILU(0) iterations.

We measure the speedup obtained in asynchronous iterations over the entire flow solution. The mass flux residual of the mean flow equations is dropped by 6 orders of magnitude unless otherwise stated. The tests are carried out on Intel Sandy-Bridge (E5-2670) nodes with 2 CPUs per node and 8 cores per CPU. The Intel C/C++ compiler version 2015b has been used with AVX vectorization enabled. Simultaneous multi-threading (SMT), also known as hyper-threading, is disabled on these nodes. Thus, we use one OpenMP thread per core, and this means that in this section, core counts are the same as thread counts.

The following points about the results to follow should be noted.

- All SGS iterations (sequential or asynchronous) are programmed according to lexicographic ordering.
- We measure only the total time taken by asynchronous iterations. This is both to leave out serial (non-threaded) operations in PETSc and to focus on the scalability of the smoother. Furthermore, for the smoothing study, we use a regular synchronized

iteration on the coarsest multigrid level - 8 iterations of ILU(0). The time taken by the coarse grid solver is not considered in the speedups reported.

- Speedup over a fixed number of multigrid cycles would not account for the quality of the smoother. Therefore we measure the total time taken by all asynchronous iterations over the entire flow solve until convergence to steady state. If the parallel smoother does not damp the high-frequency error modes well enough for some case, more multigrid cycles and thus more smoothing operations will be required to converge, and the speedup realized will suffer. Thus, the quality of the smoother is factored into the reported speedups.

- Before the main solve on the original grid starts, a full multigrid (FMG) initialization [37, chapter 2.6] is used to get a better initial solution on the original grid. This involves performing a fixed number of cycles on the coarsest level before moving to the next finer level and so on, until the finest level is reached. The reported timings and speedups include the time spent on the initial coarser solves. The smoother used on these initial coarser solves is asynchronous block SGS while the iteration on the coarsest grid is sequential ILU(0) which is not timed. The multigrid iteration counts reported in the plots are the number of multigrid cycles used on the original grid (finest level). The fixed number of iterations used for initialization on coarser levels are reported in the case descriptions below.

- Each run is repeated 3 times and we ensure that the deviation between the runs is small ($< 0.5\%$).

The cases are as follows. They are characterized by the angle of attack α, free-stream Mach number M, and the Reynolds number R_e. The convergence criterion for each of the cases is 6 orders of magnitude for the mass flux residual or density residual, except for the DLR F6 and CRM cases.

- RAE 2822 airfoil. The original description of the case is in [105]. This is a 2-dimensional transonic RANS 'case 10' with $M = 0.75, R_e = 6.2 \times 10^6, \alpha = 2.81°$. An example simulation of this case can be found in [50], where it can be seen that case 10 is the difficult one of the RAE 2822 cases. The grid has 73,728 cells. We use 50 cycles on the coarsest and medium multigrid levels each before starting the solve on the original grid using a total of 3 multigrid levels. The case was run on one node.

- NLR 7301 airfoil with flap [106]. This case is characterized by $M = 0.185, R_e = 2.51 \times 10^6, \alpha = 13.1°$. The angle of attack is ramped up from 3° to the final value of 13.1° in 200 multigrid cycles on the finest level after full multigrid initialization. The mesh is made up of 9 structured blocks for a total of 144,832 cells. 50 multigrid V-cycles are used on each of three coarser levels for initialization, and a 4-level cycle is used on the original grid. This case was run on 2 nodes.

- ONERA M6 wing [107]. This is a 3-dimensional case with $M = 0.8395, R_e = 11.72 \times 10^6, \alpha = 3.06°$. The mesh has 884,736 cells divided into 32 structured blocks. FMG initialization of 100 iterations on each coarser grid is performed before solving the original grid with 3 multigrid levels. This case was run on 4 nodes.

- DPW W1 wing [108]. This is a 3-dimensional case with $M = 0.76, R_e = 5.0 \times 10^6, \alpha = 0.6109°$. The mesh consists of $1,511,424$ cells divided into 316 blocks. FMG initialization of 100 iterations on each coarser grid is performed before solving the original grid with 3 multigrid levels. This case was solved on 16 nodes.

- DLR F6 wing-body [108]. A 3-dimensional case with $M = 0.75, R_e = 3.0 \times 10^6, \alpha = 0.5°$. The mesh consists of $3,701,760$ cells divided into 510 blocks. FMG initialization of 50 iterations on each coarser grid is performed before solving the original grid with 3 multigrid levels. This case was also solved using 16 nodes. The convergence tolerance for this case is 4 orders of magnitude.

- NASA CRM wing-body case (L3 grid) [109]. This 3-dimensional case is characterized by $M = 0.85, R_e = 5.0 \times 10^6, \alpha = 2.11°$. The mesh for this case is made up of 5,111,808 cells divided into 896 structured blocks. We use 50 multigrid cycles on two coarser levels each to initialize the flow variables on the original fine grid. The flow solve on the original grid uses a total of 3 multigrid levels. We used 32 nodes to run this case. The convergence tolerance is 5 orders of magnitude.

To illustrate the kind of cases we are solving, we show some aspects of the flow solution over the CRM wing-body. Figure 3.1 shows the pressure coefficient, while figures 3.2a and 3.2b show streamlines and regions of positive and negative skin-friction coefficient. Separation bubbles at the leading edge (3.2a), as well as at the junction of the wing root and trailing edge (3.2b), and can be seen. The grid used to solve the problem is also shown (figures 3.3,3.4).

The plots of multigrid cycles required to converge to steady-state versus the number of asynchronous iterations per grid level per cycle, for different thread counts, allow us to consider the smoothing property of the asynchronous smoother (figures 3.5, 3.7, 3.9, 3.11, 3.13, 3.15). In each of these runs, the solver on the coarsest grid is fixed at 8 iterations

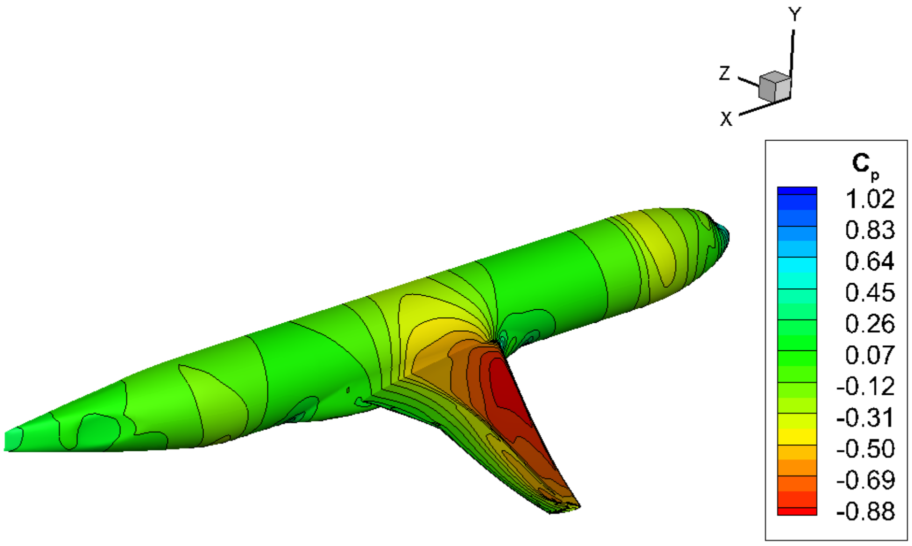
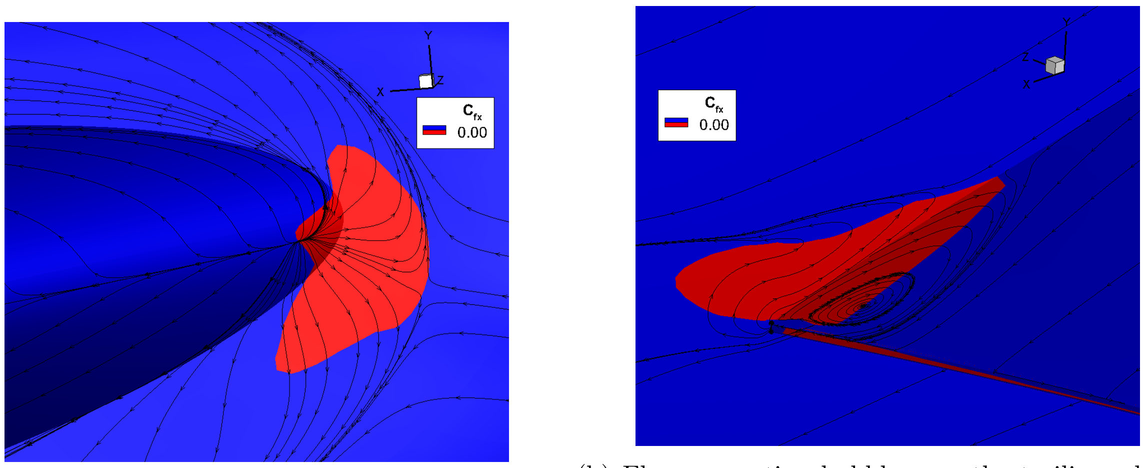


Figure 3.1: Pressure coefficient contours for the CRM wing-body case



(a) Flow separation near the leading edge of the wing

(b) Flow separation bubble near the trailing edge of the wing

Figure 3.2: Streamlines and regions of positive and negative skin-friction coefficient for the CRM wing-body case

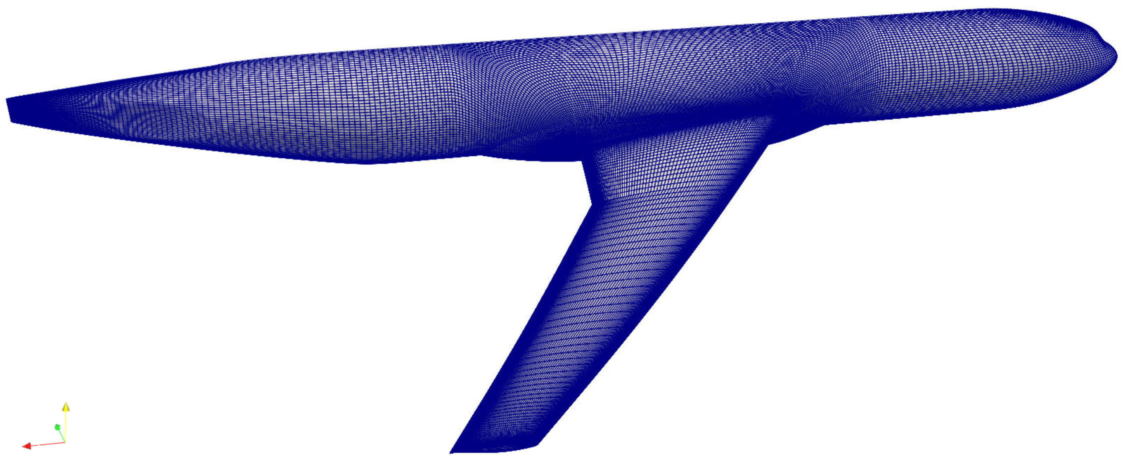


Figure 3.3: Structured grid of CRM wing-body geometry (L3 grid)

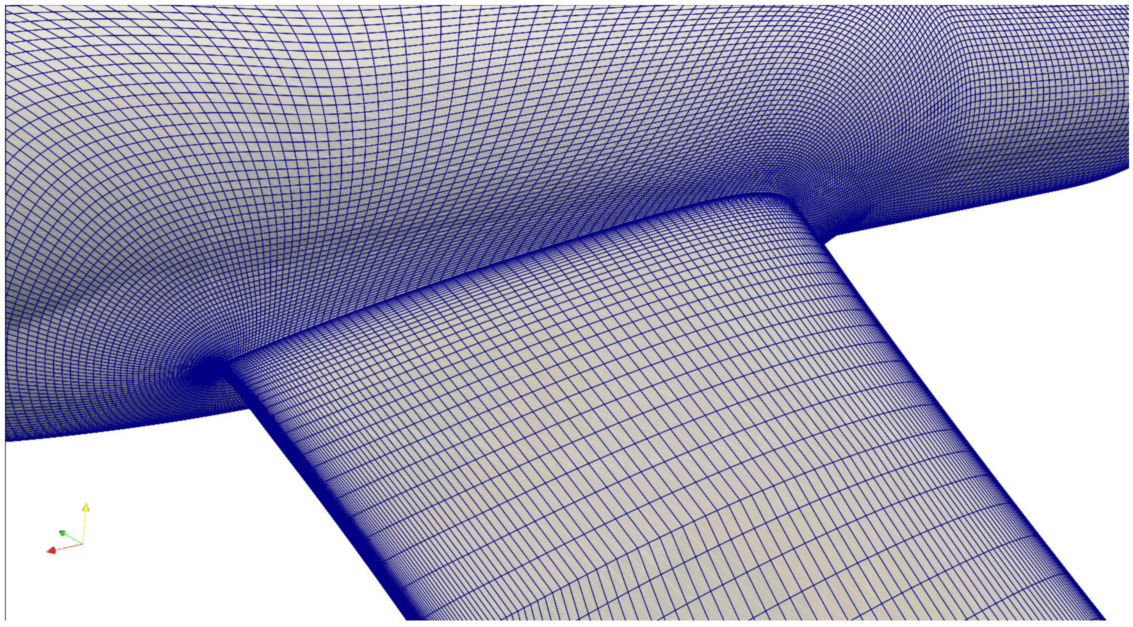


Figure 3.4: Structured grid near wing-body junction of CRM wing-body geometry (L3 grid)

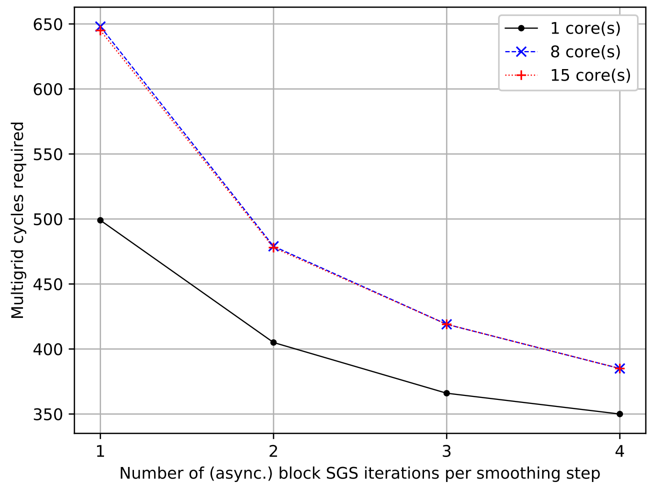


Figure 3.5: RAE2822: FAS MG cycles required to converge 6 orders of magnitude as a function of number of asynchronous sweeps used for smoothing

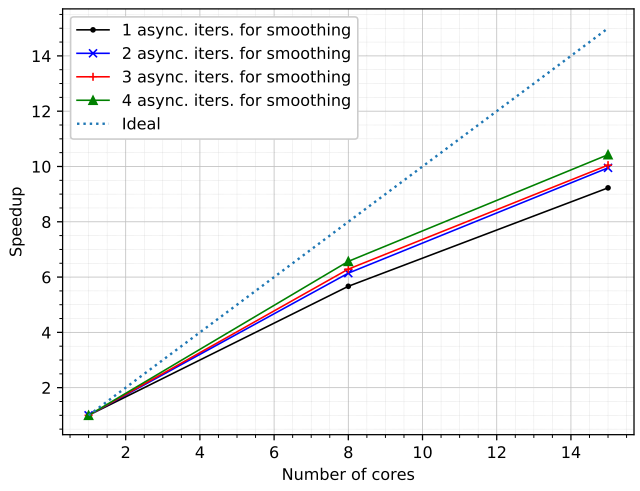


Figure 3.6: RAE2822: Strong scaling of asynchronous block SGS vs. number of cores; 73,728 cells per node

of sequential ILU(0). Hence, the difference in convergence rate is due to the difference in smoothing effectiveness, which is what we want to study in this section.

The number of sweeps used for pre-smoothing and that used for post-smoothing are always equal. For example, when we mention 1 asynchronous iteration per smoothing step, one asynchronous point-block SGS iteration was done for pre-smoothing and one for post-smoothing on all multigrid levels except the coarsest level in each multigrid cycle. The 1-core curves correspond to the case where the asynchronous iteration is the same as the regular synchronized point-block SGS iteration. For 8 and 15 cores, the smoother is asynchronous.

From this study, a common trend that emerges is that the smoothing property of the asynchronous block SGS is largely similar to that of the regular block SGS when more than

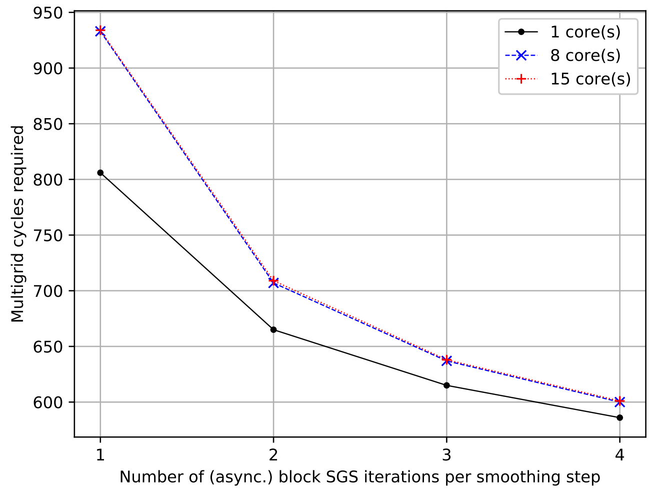


Figure 3.7: NLR 7301: FAS MG cycles required to converge 6 orders of magnitude as a function of number of asynchronous sweeps used for smoothing

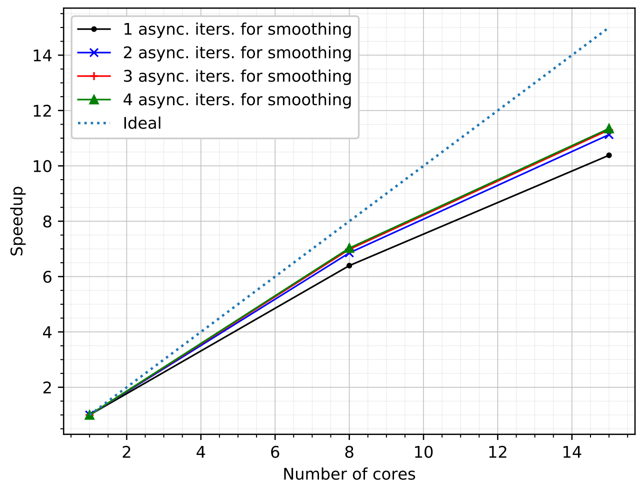
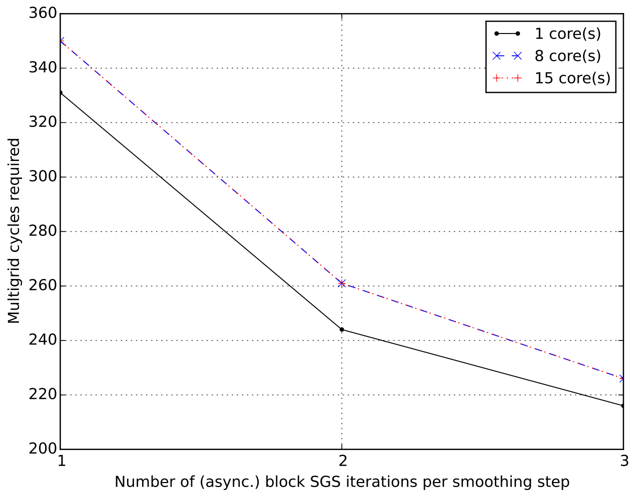
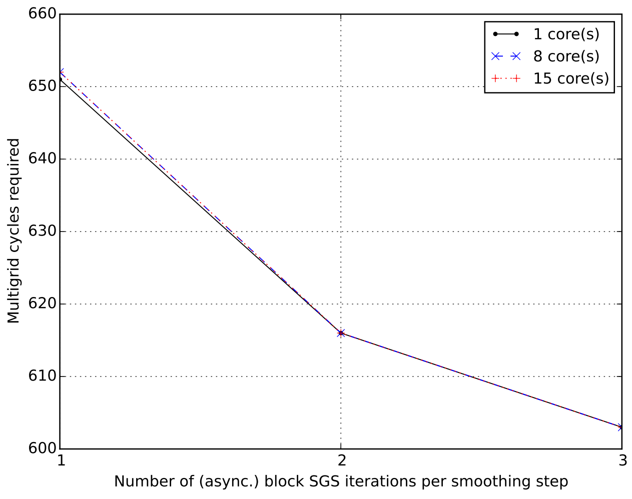


Figure 3.8: NLR 7301: Strong scaling of asynchronous block SGS vs. number of cores; 72,416 cells per node

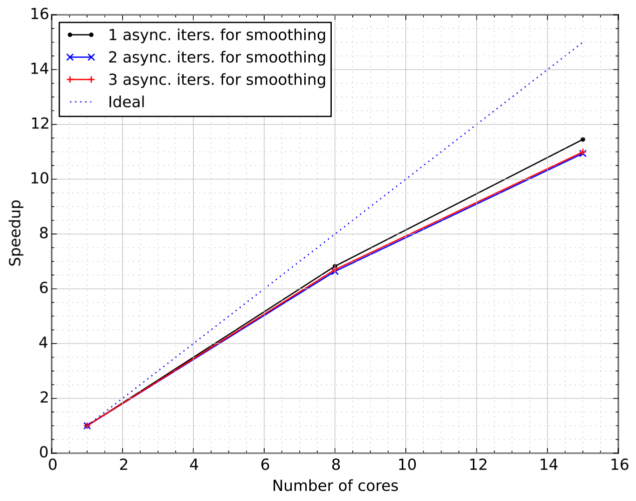


(a) FAS

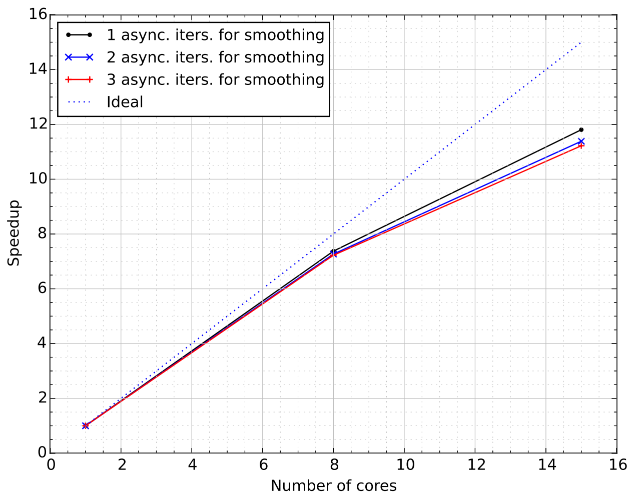


(b) Linear MG

Figure 3.9: ONERA-M6: MG cycles required to converge 6 orders of magnitude as a function of number of asynchronous sweeps used for smoothing

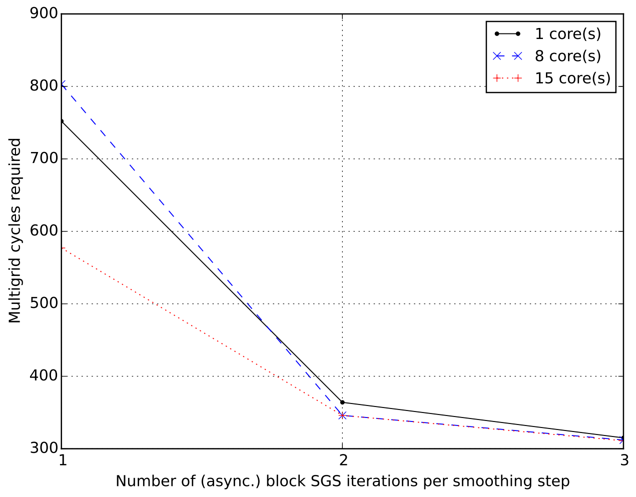


(a) FAS

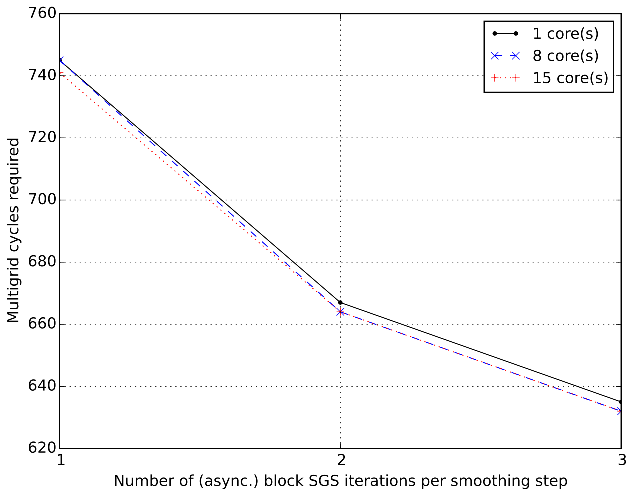


(b) Linear MG

Figure 3.10: ONERA-M6: Strong scaling of asynchronous block SGS vs. number of cores; 221,184 cells per node

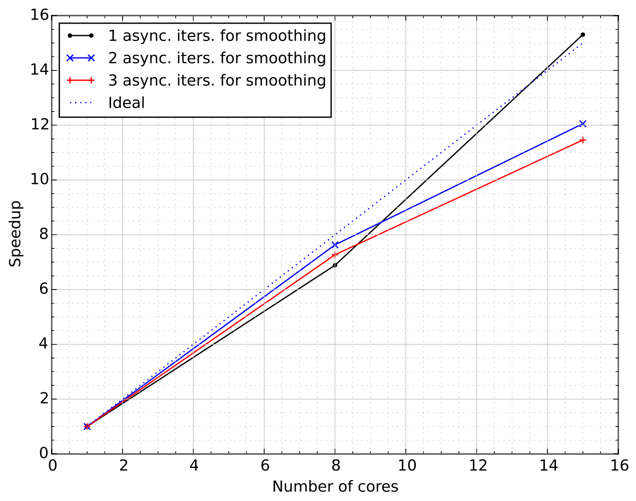


(a) FAS

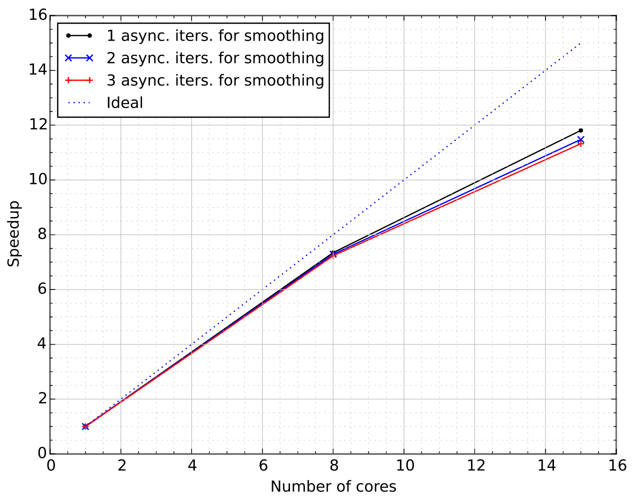


(b) Linear MG

Figure 3.11: DPW-W1: MG cycles required to converge 6 orders of magnitude as a function of number of asynchronous sweeps used for smoothing

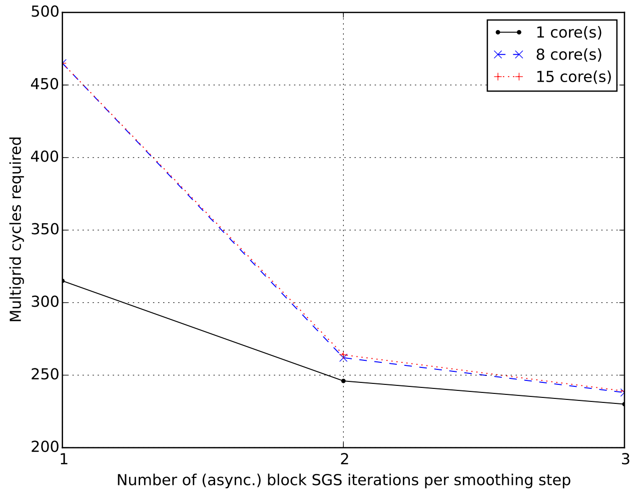


(a) FAS

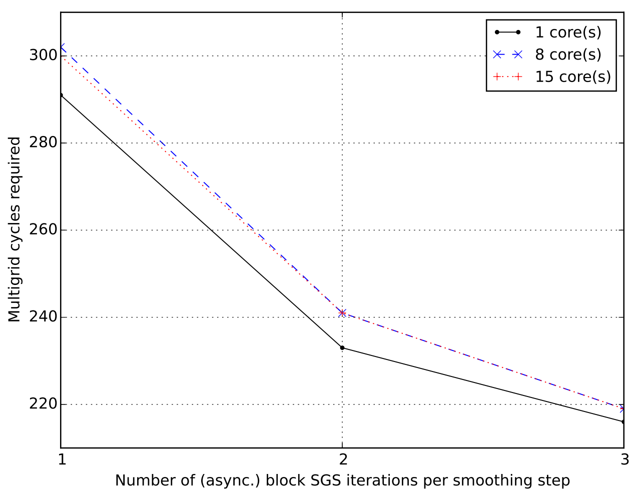


(b) Linear MG

Figure 3.12: DPW-W1: Strong scaling of asynchronous block SGS vs. number of cores; 94,464 cells per glsnode

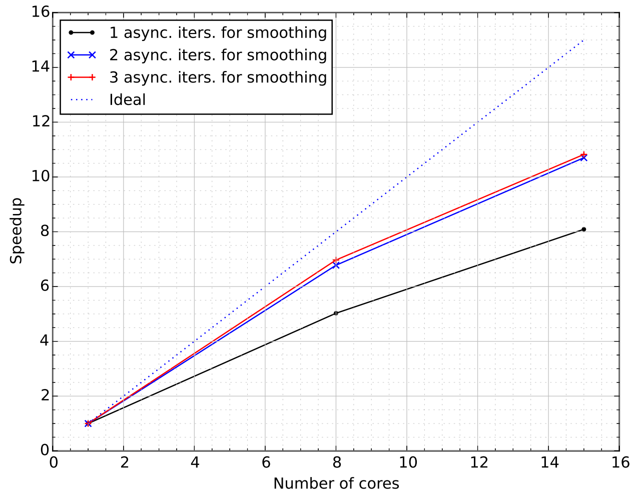


(a) FAS

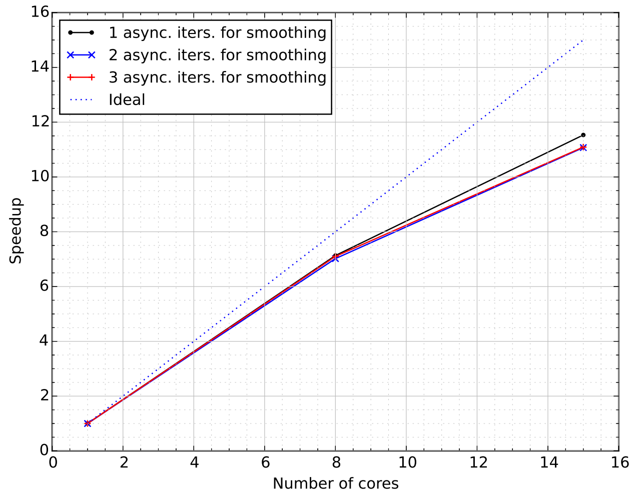


(b) Linear MG

Figure 3.13: DLR-F6: MG cycles required to converge 4 orders of magnitude as a function of number of asynchronous sweeps used for smoothing

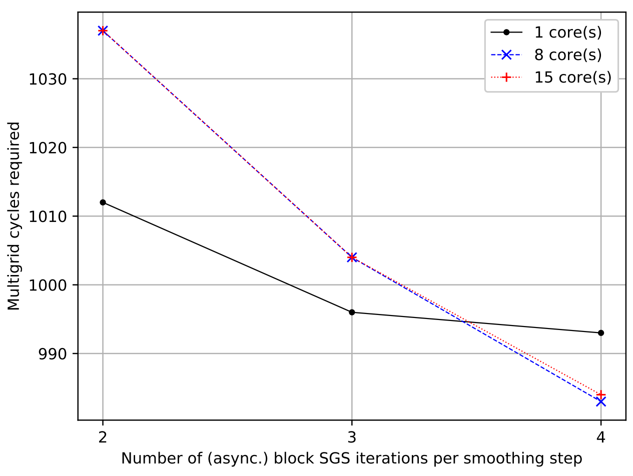


(a) FAS

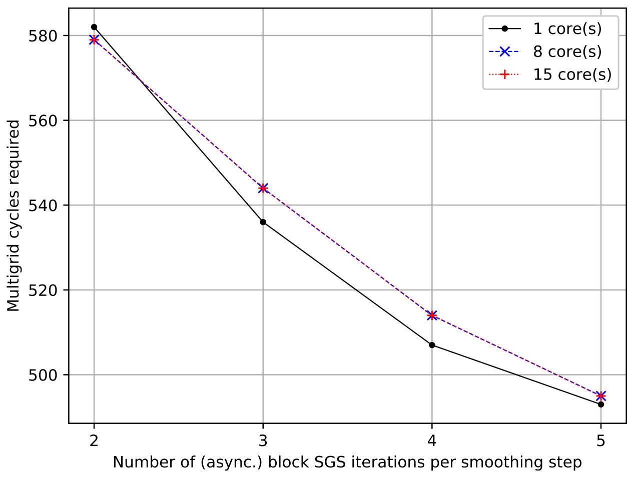


(b) Linear MG

Figure 3.14: DLR-F6: Strong scaling of asynchronous block SGS vs. number of cores; 231,360 cells per glsnode

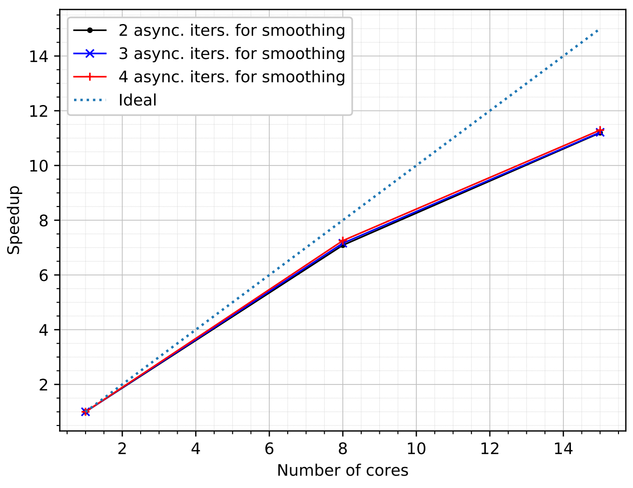


(a) FAS

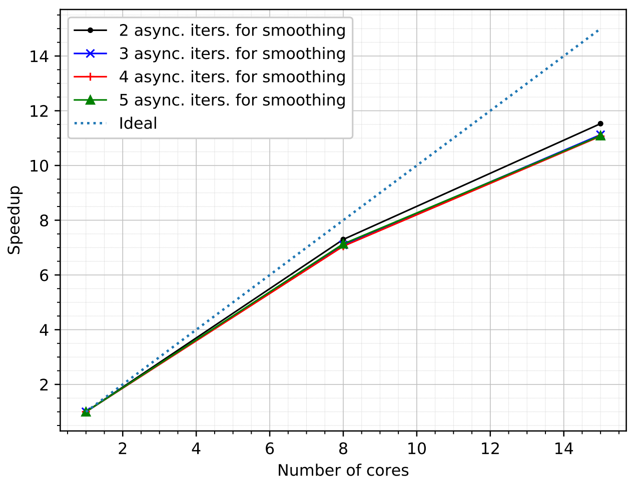


(b) Linear MG

Figure 3.15: CRM: MG cycles required to converge 5 orders of magnitude as a function of number of asynchronous sweeps used for smoothing



(a) FAS



(b) Linear MG

Figure 3.16: CRM: Strong scaling of asynchronous block SGS vs. number of cores; 159,744 cells per glsnode

one sweep is used for pre- and post-smoothing. When only one sweep is performed per smoothing operation, we observe more erratic behaviour. In this case, the specific way that updated data is made available to the different threads matters the most. As more sweeps are carried out, this erratic effect starts getting averaged out, and the convergence behaviour in case of more than one thread (the case of asynchronous iterations) tends to become the same as that for one thread (regular synchronous iterations). This is expected because over many sweeps, the asynchronous and synchronous iterations converge to the same solution. We note that just one iteration for pre- and post-smoothing each was never the best setting in terms of wall-clock time and therefore we expect this effect to rarely be of importance for this kind of solver.

The strong scaling plots (figures 3.6, 3.8, 3.10, 3.12, 3.14, 3.16) indicate 10x to 12x speedup going from 1 to 15 cores for most cases. Note that the slopes of the graphs always reduce beyond 8 cores. This is because the glsnode is a two-socket server with two Sandy Bridge CPUs with 8 cores each. Because communication between sockets is more expensive, this reduction in speedup beyond 8 cores is expected. Further, hyper-threading is not enabled on these nodes, so memory fetch latency cannot be effectively masked with computation by another thread.

To put the 10x-12x speedup in perspective, we show the speedup obtained by only the factorization phase of the linear solver over the entire flow until convergence to steady-state (fig. 3.17). Factorization for the asynchronous block SGS relaxation refers to the inversion of the small dense 5×5 diagonal blocks corresponding to each of the mesh cells. Thus it is a lot of small dense LAPACK calls made in an 'embarrassingly' parallel manner (there is no communication required). We see the the factorization achieves a speedup of 13x-14x on 15 cores of the dual-socket Sandy-Bridge server.

We achieve less speedup on the RAE 2822 case (figure 3.6), primarily because the asynchronous point-block SGS iteration requires significantly more multigrid cycles to converge than the synchronous point-block SGS iteration. However, the remaining cases provide better speedups for most asynchronous sweep counts. The speedup curves of the one-sweep runs reflect their erratic smoothing property, since the asynchronous smoother can be significantly different from the classical block SGS smoother for one-sweep runs. The DPW W1 case (figure 3.11a) is especially interesting in this regard. For the single-sweep run, the eight-thread asynchronous smoother is slightly inferior to regular block SGS smoothing, while the fifteen-thread asynchronous smoother is significantly superior. This leads to exceptionally good scaling for one asynchronous sweep (figure 3.12a).

In tables 3.1a and 3.1b, we summarize the speed-ups when three asynchronous block SGS sweeps were employed per smoothing step.

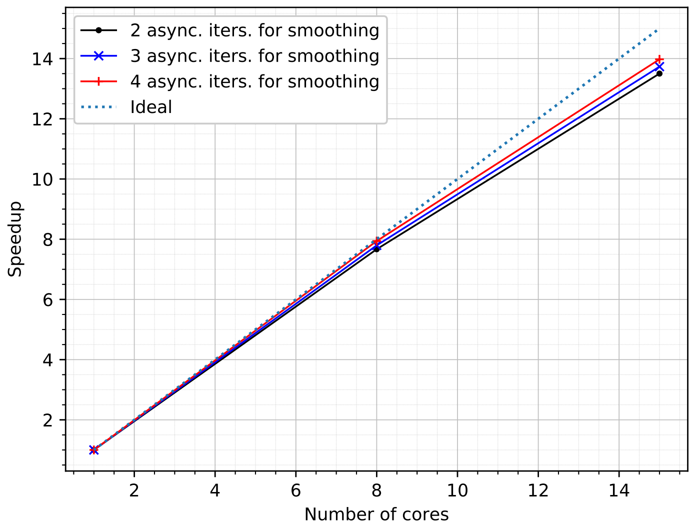


Figure 3.17: Speedup obtained by the factorization phase for the CRM wing-body case; compare with figure 3.16a

Case	8 cores	15 cores
RAE 2822	6.29	10.00
NLR 7301	6.98	11.30
ONERA M6	6.47	10.33
DPW-W1	7.27	11.45
DLR-F6	6.96	10.82
CRM L3M	7.16	11.20

(a) FAS

Case	8 cores	15 cores
ONERA M6	7.03	10.52
DPW-W1	7.23	11.32
DLR-F6	7.09	11.09
CRM L3M	7.10	11.10

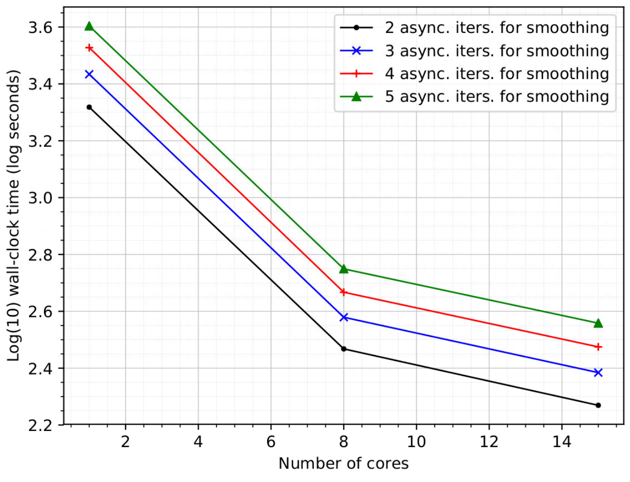
(b) Linear MG

Table 3.1: Speedup for 8 cores and 15 cores for the case of 3 asynchronous sweeps per smoothing step

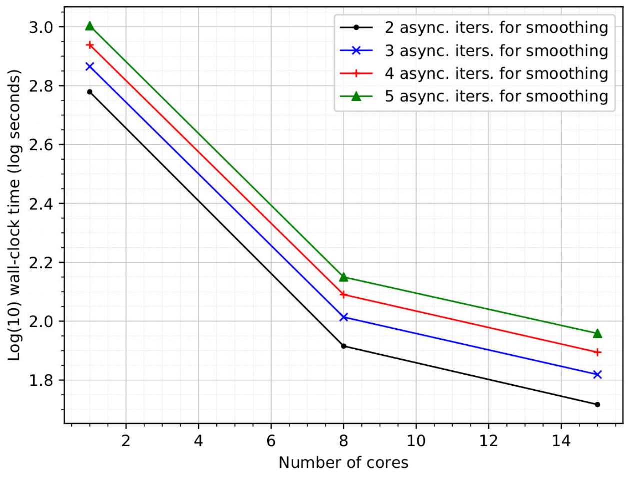
Finally, we show the impact of using different numbers of asynchronous sweeps on the actual wall-clock time required for convergence in figure 3.18 for the CRM wing-body case. Typically, fewer asynchronous sweeps lead to faster convergence in wall-clock time, though this is case-dependent.

3.4.2 Parallel scaling on the Xeon Phi Knights Landing

In this section, we demonstrate the scaling of asynchronous point-block SGS smoothing on the Intel Xeon Phi Knights Landing ('KNL') 7230 CPU under a few different conditions. The main features of this CPU that set it apart from Intel Xeon CPUs up through the Skylake family are:



(a) FAS



(b) Linear MG

Figure 3.18: CRM wing body case. Wall-clock time required to converge 5 orders of magnitude for different numbers of asynchronous sweeps used for smoothing

- High core count of 64 to 72 cores (64 in our device).
- Cores organized into tiles, each containing two cores and common L2 cache.
- 16 gigabytes of high-bandwidth memory present on the CPU die, called MCDRAM.
- Four-way simultaneous multi-threading ('hyper-threading') per core.

Apart from these, the KNL has AVX-512 vector instructions that potentially double the performance compared to older Xeon CPUs. However, for our tests here, we do not use AVX-512 vectorization; AVX is enabled instead. Our main goal in this study is to investigate scalability with threads when hyper-threading and many cores in one socket are available, which was not the case with the Sandy Bridge CPUs. The individual cores are slower than regular Xeon cores, but the large number of them and the high-bandwidth memory indicate that high throughput may be possible to achieve.

Some aspects of the KNL processor are configurable. In our case, the cores have been configured in the 'all-to-all' cluster mode. This means that there is no affinity between tiles and memory locations. Memory access latency between any KNL tile and any memory location is essentially the same as that between any other tile and any other memory location. The MCDRAM has been setup in the 'flat' mode, that is, the high-bandwidth memory is configured as a separate NUMA node [110]. By default, the system RAM is used and the MCDRAM remains unused. The 'numactl' program can be used to request exclusive use of the MCDRAM rather than system RAM without any modification to the code. This is useful if one knows that the 16 gigabytes of MCDRAM are sufficient to hold all the data required by the problem. In the flat mode, the KNL does not have any L3 cache shared among all the cores of the CPU. For all the runs in this section, we used numactl to exclusively use the MCDRAM. Also, we pin each thread to one hyper-thread context, rather than allowing threads to float among hyper-threads of a single core. For runs on the KNL, we use the 2017 version of Intel's C/C++ compiler. It may be noted in the results that we use only 60 out of the 64 cores on the device. When using all 64 cores, while the solver converges without issues, the performance in terms of wall-clock time becomes erratic due to interference from some operating system processes. We thus report results only up to 60 cores.

Similar to the studies in the previous section, we time only the asynchronous point-block SGS iterations for the scaling tests. To determine speedups for the strong scaling tests, we determine the sum of time taken by all the asynchronous iterations required for convergence of the mass-flux residual on a certain number of cores. This is compared to the time required, for the same settings and the same mass-flux residual convergence tolerance, on one core.

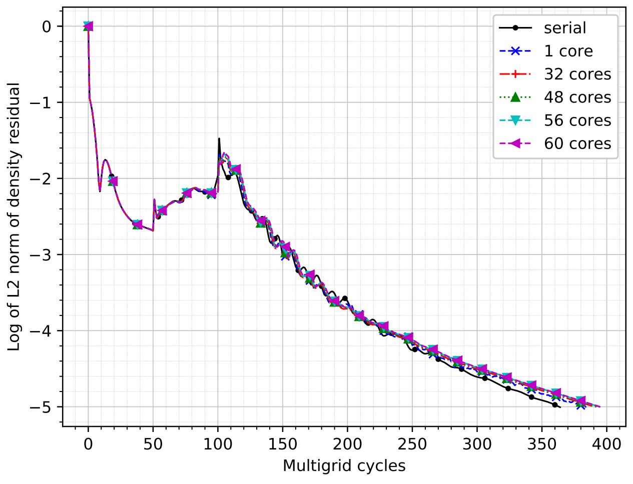
Note that when multiple threads are used per core, 1-core runs use asynchronous iterations. Serial (or sequential) runs correspond to those using 1 thread only. All cases in this section use FAS multigrid with a total of 3 levels. In order to see the performance of the asynchronous block SGS method as a smoother, for most of the tests we use a sequential coarse grid solver and do not time it. The only exception is the case where we compare synchronous and asynchronous coarse grid solvers. For each test below, we state whether this is the case. Also, we sometimes refer to point-block SGS as simply 'block SGS' which is shortened to 'BSGS' in figures for brevity.

Convergence behaviour

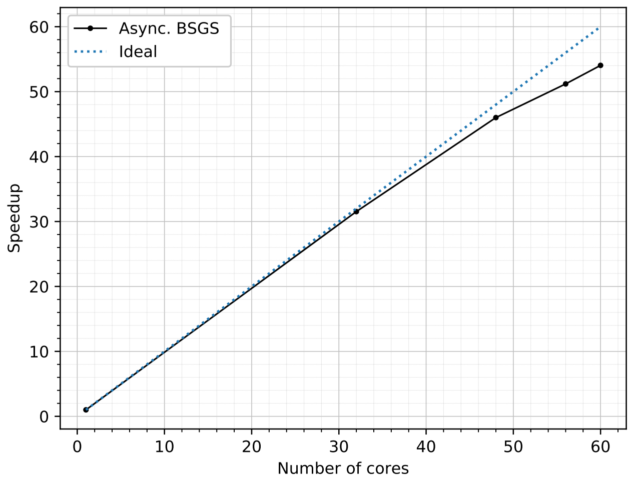
Figure 3.19 shows strong scaling and convergence history of regular block SGS (the serial run) and asynchronous block SGS (all the runs on one or more cores) on various number of cores for the 2D RAE 2822 case with 294,912 mesh cells. The flow conditions are the same as that for RAE2822 'case 10' mentioned in the previous section. Hyper-threading is used - we use 4 threads per core. Note that in all these runs, 12 sweeps of sequential point-block SGS have been used on the coarsest multigrid level.

The convergence history (figures 3.19a 3.20) shows that the convergence rate delivered by the asynchronous smoother is somewhat worse than that of the sequential smoother. However, the convergence property remains very similar with increasing parallelism. Since the RAE 2822 case 10 is one of the more difficult cases in our test suite, this is very encouraging for even further scalability of the method.

Next, we present results for the three-dimensional ONERA M6 case and show the convergence behaviour and speedup, and comment about using asynchronous iterations as a



(a) Convergence for different core counts



(b) Strong scaling with hyper-threading

Figure 3.19: Performance of asynchronous block SGS ('BSGS') on RAE2822 (294,912 cells) for different numbers of cores on the KNL processor (4 threads per core)

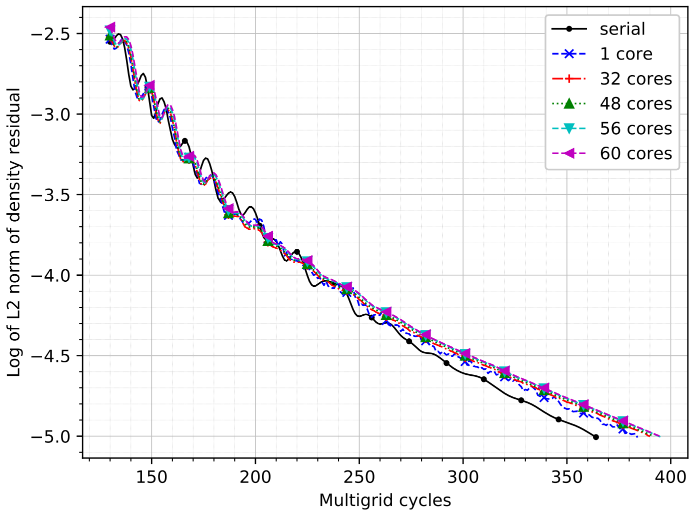


Figure 3.20: RAE2822 case: convergence with async. BSGS for different core counts, zoomed to the cycles towards the end of the run

coarse-grid solver. The flow conditions are the same as that reported for the ONERA-M6 wing case in the previous section. In figure 3.21, we show the speedup attained on a mesh with 884,736 cells. The speedup obtained when using asynchronous point-block SGS iterations on all levels including the coarse level has been compared with using sequential point-block SGS at the coarse level. In both cases, 12 point-block SGS sweeps are used on the coarse level. We see that when asynchronous point-block SGS is only used for smoothing, we get a very good speedup, nearly ideal. When it is used both as smoother and coarse-grid solver, the speedup decreases, but still remains good.

Sequential portions are not included in the timing. Hence, in the case of sequential coarse solve, the time taken by the coarse solve is not included. However, for the case where asynchronous smoothing is used on all levels, the coarse solve is included in the timing. This can contribute, to some extent, to the slowdown when using asynchronous coarse solves since the number of cells on the coarsest grid is only 13,824.

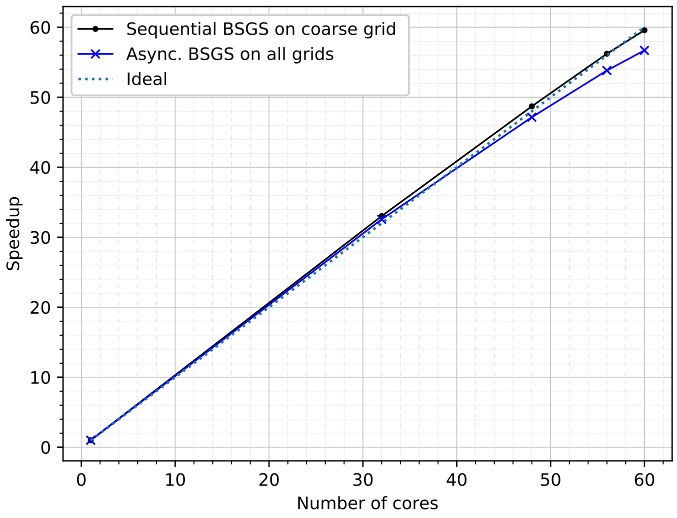
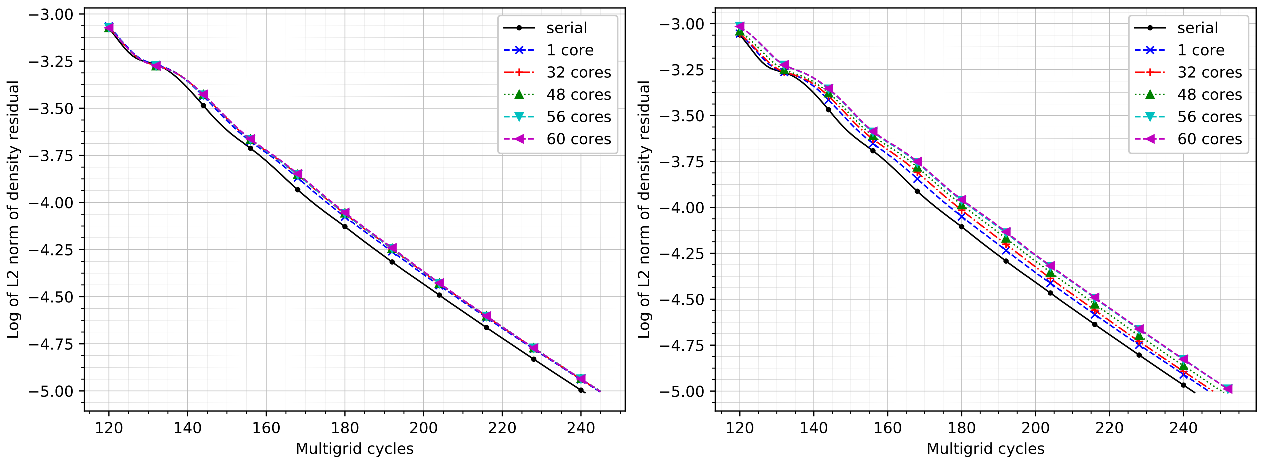


Figure 3.21: ONERA M6: Speedup comparison between asynchronous BSGS smoothing and sequential BSGS smoothing at the coarsest level (all other levels use asynchronous smoothing)



(a) Sequential BSGS coarse solve (b) Asynchronous BSGS coarse solve

Figure 3.22: Convergence behaviour of asynchronous BSGS as (left) smoother only and (right) as both smoother and coarse-grid solver on ONERA-M6 (884,736 cells). Only the final 130 cycles have been shown for clarity

However, the primary reason for the reduced speedup is seen in figure 3.22. When using a sequential coarse solve, we see that all the multi-threaded asynchronous runs have the same residual history; they are only slightly slower than the fully sequential run in terms of iterations to convergence. But when we use asynchronous iterations on all multigrid levels, the number of iterations required to converge increases slightly as we increase the number of threads. We conclude from this that while the smoothing property of the asynchronous point-block SGS is completely on par with sequential lexicographic point-block SGS, as a coarse grid solver it can be poorer. That being said, the speedup obtained using only asynchronous iterations is still very good on the KNL. It is notable that for this case, the convergence rates attained by different numbers of cores (as seen from the slopes of the curves in figure 3.22b) is almost the same when we use asynchronous iterations for the coarse-grid solve.

Effect of hyper-threading

In figure 3.23, we compare the performance of the asynchronous smoother with and without hyper-threading. We use 10 sweeps of sequential BSGS as the coarse solve so that the results are not affected by the smoother's performance on the coarse grid. For the runs in which no hyper-threading is used, each thread is pinned to one thread-context (similar to the runs with hyper-threading) but only one thread is used per core.

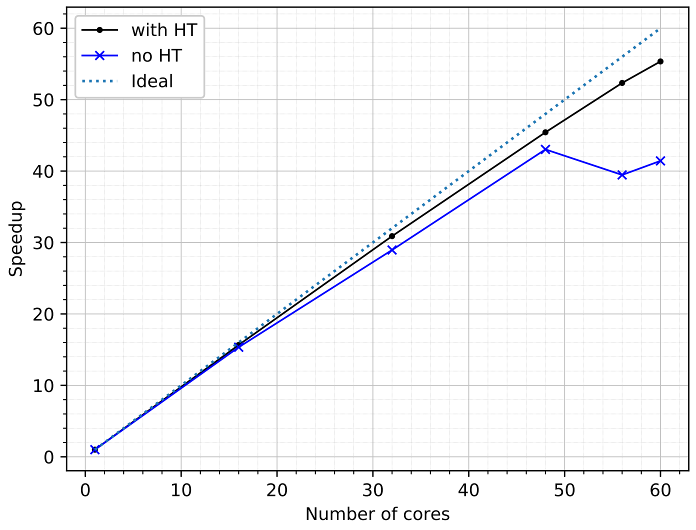


Figure 3.23: Strong scaling for ONERA M6 case with and without hyper-threading

We see that not utilizing hyper-threading leads to a worsening of strong scaling up to 48 cores and a *slowdown* thereafter, whereas the hyper-threaded run continues to scale well. In terms of number of cycles to convergence, we observed that there was absolutely no difference between the hyper-threaded and non-hyper-threaded runs. Thus, for our setup, hyper-threading is necessary for the asynchronous block-SGS smoother to hide latency that arises due to the computation and communication pattern.

Weak scaling of the asynchronous smoother

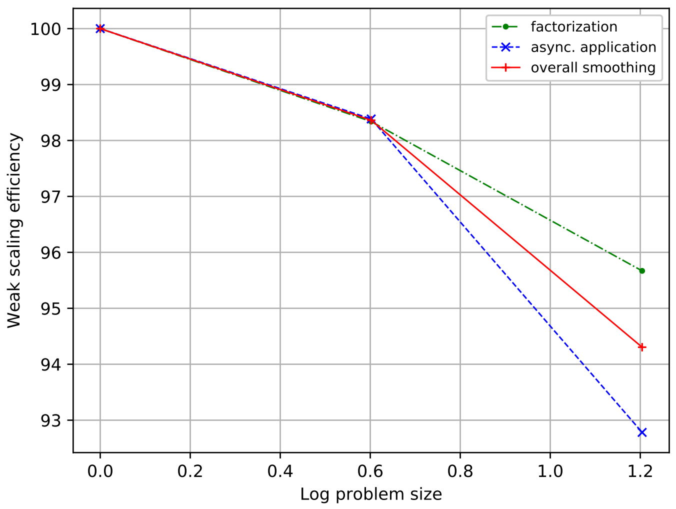


Figure 3.24: Weak scaling efficiency for 280 multigrid cycles using asynchronous point-block SGS on the RAE 2822 case 10

In this section, we show the weak scaling achieved from our implementation of asynchronous point-block SGS iterations. The weak scaling study is performed on the RAE 2822 case 10 described earlier. We use three meshes with 288×64, 576×128 and 1152×256 cells, and use 3 cores, 12 cores and 48 cores to solve the problems on the respective meshes. The simulation on each of these meshes is done using a 3-level FAS multigrid solver with asynchronous point-block SGS smoothing. This time however, instead of requiring the flow to converge to a predefined tolerance, we perform a specific number of multigrid cycles. Thus, the weak scaling studied here is a per-iteration weak-scaling which only reflects whether the cost of inter-core communication remains roughly constant with increasing parallelism and problem size. It is important to note that it does not consider the scalability of the entire multigrid solver.

For this test, we compute the weak scaling efficiency (section 2.9.5) when using $3k$ cores as

$$\text{Eff}_w(n, 3, k) = \frac{t(n, 3)}{t(nk, 3k)} \times 100 = \frac{\text{Time taken on 3 cores}}{\text{Time taken on } 3k \text{ cores}} \times 100. \tag{3.2}$$

We perform 280 multigrid V-cycles for all runs and utilize a sequential point-block SGS coarse-grid solver. Hyper-threading and MCDRAM have been enabled. Our results (figure 3.24) show a weak scaling efficiency of about 93% at 48 cores for the application. Here, application refers to the actual asynchronous relaxation and factorization refers to inversion of the 5×5 dense diagonal blocks.

The fact that the application shows worse efficiency than the factorization (which is ‘embarrassingly’ parallel with no communication) indicates that there could be some implicit synchronization in the asynchronous iteration. Currently, we surmise that this may be because the runtime (OpenMP implementation, operating system kernel, hardware etc.) forces each thread to wait for reading a needed component if it has been recently updated by another thread. This amounts to implicitly enforcing a synchronization between two threads at a time. While this is much better than the global synchronization required for traditional smoothers, it also means that the execution of the algorithm on the hardware may not be fully asynchronous. This may be what is reflected in the weak scaling. Further investigation is needed in this regard.

3.5 Conclusions

In this chapter, we formulated and demonstrated the use of asynchronous point-block symmetric Gauss-Seidel smoothing in linear and nonlinear multigrid solvers for compressible RANS flows. The results demonstrate that the smoothing property (and the multigrid convergence rate) does not degrade significantly with increasing parallelism, on both the dual-socket Sandy-Bridge CPUs and the highly parallel Knights Landing CPUs. Further, the weak scaling study shows the computation-to-communication ratio to scale favourably. Thus, asynchronous point-block smoothers show promise for effective fine-grain parallel smoothing for compressible turbulent flow problems.

We note that several areas related to this work remain to be explored. As we observed in the per-iteration weak scaling results, there is some inefficiency associated with the asynchronous iteration relative to an embarrassingly parallel kernel. As discussed at the end of the previous section, this may be due some implicit synchronization that our implementation does not consider. Future investigations will consider whether this is indeed the case, and why the weak scaling is slightly less than expected. Profiling the implementation using a profiler that can query hardware counters can enable us to identify the causes of this inefficiency and ultimately to understand what limits the performance and scaling of the asynchronous point-block SGS algorithm.

Secondly, we have not made any specific attempts to ensure or investigate the extent of vectorization. Though the Eigen library uses vectorized block operations, it has not been investigated how effective that has been. In chapter 5, we work with a different kind of solver and ensure vectorization.

Finally, the effectiveness of asynchronous iterations as coarse-grid solvers needs further inquiry. As was seen for the ONERA M6 case on the KNL processor, even when asynchronous block SGS is used on the coarsest grid, good scaling may be observed. This is in spite of the fact that convergence deteriorates slightly with increasing thread count. It remains to be seen whether this is a general trend and whether we can achieve parallel coarse-grid iterations without convergence slowdown.

Chapter 4

Block-asynchronous iterations: an asynchronous incomplete block LU preconditioner for unstructured grids

In this chapter, a block variant of the asynchronous fine-grain parallel ILU preconditioner adapted to a finite volume discretization of the compressible Navier-Stokes equations on unstructured grids is presented, and convergence theory is extended to the new variant. Experimental (numerical) results on the performance of these preconditioners on inviscid and viscous laminar two-dimensional steady-state test cases are reported. It is found, for these compressible flow problems, that the block variant performs much better in terms of convergence, parallel scalability and reliability than the original scalar asynchronous ILU preconditioner. For viscous flow, it is found that the ordering of unknowns may determine the success or failure of asynchronous block-ILU preconditioning, and an ordering of grid cells suitable for solving viscous problems is presented. The material in this chapter is the subject of an article submitted to SIAM [25] and is currently under revision.

4.1 Introduction

In case of explicit time-stepping, there has been some success in utilizing many-core devices and GPUs (eg. [111]). However, as discussed in section 2.2, explicit solvers face a restrictive time-step limit and are only suitable when resolution of unsteady high-frequency phenomena is desired. For steady-state problems, implicit time-stepping is more suitable. Implicit solvers require the solution of large sparse systems of linear equations, which in turn, requires effective parallel preconditioners.

Incomplete LU factorization is commonly used as a preconditioner to solve large sparse systems of equations due to its wide applicability. However, factorization of the matrix into upper and unit lower triangular factors (algorithm 2), and the application of the triangular factors during the solve, are performed using sequential algorithms. When only coarse-grained parallelism is necessary, sequential ILU can be used as a subdomain iteration for a domain decomposition preconditioner (section 2.7.3). But when fine grain parallelism is required, a parallel ILU algorithm is necessary. As noted in section 2.7, there are issues with existing parallel ILU methods, such as multi-colouring and level scheduling.

The idea of asynchronous iterations can be used in devising parallel iterations. An asynchronous iterative method to compute incomplete LU (ILU) factorization has been proposed by Chow and Patel [65] and applied to discretized linear partial differential equations (PDEs). They demonstrated their asynchronous ILU factorization for solving the Poisson equation and the linear convection-diffusion equation with promising results. That being said, application of asynchronous iterations specifically to fluid dynamics problems has been rare. Chaotic relaxation was applied to incompressible flows in marine engineering [60]. The solver used a pressure-based method whereby the momentum and mass flux equations are solved in separate steps. The authors demonstrate better strong scaling for chaotic relaxation compared to a Jacobi-preconditioned generalized minimum residual (GMRES) solver for certain problem sizes and a specific solution strategy. However, for the tightly coupled system of PDEs of compressible viscous flow, we show in this chapter that regular chaotic relaxation and asynchronous ILU factorization may be insufficient.

This chapter is concerned with fine-grain parallel subdomain preconditioners for implicit solvers of steady-state compressible flow problems on unstructured grids. By 'subdomain preconditioner', we refer to the iteration applied locally on each subdomain of a domain decomposition preconditioner; the latter is used to parallelize across the nodes of a cluster. Global domain decomposition preconditioning (such as additive Schwartz) is not the subject of study. In fact, all the numerical results presented in this chapter are carried out on one compute node and therefore just one domain.

We propose a point-block variant of Chow and Patel's asynchronous ILU(0) factorization method [65]. We also propose a point-block asynchronous iteration for applying the L and U factors. In this context, Chow et al. [112] showed that while using Jacobi iterations to apply triangular factors, it is advantageous to find blocks in the matrix and invert the diagonal blocks exactly. However, this was done only in the context of symmetric positive-definite matrices.

Some details of the CFD solver are given in section 4.2. We review asynchronous triangular solves (section 4.3.1) and asynchronous ILU factorization (section 4.3.2) preconditioners as potential solutions to the problem of parallel subdomain preconditioning in CFD. Next, we present block versions (section 4.4) of asynchronous iterations and extend convergence proofs to these block versions. We then describe orderings of grid cells for effective solution of viscous problems by asynchronous block ILU(0) preconditioning. Finally, we show some experimental results for both inviscid and viscous compressible flow problems (section 4.6), highlighting the necessity of the block variant and the effect of re-ordering on the performance of asynchronous ILU preconditioners.

4.2 The solver setup

We solve the compressible Navier-Stokes equations expressed in terms of the conserved variables density, momentum per unit volume in each spatial direction and total energy per unit volume as described in chapter 2.

The steady-state equations are discretized in space to obtain a nonlinear system of equations as in equation (2.12). These are solved using pseudo-time stepping, as described in section 2.2. The factor $\omega \in [0.2, 1)$ is a relaxation factor meant to prevent very large relative changes of density or pressure. The pseudo-time step $\Delta\tau$ is determined using a CFL (Courant-Friedrichs-Lewy) number. The CFL number starts at a relatively small prescribed value and is ramped exponentially with respect to the ratio of residual norms from one time step to the next:

$$c_{fl}^{n+1} := c_{fl}^{n} \left(\frac{\|\boldsymbol{r}^n\|}{\|\boldsymbol{r}^{n+1}\|} \right)^{r}, \tag{4.1}$$

where

$$r := \begin{cases} 0.25 & \text{if } \frac{\|\boldsymbol{r}^n\|}{\|\boldsymbol{r}^{n+1}\|} > 1 \\ 0.3 & \text{if } \frac{\|\boldsymbol{r}^n\|}{\|\boldsymbol{r}^{n+1}\|} \le 1 \end{cases} . \tag{4.2}$$

The exponents above were chosen empirically as they worked well for a number of test cases. The CFL number is kept between a starting minimum value and a maximum value which depend on the case.

The FVENS unstructured grid code has been used in this chapter. The spatial discretization is a cell-centred finite volume scheme over two-dimensional unstructured hybrid grids, as described in section 2.1. The Jacobian matrix used in (2.18) is computed ignoring the reconstruction; that is, the first-order inviscid numerical flux and 'thin-layer' first-order viscous flux are linearized to compute the Jacobian.

Similar to the previous chapter, the ordering of the conserved variables $\boldsymbol{w}$ and of residuals in $\boldsymbol{r}$ are such that all $d+2$ variables of one cell are placed consecutively, followed by those of the next cell, and so on. (d is the number of dimensions.) For example in two dimensions,

$$\boldsymbol{w}^T = [\rho_1, \rho v_{x1}, \rho v_{y1}, \rho E_1,\ \rho_2, \rho v_{x2}, \rho v_{y2}, \rho E_2, \ ...] = [\boldsymbol{u}_1^T\, \boldsymbol{u}_2^T \ ... \, \boldsymbol{u}_N^T], \tag{4.3}$$

where subscripts denote cell indices. The residuals are ordered in the same way, with the mass, momentum and energy fluxes for a particular cell placed consecutively. This ordering leads to a block structure of the Jacobian with small dense blocks (figure 4.1).

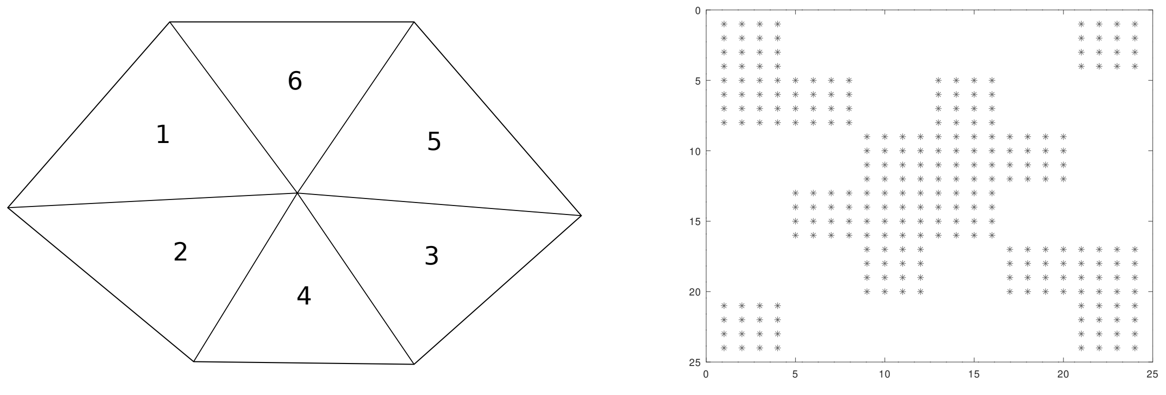


Figure 4.1: A small 2D mesh and its Jacobian non-zero pattern for a cell-centred scheme

Such a logical ordering enables point-block preconditioning, which is advantageous for systems of PDEs because the small dense blocks can be inverted exactly to resolve the local coupling between the different physical variables at one mesh location. Once the required operations on blocks (inversion, matrix-vector products etc.) are available, this allows an extension of iterations for scalar PDEs to effectively deal with systems.

The question of storage layout is independent of the above discussion on point-block matrices. In this chapter, we choose to store the matrix such that the entries in a block are stored contiguously. In case of point-block solvers for unstructured grids on many-core CPU architectures, this is advantageous for cache-locality and could be advantageous for vectorization depending on the block size. The asynchronous block ILU preconditioner presented in section 4.4 can also be used with different layouts suitable for other devices such as GPUs. This is explored in the context of structured grids in later chapters.

It has already been noted that traditional algorithms for computing ILU preconditioners (eg., 2) are sequential and difficult to parallelize. Furthermore, the application of ILU preconditioners depends on the solution of triangular systems, but exact solution of triangular systems by successive substitution also has inherent data dependency and is difficult to effectively parallelize (algorithm 6). While multi-colouring and level-scheduling have been used to parallelize triangular solves, they have the same drawbacks as those described in case of ILU factorization.

Algorithm 6 Forward substitution $\boldsymbol{Lx} = \boldsymbol{b}$

```
Require: $\boldsymbol{L}$ is lower triangular, $\boldsymbol{b}$ is the right-hand side vector
1: $x_1 \leftarrow b_1/L_{11}$
2: for $i$ from 2 to $n$ do
3:     $x_i := (b_i - \sum_{j=1}^{i-1} L_{ij}x_j)/L_{ii}$          ▷ Data dependency!
4: end for
```

A parallel alternative can be derived using the concept of asynchronous iteration. We discuss this in the next section.

4.3 Review of asynchronous iterations in preconditioning

In this section, we describe two known ways of using asynchronous iterations to devise parallel preconditioners. The first is useful for approximately applying ILU-type preconditioners in parallel, while the second builds an approximate ILU preconditioner in parallel.

4.3.1 Asynchronous triangular solves

To solve a lower triangular system, the matrix $\boldsymbol{L}$ can be split as $\boldsymbol{D} + \boldsymbol{E}$, with $\boldsymbol{D}$ diagonal non-singular and $\boldsymbol{E}$ strictly lower triangular. We can solve this by chaotic relaxation ((2.59)). A discussion of asynchronous iterations applied to triangular solves can be found in [91], in which Anzt et al. proved that an asynchronous iteration for a triangular system always converges [91, section 2.1].

We show below the algorithm for an asynchronous 'forward' (lower triangular) solve (algorithm 7).It can be used for both Gauss-Seidel and ILU factorization preconditioner applications. The asynchronous forward triangular solve may be compared with the forward substitution algorithm 6.

Algorithm 7 Asynchronous forward triangular solve $\boldsymbol{Lx} = \boldsymbol{b}$

```
Require: $L$ is lower triangular, $x$ is an initial guess for the solution, $b$ is the right-hand
    side vector
 1: function ASYNC_FORWARD_TRIANGULAR_SOLVE($n_{swp} \in \mathbb{N}$, $L \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $x \in \mathbb{R}^n$)
 2:     Begin parallel region and launch threads
 3:     for integer $i_{swp}$ in $1..n_{swp}$ do
 4:         for integer $i$ in $1..n$ do in parallel dynamically:
 5:             $x_i \leftarrow (b_i - \sum_{j=1}^{i-1} L_{ij} x_j)/L_{ii}$
 6:         end for   (no synchronization)
 7:     end for   (no synchronization)
 8:     End parallel region
 9:     return $x$
10: end function
```

The backward triangular solve is similar, except that the loop starting at line 4 is ordered backwards.

Several 'sweeps' over all unknowns are carried out, where a sweep updates each unknown once. The 'dynamically parallel' loop in these algorithms implies that the work items are not divided among processing elements (cores) *a priori*; rather, new work items are assigned to processing elements as and when the latter become free. Note that the loop over the sweeps is started inside the parallel region, which means each thread keeps count of its own sweeps. There is no synchronization at the end of the loop - even if some threads are computing entries for the first sweep, other threads may start executing work-items for the next sweep. We observe that for such a method to be useful, it must converge sufficiently in a small number of sweeps independent of the number of parallel processing elements.

4.3.2 Asynchronous ILU factorization

Chow and Patel proposed [65] a highly parallel ILU factorization algorithm based on comparing the left and right sides of the equation

$$[\boldsymbol{LU}]_{ij} = A_{ij}. \tag{4.4}$$

Suppose we restrict the sparsity pattern of the computed incomplete LU factorization to an index set S, which necessarily contains the diagonal positions $(j,j)\,\forall\, j \in \{1, 2, ..., n\}$. Let $m := |S|$, the number of non-zeros in the factorization. Then, the above component-wise equality leads to

$$\begin{aligned} L_{ij} &= \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj} \right) / U_{jj}, && \text{if } (i,j) \in S,\ i > j \\ U_{ij} &= A_{ij} - \sum_{k=1}^{i-1} L_{ik} U_{kj} && \text{if } (i,j) \in S,\ i \leq j. \end{aligned} \tag{4.5}$$

This can be written as

$$\boldsymbol{x} = \boldsymbol{g}(\boldsymbol{x}), \tag{4.6}$$

where $\boldsymbol{x} \in \mathbb{R}^m$ is a vector containing all the unknowns L_{ij} and U_{ij} in some order. These lead to a fixed point iteration of the form

$$\boldsymbol{x}^{n+1} = \boldsymbol{g}(\boldsymbol{x}^n). \tag{4.7}$$

An asynchronous form of the above nonlinear fixed-point iteration, as given by Chow and Patel, was shown in algorithm 3. Note that the ordering of the inner parallel loop still matters when the number of processing elements is less than the number of non-zeros in S. This ordering can be chosen for more effective preconditioning rather than to alleviate data dependence. Depending on the local imbalance and number of processors, the resulting iteration could range from nonlinear Jacobi to nonlinear Gauss-Seidel. If one processor is used (resulting in nonlinear Gauss-Seidel) and the loop is ordered in 'Gaussian elimination' ordering (eg. row-wise or column-wise ordering), traditional ILU is recovered. As an initial guess, we use the entries of $\boldsymbol{A}$. In a CFD simulation, the approximate LU factorization from the previous time step can also be used as the initial guess.

We briefly describe the framework used by Chow and Patel. Let an ordering of the unknowns $\boldsymbol{x}$ (the entries of the triangular factors) be given by the bijective map $\alpha : S \rightarrow \{1, 2, 3, ..., m\}$ (where $m = |S|$). $\boldsymbol{x}$ can now be expressed as

$$x_{\alpha(i,j)} = \begin{cases} L_{ij} & \text{if } i > j \\ U_{ij} & \text{if } i \leq j \end{cases}. \tag{4.8}$$

Further, the mapping $\boldsymbol{g} : D \rightarrow \mathbb{R}^m$ can be expressed, for $(i,j) \in S$, as

$$g_{\alpha(i,j)}(\boldsymbol{x}) = \begin{cases} (A_{ij} - \sum_{k=1}^{j-1} x_{\alpha(i,k)} x_{\alpha(k,j)}) / x_{\alpha(j,j)} & \text{if } i > j \\ A_{ij} - \sum_{k=1}^{i-1} x_{\alpha(i,k)} x_{\alpha(k,j)} & \text{if } i \leq j \end{cases}. \tag{4.9}$$

The domain of definition of $\boldsymbol{g}$ is

$$D := \{\boldsymbol{x} \in \mathbb{R}^m \,|\, x_{\alpha(j,j)} \neq 0 \,\forall\, j \in \{1, 2, ..., m\}\}. \tag{4.10}$$

We now select any one of the 'Gaussian elimination orderings' for α, such as

$$(1,1) \prec (1,2) \prec ... \prec (1,n) \prec (2,1) \prec (2,2) \prec ... \prec (n,n-1) \prec (n,n). \tag{4.11}$$

This is a row-major ordering; other possible Gaussian elimination orderings are column-major ordering and the partial ordering chosen by Chow and Patel [65].

Chow and Patel proved local convergence of the asynchronous ILU iteration; that is, a fixed point of $\boldsymbol{g}$ is a point of attraction of the iteration. We provide an outline of the proof below; the full proof may be found in their paper [65, theorems 3.3, 3.4, 3.5]

Theorem 3 (Chow, Patel)**.** *If $\boldsymbol{g}$ has a unique fixed point, it is a point of attraction of iteration* (4.7)*.*

Proof. It can be shown from its structure that the Jacobian matrix $\boldsymbol{g}'$ is always strictly lower triangular in Gaussian elimination ordering, and thus has zero spectral radius. Then by theorem 2, the fixed point is a point of attraction of the asynchronous iteration. □

In the event of asynchronous updates, some diagonal entries may sometimes be set to zero. To take this into account, Chow and Patel define a 'modified Jacobi-type iteration' corresponding to the iteration in equation (4.5), in which whenever a zero diagonal entry is encountered, it is replaced by an arbitrary non-zero value. They showed that a synchronized modified Jacobi-type iteration corresponding to equation (4.7) converges in at most $m = |\mathcal{S}|$ iterations irrespective of the initial guess. We state the theorem and a summary of the proof [65, theorems 3.6, 3.7].

Theorem 4 (Chow, Patel)**.** *If a fixed point $\boldsymbol{x}_*$ of the mapping $\boldsymbol{g}$ exists, then the fixed point is unique and the modified iteration corresponding to* (4.7) *converges to $\boldsymbol{x}_*$ in at most $|\mathcal{S}|$ iterations.*

Proof. For the uniqueness of the fixed point of $\boldsymbol{g}$, Chow and Patel consider [65] a Gaussian elimination ordering of the unknowns. Then in a sequential iteration, we can compute the fixed point by successive substitution. If we encounter a zero diagonal, the process breaks down and no fixed point exists. If not, the computed fixed point is the unique fixed point.

Further, it can be shown that in the parallel iteration, by the kth iteration, the kth entry of the sparsity pattern will have achieved its final value. Since we start with an arbitrary initial guess, we may encounter zero diagonal entries. In that case, we can replace any zero diagonal entries with an arbitrary non-zero value. Thus, by the mth iteration, where $m = |\mathcal{S}|$, all entries will achieve their final values. □

Algorithm 8 Asynchronous ILU factorization

```
Require: Assign initial values to $L_{ij}$ and $U_{ij}$. Let $S$ be a set of non-zero $i, j$ indices.
 1: Begin parallel region
 2: for $i_{swp}$ in $1..n_{swp}$ do
 3:     for $i$ in $1..n$ do in parallel dynamically:
 4:         for $j$ in $1..n$, $(i, j) \in S$ do
 5:             if $i > j$ then
 6:                 $L_{ij} \leftarrow \left( A_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj} \right) / U_{jj}$
 7:             else
 8:                 $U_{ij} \leftarrow A_{ij} - \sum_{k=1}^{i-1} L_{ik} U_{kj}$
 9:             end if
10:         end for
11:     end for  (no synchronization)
12: end for  (no synchronization)
13: End parallel region
```

In our implementation (algorithm 8), the loop over entries of the matrix is always ordered in the row-major ordering. Each work-item consists of computing all the entries in one row of the matrix. In OpenMP, a number of consecutively ordered work-items (determined by the loop ordering) are placed in one 'chunk'. In case of dynamic scheduling, when a free thread requests work, it is assigned the next chunk of waiting work-items [82, section 2.7.1]. For our problem, each chunk consists of a number of consecutive rows of the matrix. Thus, the row-major ordering of the loop means that all entries in the rows in one chunk are computed in a Gaussian elimination ordering (the row-major ordering) with respect to entries in that chunk.

To apply the ILU preconditioner (ie., to solve $\boldsymbol{LUx} = \boldsymbol{b}$ given $\boldsymbol{L}$ and $\boldsymbol{U}$) in parallel, Chow and Patel used synchronous Jacobi iterations to solve the triangular systems $\boldsymbol{Ly} = \boldsymbol{b}$ and $\boldsymbol{Ux} = \boldsymbol{y}$ [65]. In our work, we use asynchronous triangular solves, as described in section 4.3.1. Thus, both the factorization (build) of the preconditioner and the application of the preconditioner are asynchronous. We therefore refer to 'build sweeps' and 'apply sweeps' as the number of asynchronous sweeps used to build and apply the preconditioner respectively. Typically, a preconditioner is built once for a linear system but is applied once every linear solver iteration. Many linear iterations are normally performed in order to achieve some level of convergence of the linear problem. Thus, greater numbers of application sweeps can have a larger negative impact on performance than greater numbers of build sweeps.

4.4 Asynchronous block preconditioners

As we discussed in section 3.2, our Jacobian matrix has a natural block structure with small dense blocks. For the two-dimensional problems shown in this chapter, the block size b is 4. For three-dimensional problems, it is 5. We would like to take advantage of this fact in asynchronous preconditioning. As we show later, asynchronous block iterations are generally more robust and converge in fewer iterations than their scalar counterparts. Further, blocking may improve cache utilization and vectorization.

4.4.1 Asynchronous block triangular solves

We can extend the asynchronous triangular solves to block triangular solves in a straightforward manner. We present the forward block triangular solve algorithm as an example. Assume that the blocks are square $b \times b$ and that the matrix size n is divisible by the block size b. $\boldsymbol{x}_{a:b}$ denotes the subvector $[x_a, x_{a+1}, ..., x_{b-1}]^T$ of $\boldsymbol{x}$ for $a < b$, so that $\boldsymbol{x}_{(i-1)b+1:ib+1}$ denotes the i-th subvector of length b. Similarly, $\boldsymbol{L}_{(i-1)b+1:ib+1,\,(j-1)b+1:jb+1}$ denotes the (i,j)th $b \times b$ sub-block of the matrix $\boldsymbol{L}$. Let us simplify this cumbersome notation by defining $\boldsymbol{x}_i := \boldsymbol{x}_{(i-1)b+1:ib+1}$ and $\boldsymbol{L}_{ij} := \boldsymbol{L}_{(i-1)b+1:ib+1,\ (j-1)b+1:jb+1}$.

Algorithm 9 Asynchronous forward block-triangular solve $\boldsymbol{Lx} = \boldsymbol{b}$

```
Require: $\boldsymbol{L}$ is lower block triangular with nonsingular diagonal blocks, $\boldsymbol{x}$ is an initial guess
    for the solution, $\boldsymbol{b}$ is the right-hand side vector, $b$ is the size of square blocks in $\boldsymbol{L}$
 1: function ASYNC_FORWARD_BLOCK_TRIANGULAR_SOLVE($n_{swp} \in \mathbb{N}$, $\boldsymbol{L} \in \mathbb{R}^{n\times n}$, $\boldsymbol{b} \in \mathbb{R}^n$,
    $\boldsymbol{x} \in \mathbb{R}^n$)
 2:     Begin parallel region and launch threads
 3:     for integer $i_{swp}$ in $1..n_{swp}$ do
 4:         for integer $i$ in $1..n/b$ do in parallel dynamically:
 5:             $\boldsymbol{x}_i \leftarrow \boldsymbol{L}_{ii}^{-1}(\boldsymbol{b}_i - \sum_{j=1}^{i-1} \boldsymbol{L}_{ij}\boldsymbol{x}_j)$
 6:         end for   no synchronization
 7:     end for
 8:     End parallel region
 9:     return $\boldsymbol{x}$
10: end function
```

Following the discussion on (scalar) triangular solves in [91, section 2.1], we can easily extend the proof of convergence to this asynchronous block triangular iteration.

Theorem 5. *The chaotic relaxation for the system $\boldsymbol{Lx} = \boldsymbol{b}$ defined by the splitting $\boldsymbol{D} + \boldsymbol{E}$, with $\boldsymbol{D}$ nonsingular block diagonal and $\boldsymbol{E}$ strictly lower block triangular, converges.*

Proof. The Jacobi iteration for the splitting can be written as

$$\boldsymbol{x}^{j+1} = \boldsymbol{D}^{-1}\boldsymbol{b} - \boldsymbol{D}^{-1}\boldsymbol{E}\boldsymbol{x}^{j}. \tag{4.12}$$

Since $\boldsymbol{E}$ is strictly lower block triangular and $\boldsymbol{D}^{-1}$ is block diagonal, $\boldsymbol{D}^{-1}\boldsymbol{E}$ is also strictly lower block triangular. Thus, $\rho(|\boldsymbol{D}^{-1}\boldsymbol{E}|) = \rho(\boldsymbol{D}^{-1}\boldsymbol{E}) = 0$. By theorem 1, the chaotic relaxation (2.59) for this splitting converges. □

Theorem 6. *The Jacobi iteration* (4.12) *for the lower block triangular linear system $\boldsymbol{Lx} = \boldsymbol{b}$ converges to the solution $\boldsymbol{L}^{-1}\boldsymbol{b}$ in at most n/b iterations irrespective of the initial guess and order of updates, where n is the number of unknowns in the system and b is the block size.*

Proof. We proceed to prove this theorem by induction. Since the iteration matrix $\boldsymbol{D}^{-1}\boldsymbol{E}$ is strictly block lower triangular, the unknowns corresponding to the first cell $\boldsymbol{x}_1$ do not depend on any other entries of $\boldsymbol{x}$. Thus, irrespective of the order of updates and the initial guess, after the first iteration of equation (4.12), the b unknowns corresponding to the first cell attain their exact values $\boldsymbol{x}_1^1 = (\boldsymbol{D}^{-1}\boldsymbol{b})_1$.

Let us assume that the first $j-1$ cells' unknowns have attained their exact values by the end of the $(j-1)$th iteration. The unknowns of the jth cell $\boldsymbol{x}_j$ depend only on those of the cells 1,2,...,$j-1$ due to the strictly block lower triangular nature of the iteration matrix. Therefore, the jth cell attains its exact values by the end of the jth iteration.

Thus, by induction, all n/b cells' unknowns attain their exact values by the end of the (n/b)th iteration. □

We now define a 'block-asynchronous' iteration. Let the vector function $\boldsymbol{\psi} : D \subset \mathbb{R}^n \to \mathbb{R}^n$ be any iteration. For the purpose of forward triangular solves, it is defined by $\boldsymbol{\psi}(\boldsymbol{x}) := \boldsymbol{D}^{-1}\boldsymbol{b} - \boldsymbol{D}^{-1}\boldsymbol{E}\boldsymbol{x}$. The definition is inspired by Frommer and Szyld's definition of asynchronous iteration [86, definition 2.2]. Again, ψ_i denotes the ith function while $\boldsymbol{\psi}_i$ (in bold face) denotes the sub-vector of b functions in the ith block.

Definition 1. *A block-asynchronous iteration is of the form*

$$\boldsymbol{x}_i^{j+1} = \begin{cases} \boldsymbol{x}_i^{j} & \text{if } i \neq u(j) \\ \boldsymbol{\psi}_i(x_1^{j-s_1(j)}, x_2^{j-s_2(j)}, ..., x_n^{j-s_n(j)}) & \text{if } i = u(j) \end{cases} \tag{4.13}$$

where $s_\alpha : \mathbb{N} \to \mathbb{N}$, $\alpha \in \{1, 2, ..., n\}$ are the shift functions and $u : \mathbb{N} \to \{1, 2, ..., n/b\}$ is the update function. (n/b is an integer equal to the number of cells.)

As before, the shift functions are assumed to satisfy criterion (2.60), while the update function satisfies a block version of (2.61):

$$\text{Given } i \in \{1, 2, ..., n/b\} \text{ and } j \in \mathbb{N}, \quad \exists\, l > j \text{ s.t. } u(l) = i. \tag{4.14}$$

Recall that a step is defined by a single read of (all the required entries from) the unknown vector $\boldsymbol{x}$ by one thread. This is followed by the update of a point-block of entries by the thread. Hence, the range of the update function consists of integers between 1 and the number of blocks. This allows exact treatment of dependencies of variables within the point-block. However, the individual entries used for computing the iterate can come from different previous updates. Therefore there is a shift function for every entry, instead of every block, of the unknown vector. This block-asynchronous iteration is equivalent to Frommer and Szyld's asynchronous iteration [86, definition 2.2] in the case when the set of indices updated in each step exactly corresponds to one point-block. As such, the established asynchronous convergence theory given by Frommer and Szyld [86] still holds.

It must be mentioned that Anzt et al. [113] used the term 'block-asynchronous' to describe a different iteration that they proposed. In their approach to linear asynchronous iterations, the block aspect accounts for features of the hardware or the programming model (thread blocks in CUDA, in their case). They perform several Jacobi iterations within a block to invert them approximately, while the coupling between the blocks has asynchronous character. In our case, a diagonal block is inverted exactly and sequentially by one processing element, because our blocks are smaller and dense. As mentioned before, the diagonal blocks represent coupling between different physical quantities at one mesh location. The above block-asynchronous iteration is also applicable to non-linear iterations, as required by the iterative block-ILU method. The similarity between our block-asynchronous framework and the iteration of Anzt et al. is that the coupling between blocks is asynchronous.

Theorem 7. *The block-asynchronous linear iteration* (4.13) *for solving the lower block triangular system converges in a finite number of steps to the solution* $\boldsymbol{L}^{-1}\boldsymbol{b}$.

Proof. Because of the strictly lower block-triangular nature of the iteration matrix, the first cell's unknowns (the b-block $\boldsymbol{x}_1$) do not depend on any other unknowns. Whenever this block is updated for the first time, it attains its exact values.

Next, let us assume that at the end of the k^{th} step, all cells up to the j^{th} cell have attained their exact values. Since $\hat{s}$ is the upper bound on the delays (by condition (2.60)), starting latest at the $k + \hat{s} + 1$ step, the exact values for all cells up to the j^{th} cell will be used for any update that requires them. By condition (4.14), we know that there exists some step $l \geq k + \hat{s} + 1$ for which $u(l) = j + 1$. That is, there exists some step l at which the unknowns of the $j+1^{\text{th}}$ cell's block is updated using the exact values for all the cells that it depends on. Thus the $j+1^{\text{th}}$ cell attains its exact values after a finite number of steps.

Hence, by induction, all the unknowns equal their fixed-point values after some finite number of steps. □

Even though asynchronous iteration does not have global iterations, our implementation is carried out using several 'sweeps' of a parallel OpenMP loop over all the unknowns. If we use n_s sweeps, every unknown is updated exactly n_s times.

Since $\boldsymbol{Ly} = \boldsymbol{b}$ and $\boldsymbol{Ux} = \boldsymbol{y}$ need to be solved every time the preconditioner is applied in a solver, the above bounds on the number of fixed-point iterations or steps are only of academic interest. We hope to be able to use only a few sweeps of asynchronous updates to approximately solve the triangular systems.

4.4.2 Asynchronous block ILU preconditioner

Using the equation $(\boldsymbol{LU})_{ij} = \boldsymbol{A}_{ij}$, we can define a block LU factorization. Suppose S_B is a set of block indices which includes all diagonal blocks; it will be the block sparsity pattern imposed on the computed block LU factors. The block ILU factorization is computed as

$$\begin{aligned} \boldsymbol{L}_{ij} &= \left(\boldsymbol{A}_{ij} - \sum_{k=1}^{j-1} \boldsymbol{L}_{ik}\boldsymbol{U}_{kj}\right)\boldsymbol{U}_{jj}^{-1}, && \text{if } (i,j) \in S_B,\ i > j \\ \boldsymbol{U}_{ij} &= \boldsymbol{A}_{ij} - \sum_{k=1}^{i-1} \boldsymbol{L}_{ik}\boldsymbol{U}_{kj} && \text{if } (i,j) \in S_B,\ i \le j \end{aligned} \tag{4.15}$$

where subscripts denote block-indices. Note that the diagonal blocks of the factorization, $\boldsymbol{U}_{jj}$, are inverted exactly using dense 4×4 or 5×5 Gaussian elimination with partial pivoting.

The unknowns are $[\boldsymbol{L}_{ij}]_{kl}$, $(i,j) \in S_B$ s.t. $i > j$ and $[\boldsymbol{U}_{ij}]_{kl}$, $(i,j) \in S_B$ s.t. $i \le j$, for $1 \le k,l \le b$. Here, $[\boldsymbol{L}_{ij}]_{kl}$ denotes the (k,l) entry of the sub-matrix $\boldsymbol{L}_{ij}$. For analyzing block preconditioners, we introduce an ordering of the unknowns by the bijective map

$$\beta : S_B \times \{1,2,...,b\} \times \{1,2,...,b\} \to \{1,2,3,...,m\}, \tag{4.16}$$

where $m = |S_B|b^2$, the total number of non-zeros in the block-ILU factorization. Clearly, we assume that the number of non-zero entries is always a multiple of b^2. We can then write a vector of all the unknowns and call it $\boldsymbol{x}$, which is ordered as (compare with α defined in

(4.8))

$$x_{\beta(i,j,k,l)} = \begin{cases} [\boldsymbol{L}_{ij}]_{kl} & \text{if } i > j \\ [\boldsymbol{U}_{ij}]_{kl} & \text{if } i \leq j \end{cases}. \tag{4.17}$$

We can also define a block-ordering

$$\beta_B : S_B \to \{1, 2, ..., m/b^2\}, \tag{4.18}$$

which is bijective and related to the ordering β by $\beta_B(i,j) = \beta(i,j,b,b)/b^2$.

Let us define $\boldsymbol{h} : D_B \to \mathbb{R}^m$ (where $D_B \subset \mathbb{R}^m$) to be the function that represents the right-hand-side of equation (4.15). Denote the matrix $\boldsymbol{x}_{\beta(i,j,:,:)}$ by $\boldsymbol{X}_{ij} \in \mathbb{R}^{b\times b}$ for $(i,j) \in S_B$. Note that ':' at an indexing position denotes the entire range of indices possible for that position; here it denotes all integers from 1 to b. Similarly, we denote $\boldsymbol{h}_{\beta(i,j,:,:)} : D_B \to \mathbb{R}^{b\times b}$ by $\boldsymbol{H}_{ij}$ for $(i,j) \in S_B$. Then we can express the domain of definition of $\boldsymbol{h}$ as

$$D_B := \{\boldsymbol{x} \in \mathbb{R}^m \,|\, \boldsymbol{X}_{jj} \text{ is nonsingular } \forall\, j \in \{1, 2, ..., m/b^2\}\}. \tag{4.19}$$

With this, the mapping $\boldsymbol{h} : D_B \to \mathbb{R}^m$ can be expressed as

$$\boldsymbol{H}_{ij}(\boldsymbol{x}) := \boldsymbol{h}_{\beta(i,j,:,:)}(\boldsymbol{x}) = \begin{cases} (\boldsymbol{A}_{ij} - \sum_{k=1}^{j-1} \boldsymbol{X}_{ik}\boldsymbol{X}_{kj})\boldsymbol{X}_{jj}^{-1} & \text{if } i > j \\ \boldsymbol{A}_{ij} - \sum_{k=1}^{i-1} \boldsymbol{X}_{ik}\boldsymbol{X}_{kj} & \text{if } i \leq j \end{cases}. \tag{4.20}$$

Now the fixed-point of the ILU iteration can be written as

$$\boldsymbol{x} = \boldsymbol{h}(\boldsymbol{x}). \tag{4.21}$$

The synchronized Jacobi-type fixed-point iteration can be expressed as

$$\boldsymbol{x}^{n+1} = \boldsymbol{h}(\boldsymbol{x}^n), \tag{4.22}$$

while the block-asynchronous iteration is expressed as

$$\boldsymbol{X}_{ij}^{k+1} = \begin{cases} \boldsymbol{H}_{ij}(x_1^{k-s_1(k)}, x_2^{k-s_2(k)}, ..., x_n^{k-s_m(k)}), & \beta_B(i,j) = u(k) \\ \boldsymbol{X}_{ij}^k & \beta_B(i,j) \neq u(k) \end{cases}, \tag{4.23}$$

where $s_j : \mathbb{N} \to \mathbb{N}$ (for $1 \leq j \leq m$) are the shift functions and $u : \mathbb{N} \to \{1, 2, ..., m/b^2\}$ is the update function. Here each step refers to the update of one $b \times b$ non-zero block $\boldsymbol{X}_{ij}$. The shift and update functions are assumed to have the following properties, corresponding to the usual properties of asynchronous iterations.

$$\exists\, \hat{s} \in \mathbb{N} \text{ s.t. } 0 \leq s_i(k) \leq \min\{k-1, \hat{s}\} \quad \forall i \in \{1, 2, ...m\},\, k \in \mathbb{N}. \tag{4.24}$$

$$\text{Given } i \in \{1, 2, ..., m/b^2\} \text{ and } k \in \mathbb{N},\ \exists\, l > k \text{ s.t. } u(l) = i. \tag{4.25}$$

Let us select a 'block Gaussian elimination' ordering for β_B as the same as the Gaussian elimination ordering in (4.11), except that the indices now correspond to block indices.

$$(1,1) \prec (1,2) \prec ... \prec (1, \frac{n}{b}) \prec (2,1) \prec (2,2) \prec ... \prec (\frac{n}{b}, \frac{n}{b} - 1) \prec (\frac{n}{b}, \frac{n}{b}), \quad (i,j) \in S_B \tag{4.26}$$

(n is the dimension of the matrix $\boldsymbol{A}$). This defines a partial ordering for β, where the ordering within each block is unspecified.

We can now extend the convergence analysis given by Chow and Patel for the asynchronous ILU process to the asynchronous block-ILU algorithm.

Lemma 1. *$\boldsymbol{h}(\boldsymbol{x})$ is differentiable for all $\boldsymbol{x} \in D_B$.*

Proof. From the definition (4.20) we see that each block $\boldsymbol{H}_{ij}$ is a rational matrix function of the $\boldsymbol{X}_{kl}$. This implies that each h_i is a rational function of the x_j. Thus, $\boldsymbol{h} \in [C^1(D_B)]^m$, that is, $\boldsymbol{h}$ is continuously differentiable in its domain of definition. □

Lemma 2. *When the unknowns $\boldsymbol{x}$ and the mappings $\boldsymbol{h}$ are ordered in the block Gaussian elimination ordering* (4.26)*, $\boldsymbol{h}$ has strictly lower-triangular structure, ie., $h_k(\boldsymbol{x})$ depends only on $\{x_1, x_2, ..., x_{k-1}\}$. Thus the Jacobian $\boldsymbol{h}'(\boldsymbol{x})$ is strictly lower triangular $\forall\, \boldsymbol{x} \in D_B$.*

Proof. From equation (4.20), we see that $\boldsymbol{H}_{ij}$ depends on $\boldsymbol{X}_{pq}$ only if

$$(p,q) \in \{(\gamma,\delta) \in S_B |\, \gamma < i,\, \delta = j\} \cup \{(\gamma,\delta) \in S_B |\, \gamma = i,\, \delta < j\}. \tag{4.27}$$

This implies that $\boldsymbol{H}_{ij}$ depends on a subset of the blocks $\{\boldsymbol{X}_{pq}\,|\, p \leq i, q \leq j,\, (p,q) \neq (i,j)\}$ preceding it in the ordering (4.26). This means that for indices in the (i,j)th block, ie., for indices k such that $\beta(i,j,1,1) \leq k \leq \beta(i,j,b,b)$, $h_k(\boldsymbol{x})$ only depends on a subset of $\{x_1, x_2, ...x_{b^2[(k-1)/b^2]}\} \subseteq \{x_1, x_2, ...x_{k-1}\}$. Here, [.] is the greatest integer function. Thus $\boldsymbol{h}$ has not only a strictly lower triangular structure but a strictly lower block triangular structure.

Therefore, each h_k is a function of x_j only for $j < k$:

$$h_k(\boldsymbol{x}) = h_k(x_1, x_2, ..., x_{k-1}) \implies \frac{\partial h_k}{\partial x_l} = 0 \quad \forall\, l \geq k, \tag{4.28}$$

which means the Jacobian $\boldsymbol{h}'(\boldsymbol{x})$ is strictly lower triangular $\forall\, \boldsymbol{x} \in D_B$. □

Theorem 8. *The synchronous nonlinear fixed-point iteration $\boldsymbol{x}^{p+1} = \boldsymbol{h}(\boldsymbol{x}^p)$ is locally convergent, that is, if $\boldsymbol{x}_*$ is a fixed point of $\boldsymbol{h}$, it is a point of attraction of the synchronous iteration.*

Proof. By lemmas 1 and 2, $\boldsymbol{h}$ is differentiable and all eigenvalues of $\boldsymbol{h}'$ are zero. Since eigenvalues are unaffected by symmetric permutations of the matrix, this holds for any reordering of the equations and unknowns.

Thus, the spectral radius of $\boldsymbol{h}'$ is zero. Now from the Ostrowski theorem (theorem 10.1.3 of Ortega and Rheinboldt [114]),if $\boldsymbol{x}_*$ is a fixed point of $\boldsymbol{h}$, it is a point of attraction. □

Theorem 9. *Under the assumptions* (4.24) *and* (4.25)*, the asynchronous iteration* (4.23) *is locally convergent, ie., if $\boldsymbol{x}_*$ is a fixed point of the asynchronous iteration, it is a point of attraction of the iteration.*

Proof. In lemmas 1 and 2, we showed that $\boldsymbol{h}$ is differentiable and all eigenvalues of $\boldsymbol{h}'$ are zero. Therefore,

$$\rho(|\boldsymbol{h}'(\boldsymbol{x}_*)|) = 0. \tag{4.29}$$

Note that for a matrix $\boldsymbol{A}$, $|\boldsymbol{A}|$ denotes the matrix of absolute values of the corresponding entries.

The conditions (4.24) and (4.25) are easily seen to satisfy the requirements of asynchronous iterations given by Frommer and Szyld ([86], definition 2.2). Thus, according to their theorem [86, theorem 4.4], $\boldsymbol{x}_*$ is a point of attraction of the asynchronous iteration (4.23). □

Similar to Chow and Patels's work [65], global convergence of the synchronized block ILU iteration (4.22) can be proved.

Theorem 10. *If a fixed point of the function $\boldsymbol{h}$ exists, it is unique.*

Proof. We assume the Gaussian elimination ordering (4.26). This leads to no loss of generality because $\boldsymbol{x}_*$ is a fixed point of $\boldsymbol{h}$ in one ordering if and only if it is a fixed point in another ordering.

In this ordering, the solution can be found in one iteration through forward substitution due to the strictly lower triangular nature of $\boldsymbol{h}$. The forward substitution completes if no $\boldsymbol{X}_{jj} = \boldsymbol{U}_{jj}$ is set to a singular matrix, in which case the solution is the unique fixed point because forward substitution gives a unique solution. If any of these diagonal blocks is set to a singular matrix, no fixed point exists. □

In analogy with the scalar ILU theory, a modified iteration corresponding to the sequential or asynchronous iteration can be defined as a similar iteration except that when a diagonal block $\boldsymbol{X}_{jj}$ becomes singular, it is replaced by an arbitrary non-singular matrix.

Theorem 11. *If $\boldsymbol{h}$ has a fixed point, the modified Jacobi-type iteration corresponding to* (4.22) *converges in at most m/b^2 iterations from any initial guess $\boldsymbol{x}^0$. (m is the number of nonzero entries in all blocks in the sparsity pattern S_B and b is the block size.)*

Proof. It can be observed from (4.20) that the first block $\boldsymbol{X}^1_{1,1}$ or $\boldsymbol{U}^1_{1,1} := \boldsymbol{x}^1_{\beta(1,1,:,:)}$ of the first iterate does not depend on $\boldsymbol{x}$ at all. Thus the b^2 entries in $\boldsymbol{X}_{1,1}$ attain their exact fixed-point values after the first iteration. Other blocks may potentially be updated in some arbitrary manner, though the modified iteration ensures that diagonal blocks remain non-singular and the iteration does not break down.

Suppose the next b^2 entries (x_{b^2} to x_{2b^2}) in the ordering (4.26) correspond to the block $(k, l) \in S_B$. Then $\boldsymbol{X}_{k,l}$ depends only on $\boldsymbol{X}_{1,1}$ (at most), therefore its entries attain their fixed-point values after iteration 2, and retain those values after further iterations. Thus, the first $2b^2$ entries of $\boldsymbol{x}$ attain their correct fixed-point values at the completion of iteration 2.

If all blocks up to the p^{th} non-zero block ($p \geq 1$) in the Gaussian elimination ordering (4.26) have attained their final values by the p^{th} iteration, the $p + 1^{\text{th}}$ non-zero block in that ordering depends only on the known blocks, due to the strictly lower triangular nature of $\boldsymbol{h}$. Thus, the $p + 1^{\text{th}}$ non-zero block attains its exact value by the $p + 1^{\text{th}}$ iteration.

Continuing in this manner to the last block in the ordering (4.26), we conclude by induction that all m entries of $\boldsymbol{x}$ attain their fixed-point values at the completion of iteration m/b^2. □

We note that the maximum number of global iterations needed for modified Jacobi-type block ILU is smaller than that for the original modified Jacobi-type scalar ILU iteration.

For the asynchronous modified iteration, we cannot make claims about the number of global iterations it takes for convergence because there are no global iterations. However, because of the property (4.25) of the asynchronous block ILU iteration, for each block there are steps at which it continues to get updated as we perform more and more asynchronous steps. By an argument similar to the proof of theorem 7, we can conclude that the asynchronous modified iteration will converge in a finite number of steps.

To parallelize the application of the asynchronous ILU preconditioner, we use asynchronous triangular solves as shown in algorithm 7, and for the asynchronous block ILU preconditioner we use asynchronous block-triangular solves as illustrated in algorithm 9. We refer to the number of asynchronous iterations used to approximately solve each (block) triangular system as the number of ‘application sweeps’ or ‘apply sweeps’.

As mentioned earlier, OpenMP divides the work items into chunks. On a CPU, whenever an idle thread is assigned work, it is assigned an entire chunk which is then processed sequentially by the thread. Therefore an interesting detail to note is that we can expect a parallel (modified) fixed-point iteration to converge in at most the same number of sweeps as the number of chunks, as long as the iteration function is strictly lower triangular in the loop ordering. The reasoning is that once the entries in all chunks before the p^{th} chunk attain their final values, all entries in the $p+1^{\text{th}}$ chunk attain their final values in the next sweep. The number of chunks is usually much less than the number of non-zero blocks, because a chunk usually contains a substantial number of work-items for performance reasons.

4.5 Orderings of mesh cells

As we will see in the results section, asynchronous (block) ILU is quite sensitive to the ordering of mesh cells. We have used the following topological orderings:

- Reverse Cuthill-McKee (RCM) [44, 115]. This is a common ordering used to solve PDEs with ILU preconditioning; it is a ‘level-set’ ordering that aims to reduce the bandwidth of the matrix.

- One-way dissection (1WD) [116]. This algorithm aims to order the grid by recursively introducing separators and sub-dividing the grid.

The implementations available in PETSc [99] were used to achieve these orderings. For efficiency, we reorder the grid itself in a pre-processing stage and avoid reordering of matrices during the nonlinear solve.

In addition to the above-mentioned algorithms, we also introduce the following algorithms which are suited to viscous fluid dynamic simulations.

4.5.1 Line ordering

Line solvers are well established in CFD [117, 118, 119]. Meshes for viscous flows are usually generated with high grid-stretching in the boundary layer near the body being studied (figure 4.2), so as to capture the highly anisotropic flow profiles in that region efficiently. Thus, the wall-normal direction, having a high density of points, is one of strong coupling, while the the wall-tangent direction is more loosely coupled. Mavriplis [119] used an algorithm for finding lines of strong coupling in the grid, based on the physical locations of the grid points. In that work, lines made up of grid vertices are found for a vertex-centred discretization. Since we use a cell-centred discretization, we apply the same algorithm to cell-centres to find lines of cells which are tightly coupled. Note that only cells that lie in regions of high anisotropy are incorporated into lines. Thus lines start at boundary cells and continue towards the interior only while a local anisotropy threshold is met. These truncated lines are sometimes called linelets in the literature (eg. [118]). For simplicity, unlike Mavriplis, we only consider lines near boundaries and do not consider shear layers in the interior of the flow; our lines are limited to the boundary layer region. In this way, the grid is divided into lines of anisotropic cells and individual isotropic cells.

Once lines are found, the grid is reordered so that cells that make up a line are contiguous in the ordering. The Jacobian matrix can then be broken up into line blocks (which are block tridiagonal, representing the one-dimensional strong coupling along the line), point (cell) blocks for cells not belonging to any line, and the blocks coupling them. In a traditional line solver, a block-Jacobi iteration is applied. The individual line blocks are inverted exactly using Thomas' algorithm, and point blocks are also easily inverted exactly, but the couplings between neighbouring lines and isotropic points are not part of the preconditioner.

In our code, we reorder the grid such that cells belonging to a line are contiguous, and then apply sequential or asynchronous ILU preconditioning to this reordering. Note that a sequential ILU preconditioner would invert the line-blocks exactly. Moreover, it would more accurately relax the coupling between the lines and isotropic cells, compared to a traditional line solver.

4.5.2 Hybrid line-X ordering

The line ordering described above can be combined with a topological ordering to produce potentially even better orderings. To do this, we first find lines of anisotropic cells and order the mesh according to the line ordering described above. We then define a graph whose vertices are the lines and the individual isotropic cells; that is, each graph vertex represents either a line or an isotropic cell. If for a pair of lines, one of the lines contains a cell that neighbours at least one cell in the other line, those two lines are assumed connected directly in the graph. Similarly, isotropic cells which neighbour a cell belonging to a line are considered connected to that line in the graph. Two isotropic cells which are neighbours in the original grid are also connected in the graph (figure 4.3).

This graph is then ordered by a topological ordering such as RCM or 1WD, to produce line-RCM or line-1WD ordering respectively. Note that cells in a line remain contiguous in the final ordering - thus hybrid line-X orderings are also line orderings. We will see that

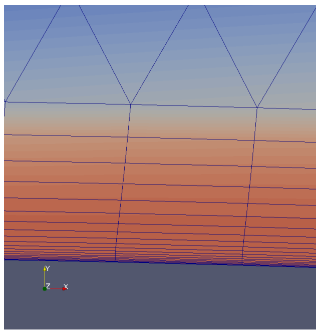


Figure 4.2: Anisotropic cells
near the boundary

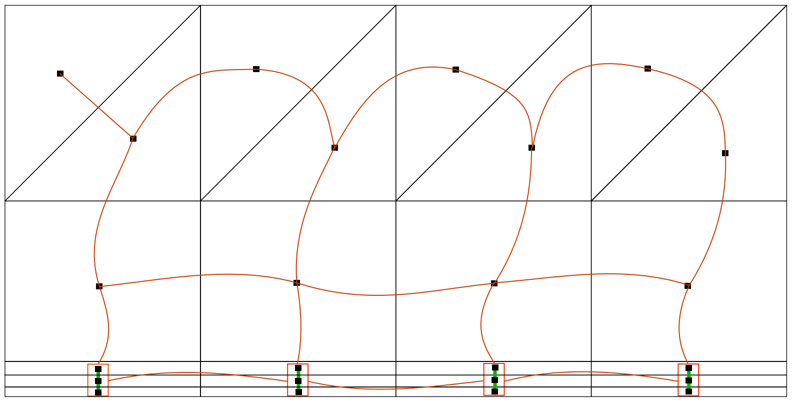


Figure 4.3: The graph used to generate hybrid line-X orderings. Black lines represented mesh edges, and red boxes represent vertices of the graph and red curves represent edges of the graph. Notice that cells in the bottom three layers of anisotropic cells are connected vertically to form lines.

hybrid line-X orderings can work very well with asynchronous point-block ILU preconditioning.

4.6 Numerical results

We provide some experimental (numerical) results which demonstrate the performance of asynchronous block ILU preconditioning, albeit for simple cases. In what follows, 'ILU' will always denote fixed-pattern incomplete LU factorization where the non-zero pattern of the L and U factors is the same as the original matrix; that is, no fill-in is allowed. 'ABILU(x, y)' will denote our asynchronous block ILU(0) preconditioner with x asynchronous sweeps used to build the block ILU(0) preconditioner and y asynchronous sweeps to apply triangular solves in every linear solver iteration.

We are interested in investigating the improvement in convergence and/or parallel scaling brought about by the use of the asynchronous block ILU preconditioner when compared to the Chow-Patel preconditioner for two test cases, one with inviscid flow and the other with viscous laminar flow. Secondly, we investigate the impact of the grid ordering on the convergence and parallel scaling of the different variants of the asynchronous ILU preconditioner. Thirdly, in their paper [65], Chow and Patel suggest scaling the original matrix symmetrically (using the same row and column scaling factors), resulting in unit diagonal entries, before running asynchronous ILU iterations on it. We look at the effect of symmetric scaling on our two test cases. Finally, please note that for the tests in sections 4.6.2 and 4.6.3, except figures 4.4 and 4.8, we averaged the data over three runs. For figures 4.4 and

4.8, the results were averaged over five runs. We observed that the deviation is small (less than 3.3% in all cases).

The following parameters are used for all the runs.

- Convergence tolerance for the nonlinear problem is a relative drop of 10^{-6} in the energy residual. The maximum CFL number (with regard to equation (4.1)) is 10,000.

- The FGMRES [120] solver from the Portable, Extensible Toolkit for Scientific Computing (PETSc) [121] is used. The restart length is set to 30. At every nonlinear iteration (pseudo-time step), linear solver convergence criterion is a residual ($\|\boldsymbol{Ax}-\boldsymbol{b}\|_2$) reduction by one order of magnitude, but only up to a maximum of 60 FGMRES iterations.

- For the solution of the nonlinear problem (the discretized PDE), a spatially first-order accurate solver is used to obtain an initial solution for the second-order solver. This initial first-order solver starts from free-stream conditions and is converged by 1 order of magnitude of the energy residual. It uses a sequential ILU preconditioner so that all evaluation runs start from exactly the same initial solution. This initial solver is not timed - all reported results are from the second-order solve [1].

- The initial guess for the asynchronous ILU factorization is the Jacobian matrix itself, while the initial guess for the triangular solves is the zero vector.

- For the point-block solver, the `BAIJMKL` matrix storage format is used for the Jacobian. This is PETSc's interface to block sparse matrix storage in the Intel Math Kernel Library (MKL). However, we do not use MKL in the preconditioners. The Eigen matrix library [104] is used for vectorized small dense matrix operations in the point-block preconditioner.

- The CPU is an Intel Xeon Phi 7230 processor with 64 cores. High bandwidth memory was used exclusively for all the runs. Only one thread was used per core. Each thread was bound to one hyper-thread context of a core. The OpenMP affinity is set to 'scatter' so that consecutive OpenMP threads are placed on different cores rather than different hyper-thread contexts of a single core. The OpenMP chunk size was 384 in case of block preconditioners, and 4 times that, 1536, for the scalar preconditioners (the block size is 4).

[1]'First order' and 'second order' here refer to only the right-hand side fluxes. The Jacobian matrix is always that of the first-order discretization. For a second-order solve, however, the Jacobian matrices are computed at second-order accurate solution vectors.

- The Intel C/C++ compiler version 17 was used with `-O3 -xmic-avx512` as optimization flags.

- In the graphs in this section, curves labelled as using one thread or one core correspond to regular sequential preconditioning. For example, asynchronous ILU(0) factorization when run with one thread is exactly traditional ILU(0) factorization. Note that this need not be true for all implementations of asynchronous iterations because of different ways that SIMD instructions can be used, but it is true for the one presented here.

Variants of asynchronous ILU preconditioners are evaluated and compared based on the criteria given below.

- Convergence of the nonlinear solver with respect to cumulative number of linear solver iterations.

- Asynchronous ILU residual after 1 build sweep, over all pseudo-time steps. The reason for using 1 build sweep is discussed below, where we study the effect of using different numbers of sweeps.

- Diagonal dominance of lower and upper triangular factors as a function of pseudo-time steps (nonlinear iterations). Since we use an iterative method to apply the triangular factors, their diagonal dominance is an indicator of the stability of the triangular solve.

- Strong scaling based on wall-clock time taken by all preconditioning operations until convergence of the nonlinear problem. A good scalability will show both the parallel efficiency of the asynchronous kernels and also the strength of the preconditioner to reduce the number of iterations required.

4.6.1 Number of asynchronous sweeps within the preconditioner

We now carry out a study to see how the general trend of FGMRES convergence of a linear system depends on the number of sweeps used to build and apply the asynchronous block ILU preconditioner. For this purpose, we extract the Jacobian matrix from an intermediate pseudo-time step for each test case and solve the linear system using the asynchronous block-ILU preconditioner and FGMRES(30) for two different thread settings - 16 and 62 threads. The result for each sweep setting was averaged over 10 repeated runs for these tables. The maximum relative deviation over all sweep settings is reported in the caption of each table. Relative deviation is defined here for each sweep setting as the standard deviation divided by the average number of iterations over the 10 repetitions for that sweep setting. Table 4.1

shows the iteration counts for the inviscid cylinder case with reverse Cuthill-McKee (RCM) ordering, while table 4.2 shows the same for the viscous NACA0012 airfoil case with RCM ordering and table 4.3 corresponds to the viscous NACA0012 case with one-way dissection (1WD) ordering.

The first broad observation is that higher build sweeps typically require higher application sweeps for effective preconditioning. This is more pronounced for higher thread counts. However this trend can barely be seen for the 1WD ordering in case of viscous problems. This is the first indication that orderings such as RCM that typically work well for synchronous ILU may not be the best for asynchronous ILU factorization.

	16 threads							62 threads						
Apply sweeps Build sweeps	1	2	3	5	10	20	Exact	1	2	3	5	10	20	Exact
1	438	227	174	141	130	129	130	495	252	186	149	130	129	129
2	453	228	176	146	129	129	129	514	254	189	154	130	129	129
3	459	230	177	147	133	131	131	516	257	190	155	133	131	131
5	467	237	178	148	134	132	132	522	267	192	156	135	132	132
10	467	235	179	149	134	132	132	523	268	192	156	135	132	132
20	476	237	178	148	134	132	132	527	266	192	157	135	132	132
Exact	470	237	178	148	134	132	132	526	268	191	157	135	132	132

Table 4.1: Number of FGMRES(30) iterations required for convergence to a relative tolerance of 1×10^{-2} as a function of number of sweeps used to build the asynchronous block-ILU preconditioner, for a matrix from the problem of inviscid flow over a cylinder (RCM ordering). Maximum deviation is about 2.2%.

	16 threads							62 threads						
Apply sweeps Build sweeps	1	2	3	5	10	20	Exact	1	2	3	5	10	20	Exact
1	1525	471		109	107	61	45					334	99	53
2			134	30	41	20	22				150	35	30	21
3			1051	54	33	19	18				802	51	24	19
5				495	23	17	16					23	22	18
10					146	17	16					509	17	17
20						38	16						59	16
Exact						258	16						1147	16

Table 4.2: Number of FGMRES(30) iterations required for convergence to a relative tolerance of 1×10^{-2} as a function of number of sweeps used to build the asynchronous block-ILU preconditioner, for a matrix from the problem of viscous flow over a NACA0012 airfoil. RCM ordering. Blanks indicate that the solver did not converge in 2500 iterations. Maximum relative deviation is 160% in case of 1 build and 10 apply sweeps.

	16 threads							62 threads						
Apply sweeps Build sweeps	1	2	3	5	10	20	Exact	1	2	3	5	10	20	Exact
1	319	225	224	224	224	224	224	320	225	225	224	224	224	224
2	282	204	203	203	203	203	203	283	204	203	203	203	203	203
3	282	204	203	203	203	203	203	283	204	203	203	203	203	203
5	282	204	203	203	203	203	203	283	204	203	203	203	203	203
10	282	204	203	203	203	203	203	283	204	203	203	203	203	203
20	282	204	203	203	203	203	203	283	204	203	203	203	203	203
Exact	282	204	203	203	203	203	203	283	204	203	203	203	203	203

Table 4.3: Number of FGMRES(30) iterations required for convergence to a relative tolerance of 1×10^{-2} as a function of number of sweeps used to build the asynchronous block-ILU preconditioner, for a matrix from the problem of viscous flow over a NACA0012 airfoil. 1WD ordering. Maximum relative deviation is less than 1%.

For the inviscid case with RCM ordering with 62 threads, convergence generally gets worse if we use more build sweeps for a given number of application sweeps, while it improves if we use more application sweeps. For the viscous case, RCM ordering results in poor performance of the asynchronous block-ILU preconditioner, and sensitivity to the number of sweeps is erratic (table 4.2). It can be seen that the general trend is towards worse preconditioning with increasing build sweeps for constant apply sweeps, and better preconditioning for increasing apply sweeps with constant build sweeps. However, there are clearly exceptions - the erratic nature of this table supports the results shown later that RCM ordering leads to poor and unreliable performance for viscous cases. Finally, the 1WD ordering gives robust results for this linear system of the viscous flow case. The results are almost independent of the number of threads, and all sweeps settings converge. For such cases, this ordering is clearly preferable to RCM, though the sequential (exact) preconditioning effectiveness is significantly worse. We further explore this in the context of convergence of the nonlinear problem further below (figure 4.11).

From the tables, we see that there is no consistent and significant advantage of using more than 1 sweep to build the factorization. For the viscous case with 1WD ordering (table 4.3), there is an advantage to using two build sweeps, though it performs consistently with one build sweep as well. Our objective is to use as few sweeps as possible without adversely impacting the convergence of the nonlinear problem much. Ultimately, for uniformity in the analysis of the nonlinear solves in the next subsections, we choose 1 build sweep and 3 apply sweeps for almost all the studies. The resulting preconditioner is denoted as ABILU(1,3). We also show some abridged results using a few other sweeps settings for the viscous flow case (figure 4.16).

It may be recalled that even though asynchronous iterations do not have 'sweeps' or global iterations in general, there exists such a notion for our implementation of asynchronous iterations. This is because our implementation still uses loops over all unknowns to control the parallel execution, which is made synchronization-free in the OpenMP framework. Since the loop is repeated a certain number of times, each unknown is updated exactly that many times. Considering this, we use the term 'asynchronous sweeps'.

4.6.2 Inviscid subsonic flow over cylinder

This section will demonstrate the effectiveness of the asynchronous block ILU preconditioner for an inviscid flow over a cylinder. The unstructured mesh consists of 217,330 quadrilaterals. The free-stream Mach number is 0.38. The HLLC [31] numerical flux is used for the inviscid terms. The mesh is largely isotropic, therefore we do not use any line-based orderings for this case.

Firstly, we demonstrate the convergence of the asynchronous ILU sweeps to the 'exact' incomplete L and U factors for one of the linear systems required for solving the nonlinear problem (figure 4.4). In these plots, the 'baseline' preconditioner is asynchronous ILU without the symmetric scaling used by Chow and Patel. The errors in the factors converge to a relative tolerance equal to machine precision in about 40 sweeps. This provides numerical validation of theorem 11 for our implementation of asynchronous (block) ILU iterations. Note that the data plotted has been averaged over five runs.

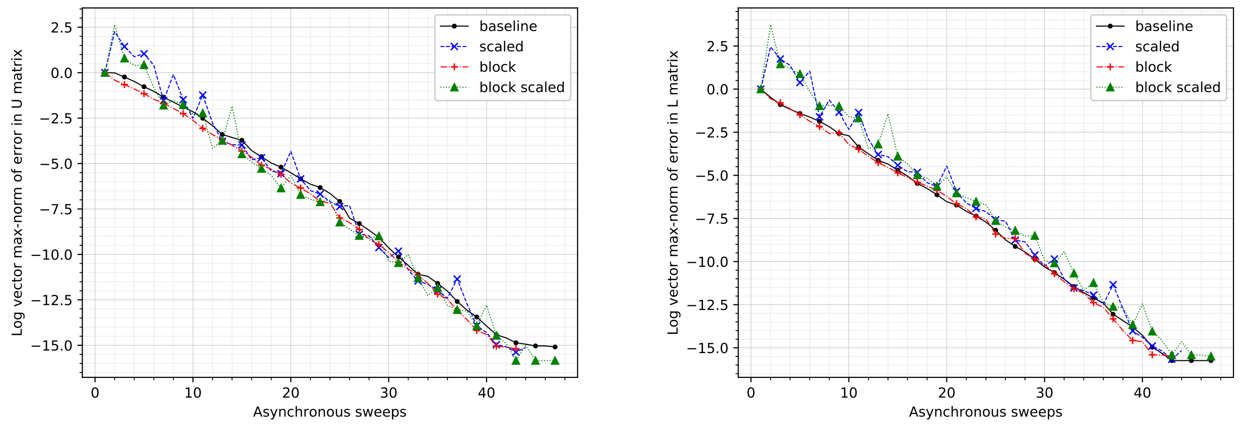
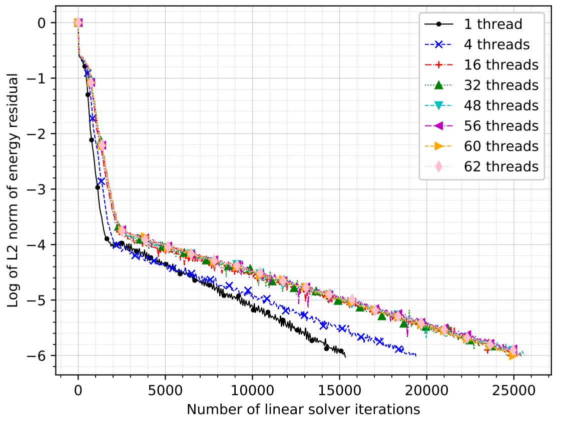


Figure 4.4: Convergence of different async. ILU fixed-point iterations (RCM ordering, 62 threads) at one pseudo-time step during the solution of the inviscid flow case. Left: Error in $\boldsymbol{U}$: $\|\texttt{vec}(\boldsymbol{U}_{async} - \boldsymbol{U}_{ilu})\|_\infty / \|\texttt{vec}(\boldsymbol{U}_0 - \boldsymbol{U}_{ilu})\|_\infty$. Right: Error in $\boldsymbol{L}$: $\|\texttt{vec}(\boldsymbol{L}_{async} - \boldsymbol{L}_{ilu})\|_\infty / \|\texttt{vec}(\boldsymbol{L}_0 - \boldsymbol{L}_{ilu})\|_\infty$

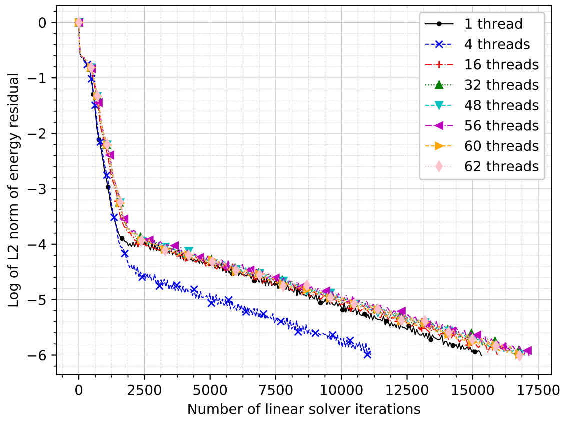
Next, we look at the convergence of the entire nonlinear CFD problem, in terms of the norm of the energy residual, with respect to the cumulative number of FGMRES iterations in figure 4.5. In case of RCM ordering of the grid cells (figure 4.5a), we observe that the asynchronous scalar ILU preconditioner causes a significant increase in the required number of FGMRES iterations as we increase the number of threads. Asynchronous block ILU, however, leads to a much smaller increase as we increase the number of threads (figure 4.5b). This is reflected in the strong scaling and wall-clock time plots shown later (figures 4.7). For this case, with the RCM ordering, the 4-thread run behaves anomalously for asynchronous block ILU - it converges even faster than the sequential ILU preconditioner. We have sometimes seen such abnormally fast convergence in previous work as well [24], but we do not generally expect an asynchronous ILU preconditioner to converge faster (in terms of number of linear solver iterations) than the corresponding standard sequential preconditioner.

However, when applied after one-way dissection (1WD) ordering, asynchronous ILU shows no such effects. In terms of the required number of FGMRES iterations for convergence, there is no significant difference between the scalar- and block-ILU preconditioners, though the block preconditioner does appear to be slightly less sensitive to the number of threads in the final few iterations (compare figures 4.5c and 4.5d). We will see that the asynchronous block ILU preconditioner is faster in terms of wall-clock time (figure 4.7b). Additionally, looking at the number of FGMRES iterations along the x-axis, we note that this ordering gives a much better ILU preconditioner than the RCM ordering for this test case.

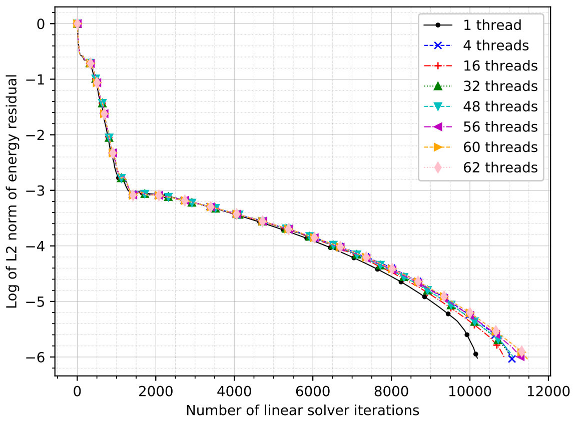
We attempt to provide some insight into the difference between scalar and block ILU in case of RCM ordering by showing the variation of the ILU fixed-point residual and the minimum diagonal dominance (described in subsection 2.3.1) of the lower and upper triangular factors. We see that one sweep of asynchronous block ILU leads to a relatively more accurate factorization (in the vector 1-norm $\|\boldsymbol{x}_1 - \boldsymbol{g}_{(b)ilu}(\boldsymbol{x}_1)\|_1 / \|\boldsymbol{x}_0 - \boldsymbol{g}_{(b)ilu}(\boldsymbol{x}_0)\|_1$, figure 4.6a) compared to the original scalar variant. Furthermore, the block L-factor has a lower max-norm for its Jacobi iteration matrix (figure 4.6c), which means it is relatively more diagonally dominant in the block case than in the scalar case. Note how the max-norm spikes to very high levels at some time steps for the scalar L-factor. (The L matrix is not actually diagonally dominant in either case - the Jacobi iteration matrix max-norm is greater than 1 in both cases - but the block case is better in this regard.) As an aside, we note that the U-factor's Jacobi iteration matrix norm increases first before stabilizing (figure 4.6b). This may be expected: since the CFL number increases until about 250 pseudo-time steps (figure 4.6d), the U-factor gets progressively less diagonal dominant until then. Recall that the CFL number is adjusted every nonlinear iteration based on the ratio of the current and previous



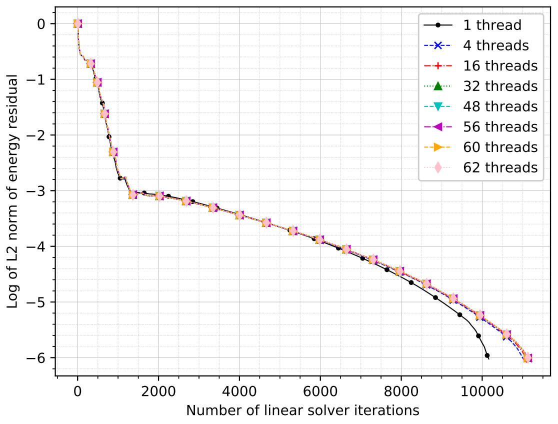
(a) Async. scalar ILU, RCM ordering



(b) Async. block ILU, RCM ordering



(c) Async. scalar ILU, 1WD ordering



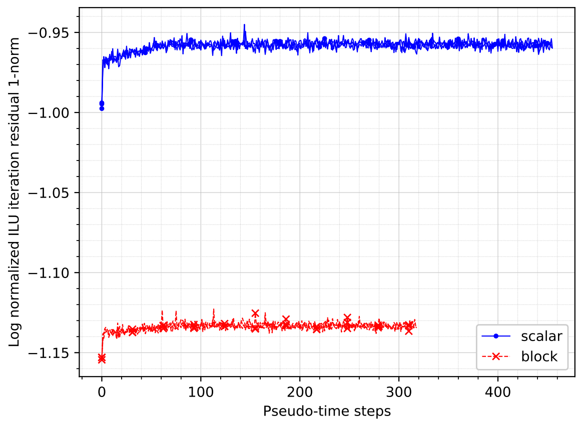
(d) Async. block ILU, 1WD ordering

Figure 4.5: Convergence of the nonlinear problem w.r.t. cumulative FGMRES iterations, for the inviscid cylinder case, showing the advantage of asynchronous block ILU (with 1,3 sweeps) over the scalar variant (no scaling was used while computing the preconditioner)

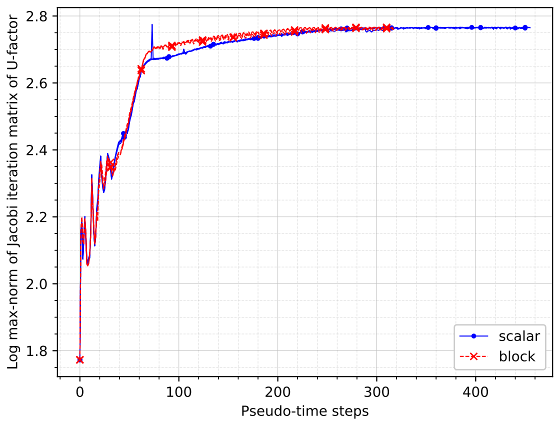
residual norms, as described in section 3.2.

Since Chow and Patel [65] favoured scaling the matrix symmetrically before computing the ILU factors, we looked at the effect of such scaling for this test case. We found that symmetric scaling of the Jacobian matrix before computing the preconditioner makes no difference to the residual history (the plot has been omitted for brevity). This may be because of the fact that we use the Euler equations expressed in terms of non-dimensional variables, and because the free-stream Mach number is moderate.

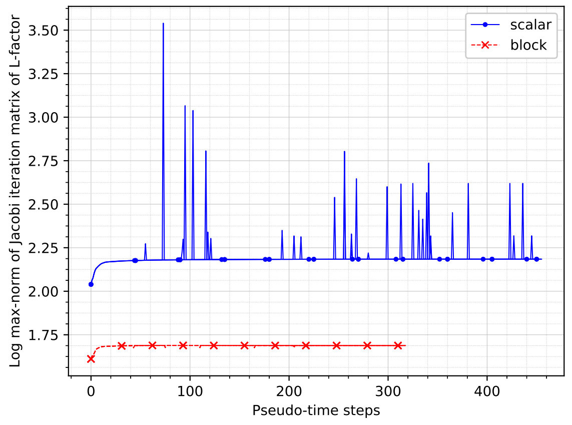
Finally, figure 4.7 shows the performance of the asynchronous ILU variants in terms of wall-clock time. Note that each data point in these graphs represents the time taken by all preconditioning operations over the entire nonlinear solver. Thus, any slowdowns because



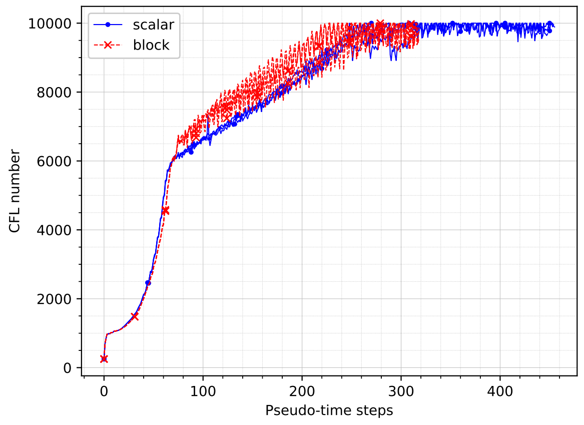
(a) Reduction in ILU fixed-point residual vector norm



(b) Norm of Jacobi iteration matrix for U-factor



(c) Norm of Jacobi iteration matrix for L-factor

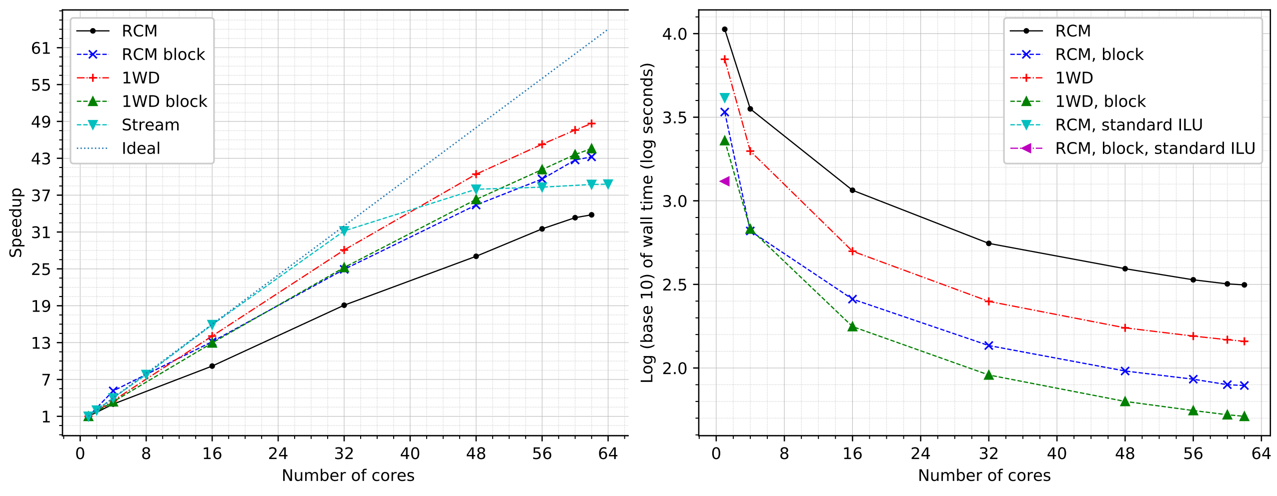


(d) CFL number

Figure 4.6: Properties of async. ILU factorization (1 build sweep), and CFL number, w.r.t. pseudo-time steps (RCM ordering, 62 threads, no scaling) for the inviscid flow case

of a weaker preconditioner are represented here. We also include the scaling of the Stream benchmark [122] to compare against the scaling of a strongly memory bandwidth limited code. In this regard, we note that in the case of a completely serial solver, a sequential block ILU(0) preconditioner with RCM ordering spends about 19% of the total wall-clock time in preconditioning operations (building and application). In the speedup plot (figure 4.7a), we observe that the scalar asynchronous ILU with RCM ordering does not show good parallel scaling. For all other variants, the parallel scaling continues to a larger number of cores compared to Stream. Since our kernels have more floating-point operations and less trivial memory access patterns than Stream, they are unlikely to be as memory-bandwidth limited. This is supported by the fact that some of the asynchronous ILU variants continue strong scaling after Stream reaches its limit. In the wall-clock time plot (figure 4.7b), we see that asynchronous block ILU with 1WD ordering performs the best at high core counts.

We include one data point each for the sequential ILU and block ILU preconditioners for reference.



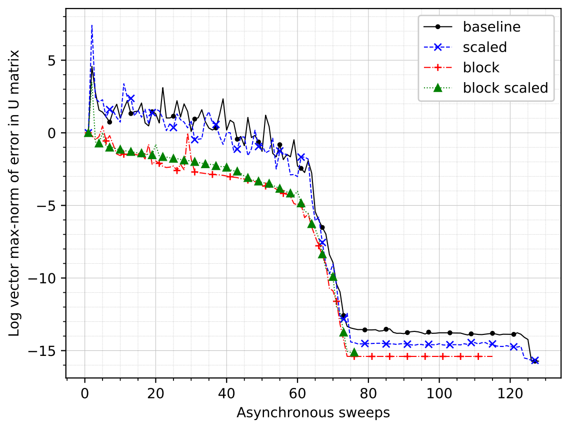
(a) Strong scaling (b) Wall-clock time

Figure 4.7: Strong scaling and wall-clock time taken by asynchronous scalar and block ILU preconditioning (1,3 sweeps) operations over the entire nonlinear solve, using RCM and 1WD orderings, for the inviscid flow case

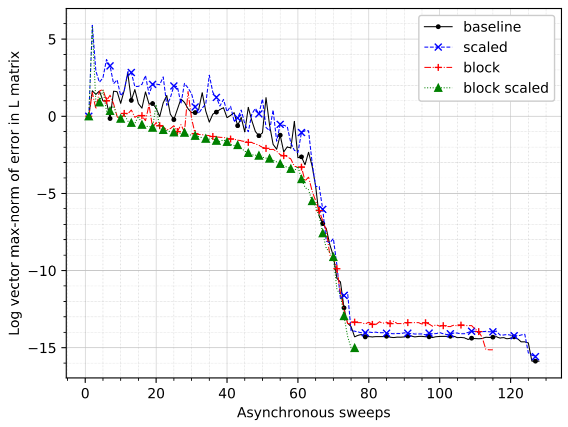
4.6.3 Viscous laminar flow over NACA0012 airfoil

We now investigate a case of viscous laminar flow over a NACA0012 airfoil. The grid is two-dimensional unstructured, made up of quadrilaterals in the boundary layer and triangles elsewhere, obtained from the SU2 [123] test case repository [124]. The Mach number is 0.5 and Reynolds number is 5000. The Roe [30] numerical flux is used for the inviscid terms. The total number of cells is 210,496, so the dimension of the problem is 841,984.

As for the previous test case, we first demonstrate the convergence of the asynchronous ILU sweeps to the 'exact' incomplete (ILU(0)) L and U factors for one of the linear systems required for solving the nonlinear problem. The grid has been ordered in the RCM ordering and 62 threads have been used (figure 4.8). The matrix is taken from a pseudo-time step at which the CFL number is 4241 and the energy residual is 1.5×10^{-4}. The 'baseline' run is a scalar asynchronous ILU(0) iteration without symmetric scaling of the original matrix. The errors in the L- and U-factors are normalized by the initial error. We see that all the iterations converge to machine precision after a sufficient number of sweeps. As opposed to the inviscid case, we see the asymptotically near-instantaneous convergence because the spectral radius of the iterations' Jacobian matrix is zero. Thus we regard this as numerical evidence not only of global convergence (theorem 11) but also of asymptotically trivial local convergence (theorem 9).



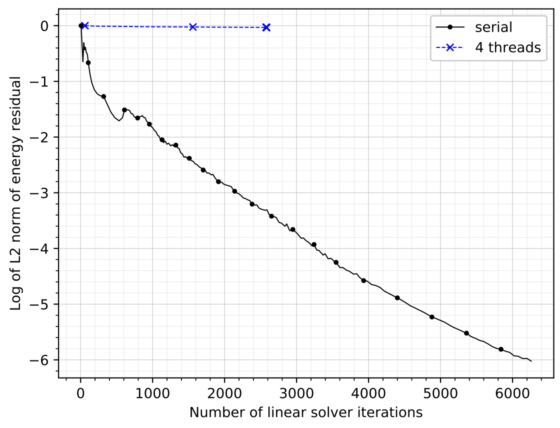
(a) $\|\texttt{vec}(U_{async} - U_{ilu})\|_\infty / \|\texttt{vec}(U_0 - U_{ilu})\|_\infty$



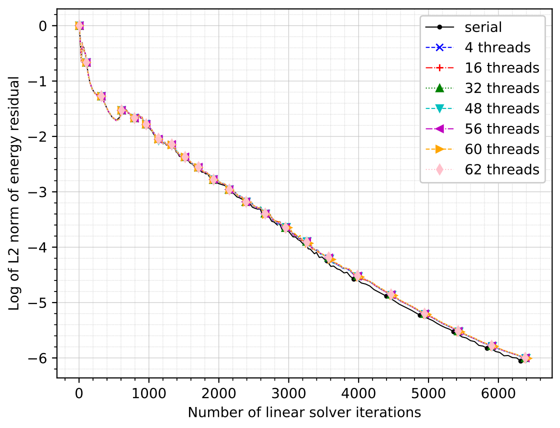
(b) $\|\texttt{vec}(L_{async} - L_{ilu})\|_\infty / \|\texttt{vec}(L_0 - L_{ilu})\|_\infty$

Figure 4.8: Convergence of different async. ILU fixed-point iterations (RCM ordering, 62 threads) for the viscous flow case

Next, we look at the convergence of the nonlinear problem, in terms of the norm of the energy residual, with respect to the cumulative number of FGMRES iterations. The first thing to note is that the scalar asynchronous ILU(0) solver does not work for this case whenever more than 1 thread is used, even when the matrix is first scaled symmetrically (figure 4.9a). Thus, the block variant is necessary for obtaining convergence with asynchronous ILU(0) preconditioning (figure 4.9b).



(a) Async. scalar ILU



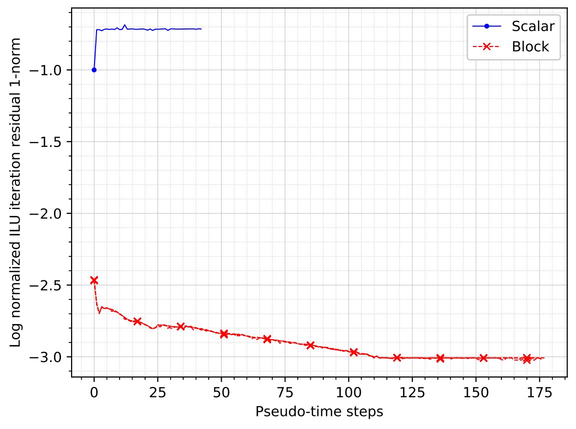
(b) Async. block ILU

Figure 4.9: Convergence (or lack thereof) of the nonlinear problem w.r.t. cumulative FGMRES iterations; 1 build sweep, 3 apply sweeps, line-1WD ordering and with symmetric scaling for the viscous flow case.

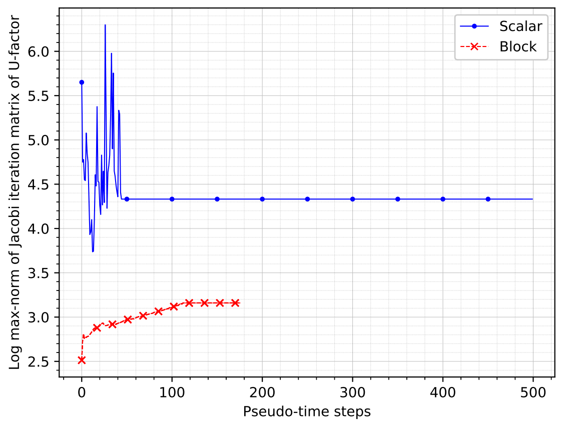
We attempt to provide some insight into this difference in convergence by showing the variation of the ILU residual and the minimum diagonal dominance of the lower and upper triangular factors. It can be seen that not only is the ILU factorization more accurate in the block case (figure 4.10a), the computed L and U factors have better diagonal dominance property (figures 4.10b, 4.10c). As noted earlier, lower max-norm of the Jacobi iteration matrix is equivalent to better diagonal dominance. As an aside, we see that the norm of the Jacobi iteration matrix for ILU factors increase until the CFL number increases.

Next, the asynchronous block ILU solver with RCM ordering stalls for any more than 1 thread (figure 4.11a). The one-way dissection (1WD) ordering is able to recover convergence, but the parallel runs still require more FGMRES iterations than the serial run (figure 4.11b). Finally, the line-based orderings converge in an essentially thread-independent number of iterations (figures 4.11c, 4.11d). The line-1WD hybrid ordering converges in the least number of iterations. We also report that with RCM ordering, sweeps settings (1,4) and (2,4) also stall at about five orders of magnitude with more than 1 thread.

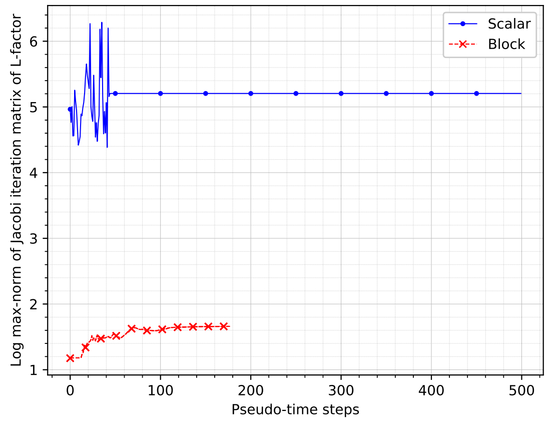
Again, we can look to the properties of the L and U factors to attempt to understand why certain orderings work better. Figure 4.12 shows properties of the asynchronous block ILU factorization for the case of 62 threads. The RCM ordering leads to a much less accurate (in the max vector norm) ILU factorization than any of the other orderings, while the hybrid line-1WD ordering results in the most accurate ILU factorization (figure 4.12a). Next, we look at the max-norm of the Jacobi iteration matrix of the U-factor in figure 4.12b. Even though the RCM ordering leads to a U that has slightly lower iteration matrix norm *on*



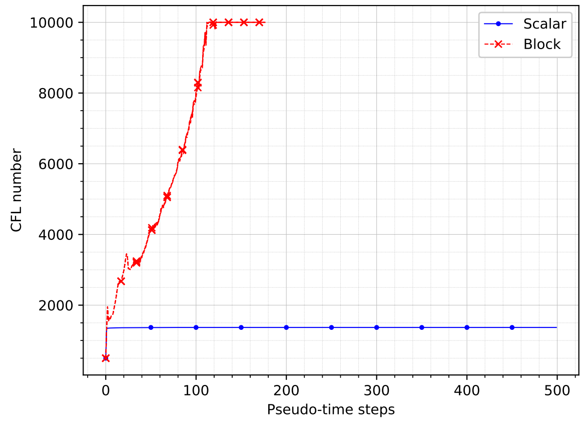
(a) Reduction in ILU fixed-point residual vector norm



(b) Norm of Jacobi iteration matrix for U-factor



(c) Norm of Jacobi iteration matrix for L-factor

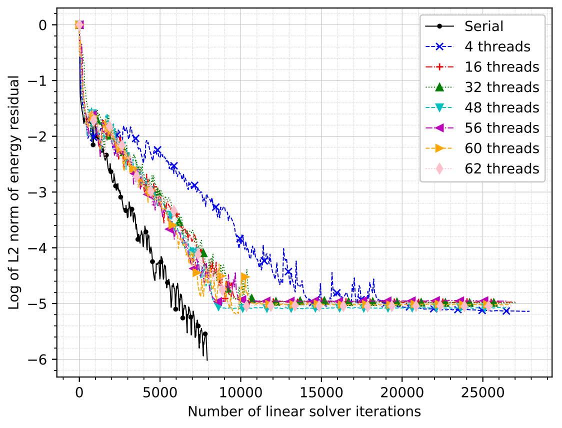


(d) CFL number

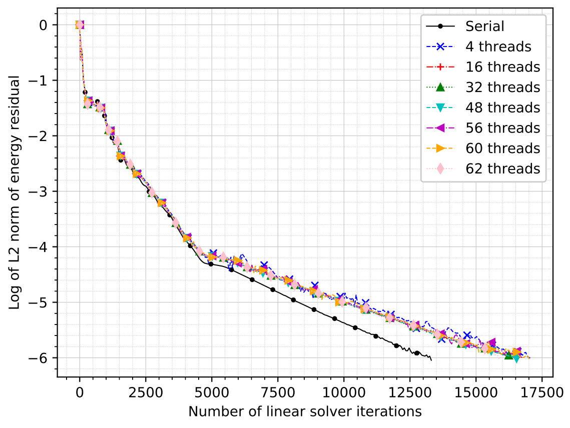
Figure 4.10: Properties of async. ILU factorization (1 sweep), and CFL number, w.r.t. pseudo-time steps for the viscous flow case with line-1WD ordering

average, it has sharp spikes at some time steps. For all the orderings the Jacobi iteration matrix norm of the U-factor rises as the CFL number increases (figure 4.12d). Once the CFL number reaches its limit of 10,000, the norm stabilizes and remains approximately constant. Finally, the norm of the iteration matrix for the L factor is clearly very unfavourable in case of the RCM ordering when compared to others (figure 4.12c), owing to very high peaks reached at many of the time steps.

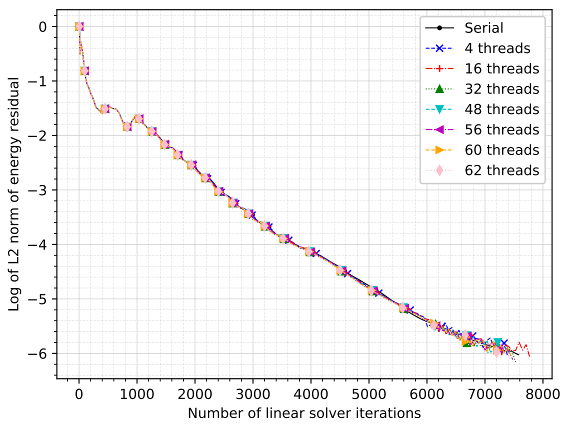
We note that scaling the original matrix symmetrically does not make a significant difference for this case either, as with the inviscid case. Figures 4.13a and 4.13b illustrate this in case of asynchronous block ILU(0) preconditioning for two orderings. RCM ordering causes a stall at a residual norm of about 10^{-5} irrespective of scaling, while the one-way dissection ordering leads to convergence in almost the same number of iterations irrespective of scaling. As with the inviscid flow case, this is not very surprising because we solve the non-dimensional Navier-Stokes equations.



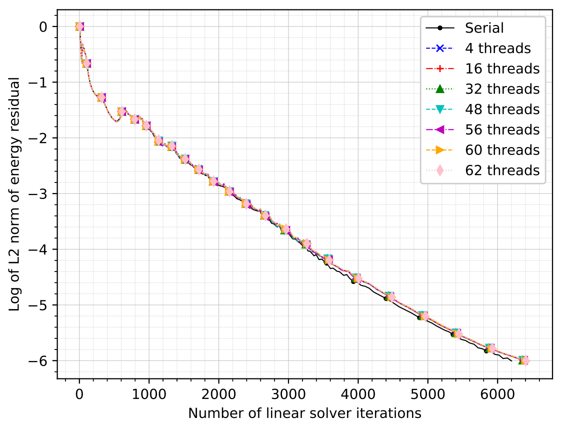
(a) RCM ordering



(b) 1WD ordering



(c) Line ordering

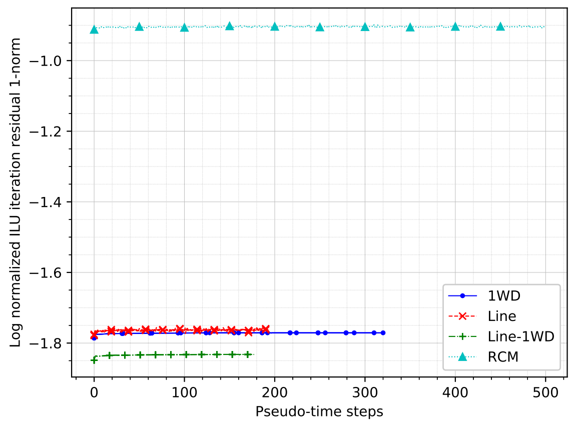


(d) Line 1WD ordering

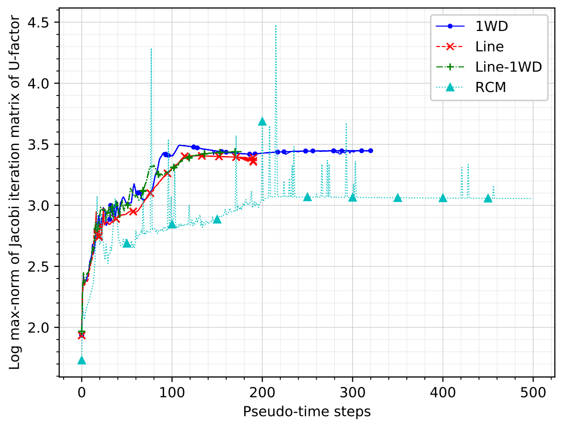
Figure 4.11: Convergence of nonlinear problem w.r.t. cumulative FGMRES iterations with different orderings (without scaling of the original matrices) for ABILU(1,3) preconditioner

Next, we run the case to a deeper level of nonlinear convergence of eight orders of magnitude using the line-1WD ordering, 1 build sweep and 2 apply sweeps (figure 4.14). We see a reduction in convergence rate after about 170 pseudo-time steps, but a brisk reliable convergence essentially independent of parallelism is still achieved, even with 2 apply sweeps.

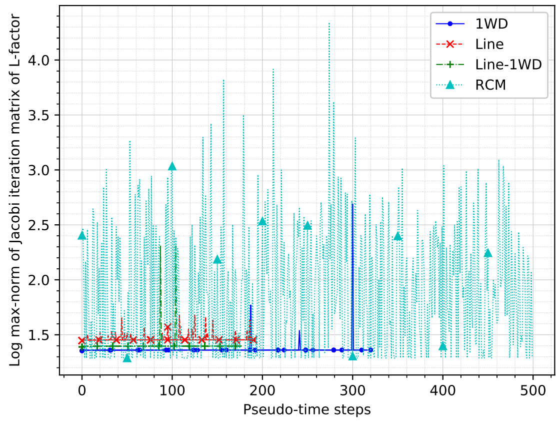
Finally, we show the performance in terms of wall-clock time and strong scaling. As before, we emphasise that data points represent the cumulative time taken by all preconditioning operations over the entire nonlinear solve. In the case of a completely serial solver, a sequential block ILU(0) preconditioner with RCM ordering spends about 16% of the total wall-clock time in preconditioning operations (building and application). The ordering has a large impact on speedups (shown in figure 4.15), as expected from convergence in terms of



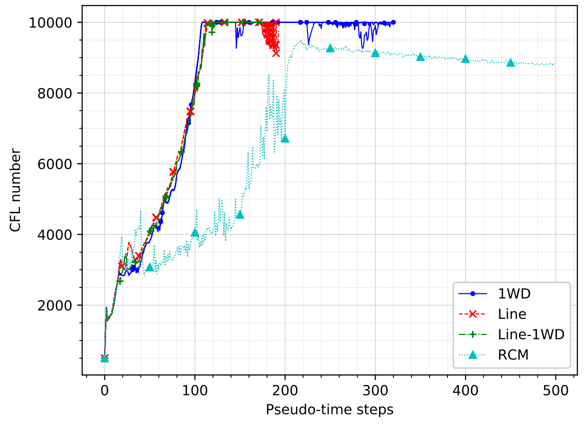
(a) ILU residual vector norm



(b) Norm of Jacobi iteration matrix for U-factor



(c) Norm of Jacobi iteration matrix for L-factor



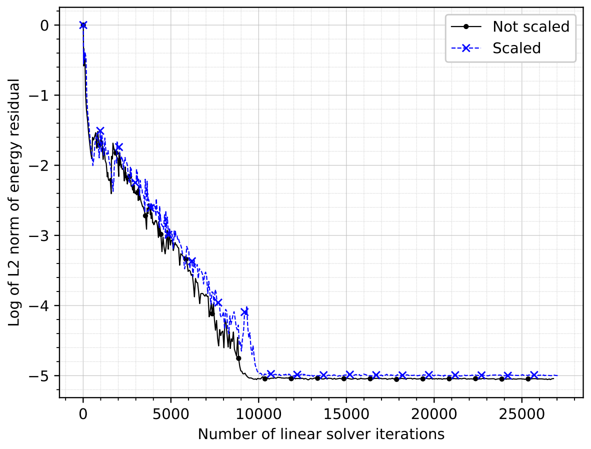
(d) CFL number

Figure 4.12: Properties of asynchronous block ILU factorization with 1 sweep, and CFL number, for the ABILU(1,3) preconditioner w.r.t. pseudo-time steps (62 threads)

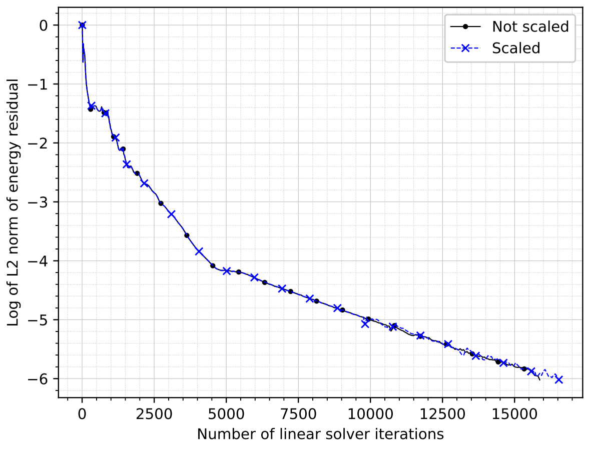
FGMRES iterations seen in figure 4.11. Though the line ordering scales best (figure 4.15a), the line-1WD ordering is actually the fastest (figure 4.15b). We include one data point for the standard sequential block-ILU preconditioner with RCM ordering in figure 4.15b for context.

In the speedup plot (figure 4.15a), we have included the scaling of the Stream benchmark. The scaling of the line-based orderings is comparable to Stream scaling up to a moderate number of cores. As in the inviscid case, the better asynchronous ILU variants continue to scale to higher core counts than Stream, which indicates that they are not quite as highly bandwidth-limited.

In figure 4.16, we compare the strong scaling and cumulative wall-clock time taken by preconditioning operations until nonlinear convergence for different sweeps settings for the line-1WD ordering. For good orderings such as line-1WD, larger numbers of sweeps do not make enough of a difference in the convergence rate to compensate for the extra work



(a) RCM ordering



(b) 1WD ordering

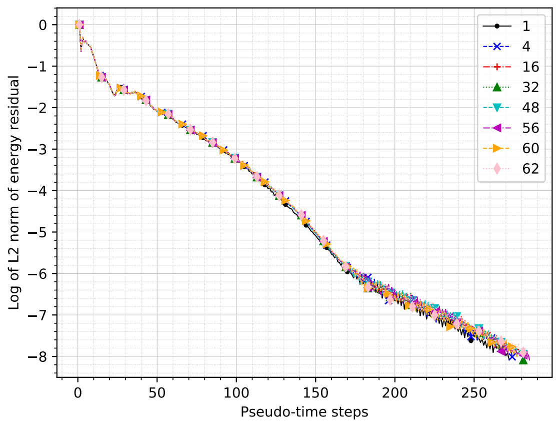
Figure 4.13: Effect of scaling on convergence of the nonlinear problem w.r.t. cumulative FGMRES iterations using ABILU(1,3) preconditioner

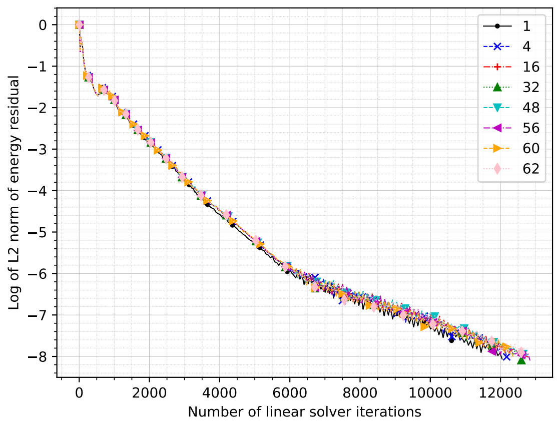
required per iteration.

In the interest of reproducibility and transparency, our codes are available online under the terms of the GNU General Public License. The asynchronous iterations are implemented as a separate library [2]. The finite volume CFD solver is available as a code [3] that optionally links to the library.

[2]`https://github.com/Slaedr/BLASTed`, commit 8522c6d26d21c8e63cc09761dd5b1362258a7b6a

[3]`https://github.com/Slaedr/FVENS`, commit a71b30a8cb9782b6f11e6e97888146d8bf4d7a5f


(a) Convergence w.r.t. nonlinear iterations


(b) Convergence w.r.t. cumulative linear iterations

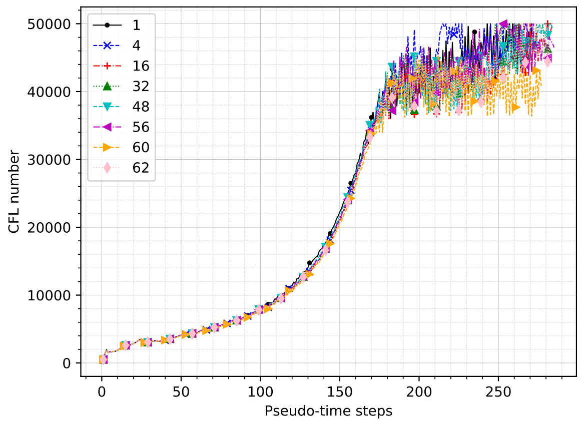
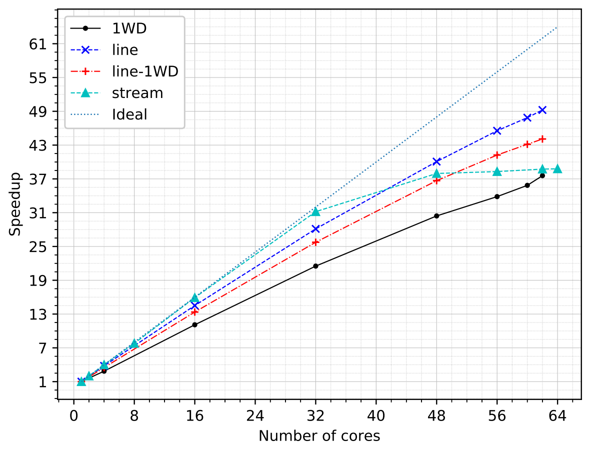

(c) Evolution of CFL number w.r.t. nonlinear iterations

Figure 4.14: Convergence of ABILU(1,2) preconditioned FGMRES for the viscous flow case with line-1WD ordering


(a) Speedup

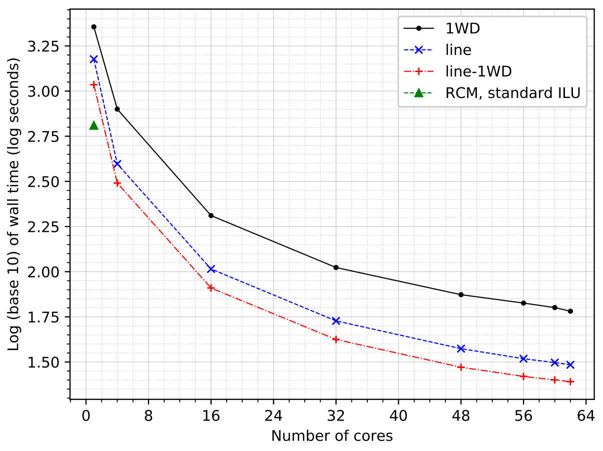
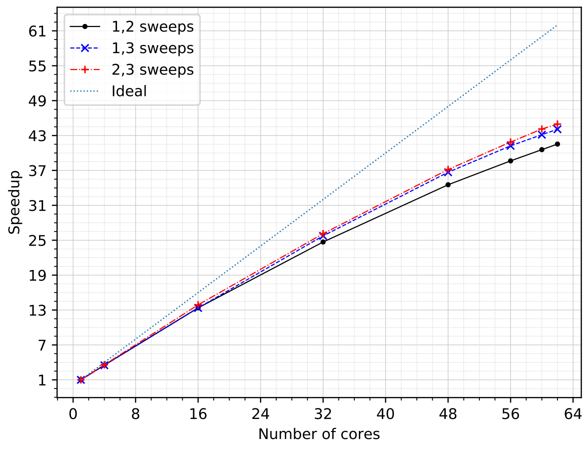
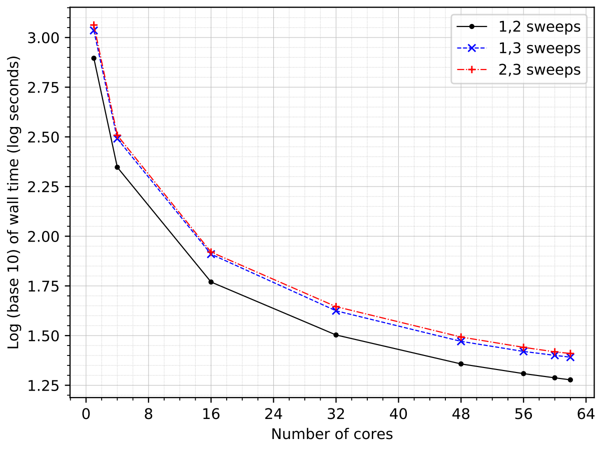

(b) Wall clock time

Figure 4.15: Performance of asynchronous block ILU (1,3) using different orderings (without scaling); though the line ordering scales best, the line-1WD ordering is the fastest up to 62 cores



(a) Speedup



(b) Wall clock time

Figure 4.16: Performance of asynchronous block ILU using different sweeps settings (without scaling) for line-1WD ordering

4.7 Conclusions

Convergence proofs for asynchronous ILU iteration have been extended to the case of asynchronous block ILU iteration; the latter shows a better theoretical convergence property. From the numerical results, it can be seen that asynchronous block ILU factorization and asynchronous forward and backward block triangular iterations hold promise for fine-grain parallel solution of compressible flow problems. For this coupled system of PDEs, the block variants are much more effective than the original asynchronous ILU preconditioning and scalar asynchronous relaxation for triangular solves.

A second conclusion that can be drawn from these results is that typical grid orderings used for sequential ILU preconditioners may not yield satisfactory results for asynchronous ILU preconditioners, especially for viscous flows. A hybrid 'line-X' ordering scheme has been introduced to deal with this issue. For external aerodynamics with viscous flow, one-way dissection (1WD), line and hybrid line-1WD orderings are seen to be good candidates. Further studies are needed on why the 1WD ordering works better than the reverse Cuthill-McKee (RCM) ordering for asynchronous ILU for these problems, even though it is inferior for sequential ILU factorization. In addition, further efforts are needed to develop an ordering and memory storage layout that work well on graphics processing units.

Chapter 5

Asynchronous iterations and incomplete sparse approximate inverses for structured grids on many-core SIMD processors

Block-asynchronous iterations, incomplete sparse approximate inverses and their combinations are proposed and their effectiveness studied for cases of external aerodynamics. In this chapter, the implementations of these solvers have focused on exploiting the wide vector units and the large number of cores on the Xeon Phi Knights Landing (KNL) processors, in the context of smoothers for nonlinear multigrid solvers specifically for the multi-block structured grid code FANSC-Lite. Two of the proposed solvers, block symmetric Gauss-Seidel and asynchronous block incomplete LU factorization, both with incomplete sparse approximate inverse application, are found to be promising.

5.1 Introduction

It was mentioned in chapter 2 that the multi-block structured grid code FANSC-Lite was parallelized over clusters of traditional central processing units (CPUs) using the Message Passing Interface (MPI). Development of a fine-grain parallel multigrid smoother is now required. Previous chapters introduced promising candidates for parallel smoothing and preconditioning for compressible flow problems; in chapter 3, a parallel asynchronous block SGS smoother was demonstrated for FANSC-Lite. However, those implementations use array-of-structures (AoS) storage (section 2.9.3) on CPUs and effective utilization of the wide vector units was not prioritized. In chapter 3, a block compressed sparse row matrix format with 5×5 blocks was used, which may not be particularly conducive to efficient 8-wide vectorization in AVX-512. Similarly, they are not applicable to graphics processing units (GPUs), which require parallel execution in lockstep for 32 work items at a time (see section 2.9.2).

In this chapter, we switch to a structure-of-arrays (SoA) storage layout, which achieves better vectorization on the KNL cores (section 2.9.4). Such an implementation can later be implemented for GPUs as well, which is the subject of the following chapter. More broadly, the changed layout enables vectorization irrespective of vector width and point-block size. Further, using a standard general matrix format for structured grids imposes unnecessary memory bandwidth overhead because of the pointer and indexing arrays required to access the coefficients. We therefore use the simplified ELL format described in section 2.9.3, omitting pointer and indexing arrays. It is important to note that unlike in the case of traditional synchronized solvers, where layout changes do not affect mathematical properties of the iterations, it turns out that they do affect the nature of the asynchronous process. This may be anticipated because the shift and update functions (section 2.8) of asynchronous iterations depend on memory access latency, which is significantly affected by layout changes. During development, we have observed that they may even depend on minor changes in the way the kernel code is written.

We use an asynchronous block ILU factorization specialized for structured grids. We also consider a different way of approximately solving triangular systems in this chapter. Recently, an 'incomplete sparse approximate inverse' (ISAI) method was proposed [72] to solve the triangular systems arising in ILU preconditioning. It was shown to be effective on some problems from the Suitesparse matrix collection [90]. This method is now explored as an alternative to asynchronous block triangular solves.

To summarize, in this chapter, we explore asynchronous point-block SGS and ILU iterations, a 'point-block SGS-ISAI' (BSGS-ISAI) iteration as well as an asynchronous point-block ILU(0)-ISAI (ABILU(0)-ISAI) iteration for use in multigrid smoothers on multi-block structured grids. All relevant objects are stored and accessed in a structure-of-arrays layout to achieve better vectorization on the KNL.

5.2 Solver framework

The multi-block structured grid code FANSC-Lite was used for this investigation. A full approximation storage (FAS) multigrid [33] solver is used for the mean flow equations. As before, we only consider the fine-grain parallelization of smoothers for the mean flow equations. The segregated Spalart-Allmaras turbulence model is solved by an alternating direction implicit (ADI) iteration on the fine grid.

The global smoother is described in algorithm 10. It consists of solving one approximate Newton iteration on a given multigrid level. The residual $\boldsymbol{r}$ is computed using the matrix dissipation (MATD) scheme [29], while the Jacobian is an approximate linearization of the 1st-order MATD scheme neglecting the pressure switch and the 4th order dissipation. It should be noted that the Jacobian matrix also ignores boundary conditions. The linear solver consists of a fixed number of preconditioned Richardson iterations with a subdomain-block Jacobi preconditioner. We are interested in developing fine-grain parallel local preconditioners for the (large) subdomains.

Algorithm 10 Implicit smoothing with preconditioned Richardson solver

```
1:  Compute fluxes $\boldsymbol{r}(\boldsymbol{w})$
2:  Compute approximate Jacobian matrix $\boldsymbol{A} := \frac{\partial r}{\partial \boldsymbol{w}}(\boldsymbol{w})$
3:  $\delta \boldsymbol{w}^0 \leftarrow \mathbf{0}$.
4:  **for** $i_{swp}$ in {1,2,...$n_{swp}$} **do**
5:      Richardson iteration with subdomain-block Jacobi global preconditioner:
6:      Linear residual:        $\boldsymbol{r}_{lin} \leftarrow -\boldsymbol{r} - \boldsymbol{A}\delta\boldsymbol{w}$
7:      Local preconditioner: $\boldsymbol{z} \leftarrow \text{blockdiag}(\boldsymbol{M}_{\text{loc}}^{-1})\boldsymbol{r}_{lin}$
8:      Update:                 $\delta\boldsymbol{w} \leftarrow \delta\boldsymbol{w} + \omega_{lin}\boldsymbol{z}$
9:      Communicate $\delta\boldsymbol{w}$
10: **end for**
11: Update state vector $\boldsymbol{w} \leftarrow \boldsymbol{w} + \omega\delta\boldsymbol{w}$
```

ω_{lin} is the linear relaxation factor while ω is the nonlinear relaxation factor. This is the scheme used for the fine-grain parallel iterations discussed in this work.

Originally, the work-horse solver used in the code was a block-Jacobi relaxation with a point-block symmetric Gauss Seidel (‘LUSGS’) local iteration, shown in algorithm 11.

Algorithm 11 Implicit smoothing with block Jacobi relaxation

```
1: Compute fluxes $\boldsymbol{r}(\boldsymbol{w})$
2: Compute approximate Jacobian matrix $\boldsymbol{A} := \frac{\partial \boldsymbol{r}}{\partial \boldsymbol{w}}(\boldsymbol{w})$
3: $\delta \boldsymbol{w}^0 \leftarrow \mathbf{0}$
4: for  $k$ in $\{1,2,...n_{swp}\}$ do
5:     Subdomain-block Jacobi global relaxation with local LUSGS:
6:     $\delta \boldsymbol{w}^* \leftarrow \boldsymbol{D}^{-1}(\boldsymbol{r} - \boldsymbol{L}_{\text{loc}}\delta \boldsymbol{w}^* - \boldsymbol{U}_{\text{loc}}\delta \boldsymbol{w}^k - \boldsymbol{L}_{\text{halo}}\delta \boldsymbol{w}^k - \boldsymbol{U}_{\text{halo}}\delta \boldsymbol{w}^k)$
7:     $\delta \boldsymbol{w}^{k+1} \leftarrow \boldsymbol{D}^{-1}(\boldsymbol{r} - \boldsymbol{L}_{\text{loc}}\delta \boldsymbol{w}^* - \boldsymbol{U}_{\text{loc}}\delta \boldsymbol{w}^{k+1} - \boldsymbol{L}_{\text{halo}}\delta \boldsymbol{w}^k - \boldsymbol{U}_{\text{halo}}\delta \boldsymbol{w}^k)$
8:     Communicate $\delta \boldsymbol{w}$
9: end for
10: Update state vector $\boldsymbol{w} \leftarrow \boldsymbol{w} + \omega \delta \boldsymbol{w}$
```

The non-linear multigrid solver uses a V-cycle. The same smoother used on finer levels is also employed on the coarse grid, though using more linear solver iterations there. While this is standard practice in aerodynamics codes, the use of a more exact coarse grid solver may affect results presented in this paper. The effect of exact coarse grid solvers has not been studied here.

5.2.1 Matrix storage layout

The matrix storage layout uses 25 different arrays for matrix coefficients, each of which corresponds to a flux-variable dependence pair and is stored in a simplified ELL format described in section 2.9.3. Each of the 25 arrays has seven columns representing the central cell and its six maximum neighbours in a 3D structured mesh block. The derivative of the mass flux in a cell with respect to density of the same cell is stored contiguously for all cells in the grid. This is followed by the derivative of mass flux in the central cell with respect to density in the first neighbouring cell for all cells contiguously, and so on, for each of the seven entries in the stencil. This is followed by the derivative of the x-momentum flux with respect to density, for each cell. This is repeated for all seven cells in the stencil. Finally, this is done for all 25 interactions. The layout is illustrated in figure 5.1.

Each SIMD lane processes all the computations needed for one cell, and thus needs to successively access each of the 7×25 coefficients associated with that cell's block-row. The next SIMD lane processes all computations needed for the next cell in the mesh and so on. Because corresponding matrix coefficients in the consecutive cells' stencils are stored consecutively in memory, a fetched 64-byte cache line contains one entry needed for each of the eight SIMD lanes in the AVX-512 unit. Thus efficient vectorization is enabled, as was

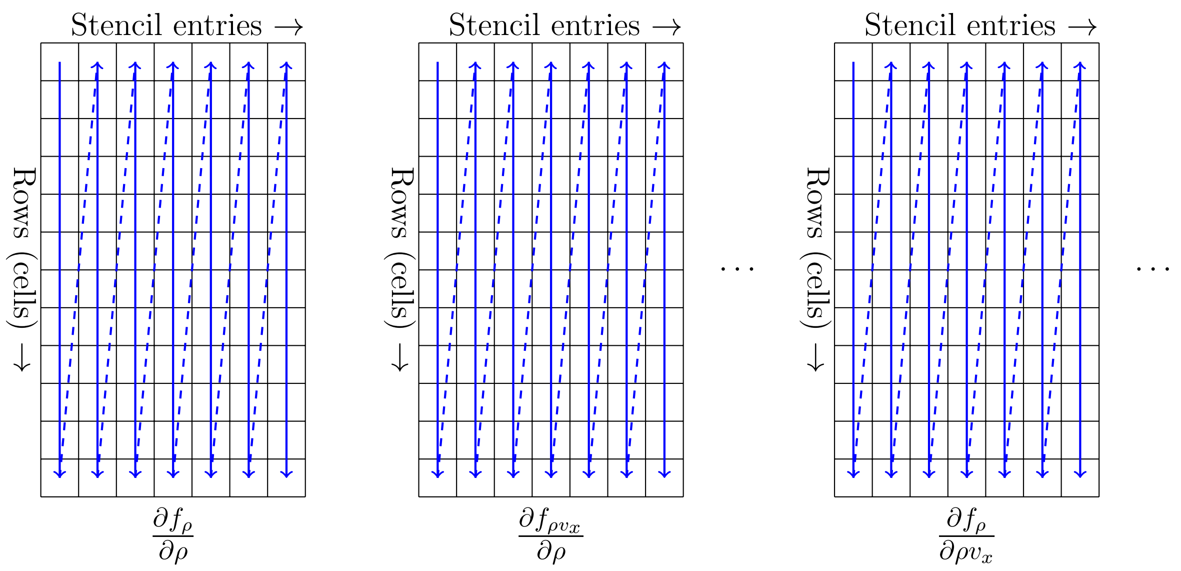

Stencil entries →
Rows (cells) →
$\frac{\partial f_\rho}{\partial \rho}$
Stencil entries →
Rows (cells) →
$\frac{\partial f_{\rho v_x}}{\partial \rho}$
. . .
Stencil entries →
Rows (cells) →
$\frac{\partial f_\rho}{\partial \rho v_x}$
. . .

Figure 5.1: Jacobian matrix storage layout

discussed in section 2.9.4. This pattern also leads to coalesced memory access required for GPUs.

5.3 Asynchronous ILU preconditioning for structured grids

Asynchronous iteration has been described in section 2.8. As mentioned there, even though asynchronous iterations are defined in terms of 'steps' rather than 'sweeps' or 'iterations', in the following, we will use the term 'iteration' of the asynchronous process to mean an update of all the entries of the solution vector. This is so that we can more naturally compare with traditional (synchronized) relaxations such as Gauss-Seidel.

One can equate $[\boldsymbol{A}]_{ij} = [\boldsymbol{LU}]_{ij}$ to get a nonlinear fixed-point iteration. An asynchronous iteration can then be applied to it, as demonstrated by Chow and Patel [65]. This was explained in section 4.3.2. There, it was noted that the synchronized Jacobi-type version of Chow and Patel's algorithm converges in at most $|\mathcal{S}|$ iterations, where $\mathcal{S}$ is the sparsity pattern (set of indices of non-zero entries) of the factorization. By theorem 11, it was shown that a modified Jacobi-type iteration corresponding to the the asynchronous block-ILU iteration converges in at most $|\mathcal{S}|/b^2$ iterations, where b is the block size.

Here, we propose an asynchronous point-block ILU(0) method (shortened to ABILU(0) hereafter) well-suited for structured grids. It is based on the fact that ILU factorization $\boldsymbol{W}_{ilu}$ of a $Nb \times Nb$ matrix $\boldsymbol{A}$ (where b is the size of the point-block and N is the number of points/cells) can be expressed as [35, section 8.5]

$$\boldsymbol{W}_{ilu} = (\tilde{\boldsymbol{D}} + \tilde{\boldsymbol{E}})\tilde{\boldsymbol{D}}^{-1}(\tilde{\boldsymbol{D}} + \tilde{\boldsymbol{F}}), \tag{5.1}$$

$$\tilde{\boldsymbol{E}}_{ij} = \boldsymbol{A}_{ij} - \sum_{k:\,(i,k)\in\mathcal{S},\,(k,j)\in\mathcal{S}}^{k<j} \tilde{\boldsymbol{E}}_{ik}\tilde{\boldsymbol{D}}_k^{-1}\tilde{\boldsymbol{F}}_{kj} \quad \text{for } (i,j)\in\mathcal{S},\ i>j \tag{5.2}$$

$$\tilde{\boldsymbol{F}}_{ij} = \boldsymbol{A}_{ij} - \sum_{k:\,(i,k)\in\mathcal{S},(k,j)\in\mathcal{S}}^{k<i} \tilde{\boldsymbol{E}}_{ik}\tilde{\boldsymbol{D}}_k^{-1}\tilde{\boldsymbol{F}}_{kj} \quad \text{for } (i,j)\in\mathcal{S},\ i<j, \text{ and} \tag{5.3}$$

$$\tilde{\boldsymbol{D}}_{i} = \boldsymbol{A}_{ii} - \sum_{k:\,(i,k)\in\mathcal{S},(k,i)\in S}^{k<i} \tilde{\boldsymbol{E}}_{ik}\tilde{\boldsymbol{D}}_k^{-1}\tilde{\boldsymbol{F}}_{ki}, \tag{5.4}$$

where $\boldsymbol{B}_{ij}$ for some matrix $\boldsymbol{B}$ represents the point-block at the $ij-$th block index in the matrix. The point-block would be 5×5 in 3D and 4×4 in 2D. $\mathcal{S}$ is now a block sparsity pattern - the set of block-indices for which the point-blocks are not identically zero.

We now consider the special case of compact discretizations on meshes which satisfy the following statement: for each cell in the mesh, no two neighbours of the cell are neighbours of each other. This is true for structured grids. In this case it can be shown that

$$\boldsymbol{W}_{ilu} = (\tilde{\boldsymbol{D}} + \boldsymbol{E})\tilde{\boldsymbol{D}}^{-1}(\tilde{\boldsymbol{D}} + \boldsymbol{F}), \tag{5.5}$$

where $\boldsymbol{E}$ and $\boldsymbol{F}$ are strictly lower triangular and upper triangular parts of $\boldsymbol{A}$ respectively (ie., $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{E} + \boldsymbol{F}$ where $\boldsymbol{D}$ is the diagonal part of $\boldsymbol{A}$).

Algorithm 12 Asynchronous block-ILU(0) iteration for structured grids

```
Require: Initial guess for $\tilde{\boldsymbol{D}}$.
1: for $i_{swp}$ in {1,2,...$n_{swp}$} do
2:     for $i \in \{1, 2, ...n\}$ do in parallel dynamically:
3:         $\tilde{\boldsymbol{D}}_i \leftarrow \boldsymbol{A}_{ii} - \sum_{k<i} \boldsymbol{E}_{ik}\tilde{\boldsymbol{D}}_k^{-1}\boldsymbol{F}_{ki}$
4:     end for
5: end for
```

$$\boldsymbol{d}^{m+1} = \boldsymbol{g}_{str}(\boldsymbol{d}^m), \tag{5.6}$$

where the ith block of the operator $\boldsymbol{g}_{str}$ is given by

$$\boldsymbol{g}_{str}(\boldsymbol{D})_i := \boldsymbol{A}_{ii} - \sum_{k<i} \boldsymbol{E}_{ik}\tilde{\boldsymbol{D}}_k^{-1}\boldsymbol{F}_{ki}. \tag{5.7}$$

The following can easily be shown.

Theorem 12. *If $\boldsymbol{g}_{str}$ has a unique fixed point, the iteration* (5.6) *is locally convergent, ie., the unique fixed point of $\boldsymbol{g}_{str}$ is a point of attraction of the iteration* (5.6).

Proof. It is easy to see that the Jacobian matrix of $\boldsymbol{g}$ with respect to the diagonal blocks, $\boldsymbol{g}'_{str}$, is strictly lower triangular, since the sum in (5.7) is over all indices less than i. This means all eigenvalues of $\boldsymbol{g}'_{str}$ are zero. Then by theorem 2, the fixed point of (5.7) is a point of attraction of the asynchronous iteration. □

Theorem 13. *If a fixed point $\boldsymbol{x}_*$ of the mapping $\boldsymbol{g}_{str}$ exists, then the fixed point is unique and the modified Jacobi-type parallel iteration corresponding to* (5.6) *converges to $\boldsymbol{x}_*$ in at most N iterations.*

Proof. On similar lines to the proof given by Chow and Patel [65], without loss of generality, we can consider a lexicographic ordering of the entries of $\tilde{\boldsymbol{D}}$. In a sequential iteration, if a singular block is encountered for any $\tilde{\boldsymbol{D}}_k$, no fixed point exists. If no singular diagonal block is encountered, the sequential iteration determines the unique fixed point.

For the case of a parallel modified Jacobi-type iteration, we consider the modified iteration in lexicographic ordering. Note that the first block of $\boldsymbol{D}$ does not depend on any other blocks, and thus it attains its final value after the first iteration. The second block may depend only on the first, so it attains its final value by the second iteration, and so on. Thus, if the fixed point exists, the ith diagonal block attains its final value by the ith iteration. □

This may be contrasted to the bound of $|\mathcal{S}|$ iterations in theorem 4. For 3D structured grids, $|\mathcal{S}| \approx 7Nb^2$. Since we only need to compute the diagonal blocks, the estimate of the maximum number of iterations required to get the exact ILU(0) factors goes down to Nb^2. Further, because of the point-block structure, the estimate drops by a factor of b^2 compared to a scalar algorithm, as demonstrated in a more general setting by theorem 11.

5.4 Application of triangular solves

Both the SGS and ILU preconditioners need to be applied by triangular solves. We consider two techniques for fine-grain parallel forward- and backward-triangular solves.

5.4.1 Asynchronous relaxation

One option is to use chaotic relaxation to derive an iterative procedure to apply triangular factors. This was already done for point-block systems in section 4.4.1. Convergence of the block-asynchronous iteration to the solution of the triangular system was proved. Further details specific to the vectorizable implementation in FANSC-Lite are given here.

Taking $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{E} + \boldsymbol{F}$ as in (5.5), we want to solve the system

$$(\tilde{\boldsymbol{D}} + \boldsymbol{E})\tilde{\boldsymbol{D}}^{-1}(\tilde{\boldsymbol{D}} + \boldsymbol{F})\boldsymbol{z} = \boldsymbol{r} \tag{5.8}$$

for $\boldsymbol{z}$. Here, $\boldsymbol{r}$ is the linear residual $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$. For block SGS, $\tilde{\boldsymbol{D}} = \boldsymbol{D}$, while for block ILU(0), $\tilde{\boldsymbol{D}}$ is given by equation (12).

The asynchronous triangular solve then consists of two steps. $(\tilde{\boldsymbol{D}} + \boldsymbol{E})\boldsymbol{y} = \boldsymbol{r}$ is solved by asynchronous linear iterations corresponding to:

$$\boldsymbol{y} = \tilde{\boldsymbol{D}}^{-1}(\boldsymbol{r} - \boldsymbol{E}\boldsymbol{y}). \tag{5.9}$$

$\boldsymbol{y}$ is initialized to zero, and the loop is executed in parallel. We order the loop from the first cell to the last, so that we retain some forward Gauss-Seidel character across AVX-512 slices and OpenMP chunks of entries in $\boldsymbol{y}$. Within each AVX-512 slice, the iteration is expected to be Jacobi-like, while across slices within one OpenMP chunk, it is expected to behave like Gauss-Seidel.

Next, $\tilde{\boldsymbol{D}}^{-1}(\tilde{\boldsymbol{D}}+\boldsymbol{F})\boldsymbol{z}=\boldsymbol{y}$ is solved by

$$\boldsymbol{z}=\boldsymbol{y}-\tilde{\boldsymbol{D}}^{-1}\boldsymbol{F}\boldsymbol{z}. \tag{5.10}$$

This time, $\boldsymbol{z}$ is initialized to $\boldsymbol{y}$ and the loop is ordered backwards from the last cell to the first, in an attempt to retain some backward Gauss-Seidel character.

By theorem 2 or the older result by Chazan and Miranker [83], convergence is easily shown, because the iteration matrix is strictly lower-triangular or strictly upper-triangular respectively.

In the results (section 5.5), a point-block SGS solver with the asynchronous triangular solve described above is referred to as a 'BSGS-async' or simply 'SGS-async' iteration. Asynchronous point-block ILU(0) iterations with asynchronous triangular solves are referred to as 'ABILU0-async'.

5.4.2 Incomplete sparse approximate inverse iteration

As mentioned earlier, incomplete sparse approximate inverse (ISAI) iterations have been proposed as triangular solvers [72]. It is related to the earlier sparse approximate inverse (SAI) method [71].

SAI

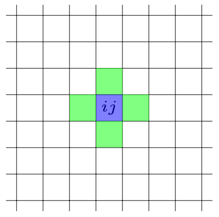
Given a $N \times N$ linear system such as $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$, the SAI method aims to compute a matrix $\tilde{\boldsymbol{M}}$ such that

$$\|\boldsymbol{A}\tilde{\boldsymbol{M}}-\boldsymbol{I}_N\|_F \tag{5.11}$$

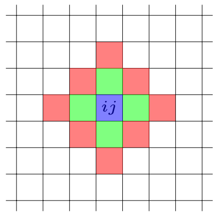
is minimized over all possible $\tilde{\boldsymbol{M}}$ obeying some sparsity constraint. Here, $\boldsymbol{I}_N$ is the $N \times N$ identity matrix. $\|\cdot\|_F$ denotes the Frobenius norm. The use of this norm allows us to decouple the minimization problem for each column of $\tilde{\boldsymbol{M}}$, which enables parallel computation. If $\mathcal{J}$ is the set of indices on which the jth column of $\tilde{\boldsymbol{M}}$ is non-zero, and $\mathcal{I}$ is the set of row-indices of $\boldsymbol{A}$ which have non-zeros in columns having indices $\mathcal{J}$, the minimization problem for the jth column is given by

$$\min_{\tilde{\boldsymbol{m}}_j}\|\boldsymbol{A}(\mathcal{I},\mathcal{J})\tilde{\boldsymbol{m}}_j(\mathcal{J})-\boldsymbol{e}_j(\mathcal{I})\|_2, \tag{5.12}$$

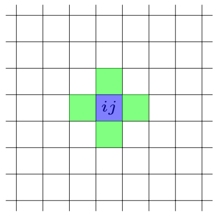
where $\boldsymbol{e}_j$ is the jth column of the identity matrix, and index sets in parentheses indicate restrictions of the corresponding matrices to those indices. Thus if the number of indices



(a) $\boldsymbol{A}$



(b) SAI(1)



(c) ISAI(1)

Figure 5.2: Stencils of the column corresponding to cell i, j

in $\mathcal{I}$ is $n_{\mathcal{I}}$ and that in $\mathcal{J}$ is $n_{\mathcal{J}}$, $\boldsymbol{A}(\mathcal{I}, \mathcal{J})$ is the $n_{\mathcal{I}} \times n_{\mathcal{J}}$ matrix containing those entries of $\boldsymbol{A}$ corresponding to the rows in $\mathcal{I}$ and the columns in $\mathcal{J}$. $\mathcal{I}$ is sometimes referred to as the 'shadow' of $\mathcal{J}$ with respect to the matrix $\boldsymbol{A}$.

In the SAI method, the matrix $\tilde{\boldsymbol{M}}$ is used as the preconditioner in a Krylov subspace solver or in a multigrid smoother. Solution of the equations (5.12) requires N small $n_{\mathcal{I}} \times n_{\mathcal{J}}$ dense least-squares solves. Suppose $\boldsymbol{A}$ corresponds to a compact discretization (stencil involving only face-neighbours for every cell). Then on a 3D structured grid with 5 unknowns per cell, if we restrict the sparsity pattern of $\tilde{\boldsymbol{M}}$ to that of $\boldsymbol{A}$, equation (5.12) corresponds to a 125×35 least-squares problem with 1225 non-zero entries. Such a problem needs to be solved for every cell to compute the sparse approximate inverse.

ISAI

In the ISAI method, the cost of computing the approximate inverse is reduced by not considering all of the rows that have non-zeros in columns of $\mathcal{J}$. Instead, the following equation is solved:

$$\min_{\boldsymbol{m}_j} \|\boldsymbol{A}(\mathcal{J}, \mathcal{J})\tilde{\boldsymbol{m}}_j(\mathcal{J}) - \boldsymbol{e}_j(\mathcal{J})\|_2. \tag{5.13}$$

This is now a square system which can be solved if $\boldsymbol{A}(\mathcal{J}, \mathcal{J})$ is non-singular. For the 3D structured grid example, the problem now becomes a 35×35 linear system with 325 non-zero entries for every cell's block-column.

Instead of using this incomplete sparse approximate inverse as a preconditioner, Anzt et al. [72] use it to solve triangular systems arising in the application of ILU factors. The ISAI iteration to solve the triangular system $\boldsymbol{L}\boldsymbol{y} = \boldsymbol{r}$ can be written as

$$\boldsymbol{y}^{n+1} = \tilde{\boldsymbol{M}}\boldsymbol{r} + (\boldsymbol{I} - \tilde{\boldsymbol{M}}_L\boldsymbol{L})\boldsymbol{y}^n. \tag{5.14}$$

$\boldsymbol{U}\boldsymbol{z} = \boldsymbol{y}$ can be solved similarly. Note that each application costs about twice as many flops as traditional forward or back substitution. We use this procedure as an option to apply both the asynchronous block ILU(0) and the block symmetric Gauss-Seidel preconditioners.

In this work, we restrict the sparsity pattern of the approximate inverses to that of the system matrix which needs to be approximately inverted, in the interest of keeping the time taken per iteration low. For structured grids, exact expressions for blocks of $\tilde{\boldsymbol{M}}$ as functions of blocks of $\boldsymbol{A}$ are easily derived. Thus, in our code, no small linear system is solved during run-time for computing $\tilde{\boldsymbol{M}}$; instead the solution is directly programmed in. Consider the ISAI $\tilde{\boldsymbol{M}}_L$ of the unit lower block triangular factor $\boldsymbol{L}$. After some algebra, we find that for any vector $\boldsymbol{x}$, the α^{th} cell-block of the product $\tilde{\boldsymbol{M}}_L\boldsymbol{x}$ is given by

$$(\tilde{\boldsymbol{M}}_L\boldsymbol{x})_\alpha = \boldsymbol{x}_\alpha - \sum_{\beta\in\mathcal{N}_L(\alpha)} \boldsymbol{E}_{\alpha\beta}\tilde{\boldsymbol{D}}_\beta^{-1}\boldsymbol{x}_\beta, \tag{5.15}$$

where $\mathcal{N}_L(\alpha)$ represents the set of cells which are 'lower' neighbours of cell α in the lexicographic ordering. For a cell with indices (i,j,k) in the structured block, $\mathcal{N}_L$ includes $(i-1,j,k), (i,j-1,k)$ and $(i,j,k-1)$ (except at block boundaries, where one or more of these three are omitted). $\boldsymbol{E}$ is the strictly block lower triangular part of the Jacobian matrix $\boldsymbol{A}$. As before, for block SGS, $\tilde{\boldsymbol{D}} = \boldsymbol{D}$ (the block diagonal part of $\boldsymbol{A}$), while for block ILU(0), $\tilde{\boldsymbol{D}}$ is given by equation (12). For the block upper triangular problem, we get

$$(\tilde{\boldsymbol{M}}_U\boldsymbol{x})_\alpha = \tilde{\boldsymbol{D}}_\alpha^{-1}(\boldsymbol{x}_\alpha - \sum_{\beta\in\mathcal{N}_U(\alpha)} \boldsymbol{F}_{\alpha\beta}\tilde{\boldsymbol{D}}_\beta^{-1}\boldsymbol{x}_\beta), \tag{5.16}$$

where now $\mathcal{N}_U(\alpha)$ represents the set of cells which are 'upper' neighbours of cell α in the lexicographic ordering, and $\boldsymbol{F}$ is the strictly block upper triangular part of the Jacobian matrix.

5.5 Experiments

We show results of using the aforementioned iterations for some benchmark transonic Reynolds-averaged Navier-Stokes problems. All runs are executed on one or two nodes with one 64-core Xeon Phi Knights Landing processor each. One MPI rank is assigned to each node; OpenMP threading is used inside each MPI process such that each core is assigned one thread. All data is stored in structure-of-arrays layout and, for the fine-grain parallel linear iterations, loops are fully vectorized with AVX-512. FANSC-Lite uses block-structured grids and each MPI partition may contain several mesh blocks. With this in mind, it should be noted that loops are fine-grain parallel only within mesh blocks, and only one mesh block is smoothed at a time. Intel compilers version 17.0 were used.

5.5.1 Wing cases

We first study the performance of the fine-grain parallel iterations on two transonic turbulent flow cases - the DPW-W1 wing [108] and the ONERA-M6 wing [107]. Both were run on one Xeon Phi Knights Landing node. Figures 5.3a and 5.5a show the number of multigrid cycles required for reduction of the mass flux residual by 5 orders of magnitude as a function of increasing parallelism.

The strong scaling plots (figures 5.3b, 5.5b) show the scaling of the linear solver only. That is, the speedups correspond to the wall-clock time taken by all linear solver iterations (on all grid levels) until the mean-flow converges by 5 orders of magnitude. Other operations in the mean flow solver such as flux computations are left out of the timing. These plots also include a curve for the Stream benchmark [122], which represents the scaling of a memory bandwidth-bound program. We expect the linear solvers to usually be limited by memory bandwidth, so this provides a useful benchmark.

For the wall-clock time results shown (figures 5.4, 5.6), all timings correspond to the total time taken by all mean-flow iterations (on all grid levels) until the mass flux residual is reduced by 5 orders of magnitude. Unlike the strong scaling plots, these include the time taken for computing the fluxes and Jacobian matrices, in addition to the linear solver. Only the grid transfer operations and the turbulence model solver are not timed.

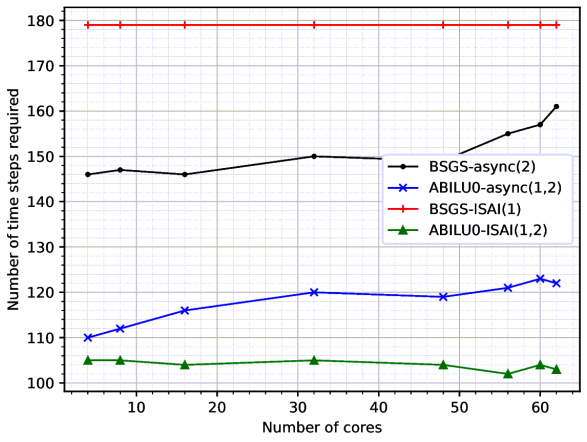
For all the runs, asynchronous application of block SGS or asynchronous block ILU(0) is carried out by 2 sweeps. The block SGS-ISAI application is done by just 1 sweep of ISAI, while asynchronous block ILU(0)-ISAI is carried out by 2 sweeps of ISAI. In all cases, asynchronous block ILU(0) factorization is carried out by 1 asynchronous sweep. The acronyms in the legends are as follows.

- 'BSGS-async': Point-block symmetric Gauss Seidel solver applied by asynchronous iterations.
- 'ABILU0-async': Asynchronous point-block ILU(0) factorization applied by asynchronous iterations.
- 'BSGS-ISAI': Point-block symmetric Gauss Seidel solver applied by incomplete sparse approximate inverse.
- 'ABILU0-ISAI': Asynchronous point-block ILU(0) factorization applied by incomplete sparse approximate inverse iterations.

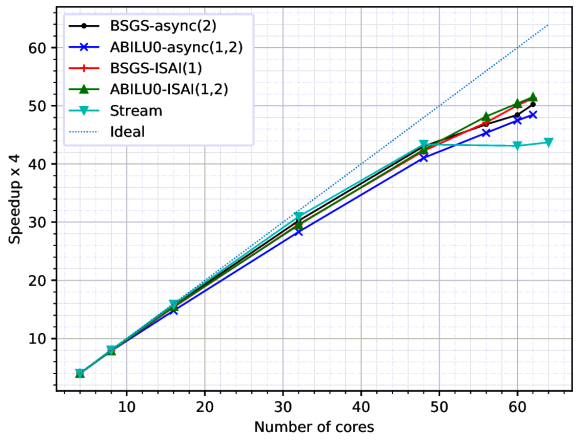
DPW-W1 wing

- $M = 0.76$, $R_e = 5.0 \times 10^6$, $\alpha = 0.6109°$

- Total number of cells = 1.51 million
- Average number of cells per block in the fine grid = 137402
- Total of 3 multigrid levels



(a) Multigrid cycles to convergence



(b) Strong scaling over all linear solver iterations in the flow solver

Figure 5.3: Iterations and speedups for DPW-W1 case

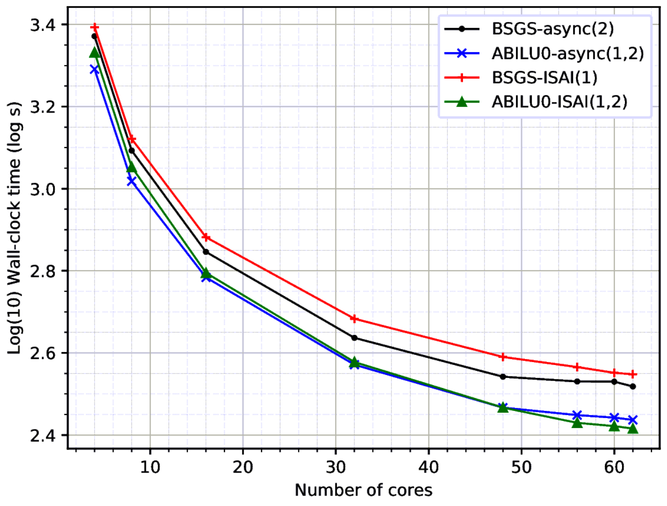


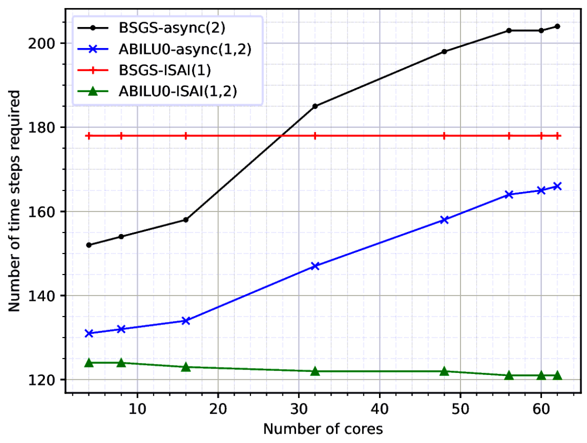
Figure 5.4: Wall-clock time spent in the mean flow solver for DPW-W1

ONERA-M6 wing

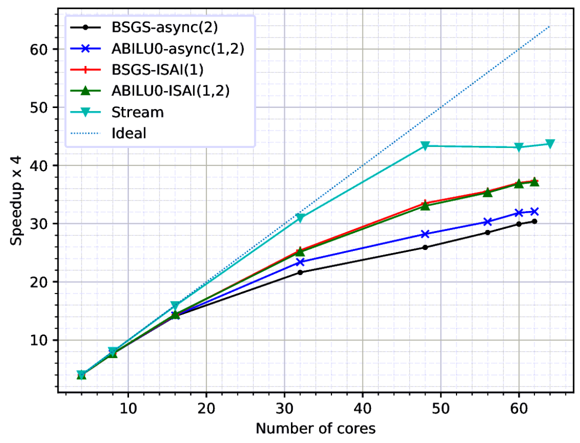
- $M = 0.8395$, $R_e = 11.72 \times 10^6$, $\alpha = 3.06°$
- Total number of cells = 884,736

- Average number of cells per block in the fine grid = 27648
- Total of 4 multigrid levels

The solution pressure contours are shown in figure 6.4.



(a) Multigrid cycles to convergence



(b) Strong scaling over all linear solver iterations in the flow solver

Figure 5.5: Iterations and speedups for ONERA-M6 case

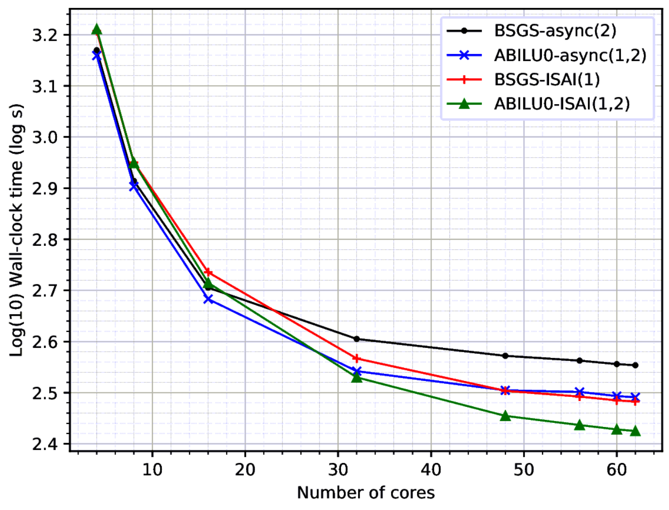


Figure 5.6: Wall-clock time spent in the mean flow solver for ONERA-M6

For both cases, we observe that the block SGS and asynchronous block ILU(0) solvers with ISAI iterative application (BSGS-ISAI and ABILU(0)-ISAI respectively) perform much better than the corresponding solvers with asynchronous iterative application. In terms of the total time taken by the mean flow solver, asynchronous block ILU(0) with ISAI application performs the best at high thread counts. However, it is seen that the actual speedups observed for the ONERA-M6 case (figure 5.5b) are lower than that indicated by memory bandwidth considerations, for all the solvers. We believe it is because mesh blocks for this case are too small, and OpenMP parallelism is currently limited to within mesh blocks in our code.

5.5.2 CRM wing-body

Finally, we compare the best of the new solvers, asynchronous block ILU(0) with ISAI iterative application ('ABILU0-ISAI'), with the current workhorse solver in the code. The latter is a point-block symmetric Gauss-Seidel (LUSGS) within each mesh block and block Jacobi relaxation to couple the different mesh blocks, as shown in algorithm 11. This 'BJac-LUSGS' smoother is not multi-threaded, but is parallelized using MPI only. Since our structured mesh partitioner is unable to exactly assign one mesh block to one MPI rank while satisfying a load imbalance bound, we have a several mesh blocks per rank. For the LUSGS runs, the mesh was repartitioned for each core count so that the load imbalance was less than 10%. This comparison is run using two Xeon Phi Knights Landing nodes, each with 64 cores.

This comparison is done on the benchmark NASA Common Research Model wing-body case [109]. The parameters for this case are $M = 0.85$, $R_e = 5.0 \times 10^6$ and $\alpha = 2.11°$. The original grid has 5 structured blocks and we run the L3 (medium) mesh with about 5.1 million cells (excluding any halo cells). We run a V-cycle multigrid solver with three multigrid levels until the mass flux residual norm drops by 4 orders of magnitude. We use 4 linear solver iterations each for pre- and post-smoothing on the fine and medium multigrid levels, while on the coarsest level we use 10 linear solver iterations.

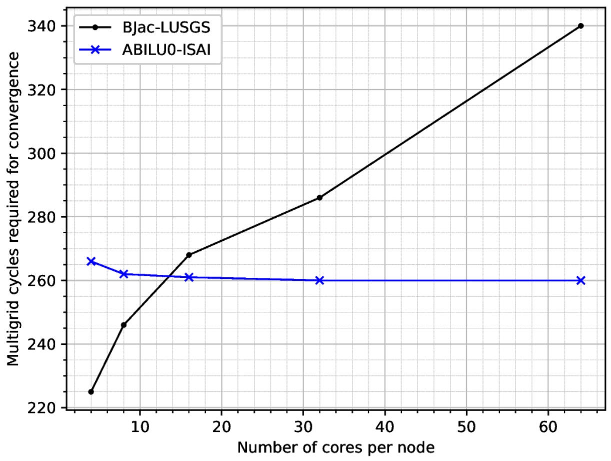
For the fine-grain parallel runs with the ABILU0-ISAI smoother, we subdivide the mesh into 14 blocks to get a good load balance on two Knights Landing nodes. Each node then has 3,027,648 cells (including halo cells), and each mesh block has between 475,200 and 240,448 cells, with a majority of the blocks having 462,400 cells each. We run this with two MPI ranks and varying numbers of threads per rank. Here too, we use one build sweep and two application sweeps every time an ABILU0-ISAI linear step is carried out.

For the workhorse Bjac-LUSGS solver, we present table 5.1 with details about the meshes. Note that the total number of cells increases with more ranks because we need more and more halo cells. In every case, there are 5,111,808 domain cells (excluding halos). We use one rank per core and split the ranks equally between the two Knights Landing nodes.

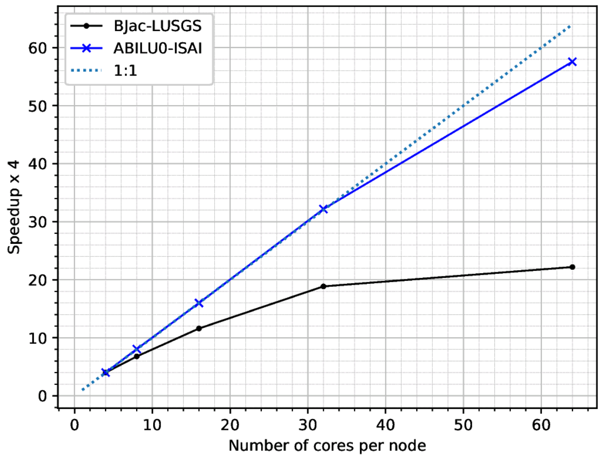
It is seen that BJac-LUSGS is not (strong-) scalable with increasing parallelism, even inside a multigrid solver, while the ABILU0-ISAI iteration makes the multigrid solver con-

Total ranks	No. mesh blocks	Total cells	Avg. cells per rank	Max. load imbalance (%)
8	28	6,297,344	787,168	7.8
16	56	6,649,344	415,584	8.1
32	112	7,050,240	220,320	8.2
64	224	7,612,416	118,944	8.5
128	448	8,458,240	66,080	8.5

Table 5.1: Details of grid partitions of the CRM wing-body case for runs with the BJac-LUSGS smoother; all cell counts include halo cells



(a) Multigrid cycles required for flow convergence



(b) Strong scaling of all linear solver iterations required for flow convergence

Figure 5.7: Iterations and speedups for CRM wing-body case

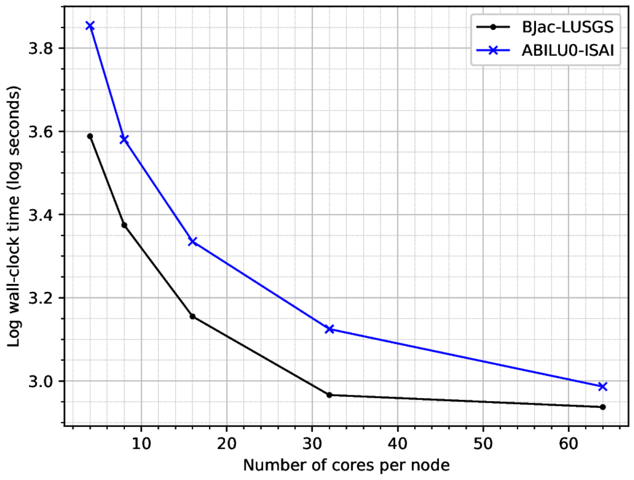


Figure 5.8: Wall-clock time spent in the mean flow solver for the CRM wing-body

verge in a constant number of cycles irrespective of the number of cores (figure 5.7a). This is reflected in the strong scaling of the linear solver operations over the entire run (figure 5.7b), in which the ABILU0-ISAI solver scales very well up to 64 cores per node, while the BJac-LUSGS solver almost saturates beyond 32 cores per node, corresponding to 64 total cores. It must, however, be noted that only one inexact Newton step is performed on the coarsest grid, albeit with more linear iterations inside. Results may change if the coarsest level is solved to some tolerance using several inexact Newton steps, for example.

We also show the total time taken by the mean flow solver, including boundary conditions, MPI communications, flux calculation and Jacobian computation, as a function of the number of cores per node(figure 5.8). The BJac-LUSGS solver (algorithm 11) is very frugal per iteration and MPI parallelism in the code has been highly optimized over the years, while the OpenMP parallelism is not yet fully optimized, eg. with regard to memory allocation in a thread-aware manner. Thus, the wall-clock time for the mean flow solver still favours BJac-LUSGS. But it can be seen that ABILU0-ISAI solver nearly breaks even at 64 cores per node. We expect that with more cores per node, or with more careful optimization of OpenMP parallelism in the code, ABILU0-ISAI will be favoured.

5.6 Conclusions

We developed and demonstrated asynchronous and incomplete sparse approximate inverse fine-grain parallel iterations for the many-core CPU Xeon Phi Knights Landing (KNL). The structure-of-arrays layout enables effective utilization of wide vector units such as AVX-512 on the KNL.

Observing the scalable performance of BSGS-ISAI and ABILU(0)-ISAI, we conclude that ISAI (with same sparsity pattern as the $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices) is effective in solving the triangular systems, at least as far as use in multigrid smoothers is concerned. Only 1 or 2 iterations per triangular solve are required. In particular, ABILU(0)-ISAI is a promising smoother. No convergence degradation with increasing parallelism is observed. It is notable that just one asynchronous sweep is enough to obtain a useful block-ILU(0) factorization. On the other hand, asynchronous triangular solves are not scalable in case of fully vectorizable codes on the Xeon Phi. Finally, from the comparison with subdomain-block-Jacobi LUSGS solver, we conclude that simple domain decomposition methods for the global smoother do not scale well to a large number of cores per node for the type of application and multigrid solver considered in this work.

Chapter 6

Asynchronous iterations and incomplete sparse approximate inverses for structured grids on graphics processing units

In this chapter, the specialized structured-grid solvers developed for many-core vector CPUs are extended to graphics processing units (GPUs). As such, this chapter is an extension to the previous chapter 5.

6.1 Solver framework

There is a difference between the formulation of the PDE system used in this chapter and that used in all of the previous chapters. In this chapter, the PDE system is expressed in terms of primitive variables (density, velocity and pressure) instead of the conserved variables (density, momentum and energy). Suppose Ω is a domain in d spatial dimensions, $\boldsymbol{u} : \Omega \to \mathbb{R}^{d+2}$ are the conserved variables, $\boldsymbol{v} : \Omega \to \mathbb{R}^{d+2}$ are the primitive variables and $\boldsymbol{c} : \mathbb{R}^{d+2} \to \mathbb{R}^{d+2}$ maps a vector of primitive variables to the corresponding vector of conserved variables ($\boldsymbol{u}(\mathbf{x}) = \boldsymbol{c}(\boldsymbol{v}(\mathbf{x}))$). Then, the governing equations

$$\frac{\partial \boldsymbol{u}}{\partial t} + \nabla \cdot \boldsymbol{F}(\boldsymbol{u}) = 0, \tag{6.1}$$

where $\boldsymbol{F}(\boldsymbol{u}) := \boldsymbol{F}^i(\boldsymbol{u}) + \boldsymbol{F}^v(\boldsymbol{u}, \nabla\boldsymbol{u})$ (see equation (2.5)), can be written as

$$\frac{\partial \boldsymbol{v}}{\partial t} + \left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{v}}(\boldsymbol{v})\right)^{-1} \nabla \cdot (\boldsymbol{F} \circ \boldsymbol{c})(\boldsymbol{v}) = 0. \tag{6.2}$$

These two systems are, of course, equivalent from a mathematical point of view. There is, however, a difference in the number of operations required per point to compute fluxes and Jacobian matrices. From an implementation point of view, the function $\boldsymbol{F} \circ \boldsymbol{c}$ is simpler (requires fewer operations) than $\boldsymbol{F}$. The new Jacobian $\frac{\partial(\boldsymbol{F}\circ\boldsymbol{c})}{\partial \boldsymbol{v}}$ is also a simpler function than the original $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{u}}$. The CUDA version of FANSC-Lite favours the primitive-variable approach. If everything else in the solver is equivalent, in practice, it is observed [70] that there is negligible difference between the two implementations in terms of number of nonlinear iterations, as expected, though the primitive variable approach takes less wall-clock time.

As opposed to previous chapters, where the first-order Jacobian matrices were always explicitly stored, an 'on-the-fly' approach is used in this chapter. Only the diagonal 5×5 blocks and their inverses are explicitly stored. The off-diagonal blocks are re-computed whenever needed in the linear iterations. Note that this is not a 'matrix-free' approach in the sense of finite differences, nor in the sense of tensor-product solvers. Moreover, the approximate inverses $\tilde{\boldsymbol{M}}$ in the case of ISAI iterations (section 5.4.2) are not stored explicitly either. Off-diagonal blocks of $\tilde{\boldsymbol{M}}$ are computed on-the-fly.

There is also a difference between the asynchronous linear solvers in this chapter versus those in chapter 5. While the asynchronous iterations there were "fully" asynchronous in the sense that there was no synchronization at the end of each sweep, here in the CUDA implementation, there is indeed synchronization. Because the loop over the sweeps executes on the host, and a kernel is launched in the same CUDA stream for each sweep, the implementation here is somewhat further away from true asynchronous iterations than the one in the previous chapter. The iterations are only asynchronous within each sweep. This holds for the implementation of both asynchronous block triangular solves as well as asynchronous block ILU factorization. Of course, this need not be the case - it is possible to write code that executes in a more asynchronous fashion; this will be explored in future work.

The other details of the solver remain the same as those described in section 5.2.

6.1.1 Asynchronous LUSGS relaxation

Since the original LUSGS + block Jacobi smoother (algorithm 11) is very efficient, an 'asynchronous LUSGS' smoother is also considered. This is similar to the asynchronous block SGS introduced in algorithm 5, but the difference is that algorithm 5 was only a local (subdomain) iteration. Here, boundary data (from halo cells) is used to compute the asynchronous updates. Finally, the solver here is implemented using a GPU-friendly structure-of-arrays layout like other solvers in this and the previous chapter, while algorithm 5 was implemented using an array-of-structures layout for multi-core CPUs. Below, $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{E} + \boldsymbol{F}$, the block diagonal, strictly block-lower and block-upper triangular parts. $\boldsymbol{r}$ is the nonlinear residual and $\delta\boldsymbol{w}$ is the nonlinear update in the inexact Newton step.

$$\delta\boldsymbol{w}^* = \boldsymbol{D}^{-1}(\boldsymbol{r} - \boldsymbol{E}\delta\boldsymbol{w}^* - \boldsymbol{F}\delta\boldsymbol{w}^n) \tag{6.3}$$

$$\delta\boldsymbol{w}^{n+1} = \boldsymbol{D}^{-1}(\boldsymbol{r} - \boldsymbol{E}\delta\boldsymbol{w}^* - \boldsymbol{F}\delta\boldsymbol{w}^{n+1}) \tag{6.4}$$

Algorithm 13 Async. LUSGS relaxation for solution of $\frac{\partial r}{\partial \boldsymbol{w}}\delta\boldsymbol{w} = -\boldsymbol{r}$

```
 1: $\delta\boldsymbol{w} \leftarrow \boldsymbol{0}$
 2: for $k$ in $\{1,2,...n_{swp}\}$ do
 3:     for $i$ from 1 to $n_{cell}$ do in parallel:
 4:         $\delta\boldsymbol{w}_i \leftarrow \boldsymbol{D}_i^{-1}(-\boldsymbol{r}_i - \sum_{j\in\mathcal{N}_L(i)} \boldsymbol{E}_{ij}\delta\boldsymbol{w}_j - \sum_{j\in\mathcal{N}_U(i)} \boldsymbol{F}_{ij}\delta\boldsymbol{w}_j)$
 5:     end for
 6:     for $i$ from $n_{cell} - 1$ to 1 do in parallel:
 7:         $\delta\boldsymbol{w}_i \leftarrow \boldsymbol{D}_i^{-1}(-\boldsymbol{r}_i - \sum_{j\in\mathcal{N}_L(i)} \boldsymbol{E}_{ij}\delta\boldsymbol{w}_j - \sum_{j\in\mathcal{N}_U(i)} \boldsymbol{F}_{ij}\delta\boldsymbol{w}_j)$
 8:     end for
 9:     Communicate $\delta\boldsymbol{w}$ in halo cells across subdomain boundaries
10: end for
```

Note the reversal in the direction of the second inner loop, which is achieved by reversing the map between CUDA threads and cell indices. Like the rest of the iterations in this chapter, algorithm 13 is not truly asynchronous in the Chazan-Miranker or Frommer-Szyld sense. It is only asynchronous within each sweep over the grid.

We use a CUDA implementation for all GPU code. For all of the solvers tested, the three-dimensional index space is manually flattened to a one-dimensional index space. One CUDA thread is assigned to one index, which maps to one mesh cell.

6.2 Experiments and preliminary results

All the runs in this section were carried out on one node with two Tesla K20 GPUs using two MPI ranks. CUDA version 8.0 was used to build the code. A limitation of the implementation is that loops are fine-grain parallel only within mesh blocks, though each GPU may be

	DPW W1 wing	ONERA-M6 wing	RAE2822 airfoil
Mach number M	0.76	0.8395	0.75
Reynolds number $R_e \times 10^6$	5.0	11.72	6.2
Angle of attack $\alpha(^\circ)$	0.6109	3.06	2.81
No. domain cells	1,510,000	884,736	294,912
Total cells (incl. halos)	1,930,000	1,290,000	1,510,000
Avg. domain cells per mesh block	116,263	27648	147,456
Avg. total cells per mesh block	148,426	36,288	754,000

Table 6.1: Physical properties and grid details of the cases

assigned more than one mesh block. This means that only one mesh block is smoothed at a time per GPU. The code already has point-block Jacobi and red-black Gauss-Seidel iterations, which we compare against the proposed solvers. The Spalart-Allmaras turbulence model is solved on the GPUs by several Jacobi iterations inside the inexact Newton loop. The number of Jacobi iteration is fixed to eight for all the cases considered in this chapter. We find that using more than eight does not improve convergence for these cases.

The physical properties are the same and grids used are very similar to those in chapter 3. They are given in table 6.1. For these three-dimensional wing cases, three multigrid levels are used. One post-smoothing nonlinear iteration is used per smoothing operation, each with 4, 4 and 10 linear solver iterations for the fine, medium and coarse levels respectively. As usual, a few fixed number of iterations are first done on coarser levels to initialize the solution on the fine grid.

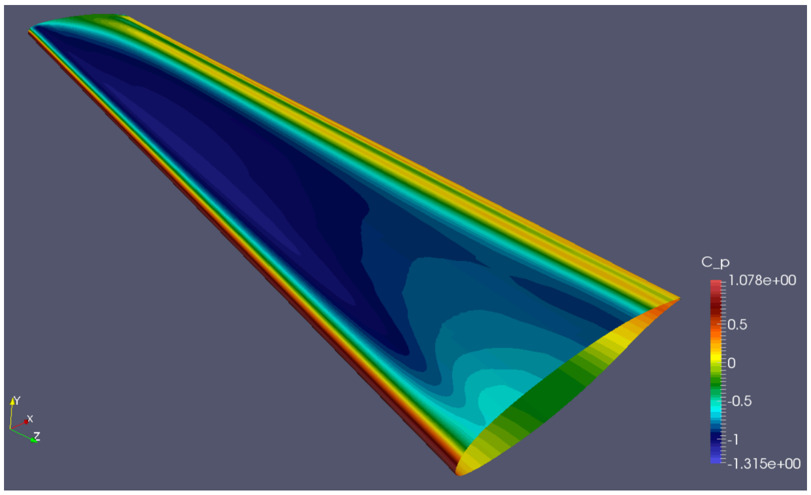


Figure 6.1: Non-dimensional pressure contours on the DPW-W1 wing

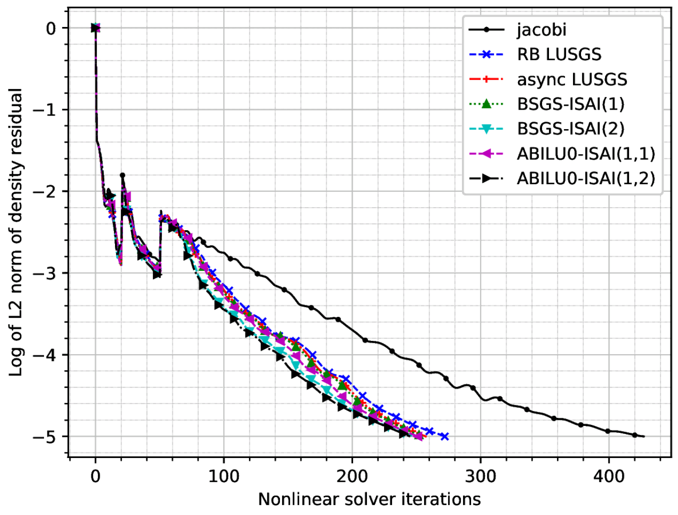


Figure 6.2: Convergence w.r.t. nonlinear iterations (FAS multigrid cycles) for different linear solvers for the DPW-W1 wing

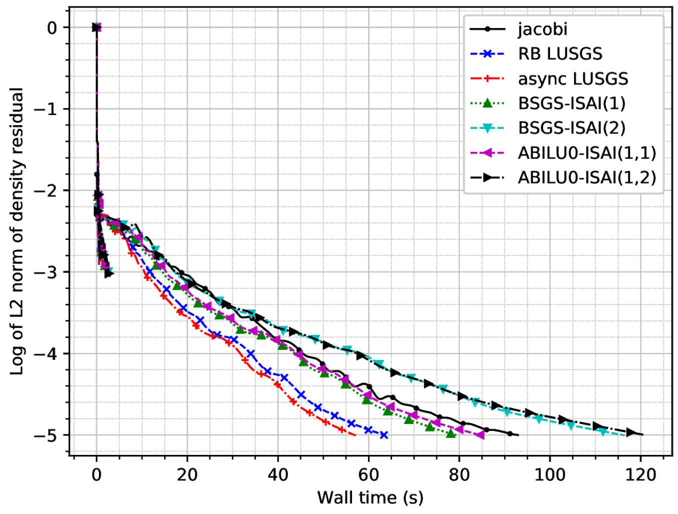


Figure 6.3: Convergence w.r.t. wall clock time for different linear solvers on the DPW-W1 wing case

The DPW-W1 wing case is relatively mild, with a low Reynolds number for external aerodynamics cases. The salient feature of the solution is a simple shock structure on the upper surface (figure 6.1). We see that in terms of iterations, this case is not very sensitive to the linear solver (figure 6.2). With this nonlinear solver setup, it can be seen that this case is not challenging by the fact that a simple point-block Jacobi smoother converges. Because the number of nonlinear iterations is almost the same for the solvers being compared, in terms of wall-clock time, the most inexpensive linear iterations largely win. The frugal LUSGS type iterations (algorithm 11) outperform the expensive inner ISAI iterations. Preconditioned

Richardson iterations (algorithm 10) have an extra matrix-vector which also increases the cost of the ABILU-ISAI and BSGS-ISAI iterations. The effects of this were seen in the results of the previous chapter as well 5.5.2). Among the LUSGS-type solvers, we note that asynchronous LUSGS is faster than red-black Gauss-Seidel by a small margin.

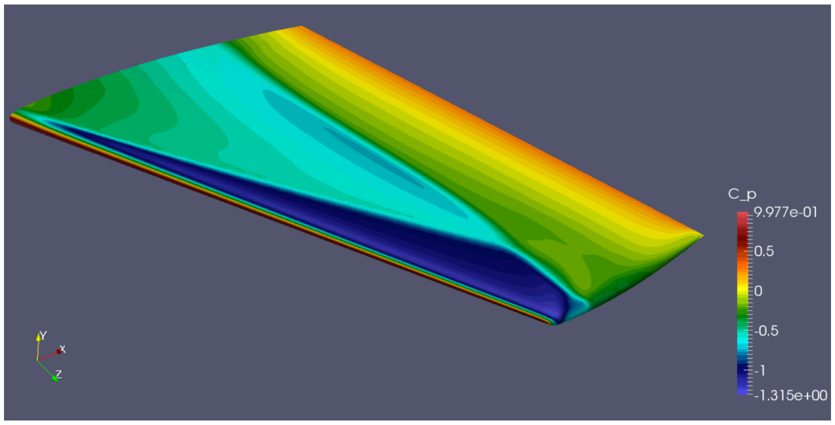


Figure 6.4: Non-dimensional pressure contours on the ONERA-M6 wing

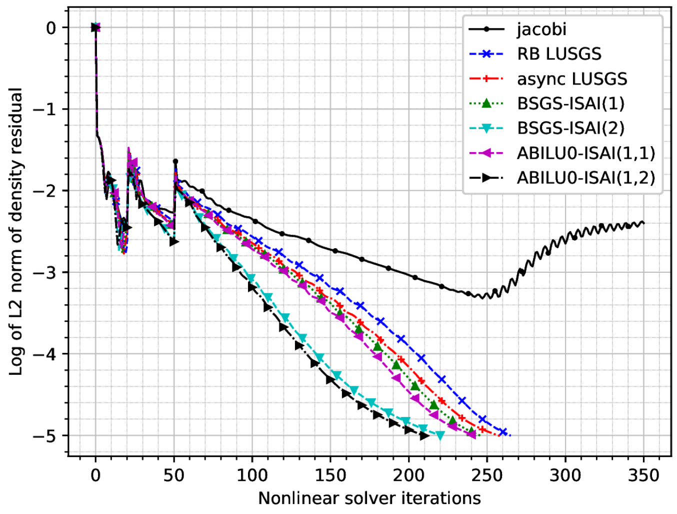


Figure 6.5: Convergence w.r.t. nonlinear iterations (FAS multigrid cycles) for different linear solvers on the ONERA-M6 wing

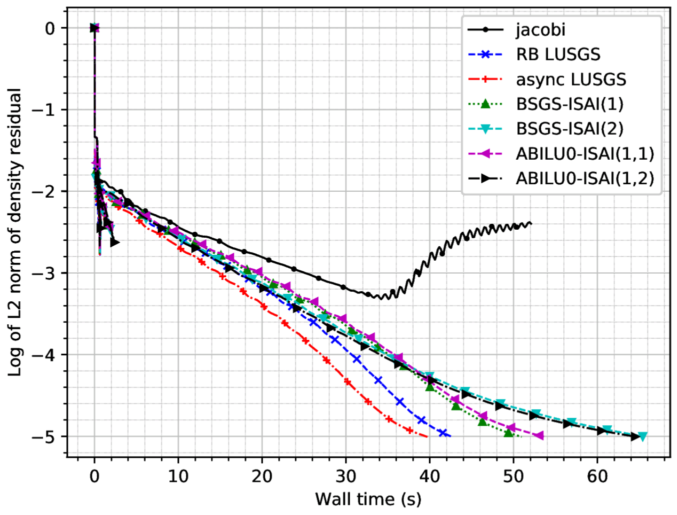


Figure 6.6: Convergence w.r.t. wall clock time for different linear solvers on the ONERA-M6 wing

The ONERA-M6 wing is a somewhat more involved case. A lambda shock (with two shocks intersecting) is formed on the upper surface (figure 6.4), and the Reynolds number is higher. The grid is shown in figure 6.8. For this case, point-block Jacobi does not converge. There is also some difference among the other solvers in terms of convergence, with the stronger solvers such as asynchronous ILU(0) with ISAI triangular solves (ABILU-ISAI) allowing the nonlinear solver to converge in less iterations (figure 6.5). However, asynchronous LUSGS and red-black Gauss-Seidel are still the fastest (figure 6.6) - the extra work done by the ISAI-based solvers has undermined the better convergence properties. Asynchronous LUSGS continues to perform better than the red-black variant.

Finally, we run a two-dimensional case, the RAE-2822 airfoil case 10 [105]. For this case, a single-grid solver is used, though coarser grid levels are still used to initialize the solution on the primary grid. Eight linear iterations are used per nonlinear iteration.

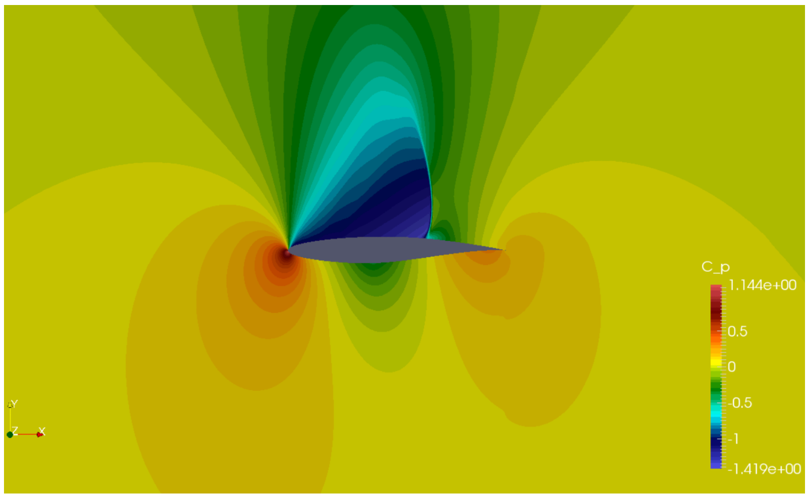


Figure 6.7: Non-dimensional pressure contours around RAE-2822 airfoil

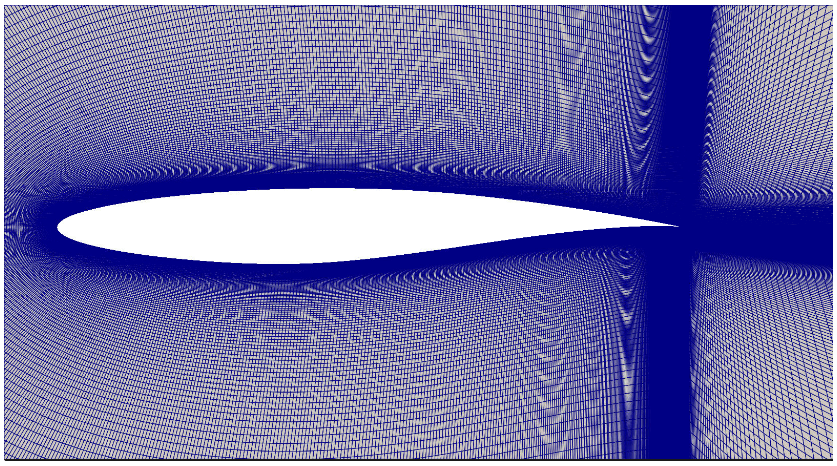


Figure 6.8: Grid used for the RAE 2822 airfoil case

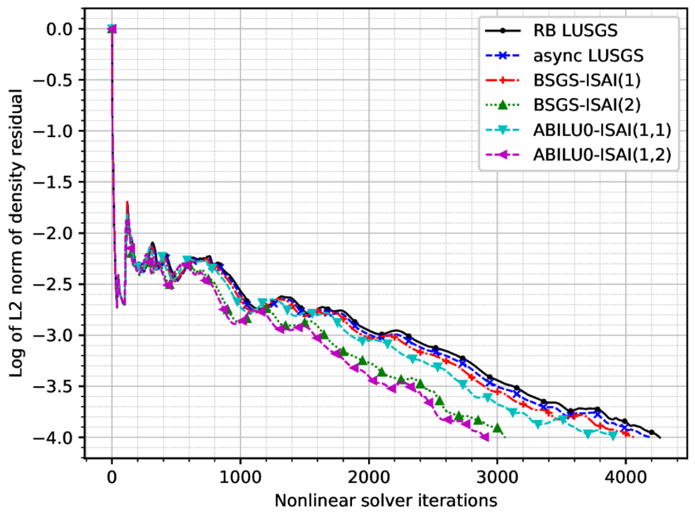


Figure 6.9: Convergence w.r.t. nonlinear iterations (inexact Newton steps) for different linear solvers on the RAE-2822 airfoil case

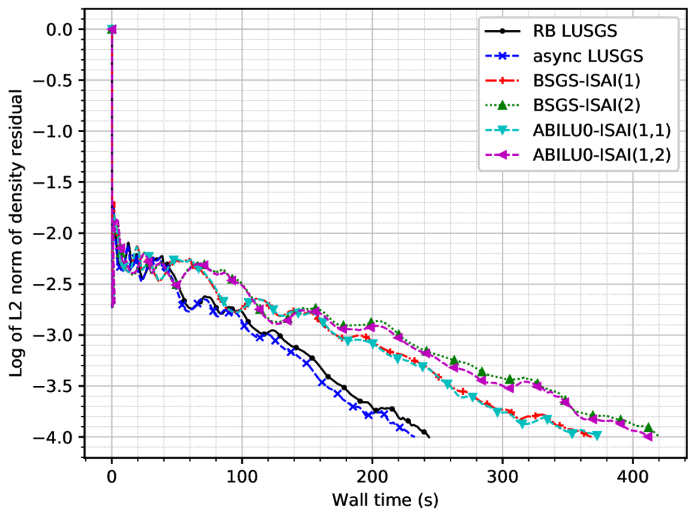


Figure 6.10: Convergence w.r.t. wall clock time for different linear solvers on the RAE-2822 airfoil case

This case is more challenging. The shape of the airfoil and the flow conditions create a relatively strong shock in a region of adverse pressure gradient (figure 6.7), causing the flow to separate. Though it is not shown, point-block Jacobi immediately blows up for this case. We see that BSGS-ISAI(2) and ABILU0-ISAI(1,2) need much fewer iterations than the other solvers (figure 6.9). However, these stronger preconditioned Richardson solvers are still slower than the relaxation-type solvers (red-black Gauss-Seidel and asynchronous LUSGS) in terms of wall-clock time (figure 6.10), once again pointing to the efficiency of the tailored relaxation implementation in FANSC-Lite.

The fact that asynchronous LUSGS is consistently slightly better than red-black Gauss-Seidel may be attributed to two reasons. In terms of iterations, red-black Gauss-Seidel decomposes each of iteration into two Jacobi-type iterations. In each Jacobi-type partial iteration, there is little propagation of signals throughout the mesh. Asynchronous LUSGS retains some Gauss-Seidel nature by using some of the new data in each linear iteration, providing somewhat greater transmission of updates in the mesh. In terms of cost per iteration, red-black Gauss-Seidel does not have a good memory access pattern; every other 'black' cell is skipped for performing an update on the 'red' cells, and vice-versa, in a chequerboard pattern. Since the cells in the structured grid are not reordered, this leads to either partial loss of coalesced memory access or thread divergence depending on the implementation. Asynchronous LUSGS does not have this issue.

6.3 CRM wing-body case

We now run a more complex geometry, the CRM wing-body case [109] shown in figures 3.1 and 3.2. This case is expected to be more demanding of the solver because of the wing-body junction and the presence of a region of recirculating flow near the wing root. However, we make some changes to the nonlinear solver. Firstly, the Jacobian matrix now contains boundary contributions from the solid wall and symmetry plane boundaries, making it more accurate. Second, the nonlinear iteration used for smoothing is now backward Euler rather than just inexact Newton. We found that this was required in case of most linear iterations once the linearized boundary conditions were added to the Jacobian matrix. The CFL number is ramped exponentially as a function of the ratio of consecutive residual norms, as was done in chapter 3.

The physical parameters remain the same as in previous chapters - $M = 0.85$, $R_e = 5\times10^6$ and $\alpha = 2.11°$. The 'L3' grid contains 5,111,808 domain cells, which grows to 6,055,296 including halo cells used for communication and boundary conditions. It is divided into 14 blocks, each with an average of 365129 domain cells. The case is run on two K20 GPUs, with each GPU having an equal number of cells and blocks. We perform a full multigrid initialization on two coarser grids to start the solver on the original fine grid, where a three-grid V-cycle FAS multigrid is used. Parameters of the nonlinear solver are given in table 6.2. In each backward Euler smoothing step, 4, 4 and 10 linear iterations are performed on fine, medium and coarse levels respectively; these numbers are unchanged from the previous section.

Figure 6.11 shows the convergence histories of the solvers tested. For this case, we see that there is a wide variation in the number of FAS multigrid iterations required for

Full multigrid level	Starting CFL	Maximum CFL	Iterations
Coarse	20	1000	20
Medium	500	10000	30
Fine	5000	100000	Until convergence

Table 6.2: Details of CFL numbers and full multigrid initialization for CRM wing-body case

convergence. We also see a consistent difference in slopes once the residual drops about 4 orders of magnitude, indicating different asymptotic convergence rates. ABILU-ISAI(1,3) is has the best convergence rate, as shown by the high slope of its convergence curve, closely followed by ABILU-async(1,3). Point-block Jacobi smoothing leads to divergence of the nonlinear solver and is omitted from the plots. We include certain variants of red-black Gauss-Seidel introduced by Nguyen et al. [70], labelled as 'RB line i' and 'RB plane j'. These are mathematically less effective than regular red-black Gauss-Seidel, but have better memory access patterns by treating i-lines and j-planes of cells, respectively, in a Jacobi fashion. Figure 6.12 shows convergence plots with respect to wall-clock time. Here, we see the asynchronous block-ILU(0) solvers outperforming all the others. We also see that a stronger triangular solve using more application sweeps pays off. Next, BSGS-ISAI(2) converges in less time than red-black Gauss-Seidel, and also does better than asynchronous LUSGS. Further, even with just one ISAI iteration for the triangular systems, BSGS-ISAI(1) performs better than red-black Gauss-Seidel in terms of iterations and similar to it in terms of wall-clock time. This case shows the limitation of the red-black Gauss-Seidel iterations.

We have added a GPU implementation of the BSGS-async solver here. As described in section 5.4.1, this is a point-block SGS preconditioned Richardson iteration with asynchronous block forward and backward triangular solves. In section 5.5.1, it was observed that this iteration fails to scale well with increasing core counts. Consistent with that, we observe worse performance for BSGS-async iterations compared to the corresponding BSGS-ISAI iterations. The BSGS-async(1) iteration has asymptotic convergence worse than red-black Gauss-Seidel. Clearly, just one application sweep is not the recommended setting, though it still converges. BSGS-async(2), however, shows significantly better performance. In a traditional synchronous setting, the BSGS preconditioned Richardson iteration would be mathematically equivalent to LUSGS relaxation. However, when carried out asynchronously, they have significantly different convergence rates depending on the number of asynchronous sweeps used per BSGS iteration.

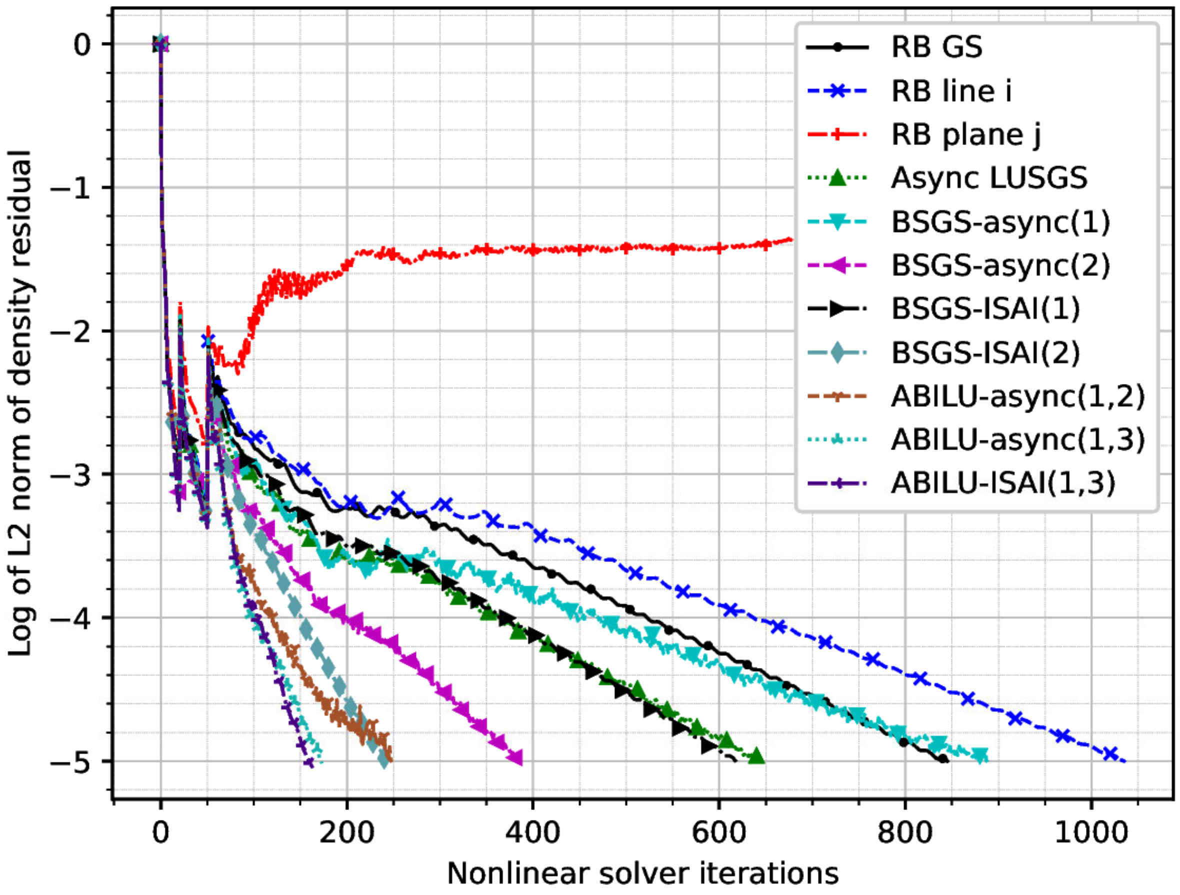


Figure 6.11: Convergence w.r.t. nonlinear iterations (FAS multigrid cycles) for different linear solvers on the CRM wing-body case

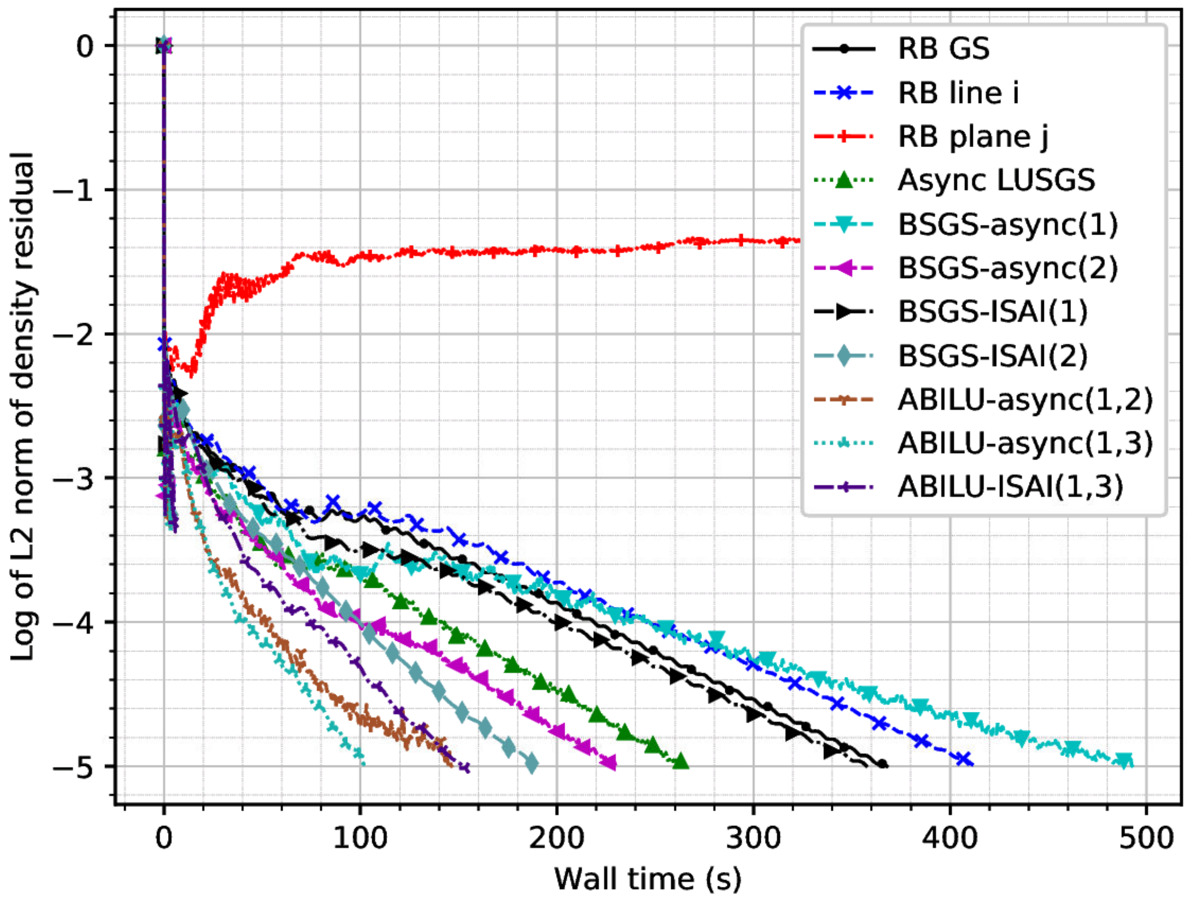


Figure 6.12: Convergence w.r.t. wall clock time for different linear solvers on the CRM wing-body case

6.4 Conclusions

From the results in this chapter, one observes that asynchronous iterations and ISAI triangular solvers are a promising option for use in multigrid solvers on GPUs for compressible flow problems.

For the simpler wing cases, asynchronous LUSGS usually performs better than red-black Gauss-Seidel. While stronger solvers such as asynchronous ILU(0) with ISAI triangular solves require less iterations, they require extra work per iteration and are thus slower in wall-clock time for these relatively simple cases with highly approximate Jacobian matrices. Asynchronous LUSGS finds a good compromise between cheap iterations and good smoothing.

The more challenging CRM wing-body case shows the advantage of the new solvers compared to traditional Jacobi and red-black Gauss-Seidel iterations. The improved smoothing of the BSGS-ISAI solvers, for example, now leads to such a large reduction in the required number of multigrid cycles that they are significantly faster in wall-clock time as well.

Chapter 7

Conclusion

The results shown in this thesis demonstrate that asynchronous iterations, and their combination with incomplete sparse approximate inverses, are strong candidates for fine-grain parallel iterations in computational fluid dynamics codes. To our knowledge, this is the first body of work that demonstrates this for compressible turbulent flows.

Block-asynchronous point-block symmetric Gauss-Seidel (SGS) and incomplete LU (ILU) factorization iterations have been analyzed and shown to perform well for compressible flow cases. An extension of the convergence theory of asynchronous iterations is provided here for our point-block iterations. The question of grid orderings for obtaining good convergence has also been explored, and a line-hybrid ordering scheme is proposed. This ordering scheme shows good potential for unstructured grid CFD codes. We saw that for the developed point-block iterations and grid orderings, the smoothing and preconditioning effectiveness remain essentially independent of the number of parallel processors, which is a very promising result.

Implementations on three major hardware architectures have been demonstrated - multi-core CPUs, many-core CPUs with wide vector units, and general-purpose graphics processing units (GPUs). Due to the nature of asynchronous iterations, differences in hardware features affect their convergence behaviour and smoothing properties. In each case it is seen that convergence and good performance can be obtained. On GPUs, the most highly parallel devices used in this work, the developed asynchronous point-block iterations provide significantly better performance compared to existing point-block red-black Gauss-Seidel iterations. A strength of the asynchronous ILU and SGS iterations is that they do not need any special ordering for parallelism. The ordering, along with the layout, can be tuned to specific applications and hardware architectures, as has been done in this work. Further, unlike other algorithms for parallel ILU and SGS, load balancing is automatically addressed by asynchronous iterations.

7.1 Future work

To be sure, there are several un-answered questions about asynchronous iterations. One apparent disadvantage of asynchronous iterations is their chaotic nature and sensitivity to the matrix structure, storage layout, hardware architecture and, at times, minute details of the kernel code. However, we have observed that for cases where an asynchronous solver converges, it reliably gives the correct solution. Moreover, for repeated runs of the same algorithm on the same hardware, the variance in the number iterations and even wall-clock time is small in percentage terms. A more detailed study of this variance is required. Strikwerda's work [125] is notable in having considered the convergence of the variance of the asynchronous iterates.

While experimental evidence of smoothing property is given in this work, a theoretical analysis of this remains to be done. A theoretical demonstration of smoothing property for the convection-diffusion equation, for instance, would help to increase confidence in these methods. However, techniques from abstract multigrid analysis or local Fourier analysis may have to be combined with statistical methods owing to the chaotic nature of these iterations. Again, the work of Strikwerda [125] may be a good starting point for this. With regard to unstructured grids, further research is needed on the interaction between grid ordering and the effectiveness of asynchronous smoothers, as the standard choice of reverse Cuthill-McKee ordering is seen to be unreliable. We expect that performance analysis will be important in shedding some light on this issue. Finally, a GPU-friendly implementation for unstructured grids is currently missing. The influence of matrix storage formats and the implementation of line- and line-hybrid orderings on GPU performance are questions to be considered.

Ultimately, the goal is the development of a scalable ($\mathcal{O}(N)$ or $\mathcal{O}(N \log N)$) solver that also shows good weak scaling in parallel. In this broader picture, the iterations developed here are potential candidates for one piece of the puzzle. Our expectation is that these iterations can be used in smoothers for parallel multi-level methods.

Glossary

cluster A networked collection of nodes, each containing a few relatively tightly integrated processor devices. 2, 36, 42, 53, 56, 79, 116

kernel A common set of instructions which is executed in every work item on different entries of an input collection to complete a task. 43, 148

node One more or less self-contained computing system or server in a networked collection; typically contains one to four central processing units (CPUs), memory, secondary storage and accelerator devices. 1, 2, 36, 42, 43, 52, 53, 55–58, 62–64, 79, 126, 130, 132, 135, 149

processing element One hardware unit (among many such units in a computer) capable of performing a sequence of operations on a given unit of data; eg. SIMD vector lane, thread context, GPU stream processor, CPU core etc.. 4–6, 43, 47, 48, 89, 149

SIMD Single Instruction Multiple Data, a paradigm of data parallelism where different processing elements execute the same instruction, in lockstep, but on different data instances. 6, 9, 36, 43, 47, 116, 119, 149

SMT Simultaneous multi-threading, a hardware feature by which processing elements can store the state associated with several different tasks simultaneously and therefore can switch between those tasks easily. 43, 57

work item A small unit of work that typically can be performed independently of other such units executing concurrently, to complete a larger task; eg. one addition performed in the process of computing the sum of two vectors. 39, 43, 46, 47, 56, 117, 149

Bibliography

[1] E. Lindholm et al. "NVIDIA Tesla: A unified graphics and computing architecture". In: *IEEE Micro* 28.2 (2008).

[2] A. Sodani et al. "Knights Landing: second-generation Intel Xeon Phi product". In: *IEEE Micro* 36.2 (2016).

[3] M. Ferronato. "Preconditioning for sparse linear systems at the dawn of the 21st century: history, current developments, and future perspectives". In: *ISRN Applied Mathematics* 2012.127647 (2012). URL: `http://dx.doi.org/10.5402/2012/127647`.

[4] *The Cerebras wafer scale engine.* URL: `https://www.cerebras.net/product/#chip` (visited on 03/29/2020).

[5] R.C. Swanson, E. Turkel, and C.-C. Rossow. "Convergence acceleration of Runge-Kutta schemes for solving Navier-Stokes equations". In: *Journal of Computational Physics* 224 (2007), pp. 365–388.

[6] Hong Luo et al. "On the computation of compressible turbulent flows on unstructured grids". In: *International Journal of Computational Fluid Dynamics* 14.4 (2001), pp. 253–270. URL: `http://dx.doi.org/10.1080/10618560108940728`.

[7] Peter Eliasson and Per Weinerfelt. "High-order implicit time integration for unsteady turbulent flow simulations". In: *Computers & Fluids* 112 (2015), pp. 35–49.

[8] Cetin Kiris and Dochan Kwak. "Numerical solution of incompressible Navier-Stokes equations using a fractional-step approach". In: *Computers and Fluids* 30 (2001), pp. 829–851.

[9] Aditya K. Pandare and Hong Luo. "A hybrid reconstructed discontinuous Galerkin and continuous Galerkin finite element method for incompressible flows on unstructured grids". In: *Journal of Computational Physics* 322 (2016), pp. 491–510. DOI: `https://doi.org/10.1016/j.jcp.2016.07.002`.

[10] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra.* Philadelphia, PA: SIAM, 1997.

[11] M. Benzi. “Preconditioning techniques for large linear systems: a survey”. In: *Journal of Computational Physics* 182 (2002), pp. 418–477.

[12] M. Naumov. *Parallel incomplete-LU and Cholesky factorization in the preconditioned iterative methods on the GPU.* Tech. rep. NVR-2012-003. NVIDIA, 2012.

[13] B.F. Smith, P.E. Bjørstad, and W.D. Gropp. *Domain decomposition - parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, 1996.

[14] W.L. Briggs, V.E. Henson, and S.F. McCormick. *A Multigrid Tutorial.* 2nd. 2000. DOI: 10.1137/1.9780898719505.

[15] Pieter Wesseling and Cornelis W. Oosterlee. “Geometric multigrid with applications to computational fluid dynamics”. In: *Journal of Computational and Applied Mathematics* 128.1 (2001), pp. 311–334. URL: http://www.sciencedirect.com/science/article/pii/S0377042700005173.

[16] Antony Jameson. “Solution of the Euler equations for two-dimensional transonic flow by a multigrid method”. In: *Applied Mathematics and Computation* 13 (1983), pp. 327–355.

[17] J.S. Cagnone et al. “Implicit multigrid schemes for challenging aerodynamic simulations on block-structured grids”. In: *Computers & Fluids* 44 (2011), pp. 314–327.

[18] Edmond Chow et al. “A survey of parallelization techniques for multigrid solvers”. In: *Parallel Processing for Scientific Computing.* Chap. 10, pp. 179–201. DOI: 10.1137/1.9780898718133.ch10. URL: http://epubs.siam.org/doi/abs/10.1137/1.9780898718133.ch10.

[19] *Whitepaper: NVIDIA Tesla P100.* Tech. rep. WP-08019-001 v01.1. NVIDIA, 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[20] Konstantinos I. Karantasis, Eleftherios D. Polychronopoulos, and John A. Ekaterinaris. “High order accurate simulation of compressible flows on GPU clusters over Software Distributed Shared Memory”. In: *Computers & Fluids* 93 (2014), pp. 18–29. ISSN: 00457930. URL: https://linkinghub.elsevier.com/retrieve/pii/S0045793014000127.

[21] Sparsh Mittal and Jeffrey S. Vetter. “A survey of methods for analyzing and improving GPU energy efficiency”. In: *ACM Computing Surveys (CSUR)* 47.2 (2015), p. 19. URL: http://dl.acm.org/citation.cfm?id=2636342.

[22] S. Huang, S. Xiao, and W. Feng. "On the energy efficiency of graphics processing units for scientific computing". In: *2009 IEEE International Symposium on Parallel Distributed Processing*. May 2009, pp. 1–8. DOI: `10.1109/IPDPS.2009.5160980`.

[23] Aditya Kashi, Syam Vangara, and Sivakumaran Nadarajah. "Asynchronous fine-grain parallel smoothers for computational fluid dynamics". In: *2018 Fluid Dynamics Conference*. 3558. Americal Institute of Aeronautics and Astronautics, 2018. DOI: `10.2514/6.2018-3558`.

[24] Aditya Kashi et al. "Asynchronous fine-grain parallel implicit smoother in multigrid solvers for compressible flow". In: *Computers and Fluids* 198.104255 (2020). URL: `https://doi.org/10.1016/j.compfluid.2019.104255`.

[25] Aditya Kashi and Siva Nadarajah. *A fine-grain parallel asynchronous incomplete block LU preconditioner for computational fluid dynamics on unstructured grids*. 2020. arXiv: `1912.00539 [math.NA]`.

[26] Aditya Kashi and Sivakumaran Nadarajah. "Fine-grain parallel smoothing by asynchronous iterations and incomplete aparse approximate inverses for computational fluid dynamics". In: *AIAA Scitech 2020 Forum*. 0806. Americal Institute of Aeronautics and Astronautics, 2020. URL: `https://doi.org/10.2514/6.2020-0806`.

[27] E. Laurendeau, Z. Zhu, and F. Mokhtarian. "Development of the FANSC Full Aircraft Navier-Stokes Code". In: *Proceedings of the 46th Annual Conference of the Canadian Aeronautics and Space Institute*. Montreal, 1999.

[28] Jiri Blazek. *Computational Fluid Dynamics - Principles and Applications*. 3rd ed. Elsevier Ltd., 2015.

[29] R.C. Swanson and Eli Turkel. "On central-difference and upwind schemes". In: *Journal of Computational Physics* 101 (1992), pp. 292–306.

[30] P.L. Roe. "Approximate Riemann solvers, parameter vectors, and difference schemes". In: *Journal of Computational Physics* 43.2 (1981), pp. 357–372.

[31] E.F. Toro, M. Spruce, and W. Speares. "Restoration of the contact surface in the HLL-Riemann solver". In: *Shock Waves* 4 (1994), pp. 25–34.

[32] P. Batten et al. "On the choice of wavespeeds for the HLLC Riemann solver". In: *SIAM Journal of Scientific Computing* 18.6 (1997), pp. 1553–1570.

[33] Achi Brandt. "Multi-level adaptive solutions to boundary-value problems". In: *Mathematics of Computation* 31.138 (1977), pp. 333–390. ISSN: 0025-5718, 1088-6842. DOI: `10.1090/S0025-5718-1977-0431719-X`. URL: `http://www.ams.org/mcom/1977-31-138/S0025-5718-1977-0431719-X/` (visited on 04/19/2017).

[34] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd. SIAM, 2003.

[35] Wolfgang Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*. Springer Verlag, 1994.

[36] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. 3rd ed. The Johns Hopkins University Press, 1996.

[37] U. Trottenberg, C.W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001. ISBN: 978-0-12-701070-0.

[38] L.C. Dutto. "The effect of ordering on preconditioned GMRES algorithm, for solving the compressible Navier-Stokes equations". In: *International Journal for Numerical Methods in Engineering* 36 (1993), pp. 457–497.

[39] Todd T. Chisholm and David W. Zingg. "A Jacobian-free Newton-Krylov algorithm for compressible turbulent fluid flows". In: *Journal of Computational Physics* 228 (2009), pp. 3490–3507.

[40] L. Luo et al. "A fine-grained block ILU scheme on regular structures for GPGPUs". In: *Computers and Fluids* 119 (2015), pp. 149–161.

[41] Wolfgang Hackbusch. *Multi-grid Methods and Applications*. Springer Verlag Berlin Heidelberg, 1985.

[42] Gabriel Wittum. "On the robustness of ILU smoothing". In: *SIAM Journal on Scientific and Statistical Computing* 10.4 (1989), pp. 699–717.

[43] M. Benzi, D. B. Szyld, and A. van Duin. "Orderings for incomplete factorization preconditioning of nonsymmetric problems". In: *SIAM J. Sci. Comput.* 20 (1999), pp. 1652–1670.

[44] Elizabeth Cuthill and James McKee. "Reducing the bandwidth of sparse symmetric matrices". In: *Proceedings of the 24th National Conference*. New York: Association for Computing Machinery, 1969, pp. 157–172.

[45] Alan George. "Nested dissection of a regular finite element mesh". In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363.

[46] Seokkwan Yoon and Antony Jameson. "Lower-upper symmetric-Gauss-Seidel method for the Euler and Navier-Stokes equations". In: *AIAA Journal* 26.9 (1988), pp. 1025–1026.

[47] Hong Luo, Joseph D. Baum, and Rainald Löhner. "A fast, matrix-free implicit method for compressible flows on unstructured grids". In: *Journal of Computational Physics* 146 (1998), pp. 664–690.

[48] Dimitri J. Mavriplis. “An assessment of linear versus nonlinear multigrid methods for unstructured mesh solvers”. In: *Journal of Computational Physics* 175.1 (Jan. 2002), pp. 302–325. ISSN: 00219991. DOI: `10.1006/jcph.2001.6948`.

[49] R. P. Fedorenko. “The speed of convergence of one iterative process”. In: *USSR Computational Mathematics and Mathematical Physics* 4.3 (Jan. 1964), pp. 227–235. ISSN: 0041-5553. DOI: `10.1016/0041-5553(64)90253-8`. URL: `http://www.sciencedirect.com/science/article/pii/0041555364902538` (visited on 04/19/2017).

[50] Cord C. Rossow. “Convergence acceleration for solving the compressible Navier-Stokes equations”. en. In: *AIAA Journal* 44.2 (2006), pp. 345–352. DOI: `10.2514/1.15636`.

[51] Mark Adams et al. “Parallel multigrid smoothing: polynomial versus Gauss-Seidel”. In: *Journal of Computational Physics* 188 (2003), pp. 593–610.

[52] Youcef Saad and Martin H. Schultz. “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems”. In: *SIAM Journal on Scientific and Statistical Computing* 7.3 (1986), pp. 856–869. URL: `http://epubs.siam.org/doi/abs/10.1137/0907058`.

[53] Henk 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 (1992), pp. 631–644. URL: `http://epubs.siam.org/doi/abs/10.1137/0913035`.

[54] Mark K. Seager. “Parallelizing conjugate gradient for the CRAY X-MP”. In: *Parallel Computing* 3.1 (1986), pp. 35–47. ISSN: 0167-8191. DOI: `http://dx.doi.org/10.1016/0167-8191(86)90005-0`. URL: `http://www.sciencedirect.com/science/article/pii/0167819186900050`.

[55] Maryam Mehri Dehnavi. “Krylov subspace techniques on graphic processing units”. PhD thesis. Department of Electrical and Computer Engineering: McGill University, July 2012.

[56] Bo Yang, Hui Liu, and Zhangxin Chen. “Preconditioned GMRES solver on multiple-GPU architecture”. In: *Computers & Mathematics with Applications* 72.4 (Aug. 2016), pp. 1076–1095. ISSN: 08981221. DOI: `10.1016/j.camwa.2016.06.027`. URL: `http://linkinghub.elsevier.com/retrieve/pii/S0898122116303595`.

[57] Kai He et al. "Parallel GMRES solver for fast analysis of large linear dynamic systems on GPU platforms". In: *Integration, the VLSI Journal* 52 (Jan. 2016), pp. 10–22. ISSN: 01679260. DOI: `10.1016/j.vlsi.2015.07.005`. URL: `http://linkinghub.elsevier.com/retrieve/pii/S016792601500084X`.

[58] *NVIDIA CUSPARSE and CUBLAS libraries*. `http://www.nvidia.com/object/cuda_develop.html`.

[59] Hartwig Anzt et al. "Acceleration of GPU-based Krylov solvers via data transfer reduction". In: *The International Journal of High Performance Computing Applications* 29.3 (2015), pp. 366–383. DOI: `10.1177/1094342015580139`. URL: `http://dx.doi.org/10.1177/1094342015580139`.

[60] James N. Hawkes et al. "Chaotic linear-system solvers for unsteady CFD". In: *VI International Conference on Computational Methods in Marine Engineering MARINE*. Ed. by F. Salvatore, R. Broglia, and R. Muscari. 2015.

[61] P. Ghysels et al. "Hiding global communication latency in the GMRES algorithm on massively parallel machines". In: *SIAM Journal on Scientific Computing* 35.1 (2013), pp. C48–C71.

[62] Mark Hoemmen. "Communication-avoiding Krylov subspace methods". PhD thesis. University of California at Berkeley, 2010.

[63] Erin Claire Carson. "Communication-avoiding Krylov subspace methods in theory and practice". PhD thesis. University of California at Berkeley, 2015.

[64] Dimitri J. Mavriplis. *Parallel performance investigations of an unstructured mesh Navier-Stokes solver*. ICASE Report 2000-13. Hampton, VA, USA: NASA, Mar. 2000.

[65] E. Chow and A. Patel. "Fine-grained parallel incomplete LU factorization". In: *SIAM Journal on Scientific Computing* 37.2 (2015), pp. C169–C193.

[66] Mohamed Aissa, Tom Verstraete, and Cornelis Vuik. "Toward a GPU-aware comparison of explicit and implicit CFD simulations on structured meshes". In: *Computers & Mathematics with Applications* 74.2 (2017), pp. 201–217. ISSN: 0898-1221. DOI: `http://doi.org/10.1016/j.camwa.2017.03.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0898122117301438`.

[67] Maxim Naumov. *Parallel sparse triangular systems in the preconditioned iterative methods on the GPU*. Tech. rep. NVR-2011-001. NVIDIA, 2011.

[68] Seiji Fujino and Shun Doi. "Optimizing multicolor ICCG methods on some vector computers". In: *Iterative Methods in Linear Algebra*. Ed. by R. Beauwens and P. de Groen. North-Holland: Elsevier Science Publishers B.V., 1992, pp. 349–358.

[69] Michele Benzi, Wayne Joubert, and Gabriel Mateescu. "Numerical experiments with parallel orderings for ILU preconditioners". In: *Electronic Transactions on Numerical Analysis* 8 (1999), pp. 88–114. ISSN: 1068-9613.

[70] M. T. Nguyen, P. Castonguay, and E. Laurendeau. "GPU parallelization of multigrid RANS solver for three-dimensional aerodynamic simulations on multiblock grids". In: *The Journal of Supercomputing* 75.5 (2019), pp. 2562–2583. ISSN: 0920-8542, 1573-0484. DOI: `10.1007/s11227-018-2653-6`.

[71] M.J. Grote and T. Huckle. "Parallel preconditioning with sparse approximate inverses". In: *SIAM Journal on Scientific Computing* 18.3 (1997), pp. 838–853.

[72] Hartwig Anzt et al. "Incomplete sparse approximate inverses for parallel preconditioning". In: *Parallel Computing* 71 (Supplement C 2018), pp. 1–22. ISSN: 0167-8191. DOI: `10.1016/j.parco.2017.10.003`.

[73] Michele Benzi and Miroslav Tuma. "A sparse approximate inverse preconditioner for nonsymmetric linear systems". In: *SIAM Journal on Scientific Computing* 19.3 (1998), pp. 968–994. URL: `http://epubs.siam.org/doi/abs/10.1137/S1064827595294691` (visited on 01/12/2017).

[74] Steven Dalton et al. *Cusp: Generic parallel algorithms for sparse matrix and graph computations*. Version 0.5.0. 2014. URL: `http://cusplibrary.github.io/`.

[75] Oliver Bröker et al. "Robust parallel smoothing for multigrid via sparse approximate inverses". In: *SIAM Journal of Scientific Computing* 23.4 (2001), pp. 1396–1417.

[76] X.-C. Cai et al. "Newton-Krylov-Schwarz methods in CFD". In: *Numerical methods for the Navier-Stokes equations*. Ed. by F.-K. Hebeker et al. Springer Fachmedien Wiesbaden, 1994, pp. 17–30.

[77] Chih-Hao Chen, Siva Nadarajah, and Patrice Castonguay. "A dynamically deflated GMRES adjoint solver for aerodynamic shape optimization". In: *Computers and Fluids* 179 (2019), pp. 490–507.

[78] Caroline Lasser and Andrea Toselli. "An overlapping domain decomposition preconditioner for a class of discontinuous Galerkin approximations of advection-diffusion problems". In: *Mathematics of Computation* 72.243 (2003), pp. 1215–1238. ISSN: 0025-5718. URL: `http://www.jstor.org/stable/4099830`.

[79] William Gropp et al. "Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD". In: *International Journal of High Performance Computing Applications* 14.2 (2000), pp. 102–136.

[80] Marzio Sala, Penelope Leyland, and Angelo Casagrande. "A parallel adaptive Newton-Krylov-Schwarz method for 3D compressible inviscid flow simulations". In: *Modelling and Simulation in Engineering* 2013.694354 (2013). URL: `http://dx.doi.org/10.1155/2013/694354`.

[81] Allison H. Baker et al. "Multigrid smoothers for ultraparallel computing". In: *SIAM Journal of Scientific Computing* 33.5 (2011), pp. 2864–2887.

[82] *OpenMP Application Programming Interface*. 4.5. OpenMP Architecture Review Board. Nov. 2015.

[83] D. Chazan and W. Miranker. "Chaotic relaxation". In: *Linear Algebra and its Applications* 2.2 (1969), pp. 199–222. ISSN: 0024-3795.

[84] John C. Strikwerda. "A convergence theorem for chaotic asynchronous relaxation". In: *Linear Algebra and its Applications* 253.1 (1997), pp. 15–24. URL: `http://www.sciencedirect.com/science/article/pii/0024379595006982`.

[85] Gerard M. Baudet. "Asynchronous iterative methods for multiprocessors". In: *Journal of the Association for Computing Machinery* 25.2 (1978), pp. 226–244.

[86] Andreas Frommer and Daniel B. Szyld. "On asynchronous iterations". In: *Journal of Computational and Applied Mathematics* 123.1 (2000), pp. 201–216.

[87] Mouhamed Nabih El Tarazi. "Some convergence results for asynchronous algorithms". In: *Numerische Mathematik* 39.3 (1982), pp. 325–340.

[88] M. N. Anwar and M. N. El Tarazi. "Asynchronous algorithms for Poisson's equation with nonlinear boundary conditions". In: *Computing* 34.2 (June 1985), pp. 155–168. ISSN: 1436-5057. DOI: `10.1007/BF02259842`. URL: `https://doi.org/10.1007/BF02259842`.

[89] Edmond Chow, Hartwig Anzt, and Jack Dongarra. "Asynchronous iterative algorithm for computing incomplete factorizations on GPUs". In: *High Performance Computing*. Ed. by Julian M. Kunkel and Thomas Ludwig. Lecture Notes in Computer Science. DOI: 10.1007/978-3-319-20119-1_1. Springer International Publishing, July 2015, pp. 1–16.

[90] T. Davis and Y. Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Transactions on Mathematical Software* 38.1 (2011), 1:1–1:25. DOI: `10.1145/2049662.2049663`.

[91] Hartwig Anzt, Edmond Chow, and Jack Dongarra. "Iterative sparse triangular solves for preconditioning". In: *Euro-Par 2015: Parallel Processing*. European Conference on Parallel Processing. DOI: 10.1007/978-3-662-48096-0_50. Springer, Berlin, Heidelberg, Aug. 24, 2015, pp. 650–661. URL: http://link.springer.com/chapter/10.1007/978-3-662-48096-0_50.

[92] Arno CN Van Duin. "Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices". In: *SIAM journal on matrix analysis and applications* 20.4 (1999), pp. 987–1006. URL: http://epubs.siam.org/doi/abs/10.1137/S0895479897317788 (visited on 01/12/2017).

[93] Suchuan Dong and George Em Karniadakis. "Dual-level parallelism for high-order CFD methods". In: *Parallel Computing* 30 (2004), pp. 1–20.

[94] *CUDA C++ Best Practices Guide*. DG-05603-001 v10.2. NVIDIA. 2019.

[95] David R. Kincaid, Thomas C. Oppe, and David M. Young. *ITPACKV 2D User's Guide*. CNA-232. University of Texas at Austin. May 1989. URL: https://web.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/.

[96] Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. *Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi*. 2013. arXiv: 1302.1078 [cs.PF].

[97] J. Hawkes et al. "On the strong scalability of maritime CFD". In: *Journal of Marine Science and Technology* 23 (2018), pp. 81–93. URL: https://doi.org/10.1007/s00773-017-0457-7.

[98] *The OpenACC Application Programming Interface*. 2.6. OpenACC-Standard.org. Nov. 2017.

[99] Satish Balay et al. *PETSc web page*. http://www.mcs.anl.gov/petsc. 2017. URL: http://www.mcs.anl.gov/petsc.

[100] Thomas D. Economon et al. "Performance optimizations for scalable implicit RANS calculations with SU2". In: *Computers & Fluids* 129 (Supplement C Apr. 28, 2016), pp. 146–158. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2016.02.003. URL: http://www.sciencedirect.com/science/article/pii/S0045793016300214 (visited on 09/28/2017).

[101] Zhi Shang. "Performance analysis of large scale parallel CFD computing based on Code_Saturne". In: *Computer Physics Communications* 184.2 (Feb. 1, 2013), pp. 381–386. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2012.09.026. URL: http://www.sciencedirect.com/science/article/pii/S001046551200313X (visited on 03/06/2019).

[102] Yong-Xian Wang et al. "Efficient parallel implementation of large scale 3D structured grid CFD applications on the Tianhe-1A supercomputer". In: *Computers & Fluids*. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011 80 (July 10, 2013), pp. 244–250. ISSN: 0045-7930. DOI: `10.1016/j.compfluid.2012.03.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0045793012000904` (visited on 03/06/2019).

[103] *Intel C++ Compiler Thread Affinity Interface.* `https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-library-support/thread-affinity-interface-linux-and-windows.html`. Accessed: 20 August 2020.

[104] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3.* http://eigen.tuxfamily.org. 2010.

[105] P.H. Cook, M.A. McDonald, and M.C.P. Firmin. *Aerofoil RAE 2822 - pressure distributions, and boundary layer and wake measurements.* Tech. rep. AGARD AR-138. 1979, A6–1–A6–77.

[106] B. van den Berg and J.H.M. Gooden. *Low-speed surface pressure and boundary layer measurement data for the NLR 7301 airfoil section with trailing edge flap.* Tech. rep. AGARD AR-303, Vol 2. 1994, A9.1–A9.12.

[107] Julien Mayeur et al. "RANS simulations on TMR 3D test cases with the Onera elsA flow solver". In: *54th AIAA Aerospace Sciences Meeting.* Vol. 2016-1357. San Diego, USA: American Institute of Aeronautics and Astronautics, 2016. DOI: `10.2514/6.2016-1357`. URL: `https://arc.aiaa.org/doi/abs/10.2514/6.2016-1357` (visited on 10/14/2018).

[108] A. J. Sclafani et al. "CFL3D/OVERFLOW results for DLR-F6 wing/body and drag prediction workshop wing". In: *Journal of Aircraft* 45.3 (May 2008), pp. 762–780. ISSN: 0021-8669. DOI: `10.2514/1.30571`. URL: `https://arc.aiaa.org/doi/10.2514/1.30571` (visited on 10/14/2018).

[109] John Vassberg. "A unified baseline grid about the Common Research Model wing/body for the Fifth AIAA CFD Drag Prediction Workshop". In: *29th AIAA Applied Aerodynamics Conference.* Vol. 2011-3508. Honolulu, Hawaii: American Institute of Aeronautics and Astronautics, 2011. DOI: `10.2514/6.2011-3508`. URL: `https://arc.aiaa.org/doi/abs/10.2514/6.2011-3508` (visited on 10/14/2018).

[110] Ryo Asai. *MCDRAM as high bandwidth memory in Knights Landing processors: developer's guide.* Tech. rep. Colfax International, 2016.

[111] F.D. Witherden, A.M. Farrington, and P.E. Vincent. "PyFR: An open source framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach". In: *Computer Physics Communications* 185.11 (2014), pp. 3028–3040.

[112] Edmond Chow et al. "Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning". In: *Journal of Parallel and Distributed Computing* 119 (2018), pp. 219–230. ISSN: 0743-7315. DOI: `10.1016/j.jpdc.2018.04.017`. URL: `http://www.sciencedirect.com/science/article/pii/S0743731518303034` (visited on 05/16/2019).

[113] Hartwig Anzt et al. "A block-asynchronous relaxation method for graphics processing units". In: *Journal of Parallel and Distributed Computing* 73 (2013), pp. 1613–1626.

[114] J.M. Ortega and W.C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, Inc., 1970.

[115] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1981.

[116] Alan George. "An automatic one-way dissection algorithm for irregular finite element problems". In: *SIAM Journal on Numerical Analysis* 17.6 (1980), pp. 740–751.

[117] Michael J. Wright, Graham V. Candler, and Deepak Bose. "Data-parallel line relaxation method for the Navier-Stokes equations". In: *AIAA Journal* 36.9 (1998), pp. 1603–1609.

[118] R. Ramamurti and R. Löhner. "A parallel implicit incompressible flow solver using unstructured meshes". In: *Computers and Fluids* 25.2 (1996), pp. 119–132.

[119] Dimitri J. Mavriplis. "Multigrid strategies for viscous flow solvers on anisotropic unstructured meshes". In: *Journal of Computational Physics* 145 (1998), pp. 141–165.

[120] Youcef Saad. "A flexible inner-outer preconditioned GMRES algorithm". In: *SIAM Journal on Scientific Computing* 14.2 (1993), pp. 461–469. URL: `http://epubs.siam.org/doi/abs/10.1137/0914028`.

[121] Satish Balay et al. *PETSc Users Manual*. Tech. rep. ANL-95/11 - Revision 3.8. Argonne National Laboratory, 2017. URL: `http://www.mcs.anl.gov/petsc`.

[122] John D. McCalpin. "Memory bandwidth and machine balance in current high performance computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995), pp. 19–25. URL: `http://tab.computer.org/tcca/NEWS/DEC95/dec95_mccalpin.ps`.

[123] Thomas D. Economon et al. "SU2: An open-source suite for multiphysics simulation and design". In: *AIAA Journal* 54.3 (2016), pp. 828–846. URL: `http://arc.aiaa.org/doi/10.2514/1.J053813`.

[124] *SU2 test cases repository.* `https://github.com/su2code/TestCases.git`. Accessed: 2019-09-01.

[125] John C. Strikwerda. "A probabilistic analysis of asynchronous iteration". In: *Linear Algebra and its Applications* 349 (2002), pp. 125–154.