UNIVERSITÀ DEGLI STUDI DI PADOVA

Sede Amministrativa: Università degli Studi di Padova

Dipartimento di Ingegneria Civile, Edile e Ambientale (ICEA)

CORSO DI DOTTORATO DI RICERCA IN:

SCIENZE DELL'INGEGNERIA CIVILE, AMBIENTALE E DELL'ARCHITETTURA

CICLO XXXVI

Chronos: an AMG solver for numerical simulations on HPC platforms

Tesi redatta con il contributo finanziario di M^{3}E srl

Coordinatore: Ch.mo Prof. Massimiliano Ferronato

Supervisore: Prof. Carlo Janna

Dottorando: Giovanni Isotton

UNIVERSITÀ DEGLI STUDI DI PADOVA

Abstract

Department of Civil, Architectural and Environmental Engineering

Doctor of Philosophy

Chronos: an AMG solver for numerical simulations on HPC platforms

Giovanni Isotton

Nowadays, numerical simulation has become a key element in solving problems of engineering and industrial interest. In fact, the need to improve the accuracy of the solution drives the development of more complex numerical models with a high degree of spatial discretization. On large complex problems, the most demanding phase of the simulation is the solution of the system of linear equations derived from the discretization of partial differential equations governing physical process. The solution of these systems requires the use of infrastructures designed for high performance computing (HPC) which, through the development of cloud platforms have become easily accessible for all engineers. This thesis work presents a general-purpose Algebraic MultiGrid (AMG) linear solver designed for distributed memory HPC systems equipped with GPU accelerators. The novelty of this research is the development and implementation of algorithms based on known numerical approaches, capable of exploiting the hardware resources of today's HPC systems. In particular, the work focuses on the redesign of algorithms developed for multi-core CPUs and their porting to GPU boards. This research project is co-funded by M^{3}E [M^{3}E, 2023], and the developments are integrated into the proprietary software Chronos. The effectiveness and performance of the proposed solver have been validated by solving a large set of problems arising from real-world applications on the Marconi100 supercomputer.

UNIVERSITÀ DEGLI STUDI DI PADOVA

Sommario

Dipartimento di Ingegneria Civile, Edile e Ambientale

Dottorato di Ricerca

Chronos: an AMG solver for numerical simulations on HPC platforms

Giovanni Isotton

Oggigiorno la simulazione numerica è diventata fondamentale per risolvere problemi di interesse ingegneristico e industriale. Infatti, la necessità di migliorare l'accuratezza della soluzione stimola lo sviluppo di modelli numerici più complessi e con un'elevata discretizzazione spaziale. Nei problemi complessi di grandi dimensioni, la fase più onerosa della simulazione è la soluzione del sistema di equazioni lineari derivanti dalla discretizzazione delle equazioni differenziali parziali che governano il processo fisico. La soluzione di questi sistemi necessita l'uso di infrastrutture progettate per il calcolo ad alte prestazioni (HPC) che, attraverso lo sviluppo delle piattaforme cloud, sono diventate di facile accesso per tutti gli ingegneri. Questo lavoro di tesi presenta un solutore lineare general-purpose di tipo Algebraic MultiGrid (AMG) progettatto per sistemi HPC a memoria distribuita dotati di acceleratori GPU. L'aspetto originale di questa ricerca è lo sviluppo e l'implementazine di algoritmi, basati su approcci numerici noti, in grado di sfruttare le risorse hardware degli odierni sistemi HPC. In particolare, gran parte del lavoro si concentra sulla riprogettazione degli algoritmi sviluppati per CPU multi-core e il loro porting su schede GPU. Questo progetto di ricerca è cofinanziato da M^{3}E [M^{3}E, 2023] e tutti gli sviluppi sono integrati nel software proprietario Chronos. L'efficacia e le performance del solutore proposto sono state validate risolvendo un ampio set di problemi derivanti da applicazioni reali sul supercalcolatore Marconi100.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Numerical simulation is widely used for solving engineering problems, both because direct measurements are often extremely expensive or even impractical, and because there is an interest in simulating processes that occurred in the past or forecasting future events. In the simulation of physical phenomena of practical industrial interest, it is often necessary to solve equations, or systems of equations, of Partial Differential Equations (PDEs). Analytical solutions can only be used in oversimplified situations, with well-defined domains and boundary conditions that are far from real-world applications. Therefore, numerical methods of discretisation are used such as the Finite Element Method (FEM) [Zienkiewicz et al., 2013] which is based on approximating the solution over the global domain as the integral of local element-level contributions or the Finite Volume Method (FVM) [LeVeque, 2002] which is characterised by the enforcement of the preservation of physical properties at the level of control volumes. These methods generates sparse algebraic system of equations:

$$A\mathbf{x} = \mathbf{b} \tag{1.1}$$

that must be solved, where $A \in \mathbb{R}^{n \times n}$ is the sparse matrix, $\mathbf{b}$ and $\mathbf{x} \in \mathbb{R}^n$ are the right-hand side and solution vector, respectively, and n is the number of equations.

In current industrial applications, with the constant demand for high spatial refinement to improve the accuracy of the final solution, n can grow up to hundreds millions of unknowns. Therfore the solution of the linear system may become the most demanding stage in numerical solutions, both in terms of computation time, up to 90%

of the total time [Koric and Gupta, 2016], and requested storage. Systems with billions of unknowns have also been solved in research experiments. The main difference between industrial problems and research experiments is that the former are characterized by complex geometries, irregular discretizations and heterogeneities in the matrix coefficients, while the latter are generally obtained by successive refinements of regular or quite regular grids. Despite their smaller size, problems arising from real-world applications are very challenging and even having large computational resources may not be enough.

There are several methods to solve the system (1.1), both direct [Koric and Gupta, 2016; Amestoy et al., 2019; Rouet et al., 2020] and iterative [Saad and Van der Vorst, 2000; D'Ambra et al., 2010; Falgout and Schroder, 2014; Badia et al., 2016]. The former are generally preferred in industrial applications because they are typically more robust and require no experience from the user. The main downside is that, especially in 3D problems, the matrix factors require a huge amount of memory that becomes the limiting factor for large scale simulations. Moreover, their parallelization is not trivial because of inherently sequential algorithms needed during the set-up. The latter present by far less memory restrictions and are suitable for highly-efficient parallel implementations. However, iterative methods do not guarantee convergence of the scheme unless a proper preconditioner is adopted. Algebraic MultiGrid (AMG) preconditioning is the most widely used since, if well tuned, it guarantees convergence in a number of iterations that does not depend or only slightly depends on the mesh size (under certain assumptions about the underlying PDE and discretization technique) [Trottenberg et al., 2001; Brezina et al., 2006b; Xu and Zikatanov, 2017; D'Ambra et al., 2021], a property of paramount importance for the extreme-size simulations that are foreseen in the near future. The main drawback of AMG preconditioning is that it is still far from being a black-box method, requiring an experienced user and sometimes a fine tuning of the set-up parameters. For most AMG solvers, a wrong set-up can easily lead to slow convergence or overly expensive preconditioners, and, in the worst case, even convergence failure or divergence [Koric et al., 2014].

The solution of large linear systems requires the use of infrastructures designed for High-Performance Computing (HPC). An HPC cluster is a set of computing nodes connected to each other by a network that allows a massive amount of resources to be exploited in parallel. Each computing node is composed of one or multiple multi-core CPUs and often it is supported by *Graphic Processing Units* (GPU) accelerators. For large-size problems, GPU boards prove extremely suitable for parallel computing, providing a good balance between performance, price, ease of software development and power consumption. In the recent past, the use of these HPC resources was limited and not trivial for a common engineer: few large companies could afford to maintain these infrastructures and proper use required specific knowledge. Now, through the growth of cloud computing, a huge amount of resources can be rented at relatively modest prices, and there are many services, for a fee, that act as an interface between the engineer and the cluster to ensure optimal exploitation of resources.

For years US national labs have provided open source software for the solution of linear systems of equations on distributed-memory systems [Balay et al., 2023; Falgout and Yang, 2002; Trilinos Project Team, 2023]. However, they do not yet exploit the use of GPU accelerators to their full potential. On the other hand, the available GPU-only AMG solvers [NVIDIA, 2023; Bernaschi et al., 2023] are mainly suitable for fluid-dynamic problems, thus not meeting the needs of the industrial world that is interested in mechanical problems.

1.1 Objectives

The main objective of this research project is the development of a general-purpose AMG solver for distributed-memory HPC systems equipped with GPU accelerators. The novelty is not in the definition of new theoretical frameworks, but in the development of algorithms, based on well known methods, that are able to exploit the hardware resources of present and near future HPC systems. In particular, a large part of the work is focused on redesigning the multi-core CPU algorithms and their porting to GPU boards. This work is cofunded by M^{3}E [M^{3}E, 2023] and the outcomes of development activities are integrated into the proprietary software Chronos [the Chronos Project Team, 2023]. Chronos is a library of iterative methods designed for HPC platforms whose development started in early 2019 and many of its features were designed and finalized during this research project. Particular care is spent in the general design of the library in order to make it easily maintainable and amenable to improvements, without sacrificing performance. Chronos has a strong object-oriented framework to enable an easy interface to other software, to be used as an innermost kernel in more complex approaches, such as block preconditioners for multiphysics [Ali Beik and Benzi, 2018; Ferronato et al., 2019; Frigo et al., 2019; Roy et al., 2020; Wathen and Greif, 2020], and to be easily modified to support emerging hardware. The focus is restricted to Symmetric and Positive Definite (SPD) matrices which arise commonly within many fluid simulations and most mechanics applications. Note that the extension to the non-symmetric case can be done by modifying the AMG set-up at a high level, using at a low level the same kernels developed for the SPD case [Sala and Tuminaro, 2008; Lin et al., 2010], adding just few adaptations. The minimum size of the problems presented herein is on the order of $O(10^6)$ unknowns. Moreover, as to the GPU-accelerated algorithms, priority was given to problems arising from the discretization of elasticity equations.

1.2 Contributions

This work introduces an original AMG solver that proves effective on real-world engineering problems and capable of making the most of HPC systems equipped with multi-core CPUs and GPU accelerators. To be effective on a wide range of different problems, Chronos AMG allows for the choice of several options, from the adaptive generation of the operator near-kernel to the smoother selection, from coarsening to prolongation, all of this in the framework of the classical AMG method. In particular, it will be shown that BAMG interpolation, an interpolation whose coefficients are computed through least squares minimization and proposed for the first time in the context of bootstrap AMG [Brandt et al., 2011], makes this AMG extremely effective also on mechanical problems without the need to use an aggregation based coarsening. From the implementation standpoint, Chronos is developed for HPC, adopting a distributed sparse matrix storage scheme where smaller blocks, stored in Compressed Sparse Row format (CSR), are nested into a global CSR structure. This storage format, together with the adoption of non-blocking send/receive messages, allows for a high overlap between communications and computations thus hiding data-transfer latency even for a relatively small amount of local operations. The advantages of this implementation impact all the most important sparse linear algebra operations: matrix-by-vector product, matrix-by-matrix product and transposition, and also in all the information-gathering stages preliminary to smoother and prolongation set-up. Moreover, it will be shown the effectiveness and scalabilty of the Chronos-implemented smoother, the so-called FSAI, that is so far not implemented in any other AMG solvers. Finally, much of the AMG set-up was redesigned to take advantage of GPU accelerators, making Chronos one of the first linear solvers of its kind.

1.3 Outline

The reminder of the thesis is organized as follows. Chapter 2 is an overview of the Chronos package, describing the main classes used to store the objects on a distributed-memory environment. Moreover, the classical AMG framework implemented in Chronos is outlined, focusing on the numerical algorithms adopted to increase effectiveness. Chapter 3 describes the implementation details of the matrix-by-vector product, frequently the most time-consuming kernel in any iterative method, and the main stages of the AMG set-up: smoother, prolongation and coarse operator se-up. The focus will be on the design of algorithms to exploit GPU boards. Chapter 4 collects all the numerical tests that have been performed. In particular, the discussion is subdivided into two sections: the former asseses the performance of the CPU-only version, comparing Chronos AMG against other state-of-the-art AMG solvers while the latter focuses on the GPU-accelerated implementation. Finally, Chapter 5 closes the thesis with some concluding remarks and ideas for future work.

Chapter 2

Chronos Overview

This chapter provides an overview of the Chronos package. First, the design of the library is described from an implementation point of view highlighting the main classes and the startegies used to store the various objects on distributed memory environments. Then, the classical AMG method is briefly outlined focusing on the specific numerical algorithms implemented to increase effectiveness.

2.1 Chronos Library Description

The Chronos package is a collection of classes and functions that implement linear algebra algorithms for distributed memory parallel computers. The library is written in C++, with Message Passing Interface (MPI) directives used for communication among MPI processes and OpenMP and CUDA for multithread execution. The hybrid MPI-OpenMP-CUDA implementation is more flexible in the use of modern computing resources and it is generally more efficient than pure MPI due to its better exploitation of fine-grained parallelism.

Chronos has been developed using the potential of Object-Oriented Programming (OOP). The abstraction introduced by Chronos through OOP allows the use of the same distributed matrix object to represent a linear system, a smoother, an AMG hierarchy or a preconditioner itself. Another advantage of this approach is the possibility to use simpler classes to derive more advanced elements, such as block preconditioners. Moreover, whatever the preconditioner, the same iterative methods can be used for the linear system or eigenproblem solution. In addition, the modular structure allows to easily integrate the CPU kernels with GPU ones leaving the overall structure of the library unchanged.

2.1.1 Main classes

The level of abstraction and the hierarchy of the main classes are sketched in Figure 2.1. All these classes are exposed to the user to access the full range of Chronos functionalities.

The Distributed Dense Matrices (DDMat) and Distributed Sparse Matrices (DSMat) are managed by the DDMat and the DSMat classes, respectively. Both DDMat and DSMat storage schemes require the matrix to be subdivided into n_p horizontal stripes of consecutive rows, where n_p is the number of active MPI processes. In the DDMat, each stripe is stored row-wise on a different MPI rank to guarantee better access in memory during multiplication operations. This makes the DDMat very efficient for linear systems with multiple right-hand-sides and eigenproblems, and distributed vectors are stored as one-column DDMat. In the DSMat each stripe is subdivided into an array of CSR matrices. The CSR format is the Chronos standard format for shared sparse matrices and the CSRMAT class is responsible for their management. The DSMat storage scheme adopted in Chronos is very effective in both the preconditioner set-up and the Sparse Matrix-by-Vector (SpMV) product because it allows much of the communication and computation to occur simultanously, as described in the next Chapter.

The Preconditioner class manages the approximation of the inverse of a Distributed Sparse Matrix. It requires in input a Distributed Sparse Matrix as DSMat-type object and an optional test space as a DDMat-type object. The classes derived from Preconditioner are Jac, aFSAI and aAMG, for Jacobi, aFSAI, and AMG, respectively. A useful feature is that each of these classes can be used as a smoother in the AMG.

Both the DSMat and Preconditioner classes are derived from the MatrixProd class which manages the Sparse Matrix-Vector (SpMV) product at the highest level of abstraction. The SpMV is frequently the most expensive operation in any preconditioned iterative solver and its management has defined the design of the whole library. With reference to Figure 2.1, the MatrixProd class leads the Chronos structure together with the iterative solvers. Furthermore, more general MatrixProd elements can be readily built using the MatrixProdList class, that manages an implicit MatrixProd object defined as the product of a sequence of MatrixProd objects ordered into a list.

At the top of the hierarchy, there are also the iterative solvers for linear systems and eigenproblems, LinSolver and EigSolver, respectively. Currently, the classes derived from LinSolver are PCG and BiCGstab, for the Preconditioned Conjugate Gradient (PCG) and the Preconditioned Biconjugate Gradient Stabilized (BiCGstab), respectively.

Finally, the Power Method and the Simultaneous Rayleigh Quotient Minimization (SRQCG) are implemented in the two classes PowMeth and SRQCG derived from EigSolver.

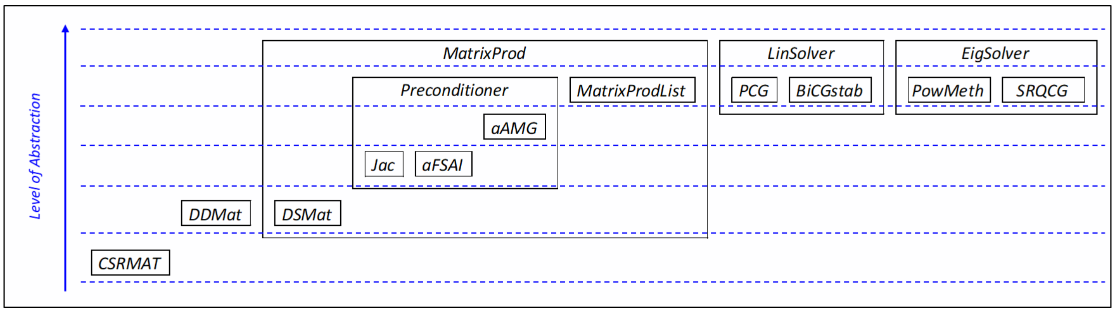


FIGURE 2.1: Chronos main classes and hierarchies.

2.1.2 Distributed Sparse Matrix (DSMat) Storage Scheme

The DSMat storage scheme implemented in Chronos consists of a partitioning of the matrix into n_p horizontal stripes of consecutive rows. Each stripe is then divided into blocks by applying the same subdivision to the columns, as schematically shown in Figure 2.2, and each block is stored as a CSR matrix.

The CSR matrices have a local numbering, i.e., rows and columns of block IJ are numbered from 0 to n_{I-1} and from 0 to n_{J-1}, where n_I and n_J are the number of rows assigned to MPI processes I and J, respectively. This expedient allows the use of 4-bytes integers, saving memory and increasing efficiency.

Each process stores the diagonal block and the list of *left* (with a lower index) and *right* (with a higher index) blocks corresponding to the connections with neighboring processes. With reference to Figure 2.2, rank 3 stores the 5 blocks highlighted in red:

0, 1 and 2 as left neighbors, the diagonal block with only internal connections and 6 as right neighbor.

This blocked scheme, although a bit cumbersome to implement, allows to stress non-blocking send/receive communications with a large overlap between data-tranfer and computation. It has proven very effective in all basic operations involving a DSMat: SpMV product, Matrix-by-Matrix product and matrix transposition.

If GPU accelerators are used, the DSMat object is stored on both the *Host* (CPU RAM) and *Device* (GPU global memory). This allows to simplify the information exchange between the MPI processes and reduces copies *DeviceToHost-HostToDevice* in many stages of the AMG set-up, as it will be shown in the next Chapter. Note that having a copy of the DSMat on the *Host* does not generate any storage issue since the current bottleneck in terms of storage is related to the global memory of the *Device*.

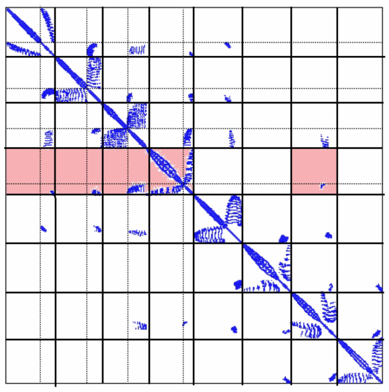


FIGURE 2.2: Schematic representation of the standard DSMat matrix storage scheme implemented in Chronos using 8 MPI processes. The blue dots are the non-zeroes entries of the DSMat. The red colored blocks are assigned to rank 3.

Other classes can be derived from the DSMat class in order to handle distributed matrices with special features. In the applications investigated in this work, matrices with a block-diagonal structure are also handled. On such matrices, each MPI rank stores a finite number of consecutive blocks and no block is split between different ranks. This subdivision then guides the partitioning of the standard DSMat as shown in Figure 2.3. Moreover, since in applications arising from numerical discretization the number of blocks is very large and their size small, the exact Cholesky factor can be computed by factorizing in parallel all its diagonal blocks: several OpenMP threads are used to factor a chunk of blocks. The sequential routine `cholmod_factorize`, provided by the SuiteSparse library [Chen et al., 2008], is used to factorize the single CSR blocks. Similary, the forward and backward substitutions are performed block-by-block using `cholmod_solve2` from SuiteSparse. Note that these operations do not require any communication between the MPI processes, and on each rank they are executed by multiple OpenMP threads.

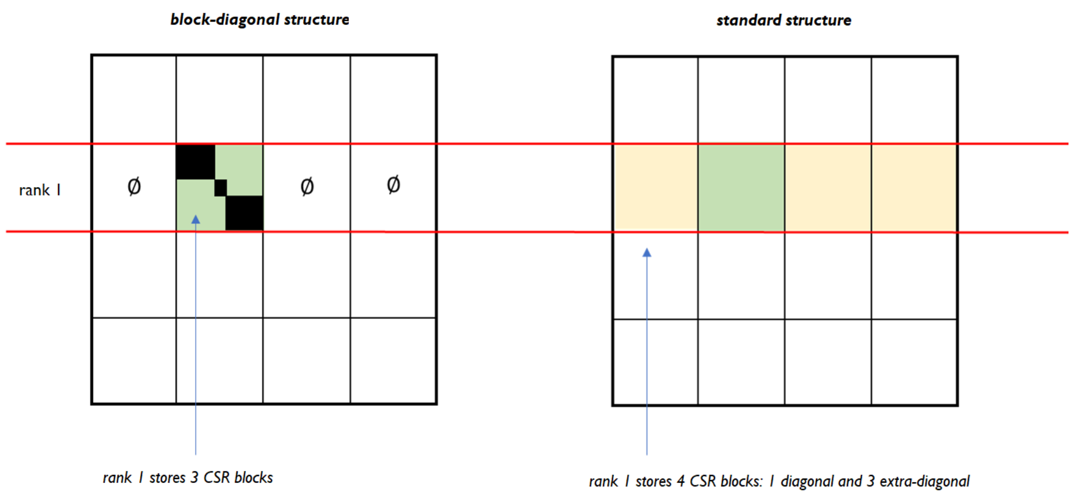

block-diagonal structure
standard structure
rank I
Ø
Ø
Ø
rank I stores 3 CSR blocks
rank I stores 4 CSR blocks: I diagonal and 3 extra-diagonal

FIGURE 2.3: Schematic representation of the DSMat storage scheme with block-diagonal structure using 4 MPI ranks. The portions of matrices stored by MPI rank 1 are highlighted with different colors.

2.2 Chronos AMG Framework

Any AMG method is generally built on three main components whose interplay gives the effectiveness of the overall method:

- Smoothing, where an inner preconditioner is applied to damp the high-frequency error components;
- Coarsening, in which coarse level variables are chosen for the construction of the next level;
- Interpolation, defining the transfer operator between coarse and fine variables.

In Chronos a fourth component, borrowed from the context of bootstrap [Brandt et al., 2011] and adaptive AMG [Brezina et al., 2005, 2006a], is added to the above three and consists in a method to improve the near kernel of the linear operator or unveil hidden components whenever they are not a priori available. Therefore, one of the strengths of this library is that it offers several options for each AMG component to allow the user to choose the best combination for any specific problem.

The present work is focused on the classical AMG setting, and below the basic concepts behind this method are briefly recalled , referring the interested reader to more detailed and rigorous descriptions in the works by Stüben [2001]; Trottenberg et al. [2001]; Xu and Zikatanov [2017]. For the sake of clearness, this section is restricted to a two levels only scheme, as the multilevel version can be readily obtained by recursion.

The first component that has to be set-up in AMG is the smoother, a stationary iterative method responsible for eliminating the error components associated with large eigenvalues of A, referred also as the high-frequency errors. The smoother is generally defined from a rough approximation of $A^{-1} \simeq M^{-1}$ and the associated error propagation operator is given by the following equation:

$$S = I - \omega M^{-1} A, \tag{2.1}$$

where I is the identity matrix and ω a relaxation factor to ensure:

$$\omega \rho(M^{-1}A) < 2 \tag{2.2}$$

see for instance [Franceschini et al., 2019] for a short explanation. Generally, the smoother is given by a simple pointwise relaxation method such as (block) Jacobi or Gauss-Seidel, with the second one often preferred even though its use on parallel computers is not straightforward, or by a Chebyshev polynomial [Adams et al., 2003]. Unlike other AMG packages such as BoomerAMG [Henson and Yang, 2002] or GAMG [Balay et al., 2023] where traditional smoothers like Gauss-Seidel or Chebyshev are selected by default, Chronos implements the adaptive Factorized Sparse Approximate Inverse (aFSAI) with the matrix M^{-1} taking the explicit form:

$$M^{-1} = G^T G \tag{2.3}$$

with G lower triangular. This choice is dictated by its almost perfect strong scalability in computation and application and by its proven robustness in real engineering problems [Janna and Ferronato, 2011; Janna et al., 2015a,b; Baggio et al., 2017]. Moreover, the cost of aFSAI application is usually much lower than that of Gauss-Seidel and Chebyshev since, generally, the number of non-zeroes of M^{-1} is only 20-40% the number of non-zeroes of A.

The second component of AMG is the so-called Coarse-Grid Correction (CGC), which is the A-orthogonal projection operation that should take care of the low-frequency components of the error. To build CGC in classical AMG, the unknowns of a given level are partitioned into Fine and Coarse (F/C), with coarse variables becoming the unknowns of the next level. The choice of coarse variables is crucial in AMG, as it determines both the rate at which the problem size is reduced and strongly affects the convergence of the method. Here, let's introduce the concept of Strenght of Connection (SoC), i.e., to each edge of the adjacency graph of A a measure of its relative importance is associated. Then, using SoC, the graph connections are ranked and those deemed less important are filtered out. A Maximum Independent Set (MIS) is finally constructed on the filtered SoC graph to determine coarse variables using the well-known and efficient PMIS algorithm [De Sterck et al., 2006].

To facilitate explanation, the system matrix is reordered according to F/C partitioning of unknowns with first fine variables and second coarse ones:

$$A = \begin{bmatrix} A_{ff} & A_{fc} \\ A_{fc}^T & A_{cc} \end{bmatrix} \tag{2.4}$$

with A_{ff} and A_{cc} square $n_f \times n_f$ and $n_c \times n_c$ matrices, respectively. In classical AMG, the interpolation operator P is written as:

$$P = \begin{bmatrix} W \\ I \end{bmatrix}, \tag{2.5}$$

where W is a $n_f \times n_c$ matrix containing the weights for coarse-to-fine variable interpolation. As the system matrix is SPD, the restriction operator R is defined through a Galerkin approach as the transpose of P, and the coarse level matrix A_c is simply given by the triple matrix product:

Algorithm 1 AMG Set-up

```
1: procedure AMG_SetUp($A_k$)
2:     Define $\Omega_k$ as the set of the $n_k$ vertices of the adjacency graph of $A_k$;
3:     if $n_k$ is small enough to allow for a direct factorization then
4:         Compute $A_k = L_k L_k^T$;
5:     else
6:         Compute $M_k$ such that $M_k^{-1} \simeq A_k^{-1}$;
7:         Define the smoother as $S_k = \left(I_k - \omega_k M_k^{-1} A_k\right)$;
8:         Partition $\Omega_k$ into the disjoint sets $\mathcal{C}_k$ and $\mathcal{F}_k$ via coarsening;
9:         Compute the prolongation matrix $P_k$ from $\mathcal{C}_k$ to $\Omega_k$;
10:        Compute the new coarse level matrix $A_{k+1} = P_k^T A_k P_k$;
11:        Call AMG_SetUp($A_{k+1}$);
12:    end if
13: end procedure
```

$$A_c = P^T A P \tag{2.6}$$

In practice, fast convergence and rapid coarsening, i.e., high F/C ratios, are always desired, and the construction of effective interpolations is of paramount importance to conciliate these conflicting requirements.

Having defined all the above components, the set-up phase of the two-level multigrid method is completed and the iteration matrix is given by:

$$(S)^{\nu_2} \left(I - P {A_c}^{-1} P^T A\right) (S)^{\nu_1} \tag{2.7}$$

with ν_1 and ν_2 representing the number of smoothing steps performed before and after the coarse-grid correction, respectively.

Algorithms 1 and 2 briefly report the general AMG set-up phase and application in a V-cycle, respectively, in a multilevel framework, where it is conventionally assumed that $A_0 = A$, $\mathbf{y}_0 = \mathbf{y}$ and $\mathbf{z}_0 = \mathbf{z}$. Details on all the computational kernels sketched in Algorithm 1 and their parallel implementation will be discussed in the next sections.

In the AMG context, two quantities are used to define the sparsity of the multilevel AMG hierarchy: grid complexity and operator complexity. The former is the total number of nodes on all grids divided by the number of nodes on the fine grid while the latter is the total number of nonzeroes in the linear operators on all grids divided by the number of nonzeroes in the fine grid operator.

Algorithm 2 AMG application in a V-cycle

```
1: procedure AMG_APPLY($A_k$, $\mathbf{y}_k$, $\mathbf{z}_k$)
2:     if $k$ is the last level then
3:         Solve $A_k\mathbf{z}_k = \mathbf{y}_k$ using $L_k$, the exact Cholesky factor of $A_k$;
4:     else
5:         Compute $\mathbf{s}_k$ by applying $\nu_1$ smoothing steps to $A_k\mathbf{s}_k = \mathbf{y}_k$ with $\mathbf{s}_0 = \mathbf{0}$;
6:         Compute the residual $\mathbf{r}_k = \mathbf{y}_k - A_k\mathbf{s}_k$;
7:         Restrict the residual to the coarse grid $\mathbf{r}_{k+1} = P_k^T\mathbf{r}_k$;
8:         Call AMG_Apply($A_{k+1}, \mathbf{r}_{k+1}, \mathbf{d}_{k+1}$);
9:         Prolongate the correction to the fine grid $\mathbf{d}_k = P_k\mathbf{d}_{k+1}$;
10:        Update $\mathbf{s}_k \leftarrow \mathbf{s}_k + \mathbf{d}_k$;
11:        Compute $\mathbf{z}_k$ by applying $\nu_2$ smoothing steps to $A_k\mathbf{z}_k = \mathbf{y}_k$ with $\mathbf{z}_0 = \mathbf{s}_k$;
12:    end if
13: end procedure
```

2.2.1 Unveiling of the near kernel operator

The near kernel plays an important role in most AMG methods. Most smoothers fail in damping error components that lie in the space of the near kernel and therfore AMG methods consider the near kernel when constructing interpolation. The kernel (or null space) associated with the homogeneous discretized operator arising from the most common PDE or systems of PDE is generally a priori known. For instance it is well-known that the constant vector is the kernel for the Laplace operator and rigid body modes constitute the kernel for linear elasticity problems. The information needed to build these spaces, usually referred to as test spaces in the adaptive AMG terminology, is readily available to the user from nodal coordinates or other data retrievable from the discretization.

However, the homogeneous operator kernel is only an approximation of the true near kernel associated with the fully assembled matrix and does not take into account all the peculiarities of the problem such as boundary conditions or the strong heterogeneities in the material properties that often arise in real-world problems. In many circumstances, a better test space can be obtained by simply modifying the initial near kernel suggested by the PDE. In the adaptive AMG literature [Brandt et al., 2011; Brezina et al., 2005, 2006a; Lee, 2020], the test space is found by simply running a few smoothing steps over a random test space or the initial near kernel, whenever available. From a theoretical standpoint, the test space should be computed by solving the generalized eigenproblem [Brannick et al., 2018]:

$$A\varphi = \lambda M\varphi \tag{2.8}$$

A way to extract an effective test space could be by relying on an iterative eigensolver [Frommer et al., 2021]. In the present implementation, we opt for the Simultaneous Rayleigh Quotient minimization (SRQCG) [Franceschini et al., 2019] whose cost per iteration is only slightly higher than a smoothing step. By contrast, SRQCG can provide a much better approximation of the smallest eigenpairs especially if a good preconditioner is provided. Since an approximation of A^{-1} is already available through the smoother, we simply reuse the previously computed M^{-1} inside the SRQCG iteration.

Unfortunately, extracting the eigenpairs of (2.8) with high accuracy is generally more expensive than solving the original system (1.1). Moreover, the SRQCG solution to (2.8) needs the multiplication of M by a vector which, due to our choice of M (2.3), would result in a forward and backward triangular solve whose parallelization may represent an algorithmic bottleneck. For this reason, to limit the set-up cost, we only approximately solve the eigenproblem:

$$A\varphi = \lambda\varphi \tag{2.9}$$

with a predetermined and small number of SRQCG iterations. This simple strategy usually gives satisfactory results, whenever an initial test space is not available or boundary conditions and heterogeneity exert a strong influence, such as in geomechanical problems. Another appealing idea, though not explored in this work, is bootstrapping [Brezina et al., 2005, 2006a; Brandt et al., 2011; D'Ambra et al., 2018], which consists in computing a relatively cheap AMG preconditioner from a tentative test space, and then using AMG itself to better uncover the near null space and rebuild a more effective AMG.

Operatively, once the test space is found, an orthonormal basis of it is computed and the basis vectors are collected into a (skinny) matrix V that may be subsequently used for the calculation of strength of connection and the prolongation.

2.2.2 Strength of Connection (SoC)

The construction of the coarse problem in Chronos is based on the definition of a SoC matrix, that is used to filter-out weak connections from the adjacency graph of A. There are three different SoC definitions available in the library:

1. Classical strength of connection:

$$s_{ij} = \frac{-a_{ij}}{\max(\min_{j \neq i} a_{ij}, \min_{j \neq i} a_{ji})} \tag{2.10}$$

2. Strength of connection based on strong couplings:

$$s_{ij} = \frac{|a_{ij}|}{\sqrt{a_{ii}a_{jj}}} \tag{2.11}$$

3. Affinity-based strength of connection:

$$s_{ij} = \frac{(\sum_k v_{ik}v_{jk})^2}{(\sum_k v_{ik}^2)(\sum_k v_{jk}^2)} \tag{2.12}$$

where s_{ij} denotes the SoC between node i and j and a_{ij} and v_{ij} denote the entries in row i and column j of the matrices A and V, respectively. SoC (2.10) is particularly effective for Poisson-like problems where the system matrix is close to an M-matrix. SoC (2.11) is generally used in smoothed aggregation AMG [Vaněk et al., 1996] and usually gives good results in structural problems. Finally, SoC (2.12) has been introduced by Livne and Brandt [2012] and, though requiring a rather expensive computation, it is able to accurately capture anisotropies as is shown by Paludetto Magri et al. [2019].

After SoC is computed for every pair of nodes, weak connections, those whose SoC value is below a user-defined threshold, are eliminated. The MIS algorithm is applied to the resulting graph that retains only the strong connections and these nodes define the unknowns on the next level. The more aggressively the connections are filtered, the higher number of nodes are left in the next level. There are two ways of controlling SoC filtering in Chronos:

1. by a threshold, the traditional way of filtering, where connections below a given θ are simply dropped;

2. prescribing an average number of connections per node.

On one side, guaranteeing an average number of connections per node is trickier, since it requires a preliminary sorting of all the SoC, but it ensures a more regular coarsening through levels with an almost constant coarsening ratio. Moreover, in affinity-based SoC, the strength values usually lie in a narrow interval close to unity so that a proper choice of the drop threshold is almost impossible.

Finally, MIS construction is performed by using the PMIS strategy which is perfectly parallel and generally gives rise to lower complexities than the Ruge-Stüben coarsening [De Sterck et al., 2008]. More aggressive coarsening methods require special care in the interpolation construction, as we will see in the next section.

2.2.3 Interpolation operators

We recall that the prolongation operator P should satisfy:

$$\mathcal{V} \subseteq \text{range}(P) \tag{2.13}$$

where $\mathcal{V}$ is the near-kernel of A or, more precisely, for a coarse space of given size n_c the optimal two-level prolongation as stated in [Brannick et al., 2018; Xu and Zikatanov, 2017] should be such that:

$$\text{span}(\mathbf{v}_i) = \text{range}(P) \tag{2.14}$$

where $\mathbf{v}_i$ are approximations of the eigenvectors associated with the smallest n_c eigenvalues of the generalized eigenproblem (2.8). To this aim, depending on the problem, we use two different strategies.

If a test space is available or it is relatively cheap to obtain a reasonable approximation of the near-kernel, then the so-called BAMG interpolation is used. The name BAMG is used because this interpolation based on least squares has been proposed for the first time in the context of bootstrap AMG [Brandt et al., 2011]. In this approach, the weights of prolongation w_{ij}, i.e., the entries of the W block in (2.5), are found through least square minimizations:

$$w_{ij} = \underset{j \in C_i}{\text{argmin}} \, \|\mathbf{v}_i - \sum_{j \in C_i} w_{ij} \mathbf{v}_j\|^2 \qquad i = 1, \ldots, n \tag{2.15}$$

with $\mathbf{v}_k$ the k-th row of V and C_i the interpolatory set for i. From experience, it is imperative for an effective interpolation that the norm $\|\mathbf{v}_i - \sum_{j \in C_i} w_{ij} \mathbf{v}_j\|$ must be reduced to zero and, in the general case, this can be accomplished only if the cardinality of C_i, $|C_i|$, is equal or larger than n_t, the number of test vectors. To guarantee an exact interpolation, it is often necessary to use neighbors at a distance larger than one, especially when dealing with systems of PDEs, with a consequent increase of the overall operator complexity. Moreover, it may happen that, even if $|C_i| \geq n_t$, some of the vectors $\mathbf{v}_k$ are almost parallel and may produce large jumps in the weights. In turn, large jumps in P introduces high frequencies in the next level operator that the smoother hardly handles. To overcome these difficulties, BAMG interpolation is computed with an adaptive procedure similar to those described in [Franceschini et al., 2019; Paludetto Magri et al., 2019]. More in detail, let's define the dense matrix Φ whose entries φ_{ij} correspond to the j-th component of the i-th test vector $\mathbf{v}_i$ for any j in the interpolatory set. For each fine node $i \in \mathcal{F}$ to be interpolated, the adaptive procedure starts by including in the interpolatory set all its coarse neighbors at a distance no larger than l_{min}, and select a proper basis for Φ by using a *maxvol* algorithm [Goreinov et al., 2010; Knuth, 1985]. If either the relative residual:

$$r_i = \frac{\|\varphi_i - \Phi_i \mathbf{w}_i\|}{\|\varphi_i\|} \tag{2.16}$$

or the norm of the weights, $\|\mathbf{w}_i\|$, are larger than the user-defined thresholds ϵ and μ, respectively, then the interpolation distance is increased by one. The interpolation distance increases up to l_{max} to limit the computational cost. This procedure, briefly sketched in Algorithm 3, though slightly expensive, allows to compute an accurate and smooth prolongation without impacting too much operator complexity. In fact, including in C_i all the coarse nodes within a large a priori selected interpolation distance usually leads to higher operator complexities because several fine nodes may be interpolated with excessively large support. Moreover, limiting the number of non-zeroes in the rows of P has the additional advantage that it is possible to perform prolongation

Algorithm 3 **BAMG prolongation adaptive set-up**

```
1: procedure BAMG_PROLONGATION($S$, $V$, $l_{min}$, $l_{max}$, $\epsilon$, $\mu$)
2:     for all $i \in \mathcal{F}$ do;
3:         Set $l = l_{\text{min}}$;
4:         Set $\mathbf{r}_i = \varphi_i$;
5:         while $l \leq l_{\text{max}}$ and ( $r_i > \epsilon$ or $\mathbf{w}_i > \mu$ ) do
6:             Include in $C_i$ all the coarse nodes at a distance at most $l$;
7:             Collect all the $\varphi_k$ such that $k \in C_i$;
8:             Select from $\varphi_k$ a maxvol basis $\Phi_i$;
9:             Find the vector of weights $\mathbf{w}_i$ by minimizing $\|\Phi_i \mathbf{w}_i - \varphi_i\|$;
10:            $l = l + 1$;
11:        end while
12:    end for
13: end procedure
```

smoothing without an exponential growth of the operator complexity. Prolongation smoothing is a very common practice in solving elasticity problems with aggregation-based AMG [Gee et al., 2009] and numerical results will show that it can be beneficial also in the context of classical AMG.

On the other hand, when there is no explicit knowledge of the test space or when the matrix at hand arises from the discretization of a Poisson-like problem, Chronos can also rely on more classical interpolation schemes. Below the expressions of some well-known interpolation formulas are briefly recalled. First, using the concept of SoC, let's define the following sets:

- $N_i = \{j \mid a_{ij} \neq 0\}$, the set of direct neighbours of i;
- $S_i = \{j \in N_i \mid j \text{ strongly influences } i\}$, the set of strongly connected neighbours of i;
- $F_i^S = F \cap S_i$, the set of strongly connected fine neighbors of i;
- $C_i^S = C \cap S_i$, the set of strongly connected coarse neighbors of i;
- $N_i^W = N_i \setminus (F_i^S \cup C_i^S)$, the set of weakly connected neighbors of i.

A generally accurate distance-one interpolation formula, introduced by Ruge and Stüben [1987], is the so-called classical interpolation. Unlike other distance-one formulas, here, the interpolation takes care of the contribution from strongly influencing points F_i^S, and the expression for the interpolation weight is given by:

$$w_{ij} = -\frac{1}{a_{ii} + \sum_{k \in N_i^w \cup F_i^{S*}} a_{ik}} \left(\sum_{k \in F_i^s \setminus F_i^{S*}} \frac{a_{ik}\bar{a}_{kj}}{\sum_{m \in C_i^s} \bar{a}_{km}} \right), \qquad j \in C_i^S, \tag{2.17}$$

where:

$$\bar{a}_{ij} = \begin{cases} 0 & \text{if } sign(a_{ij}) = sign(a_{ii}) \\ a_{ij} & \text{otherwise} \end{cases} \tag{2.18}$$

It is worth noting that the original formula proposed in [Ruge and Stüben, 1987] is here corrected accordingly with the modification introduced in [Henson and Yang, 2002] where the set of strongly connected neighbors F_i^{S*}, that are F-points but do not have a common C-point, are subtracted to the fine strong neighbors F_i^S. This modification of the interpolation formula is needed to avoid that the term $\sum_{m \in C_i^s} \bar{a}_{km}$ vanishes. Indeed, using the PMIS-coarsening method no longer guarantees that two strongly connected F-points are interpolated by a common C-point. However, even if for a large class of problems the classical interpolation is very effective, it can lose efficiency for challenging problems such as rotated anisotropies or problems with large discontinuities. Hence, some more advanced interpolation formulas are required to overcome these difficulties. A widely used interpolation strategy, mainly for very challenging Poisson-like problems, is the Extended+i interpolation. This interpolation formula extends the interpolatory set including C-points that are at distance two apart from the F-point considered. Furthermore, also coarse nodes connected to the fine neighbors of the fine point interpolated are considered. Hence, denoting with $\hat{C}_i = C_i \cup \bigcup_{j \in F_i^S} C_j$ the set of distance-two coarse nodes, the interpolation Extended+i formula takes the following form:

$$w_{ij} = -\frac{1}{\tilde{a}_{ii}} \left(a_{ij} + \sum_{k \in F_i^s} \frac{a_{ik}\bar{a}_{kj}}{\sum_{l \in \hat{C}_{i \cup \{i\}}} \bar{a}_{kl}} \right), \qquad j \in \hat{C}_i \tag{2.19}$$

with

$$\tilde{a}_{ii} = a_{ii} + \sum_{n \in N_i^w \setminus \hat{C}_i} a_{in} + \sum_{k \in F_i^s} a_{ik} \frac{\bar{a}_{ki}}{\sum_{l \in \hat{C}_i \cup i} \bar{a}_{kl}}. \tag{2.20}$$

This Extended+i interpolation remedies many problems occurring with distance-one

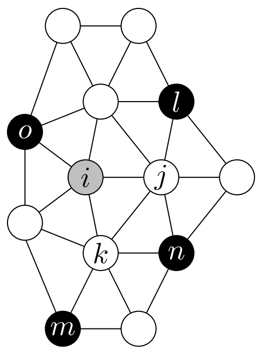


FIGURE 2.4: Example of the interpolatory points. The gray point is the point to be interpolated, black points are C-points and white points are F-points.

interpolation and provides better weights than other distance-two interpolation formulas at the cost of larger operator complexities. A possible way to reduce the complexities without or mildly affecting the convergence rate is to consider a different interpolatory set, i.e., an interpolatory set larger than the distance-one set C_i^S, but smaller than the distance-two $\hat{C}_i$. The idea is to consider an interpolatory set that only extends C_i^S for strong F-F connections without a common C-point, since in the other cases the point i is already surrounded by interpolatory points belonging to C_i^S. Here, we propose to enrich the set C_i^S including the minimum number of distance-two coarse nodes guaranteeing that each F-F strong connection is covered by at least a C-point. Let us consider the example in Figure 2.4.

Notice that using the classical interpolation, the interpolatory set would be $C_i^S = \{o\}$ and there would be two fine neighbors of i, j and k, that do not share a C-point with i. On the other hand, using the Extended+i interpolation, $\hat{C}_i = \{m, n, l, o\}$ and each F-node, strongly connected with i, would share at least one C-node of the interpolatory set $\hat{C}_i$. However, to guarantee this last condition, it would be sufficient to only include the node n to the set C_i^S, so that the extended interpolatory set would become $\hat{C}_i^h = \{o, n\}$. In other words, the set C_i^S is extended by including the maximum independent set of distance-two C-nodes to accomplish the above condition. In the following this interpolation is referred as Hybrid, and the procedure to construct the interpolatory set $\hat{C}_i^h$ is reported on Algorithm 4.

Algorithm 4 **Computation of extended interpolation set $\hat{C}_i^h$**

```
1: Set the initial interpolatory set $\hat{C}_i^h = C_i^S$
2: Compute the initial set $F^{'}$, such that:
3:     $F^{'} = \{j \in F_i^S \mid j$ is not strongly connected with at least one node in $C_i^S\}$
4: Compute the initial set of $C^{''}$, i.e., the set of the distance-two coarse nodes strongly
   connected with a $F^{'}$-point
5: Compute the vertex degree of each element in $C^{''}$ by taking into account only the
   connections with $F^{'}$
6: while $F^{'} \neq \emptyset$ do
7:     Choose the node with maximum degree in $C^{''}$ and add it to $\hat{C}_i^h$
8:     Update the set $F^{'}$
9:     Update the set $C^{''}$
10: end while
```

2.2.4 Filtering

One problem that may affect AMG methods, especially in parallel implementation, is the excessive stencil growth occurring in lower levels. This drawback is more pronounced if long-range interpolation or prolongation smoothing are used. Some authors have explored interesting solutions to reduce AMG complexity without detrimental effects on convergence [Bienz et al., 2016; Falgout and Schroder, 2014]. Simply eliminating small entries from the operators, as is done for instance with Incomplete LU factorizations (ILU) or some approximate inverse preconditioners, may completely harm the effectiveness of CGC. This happens because removal of small entries from P_k or $A_{k+1} = P_k^T A_k P_k$ may induce an inaccurate representation of the near kernel of A.

As a remedy, the authors in [Falgout and Schroder, 2014] propose to compensate the action of eliminated entries through a sort of stencil collapsing to guarantee that the filtered operator, say $\tilde{A}_{k+1}$, exerts the same action of A on the near kernel:

$$\tilde{A}_{k+1} V = A_{k+1} V \tag{2.21}$$

with V a matrix representation of the near kernel. While it is relatively simple to enforce condition (2.21) for one dimensional near kernels, it is not immediate to accommodate the action on several vectors at the same time. With multiple vectors, first the smallest entries of A_{k+1} are dropped to determine the pattern of $\tilde{A}_{k+1}$, then a correction to $\tilde{A}_{k+1}$,

Δ_{k+1}, is computed by using least squares on:

$$\|(A_{k+1} - \tilde{A}_{k+1})V - \Delta_{k+1}V\|_2 \tag{2.22}$$

In more detail, $\tilde{A}_{k+1}$ is computed row-wise such that the absolute norm of each row is a given percentage ρ of the norm of the original one, to avoid changing too much the original matrix, and then the correction is computed for the same row. The same procedure is used on the prolongation operator P_k with the only exception that instead of V, its injection in the coarse space is used. Operatively, the test space V is used when available while in cases where V is not computed, such as in Poisson problems, we simply replace V with a constant vector.

Finally, note that $\tilde{A}_{k+1}$ is no longer guaranteed to be SPD and, especially when an aggressive dropping is enforced, the use of a non-symmetric Krylov solver, such as GMRES or BiCGstab, is often needed instead of CG. Obviously, such care is not needed when only the prolongation is filtered as $\tilde{P}_k^T A \tilde{P}_k$ is always SPD for any choice of ρ.

Chapter 3

Multi GPU implementation

This chapter describes the implementation for distributed memory GPU-accelerated systems of the SpMV product and some critical stages of the AMG set-up. Regarding the AMG computation, this work focused on the most expensive stages in terms of computing time and memory demand:

1. Smoother computation: the most demanding part in the smoother set-up is the computation of an approximation of A^{-1}, that is the aFSAI (2.3).

2. Prolongator computation: in this work only the BAMG prolongator (2.15) has been investigated for GPU acceleration as the mechanical problems are of most interest to the author. Note that in Chronos, the Extended+i interpolation is also accelerated following the Matrix-by-Matrix approach proposed by Li et al. [2021].

3. Matrix-by-Matrix (MxM) product: the MxM product plays a major role in the computation of the coarse matrix (2.6).

On distributed memory systems, the design of an algorithm involves both communications and computational kernels. The former, handled through MPI directives, usually represents a preliminary stage which is necessary for the computation that follows, as it occurs in the aFSAI, BAMG set-up and MxM product. On the other hand, in SpMV product communications and computations can be successfully overlapped. The communication kernels are the same in both the CPU-only and GPU accelerated versions and the information is exchanged between the MPI processes always passing through the *Host*. If GPU accelerators are used, the copies between *Device* and *Host* are handled through CUDA directives. Note that, as decribed in the Chapter 2, the DSMat object is always replicated on the *Host* when accelerators are used and this allows to avoid frequent *HostToDevice* copies throughout the AMG set-up. Direct communication between GPU boards was not investigated in this work because this feature was not yet available when the Chronos library was originally designed.

Regarding the accelerator, the GPU board used during the development of the present work is NVIDIA Tesla V100 (based on the Volta CUDA architecture) equipped with 80 Stream Multiprocessors for a total of 5120 CUDA cores and 16 GByte of GDDR5 memory.

3.1 Distributed memory Sparse Matrix-Vector product (SpMV)

The SpMV product $A\boldsymbol{y} = \boldsymbol{z}$ between a DSMat A and a DDMat $\boldsymbol{y}$ is the most expensive operation in any preconditioned iterative solver. On a distributed memory computer, it consists of a $communication$ stage, mainly handled by the CPUs, and a $computation$ stage, that can be performed either by the CPU or the GPU accelerator (or, in some implementations by both).

With reference to Fig. 3.1, the MPI rank 3 gathers the $\boldsymbol{y}$ terms highlighted in green and computes its stripe of $\boldsymbol{z}$, highlighted in violet, as the sum of products between the extra-diagonal CSR blocks, in red, and the entries received from neighbouring processes, plus the contribution of the product between diagonal CSR blocks, in orange, and the stored $\boldsymbol{y}$ terms, in light blue. As detailed below, communication and computation can be conveniently overlapped thanks to non-blocking communications and the DSMat storage scheme:

1. First, local elements of $\boldsymbol{y}$ to be sent are moved into a device buffer and copied on the *Host*;
2. Then the *Host* starts non-blocking send/receive communications to/from neighboring processes;
3. in the meantime, the local component of $\boldsymbol{z}$ is initialized with the product between the diagonal CSR block and the local part of $\boldsymbol{y}$;
4. each process sequentially tests all the incoming communications and as soon as some components of $\boldsymbol{y}$ are received, the corresponding buffer is copied onto the

Device and the local part of z is updated with the product by the off-diagonal CSR block;

It is easy to observe that the third step in the above list can be overlapped with the first two, and also the product by the off-diagonal blocks partly hides the communication latency for the transfer of non-local y components. In the fourth step, the off-diagonal blocks are processed in sequence because each block is usually large enough to saturate all the computational resources of a modern GPU.

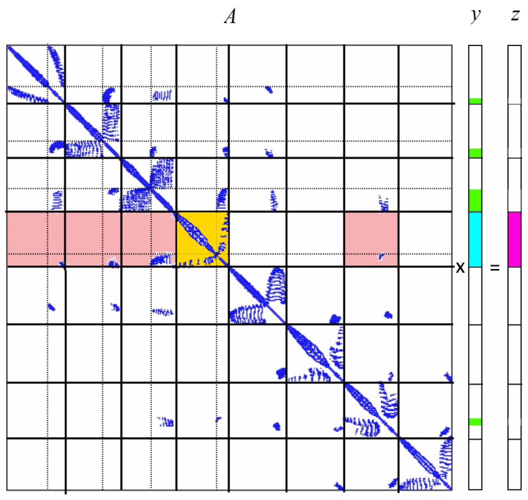


FIGURE 3.1: Schematic representation of the SpMV product.

For an efficient execution, the SpMV algorithm requires a preliminary stage, called *prepare*, during which each MPI rank communicates to its neighbors the list of y entries it needs to receive to perform the product by the off-diagonal blocks. Since this information does not change during the iterative solution of the system, it can be exchanged just once before the computation starts.

As to the local sparse-matrix-dense-vector product, a review of the different approaches developed in recent years is proposed by Filippone et al. [2017]. In the present work, a new CUDA kernel has been developed for the CSR storage format. To this purpose, a group of threads for each row of the sparse matrix is empolyed. Using a group instead of a single thread makes it possible to exploit at its best the memory bandwidth of the GPU, at least loading the elements of the matrix. The group of threads in charge of each row is called *miniwarp* in analogy with the group of 32 consecutive threads named *warp* in the CUDA jargon mentioned above. In particular, sets (having the same number of elements) of contiguous threads with a possible cardinality of 2, 4, 8, 16 or 32 are considered. Those sets of threads are then concurrently mapped to different data like in other warp-centric kernels that use a warp to manage a block of contiguous data. The threads contained in a miniwarp are able to perform cooperative computation by exploiting efficient and fine-grained intra-warp communication primitives like the *shuffle*. During the execution of the product, each row of the sparse matrix is assigned to a single miniwarp, that is, multiple rows are concurrently executed in the same full warp of 32 threads. The size of a miniwarp is dynamically dependent on the average number of nonzeroes per row of the sparse matrix. The main advantage of this miniwarp variant is that, for matrices with few nonzero entries per row, the number of idle threads decreases. With a full warp (32 threads), if a row has, on average, only $k < 32$ nonzero entries, there are, always on average, $32 - k$ threads that remain idle. Miniwarps reduce the difference significantly by using a size that is much closer to the average number of nonzero entries per row. Fig. 3.2 shows how the miniwarp works when applied to a matrix in CSR format in the case of a sparse-matrix dense-vector multiplication.

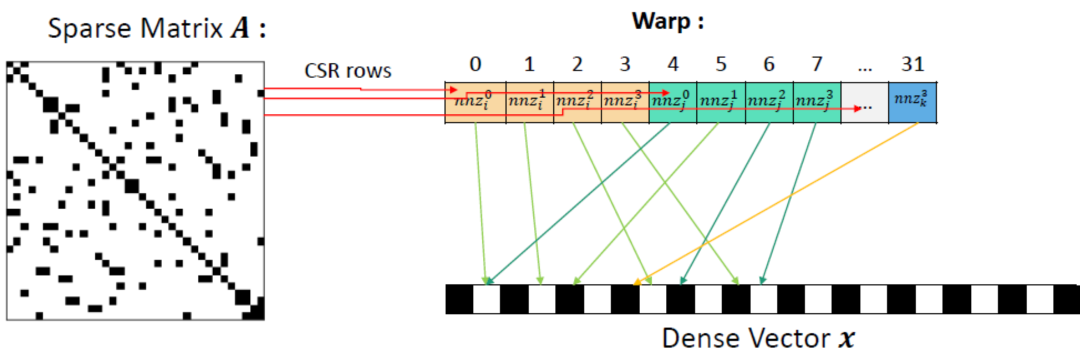


FIGURE 3.2: Each *miniwarp* is in charge of a row of the matrix stored in CSR format.

3.2 aFSAI smoother

Let's recall briefly the theoretical framework of the adaptive Factored Sparse Approximate Inverse (aFSAI).

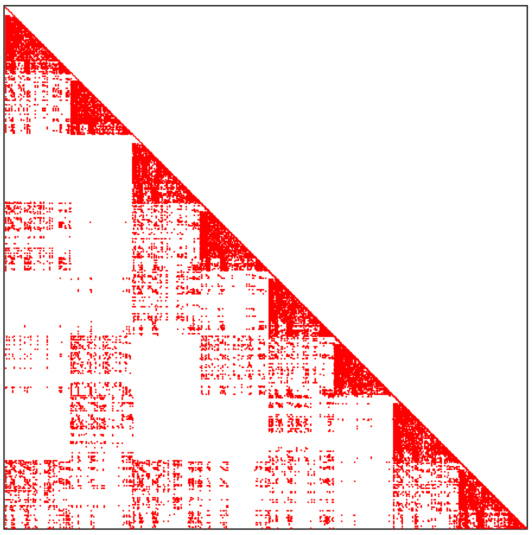


FIGURE 3.3: Example of a typical lower triangular non-zero pattern $\mathcal{S}$.

The FSAI preconditioner has been originally introduced for SPD matrices in Kolotilina and Yeremin [Kolotilina and Y., 1993], with the aim of directly approximating the inverse of A as the product of two triangular factors:

$$M^{-1} = G^T G \simeq A^{-1} \tag{3.1}$$

In the equation (3.1), G is a lower triangular matrix whose entries are computed by minimizing the following Frobenius norm:

$$\|I - GL\|_F \tag{3.2}$$

over the set $\mathcal{W}_\mathcal{S}$ of matrices having a prescribed lower triangular non-zero pattern $\mathcal{S}$, as the one depicted in Figure 3.3. The matrix L, explicitly appearing in (3.2), is the exact Cholesky factorization (lower) of A and it is not actually needed in the computation of FSAI, since it disappears during the minimization process as shown for instance in Kolotilina and Yeremin [Kolotilina and Y., 1993]. The unknown G entries, $[G]_{ij}$, are computed by solving the componentwise system:

$$[GA]_{ij} = \begin{cases} 0 & i \neq j, \quad (i,j) \in \mathcal{S} \\ [L]_{ii} & i = j \end{cases} \tag{3.3}$$

obtained through differention of (3.2) with respect to $[G]_{ij}$ and setting it equal to zero. In the equation above, the symbol $[\cdot]_{ij}$ in (3.3) is used to indicate the entry in row i and column j of the matrix between square brackets. Since $[L]_{ii}$, the i-th diagonal element of L, is unknown, we replace it in eq. (3.3) by 1. As a consequence, in place of G, we compute the matrix $\widetilde{G}$ by solving:

$$[\widetilde{G}A]_{ij} = \delta_{ij} \tag{3.4}$$

with δ_{ij} the Kronecker delta.

From a practical viewpoint, setting up the i-th row of $\widetilde{G}$, say $\widetilde{\boldsymbol{g}}_i^T$, requires:

- first, the definition of the set $\mathcal{P}_i$ collecting all the column indices that belong to the i-th row of $\mathcal{S}$, that is:

$$\mathcal{P}_i = \{j : (i,j) \in \mathcal{S}\} \tag{3.5}$$

- then, the gathering of the dense matrix $A[\mathcal{P}_i, \mathcal{P}_i]$ formed with the entries of A having row/column indices in $\mathcal{P}_i$;

- finally, solving the linear system:

$$A[\mathcal{P}_i, \mathcal{P}_i]\widetilde{G}[i, \mathcal{P}_i]^T = \boldsymbol{e}_{m_i} \tag{3.6}$$

  with $\widetilde{G}[i, \mathcal{P}_i]$ being the dense vector containing the non-zero entries of $\widetilde{\boldsymbol{g}}_i$ and the right-hand side $\boldsymbol{e}_{m_i}$ given by the last vector of the canonical basis of $\mathbb{R}^{m_i}$, with $m_i = |\mathcal{P}_i|$.

Figure 3.4 gives an idea of the gathering process used to collect the dense linear systems $A[\mathcal{P}_i, \mathcal{P}_i]$ given a set $\mathcal{P}_i$.

A more practical, although mathematical equivalent, way to implement the FSAI computation, see Janna and Ferronato [2011]; Janna et al. [2015b], consists in assuming

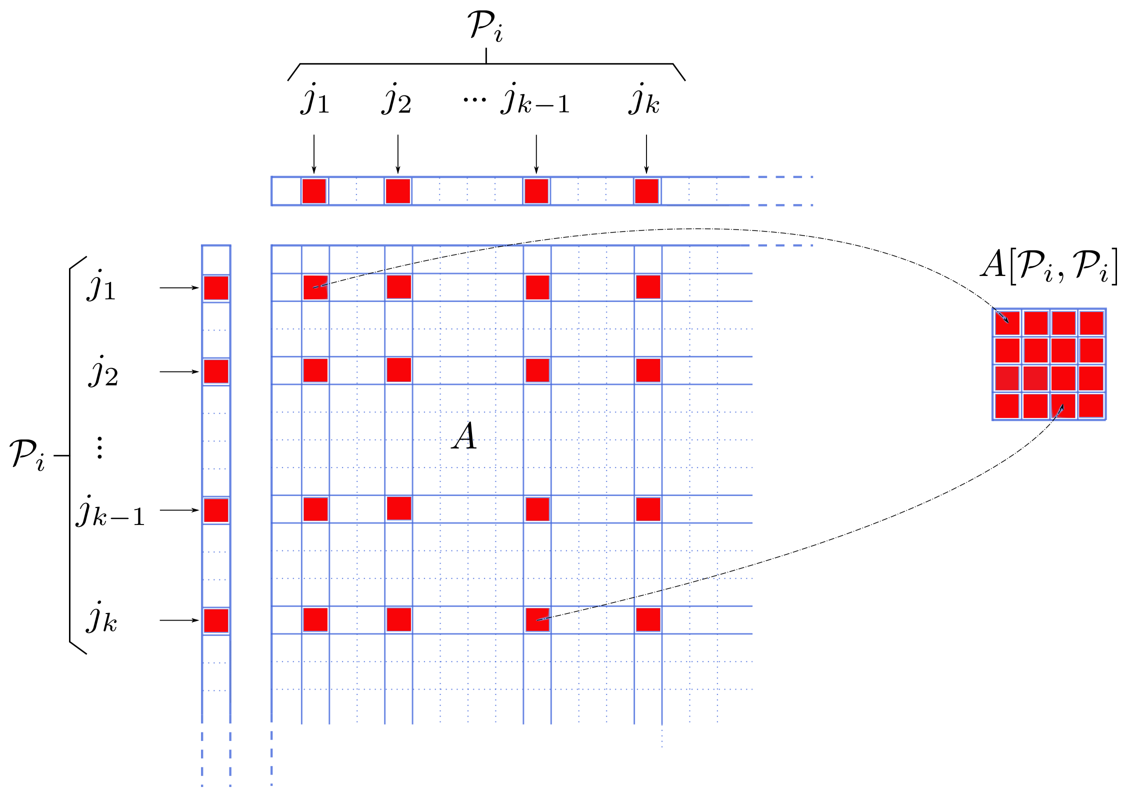


FIGURE 3.4: Schematic representation of the dense linear system gathering for a given set $\mathcal{P}_i$ (with cardinality k).

$\widetilde{G}$ unitary diagonal and solving the linear system:

$$A[\overline{\mathcal{P}_i}, \overline{\mathcal{P}_i}]\widetilde{G}[i, \mathcal{P}_i]^T = -A[\overline{\mathcal{P}_i}, i] \tag{3.7}$$

where $\overline{\mathcal{P}_i} = \mathcal{P}_i \setminus i$ and $\widetilde{G}[i, \mathcal{P}_i]$ contains the *off-diagonal* non-zero entries of $\widetilde{G}$. A diagonal scaling is finally applied to $\widetilde{G}$ in order to guarantee that all the diagonal entries of the preconditioned matrix are unitary:

$$\operatorname{diag}(D_G\widetilde{G}A\widetilde{G}^T D_G) = \operatorname{diag}(GAG^T) = I \tag{3.8}$$

where $\operatorname{diag}(\cdot)$ is the operator returning the diagonal matrix having the diagonal of its argument as entries, I is the identity matrix and the D_G entries are given by:

$$D_G[i, i] = \frac{1}{A[i, i] - \widetilde{G}[i, \mathcal{P}_i]A[\overline{\mathcal{P}_i}, \overline{\mathcal{P}_i}]\widetilde{G}[i, \mathcal{P}_i]^T} \tag{3.9}$$

Condition (3.8) ensures that G, over all the matrices $B \in \mathcal{W}_{\mathcal{S}}$, is the unique one minimizing the Kaporin number of the preconditioned matrix:

$$\kappa = \frac{\frac{1}{n}\mathrm{tr}(GAG^T)}{\det(GAG^T)^{\frac{1}{n}}} \tag{3.10}$$

which gives a measure of the PCG convergence rate Kaporin [1994].

The FSAI preconditioner for an SPD matrix is breakdown-free and possesses an extremely high degree of parallelism in both construction and application to a vector. However, to be competitive with other preconditioners, FSAI needs to be sparse with a non-zero pattern composed by only significant entries of the true inverse factor of A. The appropriate a priori choice of $\mathcal{S}$ is unfortunately very difficult thus making the original FSAI unpractical in ill-condition cases.

A better way to compute FSAI is by choosing $\mathcal{S}$ *adaptively* while computing $\widetilde{G}$ Janna and Ferronato [2011]. The basic concept in the adaptive computation of FSAI (aFSAI from now on) is the improvement of the quality of a given initial factor G_0, already satisfying equations (3.7) and (3.8), by extending its pattern with those entries that mostly contribute in reducing the Kaporin number of $G_0AG_0^T$. Let us define $\mathcal{S}_0$ the non-zero pattern of G_0 and assume the identity matrix as our default initial guess. Writing explicitly the Kaporin number κ of $G_0AG_0^T$, gives:

$$\kappa = \frac{\frac{1}{n}\mathrm{tr}(G_0AG_0^T)}{\det(G_0AG_0^T)^{\frac{1}{n}}} = \frac{\frac{1}{n}\mathrm{tr}(D_{G_0}\widetilde{G}_0A\widetilde{G}_0^TD_{G_0})}{\det(D_{G_0}\widetilde{G}_0A\widetilde{G}_0^TD_{G_0})^{\frac{1}{n}}} \tag{3.11}$$

and, recalling that $G_0AG_0^T$ has unitary diagonal entries and $\det(\widetilde{G}_0) = 1$ by construction, it follows that:

$$\kappa = \frac{1}{\det(A)^{\frac{1}{n}}}\frac{1}{\det(D_{G_0})^{\frac{2}{n}}} = \frac{\det\left[\mathrm{diag}(\widetilde{G}_0A\widetilde{G}_0^T)\right]^{\frac{1}{n}}}{\det(A)^{\frac{1}{n}}} \tag{3.12}$$

Denoting by $\widetilde{\boldsymbol{g}}_{0,i}^T$ the i-th row of $\widetilde{G}_0$, we can write the numerator of (3.12) as:

$$\det\left[\mathrm{diag}(\widetilde{G}_0\widetilde{A}\widetilde{G}_0^T)\right] = \prod_{i=1}^{n}\widetilde{\boldsymbol{g}}_{0,i}^TA\widetilde{\boldsymbol{g}}_{0,i} = \prod_{i=1}^{n}\psi_{0,i} \tag{3.13}$$

having defined $\psi_{0,i} = \widetilde{\boldsymbol{g}}_{0,i}^T A \widetilde{\boldsymbol{g}}_{0,i}$. With the above definiton, a simplified expression for the Kaporin conditioning number of the preconditioned matrix can be found:

$$\kappa = \left(\frac{\prod_{i=1}^n \psi_{0,i}}{\det(A)} \right)^{\frac{1}{n}} \tag{3.14}$$

which can be differentiated to obtain the gradient of κ with respect to $\widetilde{G}_0$. Choosing the positions of the gradient corresponding to its largest entries in absolute value is an effective heuristics, as it allows for finding an augmented pattern $\mathcal{S}_1$ giving a large reduction of κ in equation (3.14). Since each $\psi_{0,i}$ in (3.14) does not depend on any other row of $\widetilde{G}_0$ except the i-th one, the pattern expansion can be carried out on each row concurrently. The $\psi_{0,i}$ gradient, denoted as $\nabla\psi_{0,i}$, is obtained by collecting its partial derivatives with respect to the components of $\widetilde{\boldsymbol{g}}_{0,i}$:

$$\frac{\partial \psi_{0,i}}{\partial [\widetilde{G}_0]_{ij}} = 2 \left(\sum_{r=1}^n a_{jr} \, [\widetilde{G_0}]_{ir} + a_{ji} \right), \qquad \forall j = 1, \dots, i-1, \tag{3.15}$$

with the computational cost for its evaluation consisting in a single sparse-matrix by sparse-vector product. A new pattern $\overline{\mathcal{P}}_i^1$ is obtained by enlarging $\overline{\mathcal{P}}_i^0$ with the s positions corresponding to the largest entries of $\nabla\psi_{0,i}$ in absolute value and finally $\widetilde{\boldsymbol{g}}_{1,i}$ is computed by solving (3.7), after replacing the sub/superscript "0" with "1". Repeating the above procedure k_{max} times allows one to find all the rows of $\widetilde{G}_2$, $\widetilde{G}_3$, ..., up to $\widetilde{G}_{k_{max}}$. It is also possible to monitor the preconditioner quality by computing the value of $\psi_{k,i}$. Hence it is possible to stop the adaptive procedure independently for every row when

$$\frac{\psi_{k,i}}{\psi_{0,i}} = \frac{\widetilde{\boldsymbol{g}}_{k,i} A \widetilde{\boldsymbol{g}}_{k,i}^T}{\widetilde{\boldsymbol{g}}_{0,i} A \widetilde{\boldsymbol{g}}_{0,i}^T} \leq \varepsilon, \tag{3.16}$$

where ε is a user-specified tolerance. Once the approximation of $\widetilde{G}$ is satisfactory, a diagonal scaling is applied, to ensure a unitary diagonal for GAG^T.

Three user-specified parameters are generally used to control the aFSAI quality:

1. k_{max}, the maximum number of steps of the iterative procedure for each row;
2. s, the number of new positions added to the non-zero pattern of each row in a single step;

3. ε, the relative tolerance on the Kaporin number reduction used to stop the procedure.

3.2.1 Gathering

Before calling the specific GPU kernels for the aFSAI set-up on every single device, each MPI processes needs to collect some information from the others. This preliminary stage is refferred as *gathering*. As the pattern of G is computed dynamically during the set-up, in principle each MPI rank may need the entire matrix A, whose entries belonging to other processes with lower rank. Obviously, such a condition cannot be satisfied in very large problems because the whole A matrix cannot be stored on a single node. On the other hand, preliminary tests showed that using only the information from neighboring processes, as in the communication pattern of SpMV, is not enough to build a satisfactory aFSAI.

Two intermediate gathering strategies based on heuristics have been investigated: *block* and *row* gatherings. The former makes maximum use of the CSR block structure of the DSMat object while the latter is designed to reduce the required storage.

Let's start with the *block gathering* denoting by $\widehat{A}$ the communication matrix of A, that is a boolean matrix of size $n_p \times n_p$ with entries $\widehat{A}_{ij} \neq 0$ if and only if the corresponding block ij of A contains at least one non-zero element. Then, it is imposed that $\widehat{G}$, the communication graph of G, is no larger than the lower triangular part of $\widehat{A}^k$ for a small power k, typically in the range $1 \div 3$, which is simply computed through Sparse-Matrix-by-Sparse-Matrix products between the communication matrices. Once $\widehat{G}$ is known, it is used to guide the collection of information from other MPI processes. With reference to Fig. 3.5, for instance, process 3 assembles locally the matrix A_3, that it uses locally to compute its own stripe G_3 of G by gathering all the green blocks from its left-neighboring processes on $\widehat{G}$. Thanks to the symmetry, only the communication of the green blocks in the lower part is needed as a transpose can be used to obtain the additional entries.

Note that, again as a positive effect of the symmetry, each MPI rank knows in advance from the pattern of $\widehat{A}^k$ which blocks it has to receive and send to the other processes as well as the list of processes it needs to communicate with. The *block gathering*

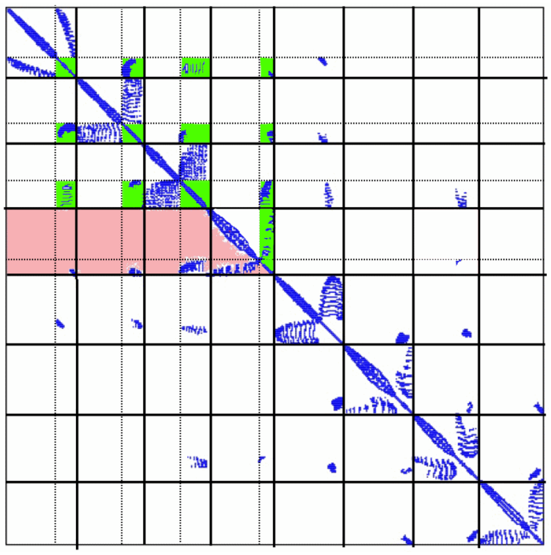


FIGURE 3.5: Schematic representation of the block collection for process 3. Green colored blocks in the lower part of the matrix A are those that need to be gathered for the assembly of A_3.

procedure consists of a preliminary computation stage of $\widehat{A}^k$ and of 2 stages of communications, that can be partly overlapped by using non-blocking communication:

1. Each MPI rank computes its part of $\widehat{A}^k$.
2. Each MPI rank sends to its right-neighboring processes (and receives from the left ones) the sizes of the blocks that have to be exchanged.
3. Each MPI rank sends to its right-neighboring processes (and receives from the left ones) the above set of blocks.

Note that, if a similar approach is extended to non-symmetric systems, each MPI rank needs to communicate to the others also the list of required blocks, which is not known, a priori, as in the symmetric case. The communicated list of blocks can be conveniently received into a 2-dimensional array data structure with size $(n_L + 1)^2$, where n_L is the number of left-neighboring processes in $\widehat{G}$. Only the block-lower part of this structure is actually filled with $({n_L}^2 + n_L)/2$ blocks received from the other MPI processes plus the (n_L+1) blocks already in place. Once all the communications have been completed, the 2-dimensional array of CSR matrices is copied into the full (lower plus upper) CSR matrix A_{I_p} that is the main input for the aFSAI GPU kernel. Observe that, during this copy, these two large structures must coexist, becoming the main part of the memory footprint of each MPI taks, that can be quantified as ~ 1.6 the size of A_{I_p}.

The *row gathering* procedure has been investigated in order to remove this storage bootleneck. Let's denote with y_{ji} the list of $\boldsymbol{y}$ entries that the j-th MPI rank sends to the i-th rank during the SpMV oparation (Section 3.1). In order to compute an effective aFSAI, the i-th MPI rank gathers from j-th rank (with $j < i$) the expanded set of rows y^k_{ji}, where k is a distance typically in the range $4 \div 8$ (for example, y^2_{ji} is the set of connections that includes y_{ji} and its neighbors). Since the *row gathering* doesn't rely on the CSR block structure of the DSMat object, this procedure is a little more complex than the previous one, and composed by the following stages:

1. Each MPI rank computes the list of rows to send to its right-neighboring processes.
2. Each MPI rank collects the rows to send to its right-neighboring processes.
3. Each MPI rank sends to its right-neighboring processes (and receives from the left ones) the sizes of the rows that have to be exchanged.
4. Each MPI rank sends to its right-neighboring (and receives from the left ones) the above set of rows.
5. Each MPI rank removes all unnecessary enrties that compromise the symmetry of the A_{I_p} pattern (the symmetry is a requirement of the local kernels described in the next section).

This approach has an approximately halved memory footprint, compared to block-gathering, while still allowing an effective aFSAI to be calculated. Block gathering is generally preferable when FSAI is used as the preconditioner itself and the problem is severely ill-conditioned

As already noted in Bernaschi et al. [2019], the aFSAI set-up can be safely performed in single precision and, in the distributed memory context, this option is particularly advantageous. In fact, beyond reducing processing time due to the use of single-precision arithmetic, it also allows a smaller amount of data movement during the *gathering* stage and the *HostToDevice* copies. Two casting operations are needed in this case. The first one is performed on the matrix A at the *Host* level: A is copied into a new matrix A_s using single precision floats for its entries, $A_s = \text{single}(A)$. Then, the matrix A_s is used for the aFSAI set-up, as described above. Once G_s, G with single precision entries, is computed, a second cast is performed at the *Device* level transforming single precision floats into doubles, $G = \text{double}(G_s)$. This last casting operation can be conveniently overlapped with the *HostToDevice* copy of the double precision A, since only A_s has been transferred before. When both the double precision representation of A and G are present on the *Device*, the Krylov method is ready to start.

3.2.2 GPU kernels

The three main kernels involved in the aFSAI set-up are the following:

1. *Kaporin*: the computation of the Kaporin gradient $\nabla\psi_i$, and the selection of its most relevant components.
2. *SystemGathering*: the collection of A entries to form $A[\overline{\mathcal{P}}_i, \overline{\mathcal{P}}_i]$ and $A[\overline{\mathcal{P}}_i, i]$.
3. *SystemSolution*: the solution of collected dense SPD systems, carried out by means of Cholesky decomposition.

Once the matrix A_{I_p} is gathered, the computation of each row of G can take place independently from the others. In order to exploit this high potential for parallelism, a specific set-up algorithm has to be designed according to the hardware features. The CPU implementation is based on the algorithm proposed in Janna et al. [2015b] where the outermost loop involves the matrix rows while for the GPU the so-called *batched* approach proposed by Bernaschi et al. [2016, 2019] has been used. According to this approach, the three main kernels of the set-up manage row batches with the same number of non-zeros. In the adaptive-FSAI the number of non-zeros of G is not known a priori, since it is controlled dynamically by ε at the row level. To address this issue, the outermost loop involves the k steps, taking care also of excluding the rows that reached the criterion (3.16) during the iterations.

The allocation of the GPU resources can be made at the beginning of the computation according to maximum number of non-zeros per row, that is the product between k_{max} and s. In particular, the *Kaporin* kernel uses a different number of CUDA threads per row than *SystemGathring* and *SystemSolution*: 32 threads (1 warp) versus 1 thread

Algorithm 5 **aFSAI set-up on CPU**

```
 1: procedure AFSAI_CPU_SETUP($k_{max}$,$s$,$\varepsilon$,$A_{I_p}$,$\widetilde{G}$)
 2:     $\widetilde{G}_0 \leftarrow I$;
 3:     $i \leftarrow 0$;
 4:     while $(i < n)$ do                                   ▷ loop over $\widetilde{G}_0$ rows
 5:         $k \leftarrow 0$;
 6:         while $(k < k_{max})$ do                         ▷ loop over $k$ steps
 7:             Compute $\nabla\psi_{k,i}$;
 8:             Form $\overline{\mathcal{P}_i}^{k+1}$ by adding to $\overline{\mathcal{P}_i}^{k}$ the $s$ largest positions of $\nabla\psi_{k,i}$;
 9:             Gather $A_{I_p}[\overline{\mathcal{P}_i}^{k+1},\overline{\mathcal{P}_i}^{k+1}]$ and $A_{I_p}[\overline{\mathcal{P}_i}^{k+1},i]$ from $A_{I_p}$
10:             Solve $A_{I_p}[\overline{\mathcal{P}_i}^{k+1},\overline{\mathcal{P}_i}^{k+1}]\widetilde{G}[i,\overline{\mathcal{P}_i}^{k+1}]^T = -A_{I_p}[\overline{\mathcal{P}_i}^{k+1},i]$;
11:             if $(\psi_{k,i} \leq \psi_{0,i} \cdot \varepsilon)$ then       ▷ check convergence
12:                 break;
13:             end if
14:             $k \leftarrow k+1$;
15:         end while
16:         $i \leftarrow i+1$;
17:     end while
18: end procedure
```

per row, respectively. Therefore, two different sizes for the batches are defined according to the available global memory, $size_{kap}$ and $size_{sys}$ and the total number of batches are simply $nb_{kap} = n/size_{kap}$ and $nb_{sys} = n/size_{sys}$, respectively.

The CPU and GPU procedures are provided in Algorithm 5 and 6, respectively. Regarding the GPU algorithm, the array `Done` of size *n* is used to mark the rows that fullfill the criterion (3.16) and can be neglected in the next k steps. Lines 11 and 12 of the algorithm call the function for *Kaporin*, while lines 23 and 24 call the two functions for *SystemGathering* and *SystemSolution*, refer to the works by Bernaschi et al. [2016, 2019] for a detailed description of the algorithm.

As shown in Bernaschi et al. [2019], the computation of the Kaporin gradient is the part of the preconditioner set-up that requires more time. Although the algorithm for the evaluation of the Kaporin gradient is the same as in Bernaschi et al. [2019], its CUDA implementation is different due to changes introduced in the *warp-level* primitives starting on the CUDA toolkit version 9.0. Since CUDA has been around for a while, we are not going to describe it in detail. However, is important to recall that, in the CUDA, GPUs execute warps of 32 parallel threads according to an execution model that NVIDIA calls Single Instruction Multiple Thread (SIMT), in other words,

Algorithm 6 aFSAI set-up on GPU

```
1: procedure AFSAI_GPU_SETUP($k_{max}$,$s$,$\varepsilon$,$A_{I_p}$, $nb_{kap}$,$size_{kap}$,$nb_{sys}$,$size_{sys}$,$\widetilde{G}$)
2:     $\widetilde{G}_0 \leftarrow I$;
3:     $Done \leftarrow 0$;
4:     $k \leftarrow 0$;
5:     while $(k < k_{max})$ do                                    ▷ loop over $k$ steps
6:         $j, i \leftarrow 0$;
7:         while $(j < nb_{kap})$ do                               ▷ loop over kaporin batches
8:             while $(i < size_{kap} \cdot (1+j))$ do             ▷ loop over batch rows
9:                 if $(Done[i] = 0)$ then
10:                    Compute $\nabla\psi_{k,i}$;
11:                    Form $\overline{\mathcal{P}_i}^{k+1}$ by adding to $\overline{\mathcal{P}_i}^{k}$ the $s$ largest position of $\nabla\psi_{k,i}$;
12:                end if
13:                $i \leftarrow i + 1$;
14:            end while
15:            $j \leftarrow j + 1$;
16:        end while
17:        $j, i \leftarrow 0$;
18:        while $(j < nb_{sys})$ do                               ▷ loop over system batches
19:            while $(i < size_{sys} \cdot (1+j))$ do             ▷ loop over batch rows
20:                if $(Done[i] = 0)$ then
21:                    Gather $A_{I_p}[\overline{\mathcal{P}_i}^{k+1}, \overline{\mathcal{P}_i}^{k+1}]$ and $A_{I_p}[\overline{\mathcal{P}_i}^{k+1}, i]$ from $A_{I_p}$
22:                    Solve $A_{I_p}[\overline{\mathcal{P}_i}^{k+1}, \overline{\mathcal{P}_i}^{k+1}]\widetilde{G}[i, \overline{\mathcal{P}_i}^{k+1}]^T = -A_{I_p}[\overline{\mathcal{P}_i}^{k+1}, i]$;
23:                    if $(\psi_{k,i} \le \psi_{0,i} \cdot \varepsilon)$ then        ▷ check convergence
24:                        $Done[i] \leftarrow 1$;
25:                    end if
26:                end if
27:                $i \leftarrow i + 1$;
28:            end while
29:            $j \leftarrow j + 1$;
30:        end while
31:        $k \leftarrow k + 1$;
32:    end while
33: end procedure
```

a variant of the SIMD (Single Instruction, Multiple Data) model, which is one of the four classes defined by the classic Flynn's taxonomy. CUDA toolkits prior to version 9.0 relied on the implicit assumption that the threads within a warp worked in a fully synchronous way. If this is not true, a program may show unexpected side effects up to the point of being unreliable. The point can be illustrated in a simple case based on the `__ballot(`*predicate*`)` primitive that returns an `unsigned int` whose n^{th} bit is set if-and-only-if *predicate* evaluates as `true` for the n^{th} thread of the warp. In the fragment of code shown in Figure 3.6, the CUDA compiler and the hardware should try to re-converge the threads right after the `if/else` block for better performance.

```
if (thread_id % 2) {
    result = foo();
} else {
    result = bar();
}
unsigned ballot_result = __ballot(result);
```

FIGURE 3.6: Unsafe CUDA programming based on the implicit assumption that warp's threads run synchronously.

But this re-convergence is not guaranteed in the most recent versions of the CUDA toolkit. Therefore, the `ballot_result` variable may not contain the ballot result from *all* 32 threads. Starting on version 9.0, up to version 10.0 of the CUDA toolkit the legacy warp-level primitives worked synchronously (albeit with a deprecation warning at compilation time). But, starting on version 10.1, the only alternative to obtain the expected behaviour, is to employ an explicit control on the threads that participate in warp operations by using the new form of the warp-level primitives. The set of threads that participates in each primitive is specified by means of a 32-bit mask, which is always the first argument in the new syntax of the warp-level primitives. So, for instance, the new form of the `__ballot()` primitive is `__ballot_sync(mask,` *predicate*`)`. All the participating threads are synchronized before the execution if they are not already synchronized. In the simple case reported in Figure 3.6, it is enough to replace the last line with `unsigned ballot_result = __ballot_sync(0xFFFFFFFF, result);`. Unfortunately there is not a general rule to determine what is the right value of the mask argument. The set of threads to be included in the mask is determined by the program logic, and may depend on branch conditions or other information available only at execution time. That is exactly the situation that occurred with the kernel for the evaluation of the Kaporin gradient that relies heavily on warp-level primitives like *shuffle* to exchange values stored in the registers among threads belonging to the same warp (as described in Bernaschi et al. [2019] this is the technique that makes very efficient the selection of minimum and maximum values within that kernel).

Let's see an example of the contributions of the three main kernels in the following fragment of the CUDA profiler (`nvprof`) output for a medium size matrix (about 5 million unknows and an average of 45 entries per row):

```
Time(%)      Time  Calls       Avg       Min       Max  Name
 44.14%  8.15852s  17310  471.32us  3.5480us  1.3083ms Kaporin
 27.64%  5.10948s    660  7.7416ms  4.2520us  29.935ms SystemGathering
 24.74%  4.57197s    660  6.9272ms  2.6200us  35.367ms SystemSolution
...
```

The three kernels take more than 90% of the time. The remaining kernels are used to check if the convergence criterion for the refinement has been reached and to build, in the end, the preconditioner. None of them takes more than 0.5% of the total execution time and are all pretty simple.

3.3 BAMG prolongator

The BAMG prolongator, as previously described in Section 2.2, is generally used when a test space is available. For each fine node $i \in \mathcal{F}$, an adaptive procedure is implemented to compute the prolongation weights $\mathbf{w}_i$ by minimizing the residual r_i of equation 2.16.

To ensure good interpolation, this approach collects an interpolatory set of coarse nodes C_i that does not belong entirely to the domain managed by the single MPI rank. Therefore, before calling the GPU kernels, a preliminary *gathering* stage is necessary to collect all information from neighbouring domains: F/C indices and the corresponding test space. First of all, weak connections are filtered out and SoC is used to guide gathering. In BAMG, SoC based on strong couplings has proved to be very effective. Then, the portion of the domain that the i-th MPI rank gathers from the j-th rank is defined in a similar way as in the *row gathering* used for the aFSAI set-up. In particular, for strong connections only in the set y_{ji}^k, where k is typically $2 \div 4$, the i-th MPI rank gathers F/C indices and the test space. The procedure is composed by the following stages:

Algorithm 7 **BAMG set-up on CPU**

```
1: procedure BAMG_CPU_SetUp($l_{min}$,$l_{max}$,$\epsilon$,$\mu$,$FC_{I_p}$,$V_{I_p}$,$W$)
2:     $W \leftarrow 0$;
3:     $i \leftarrow 0$;
4:     while ($i < n$) do                                        ▷ loop over rows
5:         if ($i \in \mathcal{F}$) then
6:             $l \leftarrow 0$;
7:             while ($l < l_{min}$) do                          ▷ loop over $l_{min}$ distance
8:                 Update $C_i$ and assembly $\Phi_i$;
9:                 $l \leftarrow l + 1$;
10:            end while
11:            while ($l < l_{max}$) do                          ▷ loop over $l_{max}$ distance
12:                Update $C_i$ and assembly $\Phi_i$;
13:                Select a basis for $\Phi_i$ by using $maxvol$ algorithm;
14:                Compute $\mathbf{w}_i$ by minimizing $r_i$;
15:                if ($r_i < \epsilon$ or $\|\mathbf{w}_i\| < \mu$) then      ▷ check convergence
16:                    break;
17:                end if
18:                $l \leftarrow l + 1$;
19:            end while
20:        end if
21:        $i \leftarrow i + 1$;
22:    end while
23: end procedure
```

1. Each MPI rank computes the list of strong connections to send to its neighboring (left and right) processes.

2. Each MPI rank collects the the F/C indices and the test space to send to its neighboring (left and right) processes.

3. Each MPI rank sends to (and receives from) its neighboring processes the sizes of the lists that have to be exchanged.

4. Each MPI rank sends to (and receives from) its neighboring processes the above set of F/C nodes and the test space.

Once communications are completed, the received information are merged with those locally defined on the processor's subdomain creating two structures, FC_{I_p} and V_{I_p}, for the F/C indices and the test space, respectively. These are the main input for the computation kernels.

Three main kernels are involved in the BAMG set-up:

1. *FindSet*: the search for the interpolatory set C_i at a certain distance l, with $l_{min} < l < l_{max}$ and the assembly of the matrix Φ_i. This dense matrix collects the $\mathbf{v}_k$ components, where $\mathbf{v}_k$ is the k-th row of the test space V_{I_p}.

2. *MaxVolume*: the setting of a proper basis for Φ_i by the *maxvol* algorithm [Goreinov et al., 2010; Knuth, 1985] to remove almost-parallel vectors $\mathbf{v}_k$ that may produce large jumps in the weights.

3. *WeightCompute*: the weight computation carried out by means of least square minimizations.

Once each MPI ranks completes the gathering stage, the computation of each row of the W block in (2.5), $\mathbf{w}_i$, can take place independently from the others. Again, specific algorithms are developed to exploit the different hardware features of the CPU and GPU. The CPU-only implementation is based on a straightforward approach where the outermost loop involves W rows while for the GPU a *batched* approach is used. During the search procedure of the interpolatory set, each node i must store all neighbouring connections (F/C both), and due to the storage limitations of the global memory of the GPU board, only one batch of rows can be handled at a time. Therefore, in the GPU implementation the outermost loop involves the batches of rows, taking care of excluding the rows that reached one of the following criteria: $r_i < \epsilon$ or $\|\mathbf{w}_i\| < \mu$, with ϵ and μ user defined thresholds. Note that if at l_{max} the interpolation doesn't not meet any of the criteria, the node i is promoted to coarse (this generally only happens for few nodes at the first level of the AMG hierachy).

The allocation of GPU resources is made at the beginning of the computation defining the size of the batches $size_{BAMG}$ according to the maximum number of neighboring connections: this is not known a priori and it is assumed equal to 3,000 based on heuristics. All the (few) rows with a larger number of neighbor connections are marked and handled together at the end by repeating the procedure with a single batch. The total number of batches is $nb_{BAMG} = n/size_{BAMG}$.

The CPU and GPU procedure are provided in Algorithm 7 and 8, respectively. Regarding the GPU algorithm the array `Done` of size *n* is used to mark the rows that fullfill the convergence criteria and can be neglected in the next l steps. Lines 9 and 19 of the algorithm call the function for *FindSet* during the distance loop over l_{min} and l_{max}, respectively. Lines 23 and 24 call the two functions for *MaxVolume* and *WeightCompute*. In the *FindSet* kernel particular care was taken to implement the neighbor connection search in CUDA. From an algebraic point of view, this operation is equivalent to a symbolic merge of sparse matrix rows and it is not trivial to develop an algorithm that exploits the GPU board. A hash-based approach was adopted following the one proposed by Nagasaka et al. [2017] in the context of the Matrix-by-Matrix (MxM) product. Since this approach is the core kernel of the MxM product, it will be described in detail in the next Section. Finally, the GPU porting of *MaxVolume* and *WeightCompute* kernels straightforward: the former kernel implements in CUDA exactly the same algorithm of the CPU-only version using 1 CUDA thread per row while the latter exploits the `cublasDgelsBatched` provided by the CUDA Toolkit [NVIDIA et al., 2023].

Algorithm 8 **BAMG set-up on GPU**

```
 1: procedure BAMG_GPU_SETUP($l_{min}$,$l_{max}$,$\epsilon$,$\mu$,$FC_{I_p}$,$V_{I_p}$,$W$)
 2:     $W \leftarrow 0$;
 3:     $Done \leftarrow 0$;
 4:     $j, i \leftarrow 0$;
 5:     while ($j < nb_{BAMG}$) do                                  ▷ loop over batches
 6:         while ($l < l_{min}$) do                                ▷ loop over $l_{min}$ distance
 7:             while ($i < size_{BAMG} \cdot (1 + j)$) do          ▷ loop over batch rows
 8:                 if ($i \in \mathcal{F}$) then
 9:                     Update $C_i$ and assembly $\Phi_i$;
10:                 end if
11:                 $i \leftarrow i + 1$;
12:             end while
13:             $l \leftarrow l + 1$;
14:         end while
15:         $i \leftarrow i - size_{BAMG}$;
16:         while ($l < l_{max}$) do                                ▷ loop over $l_{max}$ distance
17:             while ($i < size_{BAMG} \cdot (1 + j)$) do          ▷ loop over batch rows
18:                 if ($i \in \mathcal{F}$ and $Done[i] = 0$) then
19:                     Update $C_i$ and assembly $\Phi_i$;
20:                     Select a basis for $\Phi_i$ by using $maxvol$ algorithm;
21:                     Compute $\mathbf{w}_i$ by minimizing $r_i$;
22:                     if ($r_i < \epsilon$ or $\|\mathbf{w}_i\| < \mu$) then      ▷ check convergence
23:                         $Done[i] \leftarrow 1$;
24:                     end if
25:                 end if
26:                 $i \leftarrow i + 1$;
27:             end while
28:             $l \leftarrow l + 1$;
29:         end while
30:         $j \leftarrow j + 1$;
31:     end while
32: end procedure
```

3.4 MxM product

In this section the implementation of MxM product on a multi-GPU setting is described. The MxM product plays a major role in the computation of the coarse matrix (2.6) and this problem, so far, has received limited attention. A multi-GPU algorihm has been recently proposed by Azad et al. [2022], however, this approach uses complicated 2D and 3D partitioning of the matrices that make it difficult to embed in the framework of an existing library as Chronos.

Regardless of the hardware, CPU or GPU, one of the main issue in the MxM product is represented by the number of non-zero entries as well as the sparsity pattern of the resulting matrix product that are not predictable in advance. To get the required information, many solutions resort to a so-called *symbolic* stage, in which the number of nonzeros in the result matrix is computed, postponing the actual computation of the values to a following *numeric* stage [Nagasaka et al., 2017; Mathias et al., 2020; NVIDIA et al., 2023]. Even if the sparsity pattern of the output was known, achieving a balanced work distribution and a suitable (i.e, as regular as possible) access to the *global* memory is far from being trivial on a GPU. Moreover, the situation gets worse when the size of the problem does not fit in the memory of a single GPU.

A straightforward solution to split the product computation among the GPUs is that each GPU computes the corresponding block of rows of the product matrix. As an example, in the most simple configuration with just two GPUs, the first GPU is in charge of computing the first half of the rows and the second GPU is in charge of the second half of the rows of the resulting matrix. The steps to be carried out are described below for only computing the rows of just the first half of the product matrix (i.e., from the viewpoint of the first GPU) but the same approach can be applied to any subset of rows. To finalize the scalar products of the rows of the first matrix by the columns of the second matrix, each GPU needs the rows of the second matrix corresponding to the column indices of each row of the first matrix that are above or below its range of row indices. In this example the first GPU has just the nonzeros of the first half of each column of the second matrix. If a row of the first matrix belonging to its subset has nonzeros whose column index is larger than $n/2$, where n is the total number of rows (assume that n is an even number), it needs those nonzero entries to complete the product. In the following example of two very small (4×4) matrices (the horizontal line represents the distribution of data between the two GPUs), the first GPU, in order to complete its part of the product, needs the elements, b_{41} and b_{44} of the last row in charge of the second GPU.

$$\left[\begin{array}{cccc} a_{11} & 0 & 0 & a_{14} \\ 0 & a_{22} & 0 & a_{24} \\ \hline 0 & 0 & a_{33} & 0 \\ a_{41} & a_{42} & 0 & a_{44} \end{array}\right] * \left[\begin{array}{cccc} b_{11} & 0 & 0 & b_{14} \\ 0 & b_{22} & b_{23} & 0 \\ \hline 0 & b_{32} & b_{33} & 0 \\ b_{41} & 0 & 0 & b_{44} \end{array}\right] =$$

$$\left[\begin{array}{cccc} a_{11}b_{11} + a_{14}b_{41} & 0 & 0 & a_{11}b_{14} + a_{14}b_{44} \\ a_{24}b_{41} & a_{22}b_{22} & a_{22}b_{23} & a_{24}b_{44} \\ \hline & & & \\ \dots & & & \\ \dots & & & \end{array}\right]$$

It is possible to exchange all the data necessary to complete the product on each GPU before starting the computation, so that the product appears as if it were completely local from the viewpoint of the computing kernel. Regarding the kernel that addresses the computation on the single GPU, an enhanced version of the *nsparse* [Nagasaka et al., 2017] is adopted. In fact, this *nsparse* open-source solution proved to be much more efficient of any combination of *cuSparse* (i.e., the official NVIDIA library for sparse matrix operations [NVIDIA et al., 2023]) primitives for the MxM. With the choice of exchanging all information in advance of the computing kernel, the efficiency of *nsparse* is fully exploited by calling it just once. As an alternative, it is possible to compute the local part of the product, exchanging, in the mean time, the required data, then compute the remaining part of the product and sum the two contributions. This choice could offer an advantage due to the potential overlap of the computation of the local part with the exchange of the data for the non local part. Even if this latter scheme is used on the CPU-only version, its use in the multi-GPU MxM is not convenient. Indeed, it entails a double execution of the *nsparse* kernel (with two different symbolic steps) and also the execution of an additional kernel for the sum of the two partial products. The latter kernel is not completely trivial, since the matrices are stored in CSR format.

In summary, the first procedure for the multi-GPU MxM is composed by the following steps, as sketched in Algorithm 9:

1. each GPU checks which rows of the second matrix needs to be received from other GPUs, (see lines 2 to 6 in Algorithm 9).
2. CPUs exchange information about the number of non-zeros entries of each required row, using MPI collective communication primitives (`MPI_Allgather, MPI_Alltoall, MPI_Alltoallv`) (see line 7 in Algorithm 9).
3. CPUs, after the allocation of suitable memory buffers (whose size is determined in the previous step), exchange the indices and the corresponding values of the non-local rows using MPI point-to-point non blocking communication primitives, (see lines 8 and 9 in Algorithm 9);
4. Each GPU builds the subset of rows of the second matrix that it needs, by combining the rows it already owned with those received by other GPUs in the previous step. To minimize the number of memory copy operations, a new form of sparse matrix representation is introduced, the *segmented* CSR. The idea is to maintain the local part of the matrix in its original CSR data structure and to store the non-local part, coming from other MPI processes, in an auxiliary CSR data structure. The *nsparse* has been amended so that, depending on a simple check, either the primary (local) or the auxiliary CSR is used, (see line 10 in Algorithm 9).
5. Each GPU carries out the product between its part of the first matrix and the suitable subset of the second matrix built in the previous step, (see line 11 in Algorithm 9).

3.4.1 Enhancements to the nsparse kernel

Several changes have been introduced to the original nsparse implementation while the main workflow remained the same (see, for reference, Nagasaka et al. [2017]). To recap the original version, the matrix-by-matrix product of two CSR matrices is done in two main stages: symbolic and numeric. The symbolic stage is needed to determine the number of nonzeros of the output matrix whereas the numeric stage is used for calculating the nonzero elements. In order to cope with load imbalance caused by variable number of nonzeros of the resulting matrix, the updated version relies on the

Algorithm 9 MxM product on GPU

Input: Two matrices A and B, each process owns only a block of consecutive rows (from h to k) of each matrix.
Output: Matrix $C = AB$, each process computes only its block of consecutive rows (from h to k).

```
1: $rowsToReceive \leftarrow 0$
2: for $a_{ij} \in A_{local}$ do
3:     if $j < h$ or $j > k$ then
4:         $rowsToReceive \leftarrow j$
5:     end if
6: end for
7: $nnzPerRow \leftarrow MPI_{collective}(nnz(B(rowsToReceive)))$
8: Allocate memory for $B(rowsToReceive)$ combining the information contained in
   $nnzPerRow$ and $rowsToReceive$
9: $B(rowsToReceive) \leftarrow MPI_{point-to-point}(B_{remote})$
10: $B_{segmented} \leftarrow merge(B_{local}, B(rowsToReceive))$
11: $C \leftarrow sparseProduct(A_{local}, B_{segmented})$
12: return $C$
```

binning of the output matrix rows, in the same way as the original nsparse driver. Despite keeping the original code structure, there are several significant improvements that result in a speed-up around 2. In particular:

1. updating of *shuffle* primitives
2. an enhanced hash table algorithm
3. addition of new bins
4. direct access of A rows

The nsparse implementation relies on the lagacy behaviour of CUDA *shuffle* primitives, therfore these have been updated according to their new definition introduced from CUDA-9.

The goal of the new hash table algorithm is to accelerate the access to the GPU shared memory and eliminate unnecessary branching. Instead of the original atomic access to the shared memory, a *while* loop is added to help finding a free spot with no need of an atomic operation. This allows to quickly navigate through unoccupied slots in the hash table and insert the column index *key*, and the value *val* in a smaller number of atomic steps. The performance advantage of this variant is more evident in the numeric part of the driver since in that case the bin choice is exactly determined by the symbolic stage which allows for using a precisely defined hash table size for a given row of the output matrix. Since in this situation the hash table is always filled up close to its limit, one often has to iterate several times in order to find a free spot due to the collisions. In contrast, in the symbolic stage, due to overestimation of the hash table size, the new *while* loop does not give a significant speedup since the hash table is sparse enough and finding a free spot is easier. Secondly, inside the innermost *while* loop there are only *if* and *else* clauses and the *atomicAdd* construct is taken outside the hash algorithm because all keys must be updated anyway. The original and enhanced procedure are provided in Algorithm 10 and 11, respectively. In these algorithms the hash tables for the column indeces and the values are defined as $table\_key$ and $table\_val$, respectively. They have both size t_{size}, the former is initialized with -1 while the latter with 0. The variable $HASH\_SCAL$ is just a constant number.

Algorithm 10 **Original nsparse hash operation**

```
1:  procedure HASH_ORIGINAL($table\_key[t_{size}] = \{-1\}$, $table\_val[t_{size}] = \{0.\}$, $key$, $val$)
2:      $hash = (key * HASH\_SCAL)\ \&\ (t_{size} - 1)$;
3:      while (true) do
4:          if ($table\_key = key$) then
5:              atomicAdd($table\_val[hash],\ val$);
6:              break;
7:          else if ($table\_key = -1$) then
8:              $old \leftarrow$ atomicCAS($table\_key[hash]$, $-1$, $key$);
9:              if ($old\ =\ -1$ then
10:                 atomicAdd($table\_val[hash],\ val$));
11:                 break;
12:             end if
13:         else
14:             $hash = (hash + 1)\ \&\ (t_{size} - 1)$;
15:         end if
16:     end while
17: end procedure
```

Most of the times, in AMG the matrices involved in a MxM product are sparse with the resulting matrix being sparse as well. In such a case, the numeric part resorts to the use of the smaller bins, i.e. those that are tailored for very sparse rows of the output matrix. The original nsparse implementation included a *pwarp* kernel used for rows whose number of nonzeros was less than or equal to 16 with each row processed

Algorithm 11 **Enhanced nsparse hash operation**

```
procedure HASH_ENHANCED($table\_key[t_{size}] = \{-1\}$, $table\_val[t_{size}] = \{0.\}$, $key$, $val$)
    $hash = (key * HASH\_SCAL)\ \&\ (t_{size} - 1)$;
    if ($table\_key[hash]\ != key$) then
        while ($table\_key[hash]\ != key$ and $table\_key[hash]\ != -1$) do
            $hash = (hash + 1)\ \&\ (t_{size} - 1)$;
        end while
        if ($table\_key[hash]\ != key$) then
            while (true) do
                $old \leftarrow$ atomicCAS($table\_key + hash$, $-1$, $key$);
                if ($old = -1$ or $old = key$) then
                    break;
                else
                    $hash = (hash + 1)\ \&\ (t_{size} - 1)$;
                end if
            end while
        end if
    end if
    atomicAdd($table\_val[hash]$, $val$);
end procedure
```

by 16 threads (half *warp*). The next bin was intended to be in charge of rows with up to 256 nonzeros. This was a very coarse bin splitting. Few new bins are added with a maximum count of nonzero ranging from 16 up to 256 (in powers of 2) that use the pwarp kernel. It does not result in an excessive use of shared memory while providing a higher parallelism. Additionally, in the new pwarp kernel each partial warp processes in parallel several columns of the first operand instead of the rows of the second one providing a better coalesced memory access pattern in the inner loop.

Several new kernels have been added to manage the cases in which the rows of the resulting matrix do not fit the shared hash table. In particular, there is a new *chunk* kernel in which the rows of the output matrix are split into chunks of equal size that fit the shared memory. The pointer to the last visited entry is saved to quickly jump to the end of the last visited chunk. The hash table is completely removed and replaced with a bitwise-or insertion of keys and atomic addition of values. Each chunk is compacted efficiently and dumped to the output array without sorting it. On top of that, the local data of the two operands is cached and reused for the subsequent chunks resulting in a better performance. This is by far more advantageous than using the (slow) GPU global memory as in the original implemenation of nsparse.

When the matrix has a quite regular number of non-zero elements per row or the largest bin dominates over the smaller bins, it is advisable to use just one bin. Since in this case rows are not permuted, it is possible to take advantage of data locality resulting in a performance boost. In this new version of nsparse, a mechanism to handle such cases is introduced: if the largest bin has more than 15% of the rows, than only this bin is used without permuting the rows. From practical standpoint, this situation is quite common with many matrices having a regular row size.

3.4.2 Computation scheme of the RAP operation

The MxM product is used in the computation of the coarse matrix A_c (2.6) that is also referred to as RAP product. RAP is the acronym for the double matrix product between *Restriction* R, system matrix A and *Prolongation* P, with $R = P^T$ since only SPD matrices are considered in this work. The RAP operation can be perfomed in two ways:

- $R(AP)$ where at first AP is computed and then the result is multiplied by R from the left;
- $(RA)P$ where at first RA is computed and then the result is multiplied by P from the right.

Mathematically, these two operations are clearly equivalent. Moreover, in most of the CPU-only algorithms the run times of the double matrix product are almost the same independently of the order of the operations. In fact the amount of floating point operations is identical and the parallelization is only done over the rows of the output matrix (which is the most common scheme on CPU for this kind of operations). However, with GPU accelerators the situation is quite different. This topic has just been touched by Naumov et al. [2015]; Liu and Vinter [2015] in the following a detailed discussion in the context of nsparse-algorithm is provided.

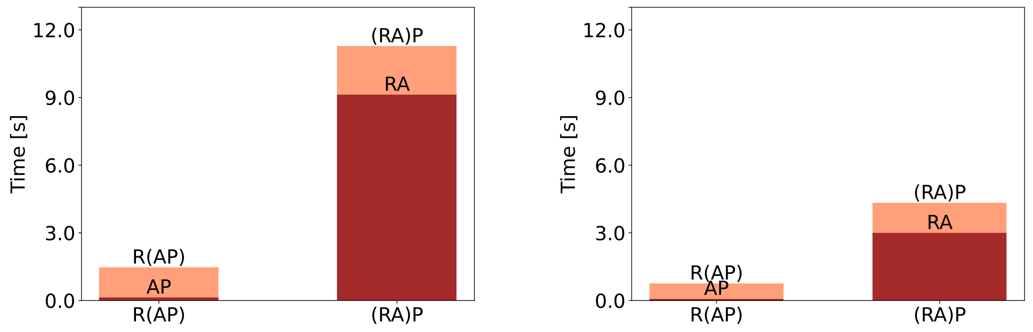
The nsparse-like algorithms on GPU execute $R(AP)$ much faster than $(RA)P$, see Tables 3.1 and 3.2 and Figure 3.7. In order to understand the reason, let's first compare the double matrix product stages AP with RA and $R(AP)$ with $(RA)P$. Since the matrix A is symmetric and $R = P^T$ (it is enough that A has a symmetric pattern and the pattern of R is the same as the pattern of P^T) we have that $AP = (RA)^T$. Therefore, it is easy to compare these two products in terms of performance. Similarly, the resulting matrices of $R(AP)$ and $(RA)P$ are the same and can be compared. Consequently, understanding the performance differences between these two stages will answer the question of why $R(AP)$ is faster than $(RA)P$ using nsparse-like algorithms on GPU.

operation	nsparse Time [s]	
	original	enhanced
AP	0.13	0.07
$R(AP)$	1.33	0.68
Total	1.46	0.75

TABLE 3.1: Execution times of AP and $R(AP)$ products for both original and enhanced nsparse.

operation	nsparse Time [s]	
	original	enhanced
RA	9.12	2.99
$(RA)P$	2.15	1.33
Total	11.27	4.32

TABLE 3.2: Execution times of RA and $(RA)P$ products for both original and enhanced nsparse.



(A) original nsparse (B) enhanced nsparse

FIGURE 3.7: Execution times of $R(AP)$ and $(RA)P$ products for both original and enhanced nsparse.

The matrices used in this test arise from a standard discretization of a small 3D mechanical problem (so that it can be handled by a single GPU, communications in fact playing a minor role), and RAP refers to the first level of the grid hierarchy in the AMG framework. The information related to the matrices involved in the RAP operation is presented in Table 3.3.

	A(left factor)				B (right factor)				C(product)		
	n	m	nnz	nnz/n	n	m	nnz	nnz/n	n	m	nnz
AP	739k	739k	30M	40	739k	33k	20M	27	739k	33k	50M
$R(A)$	33k	739k	20M	615	739k	33k	50M	68	33k	33k	15M
RA	33k	739k	20M	615	739k	739k	30M	40	33k	739k	50M
$(RA)P$	33k	739k	50M	1,536	739k	33k	20M	27	33k	33k	15M

TABLE 3.3: $R(AP)$ vs $(RA)P$ matrix information. In the matrix product, A refers to the left matrix, B refers to the right matrix, and C is the output matrix. For each matrix, the following information is provided: the number of rows, n, the number of columns, m, the number of nonzeros, nnz and the average number of nonzeros per row, nnz/n.

Performing AP and RA products produces a skinny and a fat matrix, respectively. In AP the output has many rows while in RA, the result has many columns. Even though the two resulting matrices, AP and RA, are transposes of each other(remember $R \equiv R^T$), AP is more efficiently utilizing the GPU. This is due to the fact that AP has a higher degree of parallelism with respect to the RA product since the resulting matrix of AP has few nonzeros per row ≈ 68 while RA has $\approx 1,536$. Therefore, it uses smaller bins that have a higher amount of parallelism due to an increase in the number of concurrent thread blocks residing on each streaming multiprocessor, thus providing a better usage of GPU resources. Despite the fact that the RA product, instead, has a much smaller amount of rows of the output matrix, these rows are much denser than that of the output of the AP product. Therefore, in this case, the hash table load is higher and bigger bins are used in the symbolic and numeric stages of *nsparse*. Moreover, from the algorithmic point of view, the compaction of the rows via the hash table is the most computationally intensive part, so lowering this intensity by diluting the hash table compaction over many rows is a more advantageous strategy. As a consequence, when the output is a long and skinny matrix it is preferred to have many rows instead of many columns.

The situation with the second stage matrix product of $R(AP)$ vs $(RA)P$ is more interesting since in this case the output matrices are identical, therefore, they have the same bin distribution, and the input matrices have the same dimensions. The only

Algorithm 12 **nsparse-like algorithms for** $AB = C$

```
1: $n \leftarrow A^{n \times m}$
2: for $i$ in $n$ do                                   ▷ loop over CUDA blocks
3:     for $a_{ij}$ in $A_{i*}$ do                     ▷ loop over block warps
4:         for $b_{jk}$ in $B_{a_{ij},*}$ do           ▷ loop over warp lanes
5:             $c_{ik} \leftarrow a_{ij} \cdot b_{jk}$ ▷ update the hash table
6:         end for
7:     end for
8: end for
```

thing that is different is the number of nonzeros per row of the input matrices. Consider the product A by B, the nsparse-like algorithms in exploit the so-called fined-grain parallelism over the lanes of a warp in the inner loop of the Algorithm 12 at the expense of parallelization over the warps (middle loop). This is due to the fact that the inner loop has a coalesced reading while the reading of the nonzeros of A is done in strides. Therefore, it is preferred to have a smaller number of nonzeros of A instead of B. Moreover, under the hypothesis of equal bin distribution and an equal amount of floating point operations when comparing the two matrix products, having many nonzeros per row of A will result in the merging over many corresponding rows of B which entails more synchronizations and collisions among the warps instead of lanes. Since the merging of rows over the warps is more expensive, having significantly more nonzeros per row in (RA) than of R, it results in a slowdown of the second stage of the double product of $(RA)P$ with respect to $R(AP)$.

Chapter 4

Numerical results

This Chapter collects all the numerical experiments performed to validate and test the Chronos package. The numerical experiments have been performed using large sparse matrices arising from challenging real-world problems. In particular, the Chronos AMG is benchmarked on a set of problems that can be grouped into two classes denoted as Fluid dynamic (F) and Mechanical (M). The first one consists of a series of problems arising from the discretization of the Laplace operator related to fluid dynamic problems, such as underground fluid flow (reservoir), compressible or incompressible airflow around complex geometries (CFD) or porous flow (porous flow). The second category includes problems related to mechanical applications such as subsidence analysis, hydrocarbon recovery, gas storage (geomechanics), mesoscale simulation of composite materials (mesoscale), mechanical deformation of human tissues or organs subjected to medical interventions (biomedicine), design and analysis of mechanical elements, e.g., cutters, gears, air-coolers (mechanical).

In the experiments, challenging test cases are considered, not only for the high number of degrees of freedom (DOFs), but also because of their intrinsic ill-conditioning. Indeed, in real applications, engineers usually have to deal with large jump of the physical proprieties, complicated geometries leading to highly distorted elements, heterogeneity and anisotropy. The matrices considered in the experiments are listed in Table 4.1 with details about the size, the number of non-zeros and the application field they arise from.

All these matrices are Symmetric Positive Definite (SPD), therefore, the preconditioned Conjugate Gradient (PCG) is used to solve the linear systems. In particular,

Matrix	Class	n	nnz	avg. nnz/row	Application field
`spe10`	F	3,410,693	90,568,237	26.55	3D porous flow
`geo4m`	M	4,224,870	335,738,340	79.47	3D geomechanics
`wing4m`	M	4,538,205	187,714,431	41.36	3D mechanical
`finger4m`	F	4,718,592	23,591,424	5.00	2D porous flow
`worm8m`	M	8,215,599	652,063,779	79.37	3D mechanical
`guenda11m`	M	11,452,398	512,484,300	44.75	3D geomechanics
`M10`	M	11,593,008	940,598,090	81.13	3D mechanical
`agg14m`	M	14,106,408	633,142,730	44.88	3D mesoscale
`M20`	M	20,056,050	1,634,926,088	81.52	3D mechanical
`tripod24m`	M	24,186,993	1,111,751,217	45.96	3D mechanical
`rtanis44m`	F	44,798,517	747,633,815	16.69	3D porous flow
`geo61m`	M	61,813,395	4,966,380,225	80.34	3D geomechanics
`poi65m`	F	65,939,264	460,595,552	6.99	3D CFD
`Pflow73m`	F	73,623,733	2,201,828,891	29.91	3D reservoir
`poi111m`	F	111,980,168	782,234,908	6.99	3D CFD
`c4zz134m`	M	134,395,551	10,806,265,323	80.41	3D biomedicine
`poi198m`	F	198,076,032	1,384,390,392	6.99	3D CFD

TABLE 4.1: Benchmark matrices used in the numerical experiments. For each matrix, the class, the size, n, the number of non-zeros, nnz, the average number of non-zeroes per row and the application field are provided.

the right-hand side vector is a random vector, the initial solution is zeroes and convergence is considered achieved when the $l2$-norm of the relative iterative residual becomes smaller than $10^{-8} \cdot ||b||$.

Chronos performance has been evaluated on Marconi100, a supercomputer hosted in the Italian consortium for supercomputing (CINECA). Marconi100, classified within the first ten positions of the TOP500 ranking [Strohmaier et al., 2023] at the time this manuscript was written (from July 2023 it has been replaced by Leonardo supercomputer), was composed by 980 nodes based on the IBM Power9 architecture, each equipped with two 16-cores IBM POWER9 AC922 at 3.1 GHz processors and four NVIDIA Volta V100 GPUs with Nvlink 2.0 and 16GB of memory.

The Chapter is structured as follow: first, the focus is on the CPU-only version of the AMG that is compared to other state-of-the-art solvers. These CPU-only runs draw a baseline for the GPU experiments. Specifically, all the main kernels described in Chapter 3, SpMV product, aFSAI smoother, BAMG prolongation and MxM product, are investigated individually and then the accelerated AMG as a whole is analysed. Finally, some specific applications of the Chronos package are discussed.

4.1 Performance of the CPU-only version

This section focuses on the performance of the CPU-only version and its comparison with other stat-of-the-art solvers. In particular, the discussion of the results is subdivide into two parts, the former collecting test cases from fluid dynamics and the latter from mechanics. Also strong and weak scalability analysis of the proposed implementation is provided, using large scale computational resources. The results are presented in terms of total number of computational cores used n_{cr}, the grid and operator complexities, C_{gd} and C_{op}, respectively, the number of iteration, n_{it} and the set-up, iteration and total times, T_p, T_s and $T_t = T_p + T_s$, respectively. For each test, the number of cores, n_{cr}, is selected to have a per core load of about 100-150,000 dofs and, consequently, different numbers of nodes are allocated for different problem dimensions. For all the tests, each node reserved for the run is always fully exploited by using 8 MPI tasks and 4 OpenMP threads for each task.

As a reference point to evaluate the performance of Chronos, we compare it with the state-of-the-art solvers BoomerAMG, a classical AMG, and GAMG, a smoothed aggregation-based AMG, as preconditioners in fluid dynamics and mechanical problems, respectively. BoomerAMG and GAMG have been chosen as baseline solvers because they are very well known and open-source packages whose performance have been demonstrated in several papers [Balay et al., 2023; Brezina et al., 2006b; Falgout and Yang, 2002; Henson and Yang, 2002].

4.1.1 Fluid dynamics test cases

The AMG implemented in Chronos is highly tunable, offering several set-up options to effectively solve a wide set of problems as it will be shown below.

First, let's start by comparing Chronos and BoomerAMG performance using a setup that is as similar as possible. Such comparison is intended to verify the HPC implementation and to demonstrate the efficiency of the DSMat storage scheme for SpMV product, considering the three test cases `finger4m`, `poi65m` and `Pflow73m`. The comparison takes place with the same preconditioner configuration, i.e., Jacobi smoothing, classical SoC with $\theta = 0.25$, PMIS coarsening and Extended+i prolongation. The

Matrix	n_{cr}	Solv. type	C_{gd}	C_{op}	n_{it}	T_p [s]	T_s [s]	T_t [s]
`finger4m`	32	Chr-jac	1.453	2.558	16	1.13	0.55	1.68
	32	Boomer-jac	1.454	2.574	16	0.81	0.70	1.51
`poi65m`	384	Chr-jac	1.327	4.036	16	3.81	1.65	4.46
	384	Boomer-jac	1.361	4.450	13	5.70	2.03	7.73
`Pflow73m`	480	Chr-jac	1.125	1.614	3308	14.1	611.9	626.0
	480	Boomer-jac	1.123	1.593	3576	26.1	771.7	797.8

TABLE 4.2: Solution of three fluid dynamic test cases among those reported in Table 4.1 using Jacobi smoothing and Extended+i prolongation. For each run, the following information is provided: the number of cores n_{cr}, the grid C_{gd} and operator C_{op} complexities, the number of PCG iteration n_{it}, the set-up time T_p, the iteration time T_s and the total time T_t.

only exception takes place for `Pflow73m` where the strength of connection threshold is taken as $\theta = 0.0$, that significantly increases performance.

Table 4.2 provides for each test case the results obtained with this *standard set-ups* that is denoted as Chr-jac and Boomer-jac. The grid and operator complexities with the two software are basically the same and also the iteration count turns out to be quite similar, showing that the two implementations are consistent. Only a slight difference occurs for `Pflow73m` but it is compatible with very small differences in the code implementation.

Figure 4.1 provides the time for the preconditioner set-up (left) and for PCG (right) for each solving strategy. Each time reported in the figure is normalized with respect to the corresponding Boomer-jac time, which is the baseline in these experiments. Observe that, while using Jacobi smoothing, Chronos is faster than BoomerAMG in the set-up for `poi65m` and `Pflow73m`, while BoomerAMG is better in `finger4m`. Differently, as to the iteration time, Chronos slightly outperforms BoomerAMG in all the tests. In all cases, the Chronos implementation turns out to be very efficient and the total solution time is comparable and sometimes even better than that of the BoomerAMG.

In Table 4.3, the labels Chr and Boomer identify the results obtained with Chronos and BoomerAMG when the default smoothers are selected, i.e., aFSAI and hybrid Gauss-Seidel, respectively. The use of a more elaborated/sophisticated smoother with respect to either Jacobi or hybrid Gauss-Seidel gives a significant advantage in terms of

Matrix	n_{cr}	Solv. type	C_{gd}	C_{op}	n_{it}	T_p [s]	T_s [s]	T_t [s]
`finger4m`	32	Chr	1.453	2.558	7	3.71	0.33	4.04
	32	Boomer	1.454	2.574	12	0.79	0.94	1.73
`poi65m`	384	Chr	1.346	4.496	6	27.5	1.84	29.34
	384	Boomer	1.361	4.450	14	5.88	3.18	9.06
`Pflow73m`	480	Chr	1.125	1.614	240	46.5	64.2	110.7
	480	Boomer	1.123	1.593	2777	26.5	1042.3	1068.8

TABLE 4.3: Solution of three fluid dynamic test cases among those reported in Table 4.1 using default smoothers and Extended+i prolongation. For each run, the following information is provided: the number of cores n_{cr}, the grid C_{gd} and operator C_{op} complexities, the number of PCG iteration n_{it}, the set-up time T_p, the iteration time T_s and the total time T_t.

iteration count and solving time at the price of a more expensive set-up, as shown also on Figure 4.1. The use of aFSAI always allows for achieving a faster convergence. Furthermore, the more ill-conditioned the problem is, the better aFSAI compares with other smoothers. In `Pflow73m`, which is the hardest problem in fluid dynamics, Chronos with aFSAI smoothing is 6 times faster than BoomerAMG. The set-up time is larger, but the speed-up obtained in the iteration stage may justify this effort, especially in transient simulations where the user may have to solve repeatedly the same linear system and can take advantage of preconditioner recycling.

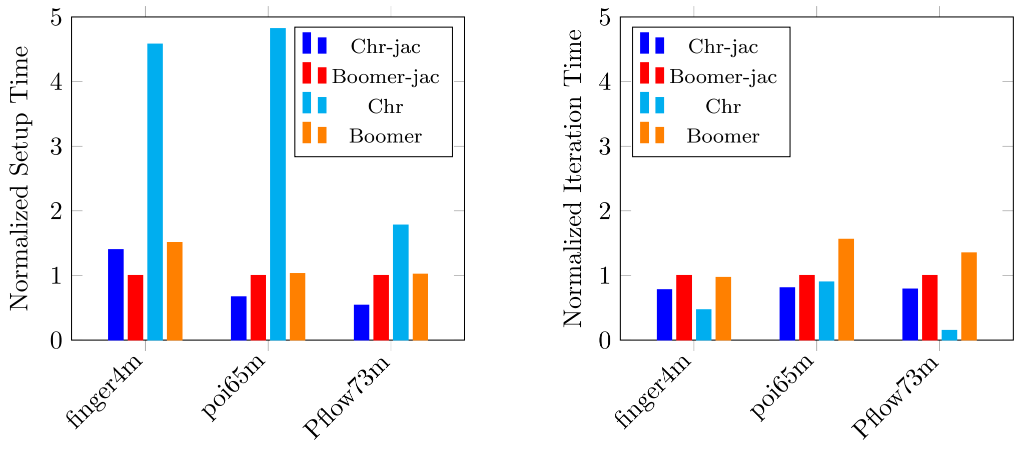


FIGURE 4.1: Comparison between Chronos and BoomerAMG by using the Extended+i prolongation and Jacobi or default smoothing. Left: normalized T_p to the Boomer-jac solution. Right: normalized T_s to the Boomer-jac solution.

Matrix	n_{cr}	Prol. type	C_{gd}	C_{op}	n_{it}	T_p [s]	T_s [s]	T_t [s]
finger4m	32	Chr-clas	1.467	1.871	31	3.57	1.32	4.89
	32	Chr-hybc	1.465	2.051	14	3.62	0.66	4.28
	32	Chr-exti	1.453	2.558	7	3.71	0.33	4.04
rtanis44m	384	Chr-clas	1.612	1.943	46	23.3	6.47	29.8
	384	Chr-hybc	1.585	2.030	36	26.6	6.09	32.8
	384	Chr-exti	1.572	2.580	16	34.0	2.90	36.9
poi65m	384	Chr-clas	1.381	2.339	21	17.9	4.69	22.59
	384	Chr-hybc	1.361	2.888	13	19.0	2.57	21.57
	384	Chr-exti	1.346	4.496	6	27.5	1.84	29.34
Pflow73m	480	Chr-clas	1.236	1.391	414	39.4	60.2	99.6
	480	Chr-hybc	1.234	1.448	416	40.4	67.1	107.5
	480	Chr-exti	1.234	2.346	410	57.7	120.9	178.6

TABLE 4.4: Comparison between different interpolation formulas in the solution of the fluid dynamic test problems from Table 4.1. For each run, the following information is provided: number of cores n_{cr}, prolongation type, grid C_{gd} and operator C_{op} complexities, number of iteration n_{it}, set-up time T_p, iteration time T_s and total time T_t.

In our fluid dynamics examples, the prolongations of choice for classical AMG are typically the classical or Extended+i interpolations. The latter is usually more effective, although more expensive, for challenging problems due to its ability to accurately interpolate fine nodes having strong fine neighbors that do not share the same strong coarse node, possibly produced by high coarsening ratios. In Table 4.4, these two well-known prolongations are compared to the hybrid one that has been discussed in Chapter 2. For `finger4m` and `poi65m`, observe that the Extended+i interpolation is the more accurate one, with higher operator complexity. As expected, this leads to a lower number of iterations, but a higher computational cost per iteration. On the other hand, the classical interpolation formula is the cheapest to compute, with very low operator complexity. However, taking into account only distance-one coarse nodes, the prolongation operator is not able to accurately reproduce the smooth error, causing an increase of the iteration count, up to twice the iteration count obtained with Extended+i. For these two tests, the best configuration lies in the middle, i.e., the hybrid interpolation formula, which keeps the operator complexity small by only taking distance-two coarse nodes that are actually useful in the interpolation process. In this way, Chronos AMG is able to obtain a more accurate interpolation formula with a computational cost comparable to the classical one.

The behavior is quite different for the other two test cases. In `rtanis44m`, a strong heterogeneity and anisotropy of permeability tensor dramatically increase the conditioning of the problem. Hence, the most accurate interpolation method, i.e., Extended+i, is needed to efficiently solve this problem. The iteration count is one third with respect to classical interpolation and the solution time is approximately one half. Unlike before, the increased accuracy of the hybrid interpolation over classical is not enough to give a sufficient benefit in terms of solving time. It is worth noting that the increased set-up cost for Extended+i is in this case largely compensated in the iteration stage. This gain is even more pronounced in cases where preconditioner recycling is possible such as in some transient or non-linear simulations.

The last test case considered in this section is `Pflow73m`, a very challenging and severely ill-conditioned problem from underground flow. Even if this is a diffusion problem, the great jumps in permeability and the distorted mesh lead to a matrix whose near-kernel is not well represented by the constant vector. For this reason, the iterations to converge are much larger than in the other tests and not even the most accurate interpolations such as Extended+i or hybrid give any benefit over classical interpolation, which, being the cheapest one, proves the most effective strategy for this test case.

4.1.2 Mechanical test cases

As seen above, Chronos allows for setting-up a very flexible AMG preconditioner, adaptable to problem types the user has to solve, with different choices available for interpolation operators and smoothing methods. In addition to different available choices for interpolation and smoothing, Chronos allows the possibility to directly smooth the prolongation and/or filter it. Moreover, for mechanical test problems, it is very helpful keeping low the grid complexity, especially in case of prolongation smoothing. This is easily achieved with default settings by dropping only a very small number of connections in the SoC graph.

As in the previous paragraph, first a baseline is defined with state-of-the-art methods such as BoomerAMG (Boomer), with Hybrid Gauss-Seidel smoothing, the unknown-based Boomer with separate treatment of unknowns relative to different directions (unk-based-Boomer) and the GAMG, a smoothed aggregation-based method. Let's

Matrix	n_{cr}	Prec. type	C_{gd}	C_{op}	n_{it}	T_p [s]	T_s [s]	T_t [s]
`tripod24m`	160	Boomer	1.244	3.207	931	64.1	611.9	676.1
		unk-based-Boomer	1.328	3.669	335	43.8	262.4	203.5
		GAMG	1.543	-	294	12.1	80.5	92.6
		BAMG-aFSAI	1.041	1.116	222	21.8	23.0	44.8
		SBAMG-aFSAI	1.041	1.322	118	36.7	16.1	52.9
		FBAMG-aFSAI	1.041	1.212	120	33.5	13.5	47.0

TABLE 4.5: Solution of the `tripod24m` test case from Table 4.1 with different approaches. For each run, the following information is provided: the number of cores n_{cr}, the preconditioner type, the grid complexity C_{gd}, the operator complexity C_{op}, the number of iteration n_{it}, the set-up time T_p, the iteration time T_s, and the total time T_t.

first refer to the test case `tripod24m`, whose results are provided in Table 4.5. With the standard Boomer, the solution is reached with a high number of iterations, more than 900 and the iteration time affects the total time the most. A significant improvement is obtained using the unknown-based version [Baker et al., 2009], where iterations are reduced by one third, and set-up and iteration times drop by 50%. The aggregation based AMG seems to be the most effective one for mechanical problems as, with GAMG, iterations are further reduced, and both T_p and T_s times decrease significantly. In this problem, Chronos with BAMG prolongation and aFSAI smoother (BAMG-aFSAI) is more effective than GAMG with a speed-up of two on the total time. The set-up time is larger, but the number of PCG iterations is lower and the cost per iteration is one third that of GAMG. It is also possible to smooth the prolongation operator with a Jacobi step. We denote this method as SBAMG-aFSAI. As could be expected, the operator complexity and the set-up time both increase but, on the other hand, the number of iterations and solution time are smaller. The increase of operator complexity and set-up time can be limited by means of filtering (FBAMG-aFSAI) without compromising effectiveness. FBAMG-aFSAI requires the same number of iterations to converge but at a lower cost per iteration. These two last strategies are particularly effective in a FEM simulation where the preconditioner can be reused several times in different time-steps so that the set-up cost becomes negligible.

Chronos proved to be robust and efficient in addressing all the mechanical test cases. A comparison of the number of iterations and times obtained with GAMG and the three BAMG strategies outlined above is shown in Table 4.6. To highlight the speed-up, Figure 4.2 shows set-up and the iteration time normalized to the GAMG times. Unfortunately, the comparison for the two largest cases, `geo61m` and `c4zz13m`, is not reported because these matrices have not been dumped on file due to their large size, and the tests have been run by linking Chronos to the FEM simulator ATLAS [Atlas Project Team, 2023]. For the three benchmarks, `guenda11m`, `tripod24m` and `M20`, the number of PCG iterations required by GAMG and BAMG is comparable, but the overall solution time is significantly lower for BAMG with a speed-up of Chronos over GAMG up to 4 in these tests. The only exception is the matrix `agg14m` where GAMG is able to produce a very effective preconditioner at the lowest set-up cost.

Finally, it can be observed that, through a proper set-up, Chronos is able to produce total solution times that depends only mildly on the problem nature but on its size only. Figure 4.3 shows for each problem the total solution time divided by the number of non-zeroes per allocated core, and this resulting time is further normalized with the average among all the experiments. In other words, the figure should show the solution time for each problem as if "exactly" the same resources were allocated for each non-zero. For a preconditioner that is totally independent of the problem nature, the same solution time for every problem is expected. Note that total solution times obtained with Chronos are very close to the average normalized solution time, thus showing only a mild dependence on the application at hand.

4.1.3 Strong and weak scalability

In this section, the strong and weak scalability of the AMG preconditioners implemented in Chronos are evaluated. All three times, i.e., set-up T_p, iteration T_s and total T_t times, are analyzed to assess scalability. The strong scalability test is shown in the top of Figure 4.4, on the left for `poi65m` with Extended+i prolongation and on the right for the `c4zz134m` test matrix with BAMG prolongation. The number of cores varies from the minimum necessary to store matrix and preconditioner up to 8 times. In both tests, the times decrease as the computing resources increase, with a trend close to the ideal one. Finally, the weak scaling is investigated with a standard 7-point finite difference discretization of the Poisson problem. Figure 4.4, bottom, shows, both the total time spent in the set-up and solve phase, on the left, and corresponding parallel

Matrix	n_{cr}	Prol. type	C_{gd}	C_{op}	n_{it}	T_p [s]	T_s [s]	T_t [s]
	64	GAMG	1.580	-	978	18.3	306.2	324.5
`guenda11m`	64	BAMG	1.041	1.118	937	27.8	105.0	133.0
	64	SBAMG	1.041	1.354	638	50.3	96.3	147.0
	64	FBAMG	1.041	1.240	638	43.5	79.8	123.0
	128	GAMG	1.644	-	26	12.5	5.8	18.2
`agg14m`	128	BAMG	1.085	1.287	135	30.6	22.2	52.8
	128	SBAMG	1.085	2.264	31	114.4	8.1	122.6
	128	FBAMG	1.085	1.670	34	53.6	7.3	60.9
	128	GAMG	1.162	-	245	211.0	391.4	602.4
`M20`	128	BAMG	1.054	1.184	775	71.2	275.0	347.0
	128	SBAMG	1.054	1.677	151	158.0	71.2	229.2
	128	FBAMG	1.054	1.292	158	93.9	55.1	149.1
	160	GAMG	1.543	-	294	12.1	80.5	92.6
`tripod24m`	160	BAMG	1.041	1.116	222	21.8	23.0	44.8
	160	SBAMG	1.041	1.322	118	36.7	16.1	52.9
	160	FBAMG	1.041	1.212	120	33.5	13.5	47.0

TABLE 4.6: Comparison between different interpolation formula in the solution of the mechanical test problems from Table 4.1. For each run, the following information is provided: number of cores n_{cr}, prolongation type, grid C_{gd} and operator C_{op} complexities, number of iteration n_{it}, set-up time T_p, iteration time T_s and total time T_t.

efficiencies, on the right. Weak scaling efficiency up to N nodes is defined as

$$E = T_1/(N \cdot T_N) \tag{4.1}$$

with T_1 the time required on a single node and T_N the time on N nodes. In this testa about 218,750 unknowns per core have been assigned.

The results show that efficiency is very good and stays almost constant in the first two doubles of the cores, whereas a smaller efficiency occurs in the last one. This performance dropdown can be ascribed to two distinct factors. First of all, while Marconi100 cores can be fully reserved for the test runs, the overall network is always shared with other users, and, consequently, the larger the resource allocation, the larger the disturbance from other running processes. Secondly, a performance dropdown is almost unavoidable in AMG methods, as the grid hierarchy ends always up with small grids. The larger the amount of resources allocated, the less efficient will be the software in dealing lower levels. Currently, to ease the implementation, Chronos uses all the allocated cores on each grid except the last one, where an `allgather` operation is

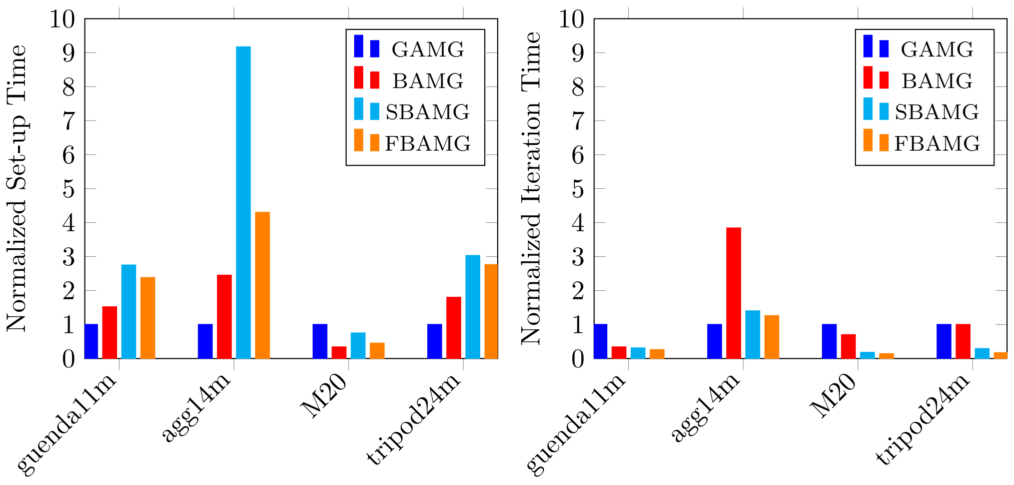


FIGURE 4.2: Comparison between GAMG and the BAMG strategies on the mechanical test cases. Left: normalized T_p to the GAMG solution. Right: normalized T_s to the GAMG solution.

called from a single core to solve the coarsest problem. In a future implementation, it is planned to progressively reduce the amount of resources with levels, thus reducing the network traffic and increasing efficiency.

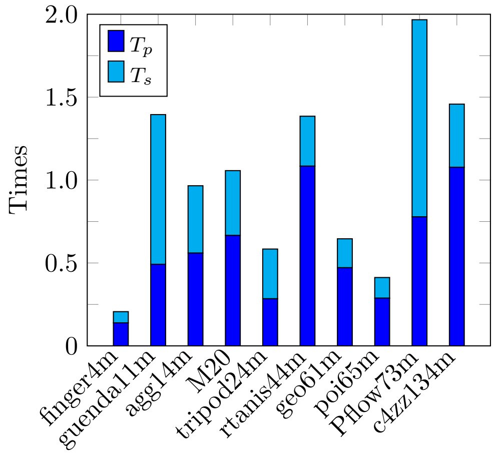


FIGURE 4.3: Set-up and iteration times, for all benchmark problems, normalized over the resources allocated per non-zero.

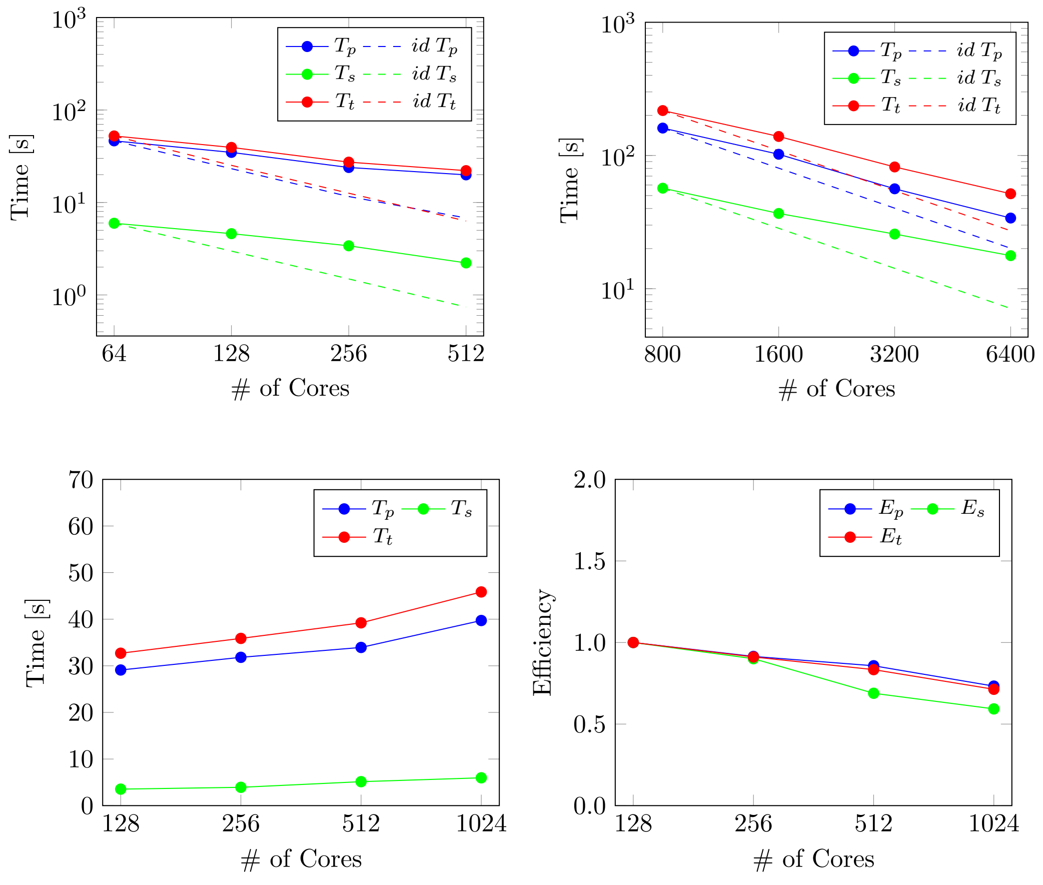


FIGURE 4.4: Strong scalability test for `poi65m` matrix and Extended+i prolongation (top-left) and `c4zz134m` matrix and BAMG prolongation (top-right). Weak scalability test on a standard 7-point finite difference discretization of the Poisson problem. Set-up, iteration and total times vs. n_{cr} on bottom-left, corresponding efficiencies vs. n_{cr} on bottom-right.

4.2 Performance of the GPU-accelerated version

This section focuses on the GPU-accelerated version of the Chronos AMG. Previously presented comparisons with other linear solvers have shown that the CPU-only implementation of Chronos is a state-of-the-art one. Therfore, the following comparisons will be made primarily between CPU-only and GPU-accelerated Chronos versions, taking the CPU-only as a baseline.

Firts, the discussion addresses individually all the main kernels described in Chapter 3: SpMV product, aFSAI smoother, BAMG prolongation and MxM product; then, the accelerated AMG as a whole is analysed. Finally, some specific applications of the Chronos package are discussed.

4.2.1 Effectiveness of the DSMat storage scheme in the SpMV product

This section shows the effictiveness of the SpMV product in the GPU-accelerated runs. To this aim, let's first compare the time required by Chronos and the well-known open source package PETSc [Balay et al., 2023] to perform 100 iterations of CG preconditioned with Jacobi. The choice of Jacobi is dictated by the fact that, since its implementation is straightforward, these tests allow to directly evaluated the SpMV efficiency. Figures 4.5 to 4.7 show the comparison between PETSc and Chronos, in both CPU-only and GPU-accelerated mode, by increasing the computing resources from a subset of a node (8 CPUs and 1 GPU) to 4 full nodes (128 CPUs and 16 GPUs) on the test matrices `spe10`, `geo4m` and `agg14m`. The speed-up of the CPU version of Chronos (Chr-CPU) over PETSc (PETSc-CPU) is on average 2.39 and reaches a maximum value of about 3.15. As expected, smaller speed-up values (about 2) are obtained as the computing resources increase due to the communication overhead becoming significant on the overall run time. The GPU-accelerated modes of Chronos (Chr-GPU) and PETSc (PETSc-GPU) have the same performance, showing that the two implementations of the SpMV and CG are equivalent. With Chronos, the GPU acceleration leads to a speed-up over the CPU-only version of about 10 for `agg14m` and `geo4m` matrices whereas it reaches a smaller value $\simeq 6$ for `spe10`. This lower performance is easily explained by considering the lower number of non-zeroes per row characterizing `spe10`, which induces a worse operation over communication ratio. The use of GPU-accelerated Chronos allows an overall speed-up over CPU-only PETSc in the range from 15 to 25 on the performed test cases.

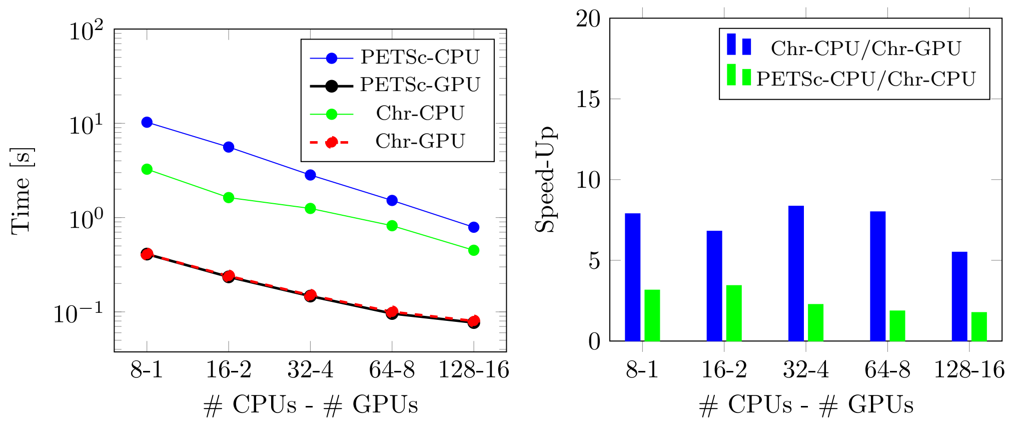


FIGURE 4.5: Scalability and Speed-Up of Jacobi preconditioned CG on the `spe10` matrix. Left: total wall time in seconds for the execution of 100 iterations of Jacobi preconditioned CG. Right: Speed-Up of GPU-accelerated Chronos (Chr-GPU) over pure CPU Chronos (Chr-CPU) (blue columns) and Speed-Up of Chronos over PETSc in pure CPU runs (green columns). The computing resources vary from a subset of a node (see text) to 4 Marconi100 nodes.

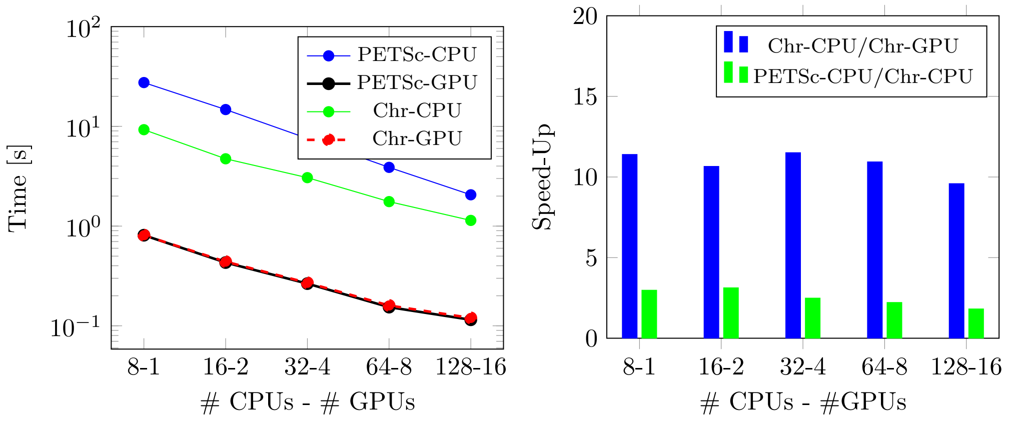


FIGURE 4.6: Scalability and Speed-Up of Jacobi preconditioned CG on the `geo4m` matrix. Left: total wall time in seconds for the execution of 100 iterations of Jacobi preconditioned CG. Right: Speed-Up of GPU-accelerated Chronos (Chr-GPU) over pure CPU Chronos (Chr-CPU) (blue columns) and Speed-Up of Chronos over PETSc in pure CPU runs (green columns). The computing resources vary from a subset of a node (see text) to 4 Marconi100 nodes.

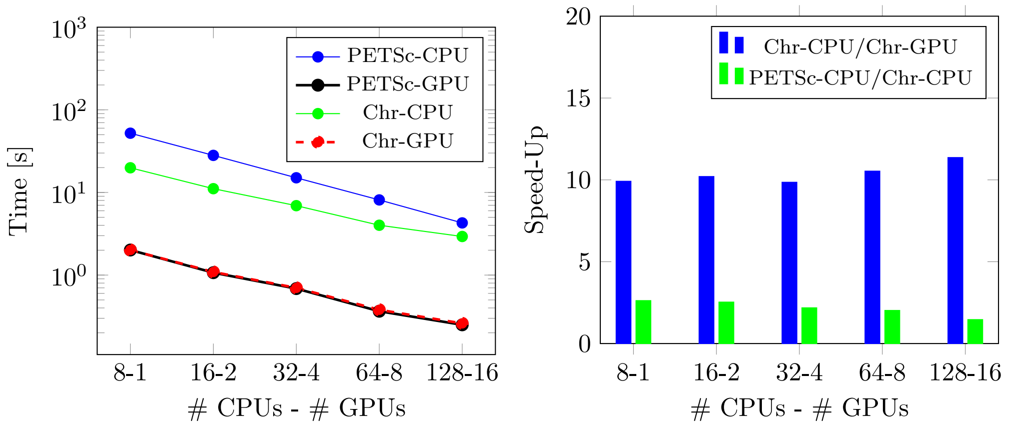


FIGURE 4.7: Scalability and Speed-Up of Jacobi preconditioned CG on the `agg14m` matrix. Left: total wall time in seconds for the execution of 100 iterations of Jacobi preconditioned CG. Right: Speed-Up of GPU-accelerated Chronos (Chr-GPU) over pure CPU Chronos (Chr-CPU) (blue columns) and Speed-Up of Chronos over PETSc in pure CPU runs (green columns). The computing resources vary from a subset of a node (see text) to 4 Marconi100 nodes.

4.2.2 Effectiveness of aFSAI

The multi-GPU implementation of the aFSAI is tested using it as a preconditioner itself. The aFSAI preconditioner, or smoother in the AMG context, is much more demanding than Jacobi in term of both implementation effort and set-up time. However, the adoption of aFSAI as preconditioner in real world problems is fully justified by its superior effectiveness in accelerating CG convergence. Figure 4.8 shows the comparison between the total solution time, including also the set-up time, and the number of iterations necessary to reduce the initial residual by 8 orders of magnitude using CG preconditioned with Jacobi and aFSAI. Only some of the smallest matrices of Table 4.1 have been considered, as their solution cost through Jacobi would have been prohibitive with the largest ones. The experiments were run using the GPU-accelerated Chronos library on a single Marconi100 node and clearly show the ability of aFSAI in reducing the number of iterations. In all the tests aFSAI outperfoms Jacobi by a factor of at least 2 in the worst case and up to 10 in the most ill-conditioned `geo4m`.

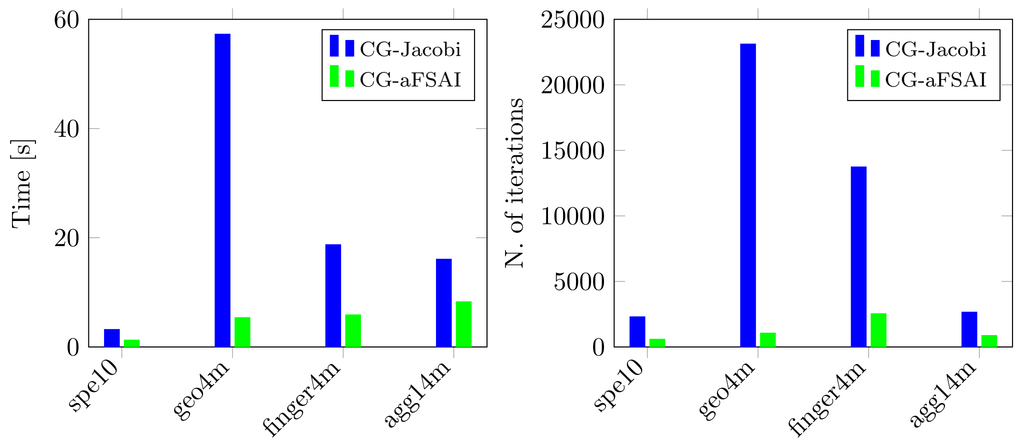


FIGURE 4.8: Comparison between the GPU-accelerated CG preconditioned with Jacobi and aFSAI using a single node of Marconi100. Left: Total solution time. Right: Number of iterations to converge.

Let's now focus on the strong and weak scalability of aFSAI-preconditioned CG. Three different times are considered to evaluate scalability: the preconditioner set-up time, T_p, the iteration time, T_s, and total time, $T_t = T_p + T_s$. The set-up time includes both the preliminary *gathering* and the *computation* stage on the GPU board, the latter dominating the set-up. Strong scalability tests have performed on some of the largest matrices of Table 4.1, by varying the number of GPUs from the minimum necessary to store the matrix and the preconditioner to a maximum of 512, corresponding to 128 nodes. Figures 4.9 to 4.12 provide execution times and parallel efficiency, ratio between real and ideal speed-up, vs the number of GPUs. In all tests both the set-up and solution time decrease inversely proportional to the computing resources with an almost ideal behaviour. The perfomrance of the set-up stage is closer to ideal as its communication has a lower impact on preconditioner set-up.

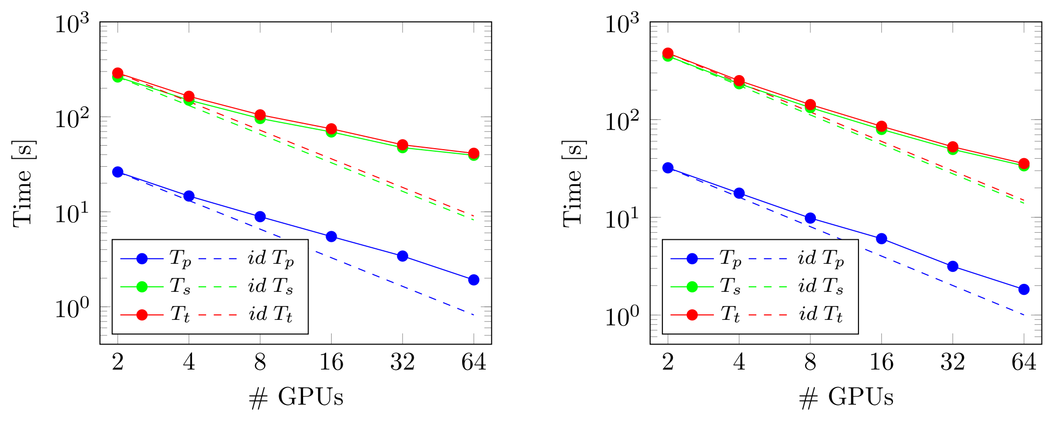


FIGURE 4.9: Strong scalability of aFSAI CG on matrices `guenda11m` (left) and `M10` (right). aFSAI set-up time T_p, CG iteration time T_s and total time $T_t = T_p + T_s$ vs. number of GPUs.

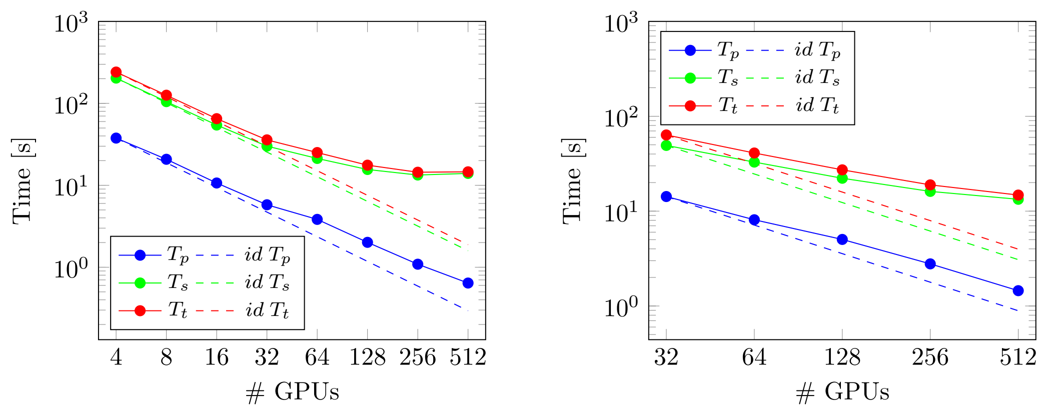


FIGURE 4.10: Strong scalability of aFSAI CG on matrices `M20` (left) and `geo61m` (right). aFSAI set-up time T_p, CG iteration time T_s and total time $T_t = T_p + T_s$ vs. number of GPUs.

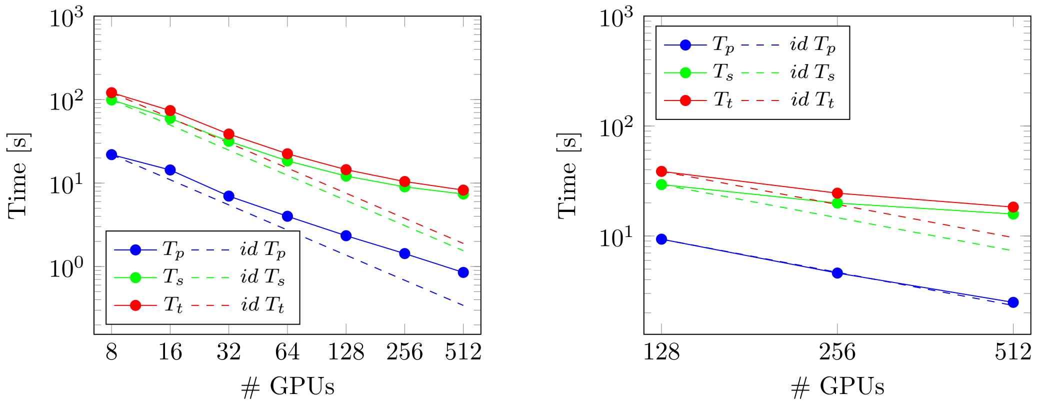


FIGURE 4.11: Strong scalability of aFSAI CG on matrices `Pflow73m` (left) and `c4zz134m` (right). aFSAI set-up time T_p, CG iteration time T_s and total time $T_t = T_p + T_s$ vs. number of GPUs.

Regarding the parallel efficiency, Figure 4.13 shows the maximum number of GPUs such that the efficiency is at least 50% for the matrices arising from structural mechanics. This particular subset of matrices have been analysed because all of them have more than 10 millions rows and have a similar number of non-zeroes per row ranging from 45 to 80. As expected, the computational resources that can be used at the same efficiency increase with the number of non-zeroes in the matrix.

The weak scalability of the implementation is tested using a 7-point stencil Finite Differences discretization of the Poisson problem on a cubic domain. In this experiment the number of equations is kept constant at 1,771,561 for each GPU, increasing the number of GPUs from 32 to 256. For larger problem sizes, the number of CG iterations increases (see Figure 4.14 upper-right), since aFSAI is not an optimal preconditioner, as for instance AMG. As a consequence, the iteration time as well as the total time correspondingly increase. However, focusing on the set-up stage and the time required by a single CG iteration (bottom-left and bottom right of Figure 4.14, respectively), it can be observed that they do not change or only slightly change when the problem size increases, showing an almost perfect weak scalability.

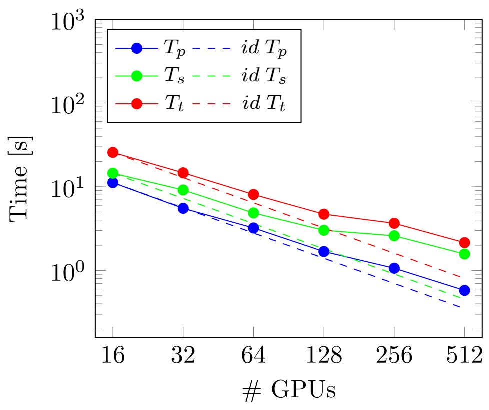


FIGURE 4.12: Strong scalability of aFSAI CG on matrix `pois198m`. aFSAI set-up time T_p, CG iteration time T_s and total time $T_t = T_p + T_s$ vs. number of GPUs.

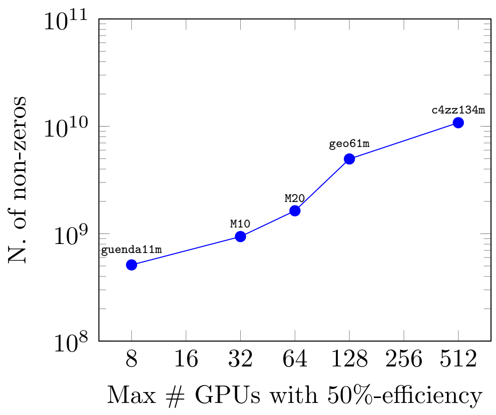


FIGURE 4.13: Maximum number of GPUs that guarantee a parallel efficiency of, at least, 50% for test matrices arising from structural mechanics.

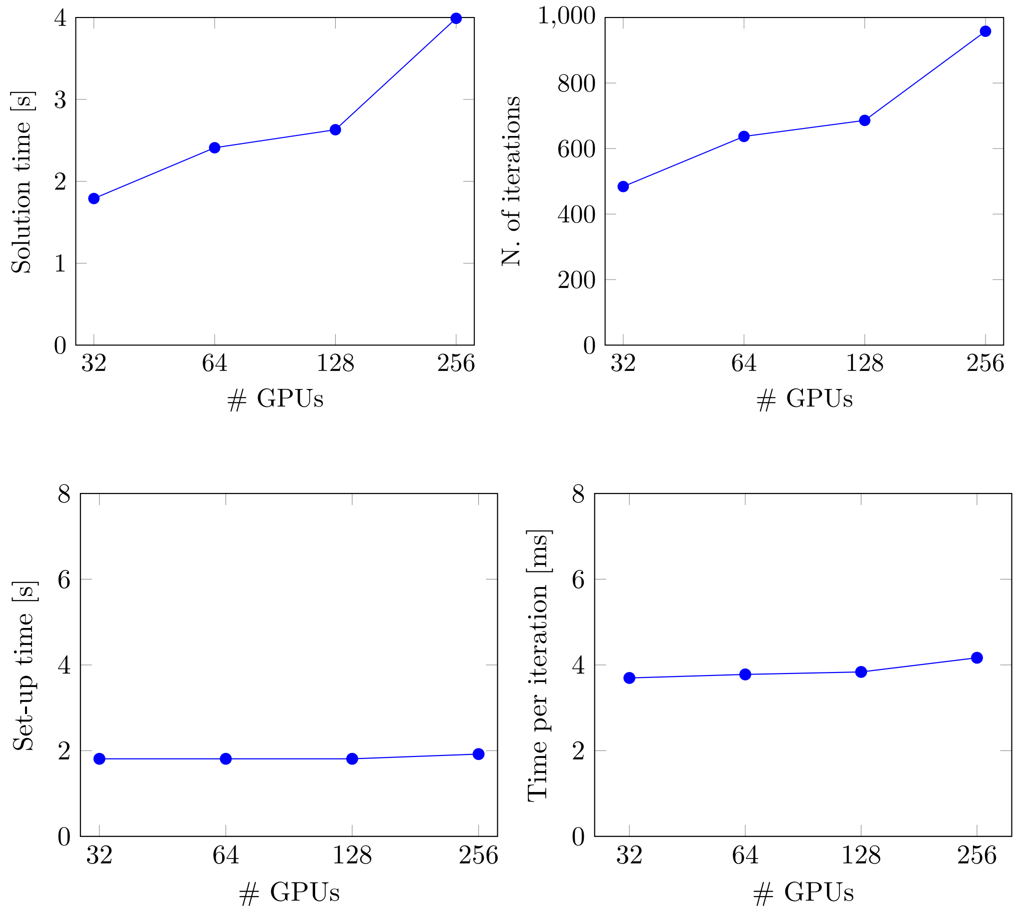


FIGURE 4.14: Weak scalability of aFSAI CG on a Poisson model problem. Iteration time [s] (top left), number of CG iteration (top right), set-up time [s] (bottom left) and time for one iteration [s] (bottom right) vs. the number of GPUs.

4.2.3 Effectiveness of BAMG prolongation

The GPU-accelerated set-up of the BAMG prolongation is tested on some of the mechanical problems collected in Table 4.1. As described in Section 3.3, the set-up is composed by a preliminary *gathering* stage on the *Host* and a *computation* stage on the *Device*. Unlike the aFSAI computation, the preliminary stage is not negligible compared to the total time, therfore, three different times are considered in the results: the gathering time, T_g, the computing time of the CUDA kernel, T_{Bk}, and total time, $T_{Bt} = T_{Bg} + T_{Bk}$. These times refer to the BAMG set-up for the first level of the hierarchy, only since the computation of the first level is about 80% of the preconditioner set-up when it comes to addressing mechanical problems.

In Table 4.7 the results for the CPU-only (Chr-CPU) and GPU-accelerated (Chr-GPU) mode of Chronos on the test matrices `wing4m`, `worm8m`, `agg14m` and `M20` are presented. In these runs, the computing resources, number of CPU cores n_{cr} and number of GPU boards n_{GPU}, are selected to have the maximum workload compatible with storage requirements: about $2 \cdot 10^{-7}$ and $2 \cdot 10^{-8}$ entries per CPU core and GPU board, respectively. Figure 4.15 shows the speed-up of GPU-accelerated version in the computation kernel only and in the total set-up.

Matrix	Chronos mode	n_{cr}	n_{GPU}	T_{Bg} [s]	T_{Bk} [s]	T_{Bt} [s]
`wing4m`	Chr-CPU	8	-	-	5.43	5.43
	Chr-GPU	8	1	-	2.26	2.26
`worm8m`	Chr-CPU	32	-	0.92	3.72	4.88
	Chr-GPU	32	4	0.92	1.39	2.77
`agg14m`	Chr-CPU	32	-	1.70	4.87	6.96
	Chr-GPU	32	4	1.70	2.35	4.82
`M20`	Chr-CPU	64	-	1.66	8.16	10.18
	Chr-GPU	64	8	1.66	1.99	4.36

TABLE 4.7: BAMG computing time in the first level of AMG hierarchy for some of the machanical test problems from Table 4.1. For each run, the following information is provided: Chronos mode, number of cores n_{cr}, number of GPU boards n_{GPU}, gathering time T_{Bg}, CUDA kernel time T_{Bk} and total time T_{Bt}.

The speed-up for the kernel-only is about $3 \div 4$ for the test cases with higher number of non-zeros per row, `worm8m` and `M20`, while it decreases to 2 for the others. These relatively low speed-ups, compared to the aFSAI one seen in the previous section, are mainly due to the intrinsic adaptivity of the algorithm that does not allow efficient scheduling of the workload: as the interpolation distance increases, the workload on the GPU board decreases since it must handles just a few fine nodes. Moreover, the larger the distance, the more expensive is the iteration. A non-adaptive approach, computing all the weights up to the maximum distance, was also investigated to facilitate workload scheduling, but the resulting increase in computational load reduced all the benefits of the optimized scheduling. The total speed-up is further reduced by the gathering stage, except for the case `wing8m` where no data movement is present since the run uses a single GPU board. The preliminary gathering plays a major role, particularly in the GPU-accelerated version: this stage is the same for the two Chronos modes but its relative magnitude on the total time increases from 20% for Chr-CPU up to more than 30% for Chr-GPU. An optimization is already planned since the gathering does not yet exploit OpenMP parallelization that proved to be very effective in the aFSAI set-up. Figure 4.16 showns the strong scaling for `agg14m` and `M20` test cases: as expected the BAMG total time decreases as computing resources are increased with almost ideal behaviour. By decreasing the work on the GPU, the behaviour deviates from the ideal, however the parallel efficiency with the minimum workload per GPU is still about 50% and 60% for `agg14m` (test case with the smallest number of non-zeros per row) and `M20` (test case with the highest number of non-zeros per row), respectively.

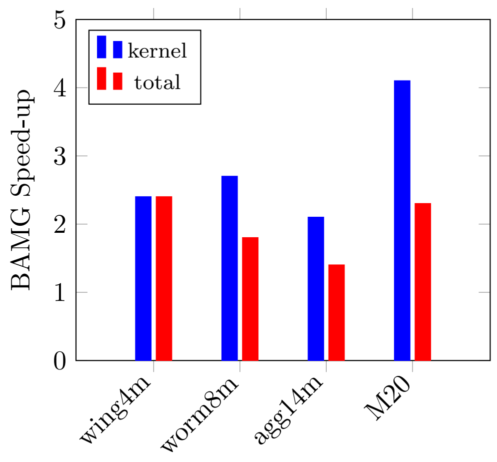


FIGURE 4.15: Speed-up of the GPU accelerated mode over the CPU-only in the BAMG set-up for the first level of the AMG hierarchy.

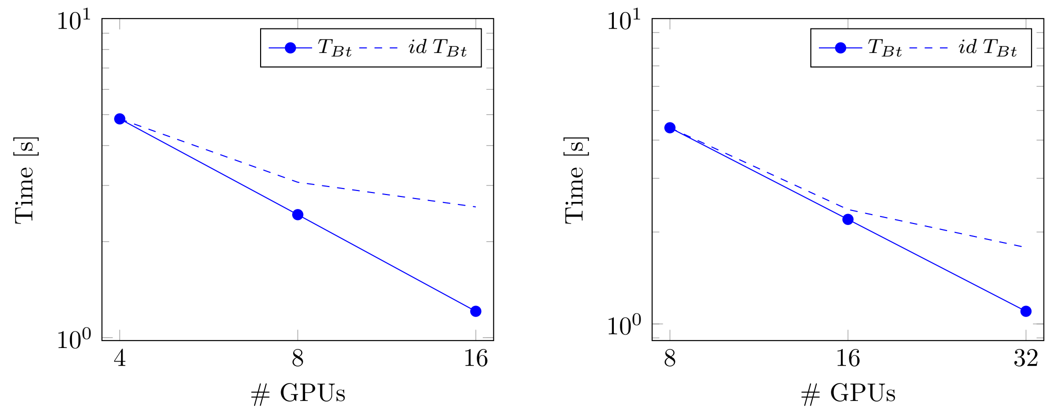


FIGURE 4.16: Strong scalability of the GPU-accelerated BAMG set-up on matrices `agg14m` (left) and `M20m` (right). Total BAMG computing time T_{Bt} vs. number of GPUs.

4.2.4 Effectiveness of MxM product

This section shows the effectiveness of the multi-GPU implementation of the MxM product in the context of the AMG set-up and, in particular, in the computation of the coarse matrix A_c (2.6). This operation, also reffered to as RAP product, involves two MxM products with both square and rectangular matrices. In the following, the results are presented in terms of computation time of the RAP product T_{RAP} for the first level of the AMG hierarchy which is the most expensive one. As pointed out in Section 3.4, the RAP operation is performed R(AP) where first AP is computed and then it is multiplied by R from the left.

Table 4.8 collects the results for the CPU-only (Chr-CPU) and GPU-accelerated (Chr-GPU) mode of Chronos on the test matrices `wing4m`, `worm8m`, `agg14m` and `M20`. For each run, further details are provided on the resulting matrix A_c, the number of rows n and the number of entries per row avg. nnz/row, and on the utilized computing resources, number of CPU cores n_{cr} and number of GPU boards n_{GPU}. The resources are selected to have about $2 \cdot 10^{+7}$ and $2 \cdot 10^{+8}$ entries per CPU core and GPU board, respectively. The speed-up of GPU-accelerated version is highlighted in Figure 4.17. Note that the largest speed-up values, around $4 \div 5$, are related to the test cases in which the resulting matrix has a larger number of non-zeros per row, `worm8m` and `M20`: for these cases communications among MPI ranks and data transfer (*DeviceToHost*

Matrix	n	A_c avg. nnz/row	Chronos mode	n_{cr}	n_{GPU}	T_{RAP} [s]
wing4m	350,115	223.18	Chr-CPU	8	-	7.90
			Chr-GPU	8	1	1.53
worm8m	525,833	486.63	Chr-CPU	32	-	11.62
			Chr-GPU	32	4	2.92
agg14m	560,026	324.10	Chr-CPU	32	-	10.14
			Chr-GPU	32	4	3.27
M20	1,027,225	434.61	Chr-CPU	64	-	15.48
			Chr-GPU	64	8	3.03

TABLE 4.8: RAP computing time in the first level of AMG hierarchy for some of the machanical test problems from Table 4.1. For each run, the following information is provided: number of rows n and number of entries per row avg. nnz/row of the coarse matrix A_c, Chronos mode, number of cores n_{cr}, number of GPU boards n_{GPU} and RAP time T_{RAP}.

and *HostToDevice*) have less impact on the total computing time, that is dominated by the computing stage. The run related to `wing4m` performs better than `agg14m` even if in the latter the A_c is denser, since the former uses a single GPU board and is not affected by any data movement. Finally, the strong scaling is investigated on `agg14m` and `M20` test cases by doubling the computing resources twice, refer to Figure 4.18-left and Figure 4.18-right, respectively. As expected the RAP time decreases increasing the computing resources with an almost ideal behaviour. With the smallest workload per GPU the parallel efficiency is about 70% in both cases.

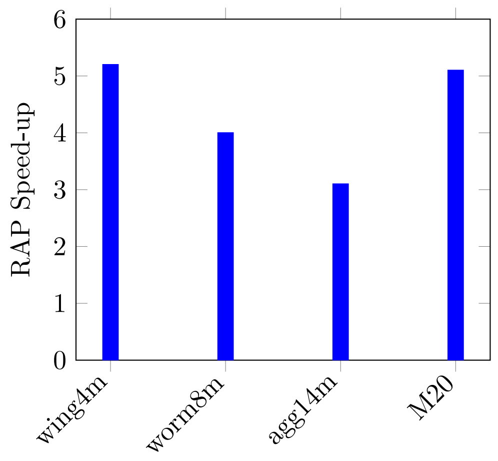
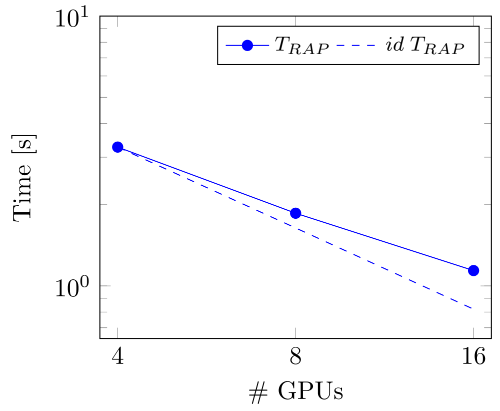


FIGURE 4.17: Speed-up of the GPU accelerated mode over the CPU-only in the RAP product for the first level of the AMG hierarchy.



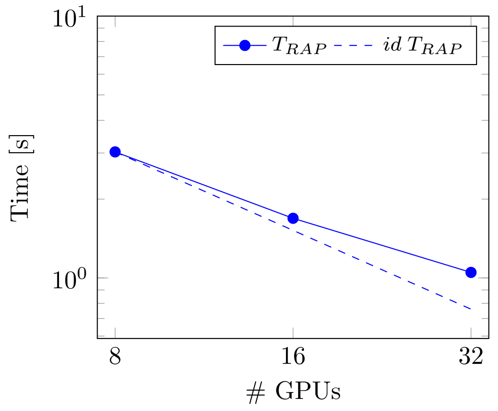


FIGURE 4.18: Strong scalability of the GPU-accelerated RAP product on matrices `agg143m` (left) and `M20` (right). RAP computing time T_{RAP} vs. number of GPUs.

4.2.5 Effectiveness of the whole AMG

In this section, the multi-GPU implementation of the Chronos AMG is investigated. The focus is more on mechanical problems, as they are more interesting for the author from a professional point of view as far as the development of GPU-accelerated algorithms is considered.

Let's first refer to the `M20` test case. Table 4.9 provides the AMG set-up times for the CPU-only (Chr-CPU) and GPU-accelerated (Chr-GPU) mode of Chronos using 64 CPU cores and, only for Chr-GPU, 8 GPU boards. The detailed time profiling consider the following stages: set-up of smoother (aFSAI) S, Test Space TS, Strenght of Connections SoC, BAMG Prolongator P, Prolongation Smoothing PS, Prolongation Filtering PF and RAP product RAP. The speed-ups of the GPU-accelerated version are highlighted in Figure 4.19. These results are provided for the first two levels, Lev_1 and Lev_2, and for the entire grid hierarchy All_H. Note that the AMG set-up is not fully GPU-accelerated, in particular, the computation of the strenght of connection, coarse node selection and filtering of the prolongation are still performed on the CPU. The first three have a negligible effect while the latter is about 25% of the total time for the GPU run and has a big influence on the overall speed-up. Prolongation filtering was not addressed in this work because it was planned to abandon this approach in favour

AMG stage	Chronos mode	Lev_1	Lev_2	All_H
S	Chr-CPU	39.66	7.76	47.96
	Chr-GPU	5.98	2.92	9.50
TS	Chr-CPU	12.03	1.52	13.66
	Chr-GPU	1.39	0.31	1.97
SoC	Chr-CPU	1.99	0.37	2.40
	Chr-GPU	1.99	0.37	2.40
P	Chr-CPU	10.19	0.68	10.93
	Chr-GPU	4.39	0.55	5.13
SP	Chr-CPU	9.57	1.50	11.15
	Chr-GPU	3.85	0.73	4.70
FP	Chr-CPU	9.03	0.79	9.89
	Chr-GPU	9.03	0.79	9.89
RAP	Chr-CPU	15.48	2.46	18.08
	Chr-GPU	3.03	0.62	3.73

TABLE 4.9: AMG set-up times for `M20` test case: set-up of smoother (aFSAI) S, Test Space TS, Strenght of Connections SoC, Prolongation Smoothing PS, Prolongation Filtering PF and RAP product RAP. A total of 64 CPU cores and 8 GPU boards have been used

of a more complex, but more efficient one based on energy minimization [Olson et al., 2011; Manteuffel et al., 2017; Janna et al., 2023]. The development of these algorithms will be addressed in future research. Analysing the speed-ups of the individual stages in the first two levels, it can be seen that there is a degradation of the performance of the GPU algorithms at the second level and a consequent decrease in relative speed-ups. This is due to the decreased workload on the single GPU board. Another improvement, that will be considered in the future, could be that of redistributing unknows on a smaller number of GPUs as the level size shrinks thus increasing GPU occupancy and reducing communication overhead. As expected, the stages that suffer the least are the prolongation smoothing and the RAP product, both of which are based on the MxM product that is designed to handle even very low workloads through a binning approach, as described in the previous Chapter.

The GPU-accelerated version of Chronos is then tested on some of the real-world problems collected in Table 4.1. In particular, Table 4.10 shows a comparison between Chr-CPU and Chr-GPU in terms of AMG set-up time T_p, iteration time T_s, and total time $T_t = T_p + T_s$. Additional information is also provided regarding grid and operator complexities, C_{gd} and C_{op}, the number of iterations required to achieve convergence n_{it}

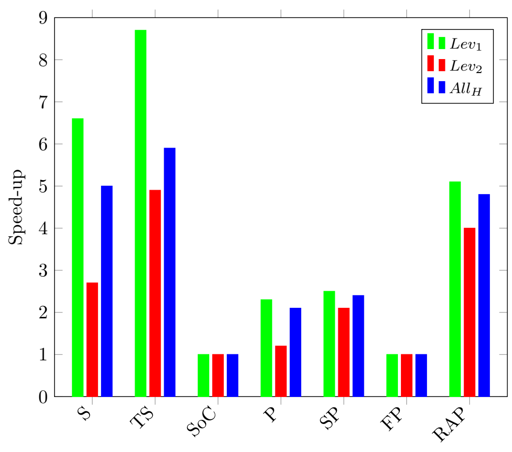


FIGURE 4.19: Speed-up of the GPU accelerated mode over the CPU-only one in the AMG set-up for `M20` test case.

and the utilized computing resources. Regarding the latter, they have been selected to have about $1-2\cdot 10^{+7}$ and $1-2\cdot 10^{+8}$ entries per CPU core and GPU board, respectively. The corresponding speed-ups are highlighted in Figure 4.20.

The number of iterations is slightly different between the run with Chr-CPU and Chr-GPU because in the GPU-accelerated version the smoother is computed in single precision, so with highly ill-conditioned problems it is a bit less effective. However, the benefit gained from single precision computation is still larger than the cost of a few more iterations in the solution stage.

Regarding speed-ups in the set-up stage, the highest values, about 3, correspond to mechanical test cases with the higher number of non-zeros per row: `wing4m`, `worm8m` and `M20`. The lowest values, about 2, occur with `agg14m` and the fluid dynamics case `poi111m`, in the latter the prolongation set-up is also not accelerated. As noted above, not all stages of the computation make use of accelerators and the workload per GPU board is low for levels of the hierarchy beyond the first. Therefore, there is still room for improvement and both of these problems will be addressed in future developments. On the other hand, the iteration stage already gives excellent results, with speed-ups up to 10 times over the Chr-CPU version. This performance actually balances the non fully-optimized set-up of the AMG on multi-GPU, particularly in all those applications where the preconditioner can be recycled.

The section concludes with a brief analysis on strong and weak scalability in terms

Matrix	Chronos mode	n_{cr}	n_{GPU}	C_{gd}	C_{gd}	n_{it}	T_p [s]	T_s [s]	T_t [s]
`wing4m`	Chr-CPU	8	-	1.084	1.473	288	89.8	140.4	230.2
	Chr-GPU	8	1			322	32.3	10.2	42.5
`worm8m`	Chr-CPU	32	-	1.069	1.430	337	83.1	107.4	190.1
	Chr-GPU	32	4			361	31.4	7.4	38.8
`agg14m`	Chr-CPU	32	-	1.042	1.374	51	51.2	16.1	67.3
	Chr-GPU	32	4			77	26.5	1.9	28.4
`M20`	Chr-CPU	64	-	1.054	1.296	294	122.9	225.9	347.0
	Chr-GPU	64	8			314	39.9	9.2	49.1
`poi111m`	Chr-CPU	64	-	1.379	2.325	29	60.3	16.6	76.9
	Chr-GPU	64	8			30	30.6	1.6	32.2

TABLE 4.10: Chronos performance on some of the test problems from Table 4.1. For each run, the following information is provided: Chronos mode, number of cores n_{cr}, number of GPU boards n_{GPU}, grid and operator complexities C_{gd} and C_{op}, number of iterations n_{it}, AMG set-up time T_p, iteration time T_s and total time T_t.

of set-up time, iteration and total. Figure 4.21 showns the strong scalability of `worm8m` and `M20` test cases. The computing resources are doubled twice from the configuration that allows the maximum workload compatible with storage requirements. As expected both the set-up and solution times decrease inversely proportional to the computing resources; with the lowest workload per GPU the parallel efficiency is about 50% and 60% for the set-up only and for the total time, respectively.

The weak scaling is investigated in Figure 4.22 with a standard 7-point finite difference discretization of the Poisson problem, using about $1 \cdot 10^{+8}$ non-zeroes per GPU board. The results show an almost ideal behaviour, with the set-up and total time that increase only slightly doubling the number of resources.

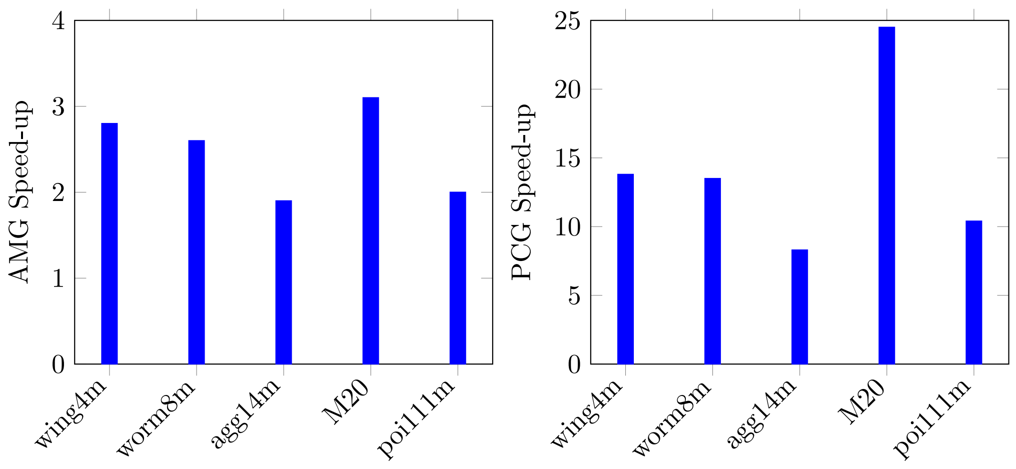


FIGURE 4.20: Speed-up of the GPU accelerated mode over the CPU-only one: AMG set-up time (left) and iteration time (right).

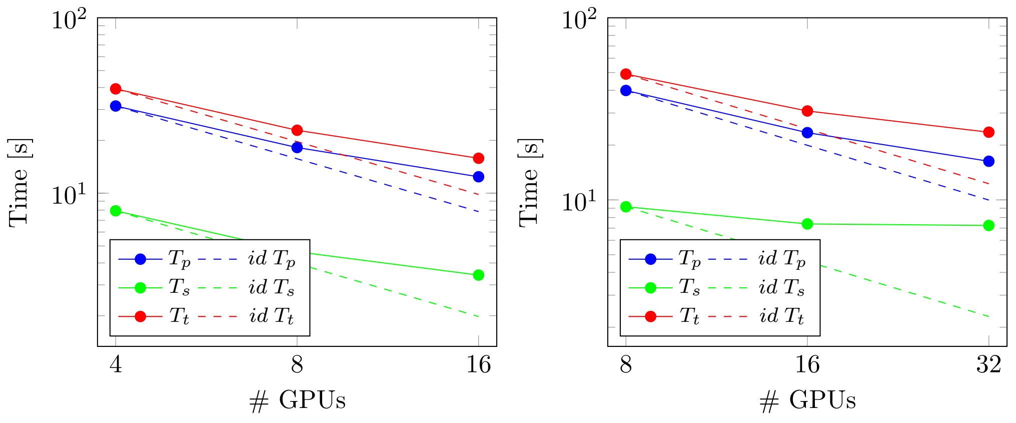


FIGURE 4.21: Strong scalability of the GPU-accelerated mode on matrices `worm8m` (left) and `M20` (right). AMG set-up time T_p, iteration time T_s and total time $T_t = T_p + T_s$ vs. number of GPUs.

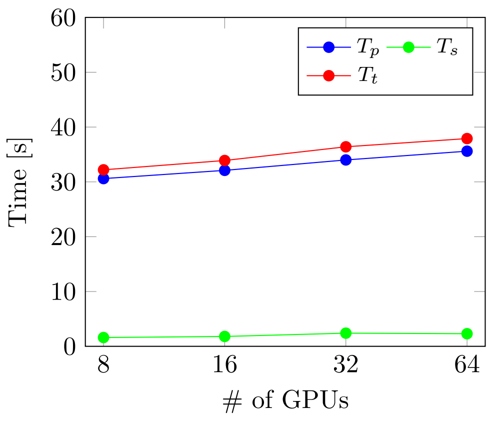


FIGURE 4.22: Weak scalability of the GPU-accelerated mode on a Poisson model problem. AMG set-up time T_p, iteration time T_s and total time $T_t = T_p + T_s$ vs. number of GPUs.

4.3 Chronos package applications

The development of a complex tool as an AMG solver for distributed memory systems was possible due to the high versatility of the Chronos package, as described in Chapter 2. The implementation of the entire software structure required a huge effort, but resulted in the successfull assembly of a library for sparse linear algebra having a very wide field of application, even outside the context of the AMG solver. This Chapter shows the robustness and parallel efficiency of the Chronos package on two different applications: the use of the aFSAI smoother as a preconditioner itself in the iterative linear solver of a geomechanical Finite Element (FE) simulator; the development a preconditioned framework to solve the algebraic block system arising in the simulation of fluid flow in Discrete Fractured Networks (DFN).

4.3.1 Acceleration of FE Geomechanical simulators

In geomechanical and basin evolution simulations, very fine meshes are needed to provide a realistic representation of complex stratigraphy and simulate the physical processes that are involved. Therfore, the exploitation of HPC infrastructure is mandatory. The CPU version of Chronos has already proven its effectiveness on this kind of problems as shown by Colombo et al. [2022], where it was interfaced with the open-source FE machanical simulator $Code\_Aster$ [EDF, 2023]. In this section the GPU-accelerated version of Chronos is tested through Atlas software [Atlas Project Team, 2023], a M3E propietary FE geomechanical simulator.

A real deep-water sedimentary model, is used as a benchmark. In particular, three geological events of the compaction model are investigated: at the beginning `TS2`, in the middle `TS5` and at the end of the deposition `TS7`. Moreover, different levels of mesh refinement are considered: low, $medium$, $high$. Table 4.11 summarizes the main information of the FE-mesh, number of nodes (nn) and number of 8-noded hexahedral elements ($nhex8$). Note that the $TS7 - high$ mesh corresponds to number of degrees of freedom ($3 \cdot nn$) greater than one billion.

Regarding the linear solver, the GPU-accelerated version of the aFSAI, described in Chapter 3, is used as a preconditioner itself. The aFSAI is preferred over AMG

FE-mesh	*low*		*medium*		*high*	
	nn	$nhex8$	nn	$nhex8$	nn	$nhex8$
`T2`	2,270,080	1,454,032	14,886,987	11,632,256	106,057,453	93,058,048
`T5`	4,585,544	3,817,800	33,612,825	30,542,400	256,619,387	244,339,200
`T7`	6,168,724	5,440,911	46,442,791	43,527,288	359,888,677	348,218,304

TABLE 4.11: Number of nodes nn and number of elements $nhex8$ for each FE-mesh.

because on geomechanical simulations, the model bottom and side boundaries are both considered fully constrained thus generating a linear system which is characterized by a reduced number of low frequencies. Therefore, coarse-grid correction is no more mandatory and a single-level preconditioner, that has a smaller set-up time, performs reasonably well even with large size models.

Marconi100 supercomputer has been used for running the tests, using a different ammount of resources according to the mesh refinement in order to assign approximately the same number of equations to each node: 1, 7 and 53 nodes for low, $medium$ and $high$ refinement, respectively.

The graph shown in Figure 4.23 collects the solution times required to solve the linear systems. In particular, the total solution time is split into set-up time of the aFSAI and iteration time of the PCG. The set-up stage, dashed portion of the columns, remains nearly constant as well as the workload per node for all the mesh refinements highlighting the high level of parallelism in the aFSAI computation. On the other hand, the iteration time grows due to the increasing number of iterations since single-level preconditioners like FSAI are non-optimal as multi-level ones. Note that the total solution time for the largest problem `TS7-high`, consisting of more than one billion unknowns, takes around half a minute.

The speed-up of the GPU-accelerated version over the CPU-only version is shown in Figure 4.24. Using GPU accelerators allows for a speed-up which is 5 in the worst case, with minimum workload, about 1,680,000 equations per GPU board. The speed-up grows up to 13 for the largest workload, about 4,920,000 equations per GPU board.

Finally, Figure 4.25 provides the strong scalability for a fixed problem size, and specifically, the trend of the total solution time T_t increasing the number of GPU boards.

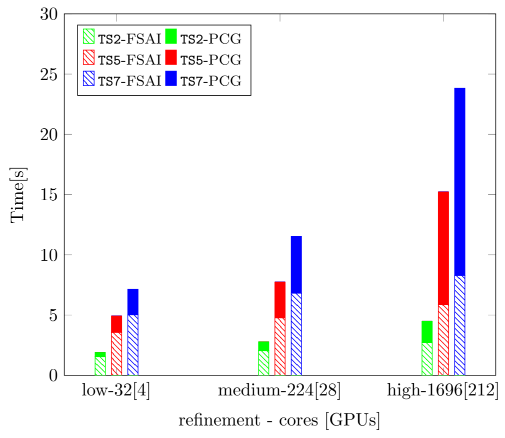


FIGURE 4.23: Total solution time in seconds, aFSAI set-up time plus PCG iteration time. The columns correspond to the six analyzed meshes (`TS-refinement`). Each refinement level is paired with the computing resources (cores [GPUs]).

In particular, the `TS5-medium` problem is solved increasing the computational resources from 224 cores and 24 GPUs to 1792 cores and 224 GPUs. The solution time T_t decreases inversely proportional to the computing resources, as expected, with a behavior close to the ideal one. Note that, even for very small workload per-GPU (about 450,000 dofs) the parallel efficiency is larger than 60%.

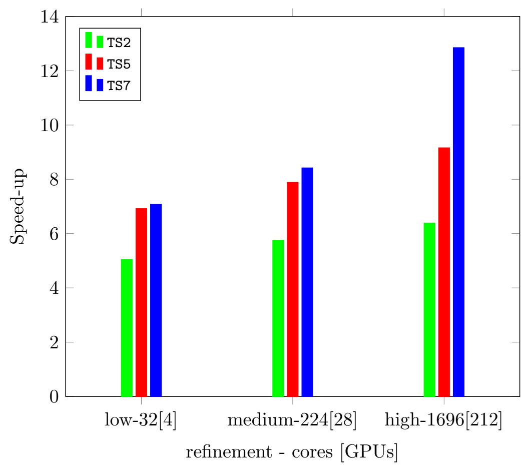


FIGURE 4.24: GPU Speed-up compared to the multi-cores CPU version. The columns correspond to the six analyzed meshes (`TS-refinement`). Each refinement level is paired with the computing resources (cores [GPUs]).

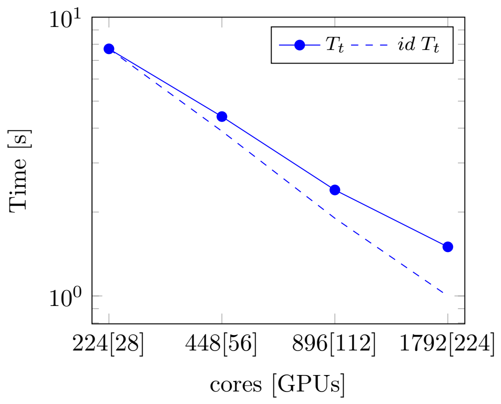


FIGURE 4.25: Strong scalability for the `TS5-medium` problem. Total solution time T_t versus computing resources (cores [GPUs]). The dashed line shows the ideal trend of the total solution time.

4.3.2 Block matrix-free preconditioner for DFN problems

This section describes how the Chronos package has been used to develop a preconditioned framework in order to solve the algebraic block systems arising in the simulation of fluid flow in large-size Discrete Fractured Networks (DFN). DFN models are usually preferred when the fracture network has a dominant impact on the fluid flow dynamics. They explicitly represent the fractures as intersecting planar polygons and neglect the surrounding rock formation, prescribing continuity constraints for the fluid flow along the fracture intersections, usually called *traces*. The discrete algebraic formulation of the DFN model proposed by Berrone et al. [2013a,b, 2019], leads to following system:

$$\mathcal{K} = \left[\begin{array}{cc|c} A & 0 & -C \\ G^h & A & -\alpha B \\ \hline -\alpha B^T & -C^T & G^u \end{array}\right], \qquad \boldsymbol{x} = \begin{bmatrix} \boldsymbol{h} \\ \boldsymbol{p} \\ \boldsymbol{u} \end{bmatrix}, \qquad \boldsymbol{f} = \begin{bmatrix} \boldsymbol{q} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix}, \tag{4.2}$$

where α is usually on the order of 1, $\boldsymbol{h} \in \mathbb{R}^{n^h}$ is the discrete hydraulic head on fractures, $\boldsymbol{p} \in \mathbb{R}^{n^p}$ are the discrete Lagrange multipliers and $\boldsymbol{u} \in \mathbb{R}^{n^u}$ is the discrete flux on the traces. The vector $\boldsymbol{q} \in \mathbb{R}^{n^h}$ includes the boundary conditions and the forcing terms. Usually, $n^p = n^h$, while according to the problem n^u can be either larger or smaller than n^h. The matrices in (4.2) are as follows:

- $A \in \mathbb{R}^{n^h \times n^h}$ is symmetric positive definite (SPD) and fracture-local, in the sense that it has a block-diagonal structure with the block size depending on each fracture dimension.
- $G^h \in \mathbb{R}^{n^h \times n^h}$ and $G^u \in \mathbb{R}^{n^u \times n^u}$ are symmetric positive semi-definite (SPSD), usually rank-deficient. The matrix G^h is fracture-local, i.e., with a block diagonal structure, while G^u has a global nature and operates on degrees of freedom related to different fractures;
- $B, C \in \mathbb{R}^{n^h \times n^u}$ are rectangular coupling blocks. The matrix C is fracture-local, with rectangular blocks whose size depends on the dimension of each fracture and the related traces, while $B = C + E$ has a global nature accounted for the

contribution of matrix E that has zero entries in the positions corresponding to the nonzero entries of the rectangular blocks of matrix C;

A reduced form of the system (4.2) is obtained by a block Gaussian elimination. The main computational burden is then the solution of the SPD system:

$$S_u(\alpha)\boldsymbol{u} = \boldsymbol{r} \tag{4.3}$$

where S_u is the Schur complement and $\boldsymbol{r}$ is the reduced forcing terms. Since S_u is SPD, the Conjugate Gradient solver, preconditioned with a Newton-Chebyshev polynomial [Bergamaschi and Martinez Calomardo, 2021; Bergamaschi et al., 2023], is employed where the diagonal of S_u, D_S, is used as *seed* preconditioner. In this context, the Chronos package is used to develop a parallel matrix-free implementation of the S_u-vector multiplication and the computation of D_S. Refer to the work by Bergamaschi et al. [2023] for a comprehensive description of the algorithm and of implementation details. The parallel efficiency of this implementation is evaluated on Marconi100 supercomputer using real-world models. In this application accelerators are not used; the parallelization relies on the MPI-OpenMP hybrid version, CPU-only. The relevant sizes and nonzeros of the test matrices are reported in Table 4.12. Note that, although the size of the models is limited, it is still the largest currently feasible because the software that generates the meshes and matrices does not support distributed memory environment.

Test case	n^u	$n^p \equiv n^h$	$nnz(\mathcal{K})$	# fractures
`Frac014`	312,518	221,144	10,854,803	1,425
`Frac151`	1,428,334	502,152	31,802,122	15,102
`Frac293`	2,777,378	994,907	44,646,710	29,370

TABLE 4.12: Size n and nonzeros nnz for each DFN test case.

First, the choice of the optimal polynomial degree m is made using `Frac014` test case on 128 cores (4 nodes) of Marconi100. Table 4.13, providing the number of iterations to converge and solution time for PCG, shows that the number of iterations always decreases with the degree of the polynomial, as expected, while the time to solution initially decreases but reaches a minimum for m = 127.

m	PCG iters	Solv. time [s]
3	2940	48.633
7	1509	50.024
15	670	44.493
31	378	49.962
63	195	51.636
127	76	40.509
255	46	49.445

TABLE 4.13: Number of iterations to converge and solution time for PCG preconditioned with a polynomials of varying degrees and 128 Marconi100 cores for test case `Frac014`.

Finally, the two cases `Frac16` and `Frac32` are solved with degree m = 127 by increasing the number of cores up to 32. The results are provided in Table 4.14. It is possible to note how the number of PCG iterations remains constant, as expected, while the solution times decreases with the increase of the number of cores. To better understand how effective polynomial preconditioning is in parallel, also the parallel efficiency η is reported, which is defined as the ratio between real and ideal speed-up. The results show an excellent strong scalability, with an efficiency of about 70% with 32 cores where the number of unknowns binded to each core is only 15,000 and 30,000 for `Frac16` and `Frac32`, respectively.

Test case	# of cores	PCG iters	Set-up time [s]	Solv. time [s]	η[%]
`Frac16`	2	105	83.9	1678.6	100.0
	4	104	43.3	866.6	96.8
	8	104	23.0	459.6	91.3
	16	103	12.5	249.7	84.0
	32	103	7.9	157.5	66.7
`Frac32`	4	107	87.5	1750.2	100.0
	8	107	46.3	924.4	94.7
	16	106	25.1	501.6	87.2
	32	108	15.0	300.5	72.8

TABLE 4.14: Number of iterations to convergence, solution time and parallel efficiency of the PCG preconditioned with polynomials of degree m = 127 with a varying number of Marconi100 cores.

Chapter 5

Conclusions

In this thesis a novel AMG solver for the solution of large and sparse linear systems of equations designed for HPC platforms has been presented. The solver is part of the Chronos package, a proprietary software from M^{3}E [M^{3}E, 2023] that cofunded the research project.

The presented AMG is based on classical methods already known in the literature, however, all of its algorithms have been revisited, tuned and optimized on the basis of a large experimentation on real-world and industrial benchmarks arising from a wide variety of application fields. Moreover, its great flexibility in the choice of the preconditioning strategy ensures that, once a proper set-up is found, total solution time depends almost solely on the problem size and the amount of computational resources allocated. Special care was given to both the design of high-level data structures and computing kernels to optimize parallel performance. A novel memory layout for the distributed matrix is proposed to enhance overlapping between communication and computation in the most important sparse linear algebra operations, matrix-by-vector product, matrix-by-matrix product and transposition, and to facilitate the gathering of the information which is preliminary to smoother and prolongation set-up. Moreover, an accelerated multi-GPU implementation was developed for all major set-up stages of the AMG: smoother, prolongation and coarse matrix set-up.

A wide set of numerical experiments was analyzed using the Marconi100 supercomputer which is equipped with both multi-core CPUs and GPU accelerators. These results show the ability of Chronos AMG to give excellent performance on a variety of applications. The CPU-only version provides solution times no worse or even better than those offered by other widely used HPC linear solvers such as BoomerAMG and

GAMG. With regard to the GPU version, the best performance, in terms of speed-up and scalability, is in the smoother set-up and in the computation of the coarse matrix (matrix-by-matrix products). Unfortunately, the overall performance of the AMG set-up is negatively affected by the low-workload per single GPU board, in the lower levels of the hierarchy and by the prolongation set-up that is still not fully GPU-accelerated. Future developments related to GPU accelerators will focus on reducing resources with levels, improving prolongation set-up and adopting nvlink communications to further reduce data transfer between *Host* and *Device*.

Finally, the object-oriented design of Chronos made it possible to exploit some of its components even outside the AMG context. The GPU-accelerated aFSAI smoother has been successfully used as a preconditioner itself in a FE geomechanical simulator. Moreover, all kernels for sparse linear algebra were used as basis for the development of a preconditioned framework to solve algebraic block system arising in the simulation of fluid flow in Discrete Fractured Networks (DFN).

Bibliography

Adams, M., Brezina, M., Hu, J., and Tuminaro, R. Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *Journal of Computational Physics*, 188(2):593–610, 2003. ISSN 0021-9991. doi: 10.1016/s0021-9991(03)00194-3.

Ali Beik, F. P. and Benzi, M. Iterative methods for double saddle point systems. *SIAM Journal on Matrix Analysis and Applications*, 39(2):902–921, January 2018.

Amestoy, P. R., Buttari, A., L'Excellent, J.-Y., and Mary, T. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Transactions on Mathematical Software*, 45(1):1–26, March 2019.

Atlas Project Team, T. Atlas web page. `https://www.m3eweb.it/atlas`, 2023. URL `https://www.m3eweb.it/atlas`.

Azad, A., Selvitopi, O., Hussain, M. T., Gilbert, J. R., and Buluç, A. Combinatorial blas 2.0: Scaling combinatorial algorithms on distributed-memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):989–1001, 2022. doi: 10.1109/TPDS.2021.3094091.

Badia, S., Martín, A. F., and Principe, J. Multilevel balancing domain decomposition at extreme scales. *SIAM Journal on Scientific Computing*, 38(1):C22–C52, 2016. URL `https://doi.org/10.1137/15M1013511`.

Baggio, R., Franceschini, A., Spiezia, N., and Janna, C. Rigid body modes deflation of the preconditioned conjugate gradient in the solution of discretized structural problems. *Computers & Structures*, 185:15–26, 2017. URL `https://doi.org/10.1016/j.compstruc.2017.03.003`.

Baker, A. H., Kolev, T. V., and Yang, U. M. Improving algebraic multigrid interpolation operators for linear elasticity problems. *Numerical Linear Algebra with Applications*, 17 (2-3):495–517, December 2009.

Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Karpeyev, D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Mills, R. T., Munson, T., Rupp, K., Sanan, P., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H. PETSc web page. https://www.mcs.anl.gov/petsc, 2023. URL https://www.mcs.anl.gov/petsc.

Bergamaschi, L., Ferronato, M., Isotton, G., Janna, C., and Martínez, A. Parallel matrix-free polynomial preconditioners with application to flow simulations in discrete fracture networks. *Computers & Mathematics with Applications*, 146:60–70, 2023. ISSN 0898-1221. doi: https://doi.org/10.1016/j.camwa.2023.06.032. URL https://www.sciencedirect.com/science/article/pii/S0898122123002845.

Bergamaschi, L. and Martinez Calomardo, A. Parallel newton–chebyshev polynomial preconditioners for the conjugate gradient method. *Computational and Mathematical Methods*, 3(6):e1153, 2021. doi: https://doi.org/10.1002/cmm4.1153. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cmm4.1153.

Bernaschi, M., Bisson, M., Fantozzi, C., and Janna, C. A factored sparse approximate inverse preconditioned conjugate gradient solver on graphics processing units. *SIAM Journal on Scientific Computing*, 38(1):C53–C72, jan 2016. URL https://doi.org/10.1137/15M1027826.

Bernaschi, M., Carrozzo, M., Franceschini, A., and Janna, C. A Dynamic Pattern Factored Sparse Approximate Inverse Preconditioner on Graphics Processing Units. *SIAM Journal on Scientific Computing*, 41(3):C139–C160, may 2019.

Bernaschi, M., Celestini, A., D'Ambra, P., and Vella, F. Multi-gpu aggregation-based amg preconditioner for iterative linear solvers. 03 2023. doi: 10.48550/arXiv.2303.02352.

Berrone, S., Scialò, S., and Vicini, F. Parallel meshing, discretization, and computation of flow in massive discrete fracture networks. *SIAM Journal on Scientific Computing*, 41(4):C317–C338, 2019. doi: 10.1137/18M1228736. URL https://doi.org/10.1137/18M1228736.

Berrone, S., Pieraccini, S., and Scialò, S. On simulations of discrete fracture network flows with an optimization-based extended finite element method. *SIAM Journal on Scientific Computing*, 35(2):A908–A935, 2013a. doi: 10.1137/120882883. URL https://doi.org/10.1137/120882883.

Berrone, S., Pieraccini, S., and Scialò, S. A pde-constrained optimization formulation for discrete fracture network flows. *SIAM Journal on Scientific Computing*, 35(2): B487–B510, 2013b. doi: 10.1137/120865884. URL https://doi.org/10.1137/120865884.

Bienz, A., Falgout, R. D., Gropp, W., Olson, L. N., and Schroder, J. B. Reducing parallel communication in algebraic multigrid through sparsification. *SIAM Journal on Scientific Computing*, 38(5):S332–S357, January 2016.

Brandt, A., Brannick, J., Kahl, K., and Livshits, I. Bootstrap AMG. *SIAM Journal on Scientific Computing*, 33(2):612–632, 2011. URL https://doi.org/10.1137/090752973.

Brannick, J., Cao, F., Kahl, K., Falgout, R., and Hu, X. Optimal interpolation and compatible relaxation in classical algebraic multigrid. *SIAM Journal on Scientific Computing*, 40(3):A1473–A1493, 2018. URL https://doi.org/10.1137/17M1123456.

Brezina, M., Falgout, R., MacLachlan, S., Manteuffel, T., McCormick, S., and Ruge, J. Adaptive smoothed aggregation (αSA) multigrid. *SIAM Review*, 47(2):317–346, 2005. URL https://doi.org/10.1137/050626272.

Brezina, M., Falgout, R., MacLachlan, S., Manteuffel, T., McCormick, S., and Ruge, J. Adaptive algebraic multigrid. *SIAM Journal on Scientific Computing*, 27(4):1261–1286, 2006a. URL https://doi.org/10.1137/040614402.

Brezina, M., Tong, C., and Becker, R. Parallel algebraic multigrids for structural mechanics. *SIAM Journal on Scientific Computing*, 27(5):1534–1554, 2006b. doi: 10.1137/040608271. URL https://doi.org/10.1137/040608271.

Chen, Y., Davis, T. A., Hager, W. W., and Rajamanickam, S. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, 35(3), oct 2008. ISSN 0098-3500. doi: 10.1145/1391989.1391995. URL https://doi.org/10.1145/1391989.1391995.

Colombo, D., Tardieu, N., Frigo, M., and Janna, C. Would like to make code_aster faster? *PRONET Update, Quarterly Report of Code_Aster Professionale Network*, 19, 2022.

D'Ambra, P., Serafino, D. D., and Filippone, S. MLD2P4: Multi-Level Domain Decomposition Parallel Preconditioners Package based on PSBLAS. *ACM Transactions on Mathematical Software*, 37(3):7 – 23, 09 2010. doi: 10.1145/1824801.1824808.

D'Ambra, P., Filippone, S., and Vassilevski, P. S. BootCMatch:a software package for bootstrap AMG based on graph weighted matching. *ACM Transactions on Mathematical Software*, 44(4):1 – 25, 06 2018. doi: 10.1145/3190647.

D'Ambra, P., Durastante, F., and Filippone, S. AMG Preconditioners for Linear Solvers towards Extreme Scale. *SIAM Journal on Scientific Computing*, 0(0):S679–S703, 2021. ISSN 1064-8275. doi: 10.1137/20m134914x.

De Sterck, H., Yang, U. M., and Heys, J. J. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM Journal on Matrix Analysis and Applications*, 27(4): 1019–1039, January 2006.

De Sterck, H., Falgout, R. D., Nolting, J. W., and Yang, U. M. Distance-two interpolation for parallel algebraic multigrid. *Numerical Linear Algebra with Applications*, 15(2-3): 115–139, 2008.

EDF. Finite element *code_aster*, analysis of structures and thermomechanics for studies and research. Open source on www.code-aster.org, 2023.

Falgout, R. D. and Schroder, J. B. Non-Galerkin coarse grids for algebraic multigrid. *SIAM Journal on Scientific Computing*, 36(3):C309–C334, January 2014.

Falgout, R. D. and Yang, U. M. Hypre: a library of high performance preconditioners. In *Proceedings of the International Conference on Computational Science-Part III*, ICCS '02, pages 632–641, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43594-8. URL http://dl.acm.org/citation.cfm?id=645459.653635.

Ferronato, M., Franceschini, A., Janna, C., Castelletto, N., and Tchelepi, H. A. A general preconditioning framework for coupled multiphysics problems with application to contact- and poro-mechanics. *Journal of Computational Physics*, 398:108887, December 2019.

Filippone, S., Cardellini, V., Barbieri, D., and Fanfarillo, A. Sparse Matrix-Vector Multiplication on GPGPUs. *ACM Transactions on Mathematical Software*, 43(4):1 – 49, 01 2017. doi: 10.1145/3017994.

Franceschini, A., Paludetto Magri, V. A., Mazzucco, G., Spiezia, N., and Janna, C. A robust adaptive algebraic multigrid linear solver for structural mechanics. *Computer Methods in Applied Mechanics and Engineering*, 352:389–416, August 2019.

Frigo, M., Castelletto, N., and Ferronato, M. A relaxed physical factorization preconditioner for mixed finite element coupled poromechanics . *SIAM J. Sci. Comput.*, 41: B694–B720, 2019.

Frommer, A., Kahl, K., Knechtli, F., Rottmann, M., Strebel, A., and Zwaan, I. A multigrid accelerated eigensolver for the Hermitian Wilson–Dirac operator in lattice QCD. *Computer Physics Communications*, 258:107615, 2021.

Gee, M. W., Hu, J. J., and Tuminaro, R. S. A new smoothed aggregation multigrid method for anisotropic problems. *Numerical Linear Algebra with Applications*, 16(1):19 – 37, 2009. doi: 10.1002/nla.593.

Goreinov, S. A., Oseledets, I. V., Savostyanov, D. V., Tyrtyshnikov, E. E., and Zamarashkin, N. L. How to find a good submatrix, April 2010.

Henson, V. E. and Yang, U. M. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155 – 177, 2002. ISSN 0168-9274. URL http://www.sciencedirect.com/science/article/pii/S0168927401001155. Developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss.

Janna, C. and Ferronato, M. Adaptive pattern research for block FSAI preconditioning. *SIAM Journal on Scientific Computing*, 33(6):3357–3380, 2011. URL https://doi.org/10.1137/100810368.

Janna, C., Ferronato, M., and Gambolati, G. The use of supernodes in factored sparse approximate inverse preconditioning. *SIAM Journal on Scientific Computing*, 37:C72–C94, 2015a. URL https://doi.org/10.1137/140956026.

Janna, C., Ferronato, M., Sartoretto, F., and Gambolati, G. FSAIPACK: A software package for high-performance factored sparse approximate inverse preconditioning. *ACM Trans. Math. Softw.*, 41(2):10:1–10:26, February 2015b. ISSN 0098-3500. URL http://doi.acm.org/10.1145/2629475.

Janna, C., Franceschini, A., Schroder, J. B., and Olson, L. Parallel energy-minimization prolongation for algebraic multigrid. *SIAM Journal on Scientific Computing, TO APPEAR*, 2023.

Kaporin, I. E. New convergence results and preconditioning strategies for the conjugate gradient method. *Numerical Linear Algebra with Applications*, 1(2):179–210, 1994. ISSN 1099-1506. URL http://dx.doi.org/10.1002/nla.1680010208.

Knuth, D. E. Semi-optimal bases for linear dependencies. *Linear and Multilinear Algebra*, 17(1):1–4, May 1985.

Kolotilina, L. Y. and Y., Y. A. Factorized sparse approximate inverse preconditioning. i. theory. *SIAM Journal on Matrix Analysis and Applications*, 14(1):45–58, 1993.

Koric, S. and Gupta, A. Sparse matrix factorization in the implicit finite element method on petascale architecture. *Comput. Methods Appl. Mech. Engrg.*, 302:281–292, April 2016.

Koric, S., Lu, Q., and Guleryuz, E. Evaluation of massively parallel linear sparse solvers on unstructured finite element meshes. *Computers & Structures*, 141:19 – 25, 2014. ISSN 0045-7949. URL https://doi.org/10.1016/j.compstruc.2014.05.009.

Lee, B. Algebraic multigrid for systems of elliptic boundary-value problems. *Numerical Linear Algebra with Applications*, 17:495–21, April 2020.

LeVeque, R. J. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002. doi: 10.1017/CBO9780511791253.

Li, R., Sjögreen, B., and Yang, U. M. A new class of amg interpolation methods based on matrix-matrix multiplications. *SIAM Journal on Scientific Computing*, 43(5): S540–S564, 2021. doi: 10.1137/20M134931X. URL https://doi.org/10.1137/20M134931X.

Lin, P. T., Shadid, J. N., Tuminaro, R. S., and Sala, M. Performance of a Petrov-Galerkin algebraic multilevel preconditioner for finite element modeling of the semiconductor device drift-diffusion equations. *International Journal for Numerical Methods in Engineering*, ED-11(1):n/a – n/a, 08 2010. doi: 10.1002/nme.2902.

Liu, W. and Vinter, B. A framework for general sparse matrix-matrix multiplication on gpus and heterogeneous processors. *Journal of Parallel and Distributed Computing*, 85: 47–61, 11 2015. doi: 10.1016/j.jpdc.2015.06.010.

Livne, O. E. and Brandt, A. Lean algebraic multigrid (LAMG): Fast graph laplacian linear solver. *SIAM Journal on Scientific Computing*, 34(4):B499–B522, 2012. URL https://doi.org/10.1137/110843563.

M^{3}E. M^{3}E web page. https://www.m3eweb.it, 2023. URL https://www.m3eweb.it.

Manteuffel, T. A., Olson, L. N., Schroder, J. B., and Southworth, B. S. A root-node–based algebraic multigrid method. *SIAM Journal on Scientific Computing*, 39(5): S723–S756, 2017. doi: 10.1137/16M1082706. URL https://doi.org/10.1137/16M1082706.

Mathias, P., Martin, W., Daniel, M., and Markus, S. speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis. Proceedings of the 25th

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 362–375, 2020.

Nagasaka, Y., Nukada, A., and Matsuoka, S. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. 2017 46th International Conference on Parallel Processing (ICPP), pages 101 – 110, 07 2017. ISBN 978-1-5386-1042-8. doi: 10.1109/icpp.2017.19.

Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Layton, S., Markovskiy, N., Reguly, I., Sakharnykh, N., Sellappan, V., and Strzodka, R. Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, 2015. doi: 10.1137/140980260. URL https://doi.org/10.1137/140980260.

NVIDIA. AmgX web page. https://developer.nvidia.com/amgx, 2023. URL https://developer.nvidia.com/amgx.

NVIDIA, Vingelmann, P., and Fitzek, F. H. Cuda toolkit, 2023. URL https://developer.nvidia.com/cuda-toolkit.

Olson, L. N., Schroder, J. B., and Tuminaro, R. S. A general interpolation strategy for algebraic multigrid using energy minimization. *SIAM Journal on Scientific Computing*, 33(2):966–991, 2011. doi: 10.1137/100803031. URL https://doi.org/10.1137/100803031.

Paludetto Magri, V. A., Franceschini, A., and Janna, C. A novel algebraic multigrid approach based on adaptive smoothing and prolongation for ill-conditioned systems. *SIAM Journal on Scientific Computing*, 41(1):A190–A219, January 2019.

Rouet, F.-H., Ashcraft, C., Dawson, J., Grimes, R., Guleryuz, E., Koric, S., Lucas, R. F., Ong, J. S., Simons, T. A., and Zhu, T.-T. Scalability challenges of an industrial implicit Finite Element code. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 505–514. IEEE, May 2020.

Roy, T., Jönsthövel, T., Lemon, C., and Wathen, A. A constrained pressure-temperature residual (cptr) method for non-isothermal multiphase flow in porous media. *SIAM Journal on Scientific Computing*, 42:B1014–B1040, 2020.

Ruge, J. W. and Stüben, K. *Algebraic multigrid*, chapter Society for Industrial and Applied Mathematics, pages 73–130. IEEE Educational Activities Department, 1987. URL http://locus.siam.org/doi/abs/10.1137/1.9781611971057.ch4.

Saad, Y. and Van der Vorst, H. A. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123:1–33, October 2000.

Sala, M. and Tuminaro, R. S. A new petrov–galerkin smoothed aggregation preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 31 (1):143–166, 2008. doi: 10.1137/060659545. URL https://doi.org/10.1137/060659545.

Strohmaier, E., Dongarra, J., Simon, H., and Meuer, M. Top500: The list of the 500 most powerful computer systems, 2023. URL https://www.top500.org.

Stüben, K. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1):281 – 309, 2001. ISSN 0377-0427. URL http://www.sciencedirect.com/science/article/pii/S0377042700005161. Numerical Analysis 2000. Vol. VII: Partial Differential Equations.

the Chronos Project Team. Chronos web page. https://www.m3eweb.it/chronos, 2023. URL https://www.m3eweb.it/chronos.

Trilinos Project Team, T. The Trilinos Project Website, 2023. URL https://trilinos.github.io.

Trottenberg, U., Oosterlee, C., and Schüller, A. *Multigrid*. Academic Press, 2001. ISBN 9780127010700. URL https://www.elsevier.com/books/multigrid/trottenberg/978-0-08-047956-9.

Vaněk, P., Mandel, J., and Brezina, M. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3):179–196, Sep 1996. ISSN 1436-5057. URL https://doi.org/10.1007/BF02238511.

Wathen, M. and Greif, C. A scalable approximate inverse block preconditioner for an incompressible magnetohydrodynamics Model Problem. *SIAM Journal on Scientific Computing*, 42(1):B57–B79, January 2020.

Xu, J. and Zikatanov, L. Algebraic multigrid methods. *Acta Numerica*, 26:591–721, 2017. URL `http://dx.doi.org/10.1017/S0962492917000083`.

Zienkiewicz, O., Taylor, R., and Zhu, J. *The Finite Element Method: its Basis and Fundamentals (Seventh Edition)*. Butterworth-Heinemann, Oxford, seventh edition edition, 2013. ISBN 978-1-85617-633-0. doi: https://doi.org/10.1016/B978-1-85617-633-0.00019-8. URL `https://www.sciencedirect.com/science/article/pii/B9781856176330000198`.