The solution of large sparse linear systems on parallel computers using a hybrid implementation of the block Cimmino method

Mohamed Zenadi

▶ **To cite this version:**

Mohamed Zenadi. The solution of large sparse linear systems on parallel computers using a hybrid implementation of the block Cimmino method. Other [cs.OH]. Institut National Polytechnique de Toulouse - INPT, 2013. English. NNT : 2013INPT0126 . tel-04301573

HAL Id: tel-04301573

https://theses.hal.science/tel-04301573v1

Submitted on 23 Nov 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Sureté de Logiciel et Calcul à Haute Performance

Présentée et soutenue par :

M. MOHAMED ZENADI

le mercredi 18 décembre 2013

Titre :

METHODES HYBRIDES POUR LA RESOLUTION DE GRANDS SYSTEMES LINEAIRES CREUX SUR CALCULATEURS PARALLELES.

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

Directeur(s) de Thèse :

M. PATRICK AMESTOY

M. DANIEL RUIZ

Rapporteurs :

M. AHMED SAMEH, PURDUE UNIVERSITY EU

M. MILOUD SADKANE, UNIVERSITE DE BRETAGNE OCCIDENTALE

Membre(s) du jury :

M. ANDREAS FROMMER, UNIVERSITE DE WUPPERTAL, Président

M. DANIEL RUIZ, INP TOULOUSE, Membre

M. IAIN S. DUFF, RUTHERFORD APPLETON LABORATORY OXFORD, Membre

M. MARIO ARIOLI, RUTHERFORD APPLETON LABORATORY OXFORD, Membre

M. PATRICK AMESTOY, INP TOULOUSE, Membre

Résumé

Nous nous intéressons à la résolution en parallèle de système d'équations linéaires creux et de large taille. Le calcul de la solution d'un tel type de système requiert un grand espace mémoire et une grande puissance de calcul. Il existe deux principales méthodes de résolution de systèmes linéaires. Soit la méthode est directe et de ce fait est rapide et précise, mais consomme beaucoup de mémoire. Soit elle est itérative, économe en mémoire, mais assez lente à atteindre une solution de qualité suffisante. Notre travail consiste à combiner ces deux techniques pour créer un solveur hybride efficient en consommation mémoire tout en étant rapide et robuste. Nous essayons ensuite d'améliorer ce solveur en introduisant une nouvelle méthode pseudo directe qui contourne certains inconvénients de la méthode précédente .

Dans les premiers chapitres nous examinons les méthodes de projections par lignes, en particulier la méthode Cimmino en bloc, certains de leurs aspects numériques et comment ils affectent la convergence. Ensuite, nous analyserons l'accélération de ces techniques avec la méthode des gradients conjugués et comment cette accélération peut être améliorée avec une version en bloc du gradient conjugué. Nous regarderons ensuite comment le partitionnement du système linéaire affecte lui aussi la convergence et comment nous pouvons améliorer sa qualité. Finalement, nous examinerons l'implantation en parallèle du solveur hybride, ses performances ainsi que les améliorations possible.

Les deux derniers chapitres introduisent une amélioration à ce solveur hybride, en améliorant les propriétés numériques du système linéaire, de sorte à avoir une convergence en une seule itération et donc un solveur pseudo direct. Nous commençons par examiner les propriétés numériques du système résultants, analyser la solution parallèle et comment elle se comporte face au solveur hybride et face à un solveur direct. Finalement, nous introduisons de possible amélioration au solveur pseudo direct. Ce travail a permis d'implanter un solveur hybride "ABCD solver" (Augmented Block Cimmino Distributed solver) qui peut soit fonctionner en mode itératif ou en mode pseudo direct.

Mots-clés : matrices creuses, méthodes itératives de résolution de systèmes linéaires, méthodes directes de résolution de systèmes linéaires, méthodes hybrides, partitionnement, hypergraphes, calcul haute performance, calcul parallèle.

Abstract

We are interested in solving large sparse systems of linear equations in parallel. Computing the solution of such systems requires a large amount of memory and computational power. The two main ways to obtain the solution are direct and iterative approaches. The former achieves this goal fast but with a large memory footprint while the latter is memory friendly but can be slow to converge. In this work we try first to combine both approaches to create a hybrid solver that can be memory efficient while being fast. Then we discuss a novel approach that creates a pseudo-direct solver that compensates for the drawback of the earlier approach.

In the first chapters we take a look at row projection techniques, especially the block Cimmino method and examine some of their numerical aspects and how they affect the convergence. We then discuss the acceleration of convergence using conjugate gradients and show that a block version improves the convergence. Next, we see how partitioning the linear system affects the convergence and show how to improve its quality. We finish by discussing the parallel implementation of the hybrid solver, discussing its performance and seeing how it can be improved.

The last two chapters focus on an improvement to this hybrid solver. We try to improve the numerical properties of the linear system so that we converge in a single iteration which results in a pseudo-direct solver. We first discuss the numerical properties of the new system, see how it works in parallel and see how it performs versus the iterative version and versus a direct solver. We finally consider some possible improvements to the solver. This work led to the implementation of a hybrid solver, our "ABCD solver" (Augmented Block Cimmino Distributed solver), that can either work in a fully iterative mode or in a pseudo-direct mode.

Keywords: sparse matrices, iterative methods for linear systems, direct methods for linear systems, hybrid methods, partitioning, hypergraphs, high-performance computing, parallel computing.

Acknowledgements

The work presented in this manuscript would not have been possible without the close association of many people, the first of which is *Daniel Ruiz* who encouraged, taught, and helped me throughout the last three years. Daniel is the advisor anybody would want, he is smart and fun to talk with. I really enjoyed all the hours we spent talking about the past, the present and the future of mankind. Although he is patient with people, when he lose patience it is usually *Ginette* (his computer) the main culprit. You will never get bored with him, best advisor and a good friend. I am also grateful to *Pr. Patrick Amestoy* and *Dr. Ronan Guivarch* my co-advisors, always present for their scientific and computer engineering advices, they are the people who made me chose this field and thanks to them, I enjoy being part of it. I also grateful to *Pr. Iain S. Duff* for all the scientific advice, all the red markings on the manuscript and papers that made me doubt I had written anything right, and especially for his fact checking. All of you guys, thank you!

Many thanks to the referees of this manuscript, Professors *Miloud Sadkan* and *Ahmed Sameh*, for reviewing and commenting the dissertation, and for traveling such long distances just to attend the defense. I also thank the other members of the committee, *Mario Arioli* and *Andreas Frommer* for their comments.

My heartfelt thanks to those who populated my work environment, Florent Lopez (flop), François-Henry Rouet (dontu), Clément Royer, Clément Weisbecker (climoune). A big thanks to everyone at the IRIT laboratory, including Alfredo Buttari, Fred Camillio, Michel Daydé, André Gaetan, Guillaume Joslin, Chiara Puglisi, and to both Sylvies. A special thanks to my friends Jérémy, Toufik, and Djalel (I love you brother!).

All my gratitude and love to the best parents, who finally are no longer asking me *"did you finish your homework?"*. The two people who took care of me the most, who taught me, and who supported me during more than two decades of studies. Love and thanks to my sister who supported me and took good care of me, to my brother with whom I had so much fun playing Mario Kart and Mario Party. A big thanks to my family-in-law who welcomed me as a member and who took care of me whenever I had the chance to visit them, especially the lovely person that is my mother in-law.

My deepest gratitude towards the person who enlightened my life, the person who supports me day and night, who understands me the most, who loves me, and who gives me strength, **Safia**, my eternal love, with whom I had the chance to smile more, to laugh more, to feel as the luckiest man on earth, and the one who gave birth to little **Haroon** the icing on the cake to conclude my thesis.

Contents

Chapter 1

Introduction

One remarkable breakthrough in solving algebraic problems dates to almost two millennia ago in a collection of essays by multiple Chinese scholars called *Nine Chapters on the Mathematical Arts*. A large number of problems of everyday life and engineering were presented along with their solutions in nine categories. The eighth chapter, *The rectangular array*, introduces a method of solving systems of linear equations through a succession of elimination steps. One of the problems as as follows

"*Now given 3 bundles of top grade paddy, 2 bundles of medium grade paddy, (and) 1 bundle of low grade paddy. Yield: 39 dou of grain. 2 bundles of top grade paddy, 3 bundles of medium grade paddy, (and) 1 bundle of low grade paddy, yield 34 dou. 1 Bundle of top grade paddy, 2 bundles of medium grade paddy, (and) 3 bundles of low grade paddy, yield 26 dou. Tell: how much paddy does one bundle of each grade yield? Answer: Top grade paddy yields 9(1/4) dou (per bundle); medium grade paddy 4(1/4) dou; (and) low grade paddy 2(3/4) dou.*" [1]

In modern times, this problem would have been transformed into a 3×3 matrix and could be solved using Gaussian elimination. In fact, the solution proposed in the textbook is similar to Gaussian elimination as it tries to nullify the coefficients of the variables to obtain as many zeros as possible. The textbook presents the solution to problems up to 6×6, and even underdetermined problems.

Today's problems are more complicated than finding the amount of paddy depending on its grade. Structural analysis, chemical engineering, network theory, fluid dynamics, data analysis and circuit theory are only a few fields where systems of linear equations are used. The main difference is that these systems yield *sparse* matrices. A matrix is said to be sparse if there is a small ratio of nonzero entries to the total number of entries in the matrix. The sparsity of the matrix is due to the loose coupling in the data it represents, in network theory this means that each node is connected to only a few other nodes, in physical problems this can be a result of weak interactions between the different physical domains. Exploiting the sparsity is essential if we want to solve large systems on modern architectures. We note that today's problems can be as large as millions or even billions in size.

During the last decades, much work was done on computer hardware to be able to execute as many operations per second as possible. As hardware becomes more and more powerful, algorithms for solving systems of linear equations had to wisely exploit the new possibilities offered by the hardware, such as faster communications between the different machine nodes and the increasing number of processing units per node. This led to a wave of algorithms that try to use the different levels of parallelism available to them and to do that efficiently.

The algorithms to solve systems of linear equations can be divided into two families: *direct* or *iterative* methods. Direct methods are based on a factorization of a permutation of the given input matrix. Depending on the numerical properties of the matrix, one can use either LU, LL^T or

LDL^T factorizations. These factors are then used in substitution processes to obtain the solution. However, direct methods suffer from a potentially very large memory footprint when solving large linear problems, in particular those arising from 3D partial differential equation simulations. Different studies have been made to reduce the memory usage, such as using and *out-of-core* approach (see [2] and [3]) or *low-rank* approximation (see [4] and [5]).

Iterative methods are essentially based on matrix-vector products with an iteration matrix that can be obtained from some particular transformations of the original matrix. They generally compute successive approximations of the solution and monitor convergence with respect to some error measure. In some cases, when some class of iterative methods are specifically suited for a given class of problems, convergence can be very fast making these techniques very appealing with respect to direct methods. However, in the general case, these methods may converge slowly unless some sort of numerical boost is applied to them. This usually means applying some form of preconditioner that improves the numerical properties of the iteration matrix so that convergence is faster. The only issue here is that finding some appropriate preconditioner is usually problem dependent and, in particular for hard cases, it is difficult to find cheap preconditioners to achieve fast enough convergence. However, one major advantage of iterative methods in general is the low memory usage as the required data structures are mainly the original matrix and few sets of extra data whose size is proportional to the size of the original matrix. This feature has made them quite attractive for very large sparse problems.

An intermediate class of methods, that emerged later, called **hybrid** methods, try to combine the advantages of both previous classes e.g. the robustness of direct methods and the memory economy of iterative methods. Hybrid methods usually decompose the original problem into subproblems which are solved independently using a direct solver. Due to the fact that these subproblems are smaller, solving them usually requires less memory with respect to solving the original problem with a direct solver. The solutions to the subproblems are then combined within an iterative process to derive approximations to the solution of the original problem. Additionally, they exploit the independence between the subproblems raising an additional straightforward level of parallelism. One or more additional levels of parallelism are achieved within the direct solver itself when solving the subproblems.

Within the class of hybrid methods, we are interested in row projection techniques which, as suggested by their name, use successive projections onto the range of subsets of rows of the matrix. These projections are computed using a direct solver and the update of the solution is iterative, which is why they are considered hybrid. For a natural first level of parallelism, we focus specifically on the block Cimmino method in which these projections are basically summed at each iteration and therefore can be computed totally independently. Row projection techniques usually require some form of acceleration because convergence can be very slow. The most common acceleration method is the *conjugate gradient* method, when the iteration matrix is similar to a symmetric positive definite matrix, which is in fact the case for the block Cimmino method. The conjugate gradient method belongs to the class of Krylov methods and guarantees the convergence in exact arithmetic after a finite number of iterations when the matrix is symmetric positive definite. However, it can also be used as an iterative technique with an appropriate monitoring of convergence, and this yields a usually robust iterative method. Its convergence can be improved even further by using a block Krylov technique such as the *block conjugate gradient* algorithm.

We start this dissertation by introducing, in Chapter 2, different aspects of row projection techniques. We recall some of their numerical properties, as described by Elfving in [6], and discuss their acceleration using the conjugate gradient method, based on early work by Bramley and Sameh in [7]. Next we analyse different types of convergence profile and explain how it is possible to improve it in some cases by means of the stabilized block conjugate gradient method presented by Arioli, Duff, Ruiz and Sadkane in [8]. Throughout this chapter, we also show in detail how matrix-vector products with the iteration matrix are performed using a direct solver

applied to augmented systems that express the row projection solutions.

Partitioning of the original system is an important issue in hybrid methods in general. Depending on the hybrid method, various partitioning strategies can be considered, their quality being directly related to the achieved efficiency. In Chapter 3, we investigate the particular case of row projection techniques and the effect of various partitioning strategies of the original linear system on the convergence profile of the method. We introduce and compare different strategies that aim to obtain partitions that improve the spectrum of the iteration matrix used by the conjugate gradient acceleration. The first strategy we present tries to form two sets of partitions that are, within each set, mutually orthogonal to each other. The second strategy lowers this constraint and tries to offer more freedom with the hope of being able to get more and better equilibrated partitions while maintaining a convergence profile similar to the first strategy. Finally, we consider a way to use hypergraph partitioning to obtain partitions that share minimum column overlap, and we also analyse its benefits compared to the two previous strategies.

With respect to very large test problems for which, as we have said earlier, memory and efficiency issues are very important, we propose a specific hybrid implementation of the block Cimmino method that interfaces with the direct solver in a clever way so that it exploits as much as possible all levels of parallelism. We present in Chapter 4 the details about the algorithms used in this hybrid approach, starting with a basic master-slave implementation to highlight some of its algorithmic aspects. We then propose a fully distributed version that can address in some specific manner the different levels of parallelism identified, with appropriate mapping of data structures across participating processes. On various sets of test problems, including some very large ones, we illustrate the parallel potential of our hybrid implementation of the block Cimmino method and show how it compares to a direct solver in terms of efficiency and memory usage.

One remaining major issue, with respect to the design of a general purpose iterative solver, is the numerical behaviour (convergence) which remains problem dependent and cannot be a priori controlled. Indeed, the efficiency of block row projection techniques relies strongly on the fact that the partitioning of the original matrix is able to capture most of the ill-conditioning within the blocks, and removes ill-conditioning across the blocks. This is not an easy target to achieve while maintaining at the same time a good degree of parallelism and low memory usage, as these latter issues usually require a minimum number of partitions. To achieve this goal, we propose a way to enforce numerical orthogonality between the sets of rows given by the partitioning (whatever one we use). We introduce in Chapter 5 a novel approach based on some form of augmentation of the original matrix. Indeed, to enforce numerical orthogonality between predefined sets of rows, we introduce some extra variables and show different ways to enlarge the system of equations and reach this orthogonality in the super-space. We also introduce an extra set of equations to recover the same solution as in the original linear system. This augmentation technique leads to an algorithm to compute the solutions, that involves the same type of operations as in usual row projection techniques, but with the introduction of a new auxiliary symmetric positive definite linear system of much smaller size. This auxiliary system can be easily generated by means of calls to the direct solver on multiple right-hand sides, and in an embarrassingly parallel manner. Factorizing this auxiliary smaller system ultimately yields an implicit direct solver, but based on tools and mechanisms specific to block iterative techniques. This novel approach results finally in a fixed number of calls to the direct solution of sets of linear systems of much smaller sizes, and numerical issues with convergence are avoided while maintaining a good degree of parallelism and low memory usage. We illustrate the benefits of this new solver on a set of large matrices and see how it compares with our classical hybrid solver.

Finally, in Chapter 6, we investigate various ways to decrease the size of this auxiliary matrix. When doing so, we relax gradually the aforementioned enforced numerical orthogonality and we are led to again require iterations to recover the solution. This approach defines a trade-off between the pure iterative technique (no augmentation) and the implicit direct method (full augmentation).

We then try to understand experimentally how this can still be beneficial. We also investigate the iterative solution of the full auxiliary symmetric positive definite linear system (without any reduction) as it is possible to use matrix-vector products with this matrix without the need to build it. These two complementary studies can indeed be of some interest when this auxiliary system is too large or too dense, which can make it difficult to build explicitly and factorize.

The various experiments in this manuscript are performed on three different architectures. We present in Table 1.1 these three architectures with their name and location, and their corresponding characteristics such as the type of processors they have, the number of cores per node and the amount of memory per node. In the rest of this dissertation we shall refer to the architectures directly by their names whenever necessary.

Name	**CPU**	**Core/node**	**Memory/node**	**Location**
`Conan`	AMD Opteron(TM) 6220	32	500GB	ENSEEIHT
`Hyperion`	Intel Xeon 5560	8	32GB	CALMIP Toulouse
`Oakleaf`	Fujitsu Sparc64-IXfx	16	32GB	Tokyo University

Table 1.1: The systems used to test our solver.

Chapter 2

Block Conjugate Gradient acceleration of the block Cimmino method

The classical and the block version of the conjugate gradient algorithm are methods that can theoretically guarantee the convergence of the solution to symmetric positive definite systems in a finite number of iterations in the absence of roundoff errors. This characteristic makes it an appealing acceleration method for different iterative schemes. Among these schemes we will study the block Cimmino one in detail and see how we can use both the classical and the block conjugate gradient algorithms as accelerators of convergence.

We start by looking at some of the row projection techniques and the theory behind them. We look at how we can accelerate them and the special case of conjugate gradient acceleration. Then we discuss the block conjugate gradient (Block-CG) acceleration and the main differences with the classical version.

2.1 Introduction to row projection techniques

In this chapter we consider the solution of the linear system

$$Ax = b \tag{2.1}$$

where A is an $m \times n$ sparse matrix, x an n-vector and b is a given m-vector.

Elfving [6] and Campbell [9] showed that the minimum norm solution (m.n.s.) to (2.1) is a solution to the least squares problem

$$\min \| x \|_2 \text{ with } x \in \{x; \| Ax - b \|_2 \text{ is minimum}\},$$

and in the case of a consistent system, this m.n.s is unique and can be obtained by computing

$$x = A^+ b \tag{2.2}$$

where A^+ is the Moore-Penrose pseudoinverse defined by

$$A^+ = A^T (AA^T)^{-1}. \tag{2.3}$$

Thus the solution (2.2) can be obtained through the solution of

$$AA^T z = b, \qquad \text{then} \qquad x = A^T z,$$

the details of obtaining the pseudoinverse are discussed in [10]. Let $P_{\mathcal{R}(A^T)}$ be the orthogonal projector onto the range of A^T written as

$$P_{\mathcal{R}(A^T)} = A^+ A, \tag{2.4}$$

and $P_{\mathcal{N}(A)}$ the orthogonal projector onto the nullspace of A defined as

$$P_{\mathcal{N}(A)} = I - P_{\mathcal{R}(A^T)}.$$

For simplicity, we will assume that A is a square $m \times m$ matrix and has full rank. Consider the partition of the matrix A into p strips of rows, and the partitioning of b accordingly, as in the following:

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix}. \tag{2.5}$$

Row Projection methods are algorithms that compute the solution to the linear system (2.1) through successive projections of the iterate $x^{(k)}$ onto the range of each A_i^T. The row projection methods fall usually into two general categories, that we recall here: the Kaczmarz [11] form, in which the next iterate is obtained through a product of projections and the Cimmino [12] form in which the iterate is obtained through a sum of projections, Bramley and Sameh did a deep comparison of both approaches in [7].

In the following we consider that we have at least several rows per partition, this approach is called block row projection and is a generalization for both Kaczmarz and Cimmino methods that consider single row projections.

The block Kaczmarz method obtains its next iterate through a product of projections of the current iterate such that, at the k-th iteration, we have the following computation

$$\delta^{(i+1)} = \delta^{(i)} + A_i^+ \left(b_i - A_i \delta^{(i)}\right) \quad \text{for } i = 1 \text{ to } p$$

where $\delta^{(1)} = x^{(k)}$ and A_i^+ is the Moore-Penrose pseudoinverse of A_i. Then the next iterate is computed as

$$x^{(k+1)} = \delta^{(p+1)}.$$

In contrast, the block Cimmino method obtains its next iterate through a sum of the projections as

$$\delta^{(i)} = A_i^+ \left(b_i - A_i x^{(k)}\right),$$

then updates the next iterate

$$x^{(k+1)} = x^{(k)} + \omega \sum_i^p \delta^{(i)}$$

with some relaxation parameter ω.

We notice that the two approaches are quite similar, however the block Cimmino approach is more amenable to parallelism as each projection can be computed independently from the others and in parallel.

We will describe in more detail the block Cimmino algorithm in Section 2.2. We recall the acceleration of this method using the conjugate gradient algorithm in Section 2.4 and we take a brief look at the stabilized block conjugate gradient algorithm as an accelerator in the same section. Finally, we will look at some numerical experiments in Section 2.5.

2.2 The block Cimmino method

Consider the partitioning (2.5) and let $P_{\mathcal{R}(A_i^T)}$ be the orthogonal projector onto the range of A_i^T defined as in (2.4) and A_i^+ the Moore-Penrose pseudoinverse of A_i defined as in (2.3). The block Cimmino algorithm can be described as in the Algorithm 2.1.

Algorithm 2.1 The block Cimmino method

Input: ω
  $x^{(0)} \leftarrow arbitrary$
  $k \leftarrow 0$
  loop
    for $i = 1 \rightarrow p$ **do**
      $\delta_i = A_i^+ \left(b_i - A_i x^{(k)}\right)$
    end for
    $x^{(k+1)} \leftarrow x^{(k)} + \omega \sum_{i=1}^{p} \delta_i$
    $k \leftarrow k + 1$
  end loop

Elfving [6] showed that if $0 < \omega < 2/\rho \left(\sum_{i=1}^{p} P_{\mathcal{R}(A_i^T)}\right)$ the method converges towards the minimum norm solution; for simplicity we consider ω as equal to one in the following. Moreover, the method converges for any generalized inverse, as Campbell showed in [9], as long as A_i has full row rank which implies the existence of $(A_i A_i^T)^{-1}$.

From a geometrical point of view, we see in Figure 2.1 the interpretation of the block Cimmino algorithm in the case of two partitions. It projects the current iterate on the different subspaces and then combines them with respect to the parameter ω to get the next approximation of the solution.

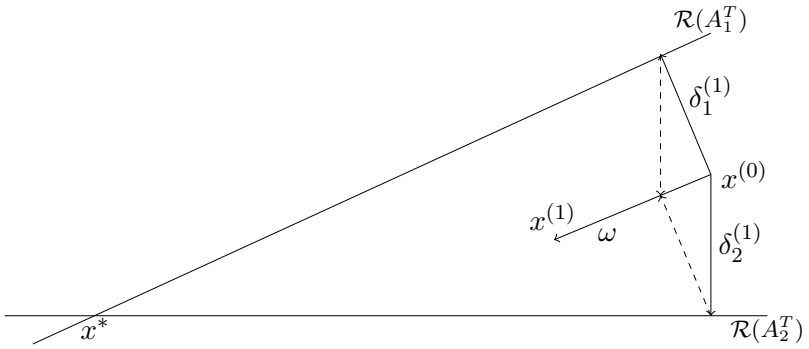


Figure 2.1: Geometric interpretation of the block Cimmino Algorithm with $p = 2$

The Algorithm 2.1 shows an important characteristic of the block Cimmino algorithm which is the independence in the computations of the different projections δ_i. This enables us to exploit two main levels of parallelism, the first one coming from the independent computation of the projections, and the second one from operations within this computation. We shall talk more about it in Chapter 4 when describing the hybrid parallelization of the block Cimmino method.

Iteration matrix. We define the iteration matrix Q of the block Cimmino iteration as

$$Q = I - H$$
$$\text{with} \qquad H = \sum_{i=1}^{p} P_{\mathcal{R}(A_i^T)} = \sum_{i=1}^{p} A_i^+ A_i.$$

It is obtained from the block Cimmino iteration defined by

$$x^{(k+1)} = x^{(k)} + \sum_{i=1}^{p} A_i^+ \left(b_i - A_i x^{(k)}\right) \tag{2.6}$$

$$= \left(I - \sum_{i=1}^{p} A_i^+ A_i\right) x^{(k)} + \sum_{i=1}^{p} A_i^+ b_i$$

$$= (I - H)\, x^{(k)} + \sum_{i=1}^{p} A_i^+ b_i$$

$$= Q x^{(k)} + \sum_{i=1}^{p} A_i^+ b_i. \tag{2.7}$$

The issue that arises now is how to compute the projections involved in the block Cimmino iteration. Four approaches can be used:

Normal equations of the partitions A_i can be used, in which it is only necessary to solve linear systems with the different $A_i A_i^T$ matrices at each iteration. However, this approach has two main drawbacks. First, building $A_i A_i^T$ induces a large preprocessing time to estimate the storage needed and can require excessive extra storage especially if A_i is sparse with some dense columns. The second drawback is that the conditioning corresponding to the L_2-norm $\kappa_2(A_i A_i^T)$ is the square of $\kappa_2(A_i)$ which is penalizing for ill-conditioned matrices.

QR **factorization** well known for its use with least-squares methods, avoids the formation of $A^T A$. In our case, consider the $Q_i R_i$ decomposition of A_i^T, we can compute $A_i^+ v$ as in

$$\begin{aligned} A_i^+ b_i &= A_i^T \left(A_i A_i^T\right)^{-1} b_i \\ &= Q_i R_i \left(R_i^T Q_i^T Q_i R_i\right)^{-1} b_i \\ &= Q_i R_i \left(R_i^T R_i\right)^{-1} b_i \\ &= Q_i R_i R_i^{-1} R_i^{-T} b_i \\ A_i^+ b_i &= Q_i R_i^{-T} b_i \end{aligned} \tag{2.8}$$

$$\text{and} \qquad A_i^+ A_i x = Q_i Q_i^T x. \tag{2.9}$$

Using this approach, one can factorize independently each partition and at each iteration compute the projection using (2.9). For the moment, there is no suitable sparse QR distributed solver available, therefore this approach has not yet been investigated numerically.

Seminormal equations are midway between normal equations and QR factorization. Suppose we have the QR factorization

$$A_i^T = Q_i R_i,$$

therefore, the normal equations can be written as

$$A_i A_i^T = R_i^T R_i$$

and solving a linear system with these amounts to solving

$$R_i^T R_i w = z.$$

We can therefore compute $A_i^+ v$ as

$$\begin{aligned} A_i^+ v &= A_i^T \left(A_i A_i^T\right)^{-1} v \\ &= A_i^T \left(R_i^T R_i\right)^{-1} v \\ &= A_i^T w \end{aligned}$$

$$\text{and} \qquad A_i^+ A_i x = A_i^T \left(R_i^T R_i\right)^{-1} A_i x.$$

Augmented systems are the simplest to use in our case. To obtain the projection

$$\delta_i = A_i^+ \left(b_i - A_i x^{(k)}\right),$$

we can solve the linear system

$$A_i \delta_i \; = \; r_i, \qquad (r_i = b_i - A_i x^{(k)})$$

using the augmented system approach:

$$\begin{pmatrix} I & A_i^T \\ A_i & 0 \end{pmatrix} \begin{pmatrix} u_i \\ v_i \end{pmatrix} \; = \; \begin{pmatrix} 0 \\ r_i \end{pmatrix}. \tag{2.10}$$

Whose solution is defined by

$$u_i = A_i^T \left(A_i A_i^T\right)^{-1} r_i \qquad \text{and} \qquad v_i = -\left(A_i A_i^T\right)^{-1} r_i.$$

By solving the linear system (2.10) at each iteration we obtain the different projections $\delta_i = u_i$. We use a sparse direct solver to factorize the augmented system and solve the linear systems at each iteration.

The augmented system approach has been proven to be more stable than the normal equations approach by Arioli et al. [13] as it avoids building and storing the normal equations for each partition, especially when reordering the matrix using the maximum weighted matching that creates oxo pivots in the LDL^T factorization of (2.10)'s matrix, as we will show in the next section.

2.3 Solving augmented systems using a direct solver

In this section we will look at the different aspects of solving augmented systems using a direct solver. The notations in this section (such as the names of the matrices and vectors) are independent from the rest of the manuscript. We consider the following system:

$$Ax = b$$

where x is the solution vector to the linear system, b a known vector and $A = (a_{ij})$ is an $n \times n$ symmetric matrix described as:

$$A = \begin{bmatrix} D & C \\ C^T & 0 \end{bmatrix}. \tag{2.11}$$

In contrast to iterative methods that look for the solution over a set of successive approximations, direct methods find the exact solution –relative to the machine precision and to the conditioning of A– to the linear system in a finite number of operations. However, direct methods could require huge amounts of memory to factorize large matrices, especially when matrices result from large 3D problems.

To solve the linear system (2.11) the direct solver goes through three phases [14]:

- *The analysis phase* in which it does a preprocessing to improve the structure of the matrix and/or its numerical properties for a faster and more stable factorization. It also estimates the amount of memory it requires for factorization and the number of floating-point operations needed.
- *The factorization phase* of the prepocessed matrix where it is decomposed into a product of two or three matrices easier to solve.
- *The solution phase* performs a set of operations to obtain the solution to the linear system using the matrices obtained during the factorization phase.

In this section we will give some details about the different algorithms involved in the three phases and which are important for maintaining a good performance when solving the augmented systems. More details on the background presented in this chapter can be found in [15, 16, 17].

2.3.1 Analyzing the matrix

The work done during the analysis can be divided into two kinds of preprocessing:

Structural preprocessing. As its name suggest, this preprocessing takes into account only the pattern (structure) of the nonzero entries in the matrix. Its main objective is either to reduce the fill-in during the factorization and therefore reduce the number of operations and the memory requirement, or to make the structure more adapted for a parallel factorization. The algorithms of this kind, like the *ordering* algorithms, are involved in a permutation process which reorders the matrix to achieve the goals previously listed. Alternately, using the structure of the matrix, they try to determine the structure of the factors and the storage space needed for it; this algorithm is called the *symbolic factorization* and was originally considered as an independent phase from the analysis as in [18].

Numerical preprocessing. The numerical preprocessing considers and uses the values of the nonzero entries in the matrix. The algorithms that belong to this kind of preprocessing try to apply transformations to the matrix making the factorization algorithm more stable due to fewer roundoff errors. The two main preprocessing steps involved in this kind of preprocessing are *scaling* and *maximum transversal* algorithms.

Before we start to describe the different steps involved in the analysis phase, we have to take a look at the roundoff propagation that arises during the factorization and that might lead to erroneous results. In [19], the authors give the following example. Assume a computer with 3 decimal digits floating-point representation, then the linear system

$$\begin{bmatrix} 0.001 & 1.000 \\ 1.000 & 2.000 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.000 \\ 3.000 \end{bmatrix}, \tag{2.12}$$

leads, after applying the Gaussian elimination process, to the matrices:

$$\hat{L} = \begin{bmatrix} 1.000 & 0.000 \\ 1000.000 & 1.000 \end{bmatrix} \qquad \hat{U} = \begin{bmatrix} 0.001 & 1.000 \\ 0.000 & -1000.000 \end{bmatrix}.$$

Using forward and backward substitution –that will be introduced later in this chapter– we get the solution $\hat{x} = (0, 1)^T$, whereas the real solution is $x = (1.002\ldots, 0.996\ldots)^T$ if computed with higher precision. This roundoff error occurs when the 0.0001 entry in the matrix is used as a pivot which is a small value relative to the off-diagonal entry of size 1. The methods discussed in this section will help to avoid this kind of situation.

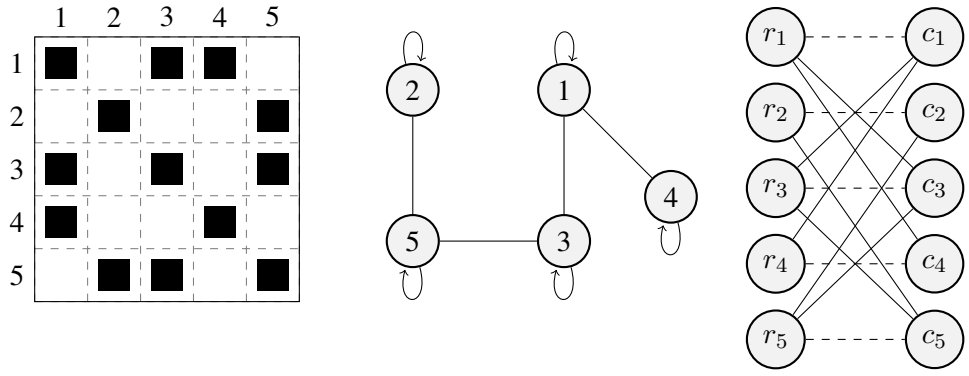
2.3.1.1 Maximum transversal

To introduce the maximum transversal, we have to go through some graph concepts.

Adjacency graph. We are familiar with the concept of the adjacency matrix that represents the connections between the vertices of a graph. An adjacency graph does the reverse thing, it represents the structure of the matrix. As presented in [16], we can represent the structure of a symmetric matrix A by the graph $G = (V, E)$ where $V = 1, 2, \ldots, n$ are the vertices representing the row/column number, and E are the edges where each pair (i, j) in E represents a nonzero element in the row i and column j and vice versa.

The same structure can be represented by a bipartite graph $G = (R, C, E)$, where $R = r_1, r_2, \ldots, r_n$, $C = c_1, c_2, \ldots, c_n$ and $(r_i, c_j) \in R \times C$ belongs to E if and only if $a_{ij} \neq 0$ where a_{ij} is the entry in the matrix A in row i and column j.

The Figure 2.2 shows how we can get the associated adjacency graph and the corresponding bipartite graph of a given symmetric matrix. For clarity, we represent diagonal elements in the bipartite graph with dashed lines.



A sparse matrix | Adjacency graph | Bipartite graph

Figure 2.2: Graph representation of a sparse matrix.

Matching. A matching in a bipartite graph is a set of edges with no pair of edges sharing a common vertex; this corresponds to a set of nonzero entries in which no pair are in the same row or column. A **maximum matching** is a matching with maximum cardinality. It is equal to n if the matrix is structurally nonsingular, it is called also **complete** –or perfect– matching.

In the same way, finding the **maximum weighted matching** amounts to finding a matching with maximum weight $w(\mathcal{M})$, where $\mathcal{M}$ is the concerned matching. In the unsymmetric case, it is done by finding the matching maximizing the product $\Pi_i a_{\sigma(i)i}$ over all permutations σ. Finding a maximum weighted matching can be used to put large entries onto the diagonal of the resulting permuted matrix. In Figure 2.3 we show a possible row permutation that puts large values on the diagonal.

However, if we want to keep the symmetry of the matrix we have to use a **symmetric maximum weighted matrix** introduced in [17]. In this matching, we compute the maximum weighted matching and then use it to get potential 1×1 and 2×2 pivots which will be exploited during the factorization.

Once the maximum weighted matching $\mathcal{M}$ is found, we store the diagonal nonzero entries that are in the matching into a set $\mathcal{M}_{1\times1}$ that holds potential 1×1 pivots. The author in [17] uses

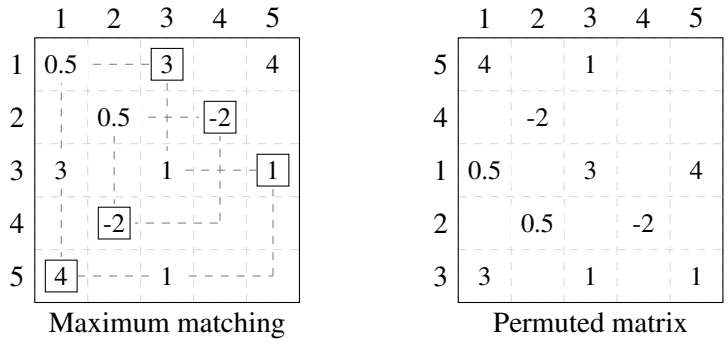


Figure 2.3: Unsymmetric weighted matching, dashed lines represent the permutation cycles.

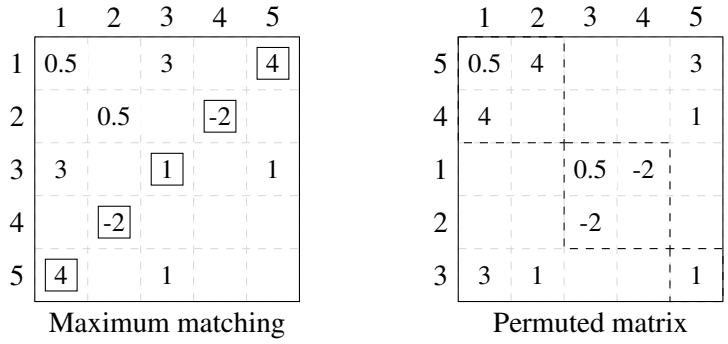


Figure 2.4: Symmetric weighted matching, dashed boxes represent the good potential pivots.

the strategy described in [20] to select 2×2 pivots and put them into a set $\mathcal{M}_{2\times 2}$. Using these two sets we create $\mathcal{M}_s = \mathcal{M}_1 \cup \mathcal{M}_2$ the symmetric matching corresponding to $\mathcal{M}$. The symmetric maximum matching can then be used to create row and column permutations to bring big values close to the diagonal which will form good initial 1×1 and 2×2 pivots during the factorization. Figure 2.4 shows a symmetric maximum weighted matching –left part– for the previous example in Figure 2.3. The corresponding permuted matrix is shown in the right part where the pivots are surrounded by thick dashed lines.

2.3.1.2 Scaling

In exact precision, an operation gives the correct result regardless of the values involved in that operation. However, due to the finite precision of floating-point arithmetic, round-off errors appear especially when there is a large difference in magnitude between these values.

Matrices containing entries with disparate magnitudes are called badly-scaled matrices. Dealing with such matrices may lead to performance issues, especially during the factorization.

As values are involved in this process it is considered as a numerical preprocessing. It mainly consists in left and right-multiplying the original matrix by diagonal matrices D_r and D_c respectively. If the matrix to be scaled is symmetric, the scaling matrices are equal. The original matrix A becomes then

$$\hat{A} = D_r A D_c,$$

which is used to solve a new linear system

$$\hat{A}\hat{x} = \hat{b}.$$

The new right-hand side is obtained by scaling the original right-hand side such as $\hat{b} = D_r b$ and the solution to the original linear system is obtained by scaling back the new solution vector such as $x = D_c^{-1}\hat{x}$.

The scaling process helps to make the coefficients of the system closer in magnitude, whereas the relationship between values in the matrix can be wide as illustrated by the system shown in equation (2.12). The scaling helps in the selection of the pivots during the factorization and thus affects the numerical stability of that process. In [16] the authors show the impact of good and bad scalings on the numerical results.

Different scaling algorithms exist and can use different norms in that process. We will use the HSL package `MC77` implementing the algorithm introduced by Knight, Ruiz and Uçar in [21] that tries to compute the scaling matrices so that the p-norms of all rows and columns of $D_r^{-1}AD_c^{-1}$ are close to 1. Notice here that this package gives the inverse of the scaling matrices.

We will discuss two main ways to get the scaling factors D_r and D_c in this section. The reader can get more details in [22, 21].

1. Using the maximum weighted matching. During the maximum transversal algorithm, it is possible also to compute scaling factors so that the scaled matrix has ones in absolute value on its diagonal and its other entries are less than or equal to one in absolute value. This is why the author in [22] describes it as an **I-matrix**. However, this algorithm does not preserve the symmetry of the matrix. The authors of [16] described an adaptation to avoid this problem. The main idea is summarized by:

Definition 2.1. *A matrix $B = (b_{ij})$ is said to satisfy the constraints of the method if and only if*

$$\exists \text{ a permutation } \sigma, \text{ such that } \forall i, |b_{i\sigma(i)}| = \| b_{*\sigma(i)} \|_\infty = \| b_{\sigma(i)*} \|_\infty = 1$$

Property 2.1. *Let $\mathcal{M}$ be the maximum matching of the symmetric matrix A and $D_r = (d_{r_i})$, $D_c = (d_{c_i})$ be the row and column scaling respectively. Let $D = (d_i) = \sqrt{D_r D_c}$, then DAD is a symmetrically scaled matrix that satisfies the constraints given by Definition 2.1.*

2. Simultaneous row and column scaling. We consider an iterative algorithm that scales the matrix to a doubly stochastic matrix see [21]. The scaled matrix uses the inverse of the scaling factors such that $\hat{A} = D_1^{-1}AD_2^{-1}$. We describe in more detail the scaling in Chapter 4 during the preprocessing phase. A parallel implementation of this algorithm has been proposed in [23], and tests have been carried out in [16] showing that the $p-$norm used in the algorithm indeed influences the factorization process, while showing that this method using the infinity norm gives good results 95% of the time whereas the maximum weighted matching is the best on 100% of the treated problems. They noticed also that bad scaling can cause severe numerical problems, increase the memory requirements and severely slow down the factorization.

2.3.1.3 Ordering

During the Gaussian elimination of sparse matrices, 1×1 pivots are used within the elimination as in

$$a_{ij} = a_{ij} - \frac{a_{ik}a_{kj}}{a_{kk}}.$$

If a_{ij} was zero before the elimination and a_{ik} and a_{kj} were nonzeros, then the updated a_{ij} becomes a nonzero. This newly created entry is called a *fill-in* and can cause severe memory problems

during the factorization process. Ordering –also called re-numbering– permutes the rows and columns of the matrix to reduce the fill-in and this is done by pre and post-multiplying the matrix by permutation matrices, that is $A_{ordered} = PAP^T$ with P the permutation matrix. This fill-in is linked to the order in which we do the elimination during the factorization process. A good ordering tries to keep the fill-in low and thus preserves the sparsity of the matrix. The matrix A on the left hand-side of Figure 2.5 results in a dense matrix after factorization. However, the reordered matrix PAP^T results in no fill-in.

$$A = \begin{pmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & & \\ \bullet & & \bullet & \\ \bullet & & & \bullet \end{pmatrix} \qquad PAP^T = \begin{pmatrix} \bullet & & & \bullet \\ & \bullet & & \bullet \\ & & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{pmatrix}$$

Figure 2.5: Matrix before and after ordering; A generates 6 fill-ins, whereas PAP^T generates no fill-in.

There are two main approaches to minimize this fill-in. The first is a local one that tries to minimize the fill-in at each step of the elimination process and is based on local heuristics like the minimum degree algorithm. The other approach treats the matrix as a whole (global approach) and tries to permute it to special structures such as the nested dissection algorithm. The ordering algorithms can be viewed as operating on the elimination graph whose initial state is equal to the adjacency graph of the matrix A.

1. Minimum degree is an algorithm [24, 18] using local heuristics. There also exists an efficient variant called Approximate Minimum Degree [25, 26]. At each step of the elimination process it selects the next node to be eliminated based on its degree (the number of nodes connected to it), thus selecting the node with minimum adjacent nodes in the current elimination graph. The permuted matrix in Figure 2.5 was obtained by using minimum degree algorithm.

2. Nested dissection is an algorithm in which we recursively dissect –cut– the graph into sets using separators. A separator S is a set of nodes that once removed, separates the graph into two distinct –disconnected– graphs. The algorithm consists in selecting a separator for the graph and reordering the matrix so that the nodes in the distinct graphs are numbered first then the separator nodes last. This leads to a bottom-right bordered structure of the permuted matrix. This way of cutting the graph restricts the fill-in to the diagonal and border blocks of the matrix [27, 28].

Ordering on the compressed graph. If we applied permutations using the maximum weight matching on the matrix as described earlier, we are able to use an ordering that will not break the resulting 2×2 pivots. Suppose that we have a matrix such as

$$A = \begin{pmatrix} \bullet & \bullet & & & \bullet \\ \bullet & & & & \bullet \\ & & \bullet & \bullet & \\ & & \bullet & & \\ \bullet & \bullet & & & \bullet \end{pmatrix},$$

where the 2×2 pivots are the two first diagonal blocks. An ordering on the compressed graph [17] consists in compressing the graph representing A so that the 2×2 pivots are grouped into a single vertex and their adjacency is the union of the two adjacencies for each node. The resulting graph is represented using its adjacency matrix R in Figure 2.6, the 2×2 pivots are represented with

a weight equal to 2, the 1×1 pivots with weights equal to 1 and the off-diagonal elements are represented by the symbol $\bullet$. If we apply an ordering on R, we are sure that the nodes 2 will not

$$R = \begin{pmatrix} 2 & & \bullet \\ & 2 & \\ \bullet & & 1 \end{pmatrix}.$$

Figure 2.6: Adjacency matrix corresponding to the compressed graph of the matrix A

be broken, and thus keeping the integrity of the maximum weighted matching.

2.3.2 Factorization and Solution

Factorizing a matrix amounts to decomposing the matrix A into a product of matrices then used by the solution phase. Different factorizations exist; we focus on factorizations based on Gaussian elimination. Depending on the property of the matrix (symmetric, diagonally dominant, positive definite) we can perform the following decompositions:

- LU factorization: the matrix A is decomposed as $A = LU$ with L a lower triangular matrix with unity diagonal, and U an upper triangular matrix,
- Cholesky factorization: when A is symmetric positive definite we can decompose it in the form $A = LL^T$, where L is a lower triangular matrix,
- LDL^T factorization: for general symmetric matrices, we decompose it in the form $A = LDL^T$, where L is a lower triangular matrix and D is a diagonal matrix with 1×1 and 2×2 diagonal blocks. As augmented systems are indefinite, we will use this approach to factorize them.

Consider the symmetric $n \times n$ matrix A and suppose that it has the following structure

$$A = \begin{bmatrix} P & C^T \\ C & B \end{bmatrix},$$

where P is an $s \times s$ matrix, C is $(n-s) \times s$ and B is $(n-s) \times (n-s)$, with $s = 1$ or 2. We define also $v = |det(P)| = |p_{11}p_{22} - p_{21}^2|$ the determinant of the matrix P, when $s = 2$.

If $v \neq 0$, then P^{-1} exists and we have:

$$\begin{aligned} A = A_0 &= L_1 D L_1^T \\ &= \begin{bmatrix} I & 0 \\ CP^{-1} & I \end{bmatrix} \begin{bmatrix} P & 0 \\ 0 & A_1 \end{bmatrix} \begin{bmatrix} I & P^{-1}C^T \\ 0 & I \end{bmatrix}. \end{aligned}$$

Applying the previous step recursively on $A_1, A_2, \ldots$ we obtain:

$$A = (L_1 L2 \ldots L_{n-1}) D (L_{n-1}^T \ldots L_2^T L_1^T).$$

Notice that D is a block-diagonal matrix with P being the diagonal matrix at each step of the decomposition. When $s = 1$ we call P a 1×1 pivot, when it is equal to 2, we call P a 2×2 pivot.

During the process of factorization, we use symmetric permutations and numerical pivoting to ensure that the P matrices have good numerical stability and avoid a zero block. Using the maximum weighted matching might provide good pivots and therefore improves the factorization

performance. The author in [17] presents a solution to obtain stable pivots that fall into four categories:

$$\begin{array}{cccc} \begin{pmatrix} x \end{pmatrix} & \begin{pmatrix} x & x \\ x & x \end{pmatrix} & \begin{pmatrix} 0 & x \\ x & 0 \end{pmatrix} & \begin{pmatrix} x & x \\ x & 0 \end{pmatrix} \\ 1 \times 1 \text{ pivot} & \text{full } 2 \times 2 \text{ pivot} & oxo \text{ pivot} & Tile \text{ pivot} \end{array}$$

Once the matrix is factorized, the solution phase goes through two steps called **forward** and **backward** substitution. The former corresponds to solving the systems $Ly = b$ and $Dz = y$, and solving $L^T x = z$ in the later to obtain the solution.

2.4 The block conjugate gradient acceleration of block Cimmino

The convergence of the row projection techniques presented in the previous section is known to be slow even after selecting the optimal parameter ω. Hageman and Young showed in [29] that polynomial acceleration such as Chebyshev polynomials or CG acceleration can be used to improve convergence of iterative techniques. Moreover, CG acceleration converges fast enough and does not require any parameter estimate (the largest and the smallest eigenvalue are required in the case of Chebyshev acceleration).

The classical CG of Hestenes and Stiefel [30] can be applied to linear systems whose matrices are symmetric positive definite (SPD). It converges, in the absence of roundoff errors, in a finite number of steps.

Bramley and Sameh [7] used CG to accelerate Kaczmarz and Cimmino methods. They showed that, in practice, these accelerations converge faster than GMRES and CGNE. They also showed that these accelerated row projection methods have a high robustness and can successfully solve large linear systems.

Let us consider the iterative scheme of the block Cimmino method (2.7)

$$x^{(k+1)} = Qx^{(k)} + \sum_{i=1}^{p} A_i^+ b_i,$$

where $Q = I - H$ is the iteration matrix and $H = \sum_{i=1}^{p} A_i^T (A_i A_i^T)^{-1} A_i$. H can be written as

$$H = A^T D^{-1} A,$$

where D is a block diagonal matrix

$$\begin{pmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_p \end{pmatrix}$$

in which each block

$$D_i = (A_i A_i^T)^{-1}$$

is symmetric positive definite. Therefore, if A is square and has full rank then the matrix H is symmetric positive definite as is Q. If it is not the case, then H will be symmetric positive semidefinite. We will study later in Chapter 5 the use of CG on semidefinite systems.

Accelerating the block Cimmino method using CG amounts to solving the linear system

$$Hx = k, \tag{2.13}$$

where $k = \sum_{i=1}^{p} A_i^+ b_i$ and x is the same solution vector as for the original linear system $Ax = b$.

The CG acceleration algorithm is described in Algorithm 2.2. In lines 5 and 7 we use $Hp^{(j)}$ which is defined as

$$Hp^{(j)} = \sum_{i=1}^{p} A_i^+ A_i p^{(j)}$$

and is equivalent to solving the augmented systems presented in (2.10) with $r_i = A_i p^{(j)}$ i.e.,

$$\begin{pmatrix} I & A_i^T \\ A_i & 0 \end{pmatrix} \begin{pmatrix} u_i \\ v_i \end{pmatrix} = \begin{pmatrix} 0 \\ A_i p^{(j)} \end{pmatrix},$$

the sum of u_i, the upper part of the solutions, is $Hp^{(j)}$.

Algorithm 2.2 CG acceleration of block Cimmino

1: $x^{(0)}$ is arbitrary
2: $r^{(0)} \leftarrow k - Hx^{(0)}$
3: $p^{(0)} \leftarrow r^{(0)}$
4: **for** $j = 0, 1, 2..$ until convergence **do**
5: $\quad \lambda_j \leftarrow \left(r^{(j)^T} r^{(j)}\right) / \left(p^{(j)^T} H p^{(j)}\right)$
6: $\quad x^{(j+1)} \leftarrow x^{(j)} + p^{(j)} \lambda_j$
7: $\quad r^{(j+1)} \leftarrow r^{(j)} - H p^{(j)} \lambda_j$
8: $\quad \alpha_j \leftarrow \left(r^{(j+1)^T} r^{(j+1)}\right) / \left(r^{(j)^T} r^{(j)}\right)$
9: $\quad p^{(j+1)} \leftarrow r^{(j+1)} + p^{(j+1)} \alpha_j$
10: **end for**

In the second line of Algorithm 2.2, when computing the initial residual

$$r^{(0)} = k - Hx^{(0)} = \sum_{i=1}^{p} A_i^+ \left(b_i - A_i x^{(0)}\right),$$

we sum the u_i resulting from solving the linear systems

$$\begin{pmatrix} I & A_i^T \\ A_i & 0 \end{pmatrix} \begin{pmatrix} u_i \\ v_i \end{pmatrix} = \begin{pmatrix} 0 \\ b_i - A_i x^{(0)} \end{pmatrix}.$$

We introduce *Projections Sum* (PS) described in Algorithm 2.3 to describe the sum of projections. It takes as input two vectors w and z and computes

$$PS_A(w, z) = \sum_{i=1}^{p} A_i^+ (w_i - A_i z), \tag{2.14}$$

where w_i is a partition of w according to the partition A_i. In this way we can rewrite $Hp^{(j)}$ and k as

$$\begin{aligned} Hp^{(j)} &= PS_A(0, p^{(j)}) \\ k &= PS_A(b, 0). \end{aligned}$$

The convergence of the CG acceleration can sometimes be slow when the CG iteration matrix, H, has small clusters of eigenvalues as we will see in practical examples in Section 2.5. CG finds it hard to target them and the convergence curve will often have long plateaux. Golub, Ruiz and Touhami [31] studied a combination of Chebyshev filters with conjugate gradients to accelerate its convergence by targeting the small clusters of eigenvalues directly. They showed

Algorithm 2.3 Projections Sum (PS_A)

Input: w and z

Output: $\sum_{i=1}^{p} A_i^+ (w_i - A_i z)$

$\delta \leftarrow \begin{bmatrix} 0 & \dots & 0 \end{bmatrix}^T$

for $i = 1 \rightarrow p$ **do**

$\begin{bmatrix} u_i \\ v_i \end{bmatrix} \leftarrow direct\_solve \left(\begin{bmatrix} I & A_i^T \\ A_i & 0 \end{bmatrix} \begin{bmatrix} u_i \\ v_i \end{bmatrix} = \begin{bmatrix} 0 \\ w_i - A_i z \end{bmatrix} \right)$ /* Forward and backward substitution */

$\delta \leftarrow \delta + u_i$

end for

return δ

that this approach can be helpful when solving several linear systems that have the same matrix with different right-hand sides. This is of particular interest in the case of the block Cimmino acceleration, since the block Cimmino iteration matrix H has a good clustering of eigenvalues around 1, but with some trailing eigenvalues clustered at the extremes of the spectrum. Arioli, Duff, Ruiz and Sadkane in [8] introduced a block version of the CG acceleration (Block-CG), and stabilized it for large block sizes, as given in Algorithm 2.4.

The stabilization process is performed at the pairs of lines 11, 12 and 17, 18. The issue with non-stabilized Block-CG algorithm is that the residual $R^{(j+1)}$ matrices are maintained orthogonal block-wise (e.g. $R^{(j)} \perp R^{(j+1)}$) but not within the blocks and thus the $R^{(j+1)^T} R^{(j+1)}$ matrices can become very ill-conditioned as the algorithm converges. To avoid this problem, the authors in [32] enforced the stability of the method by forcing the matrices $\bar{R}^{(j+1)}$ to have orthogonal columns as in

$$\gamma_{j+1}^T \gamma_{j+1} = chol \left(R^{(j+1)^T} R^{(j+1)} \right)$$

where $chol$ is the Cholesky decomposition and γ_{j+1} is the resulting upper triangular matrix. By setting

$$\bar{R}^{(j+1)} = R^{(j+1)} \gamma_{j+1}^{-1}$$

we obtain

$$\begin{aligned} \bar{R}^{(j+1)^T} \bar{R}^{(j+1)} &= \gamma_{j+1}^{-T} \left(R^{(j+1)^T} R^{(j+1)} \right) \gamma_{j+1}^{-1} \\ &= \gamma_{j+1}^{-T} \gamma_{j+1}^T \gamma_{j+1} \gamma_{j+1}^{-1} = I \end{aligned} \tag{2.15}$$

This property shown in (2.15) is, in practice, what maintains the stability of the algorithm and is also used for the P matrices making them H-orthogonal. The stability of this algorithm has been studied by the same authors and they showed that it deteriorates on really large block-sizes due to the fact that the conditioning increases once the algorithm reaches superlinear convergence; moreover some columns might converge earlier than the others in the set of linear systems solved simultaneously, see O'Leary [33].

If the Cholesky decomposition fails due to ill-conditioning, we employ a fall-back solution using the orthogonalization process described in [34] based on a modified Gram-Schmidt that avoids computing square roots. However, if some vectors converge earlier than the others, we can use a reduction of block-size to avoid issues as suggested in [33]. Nevertheless, we did not observe (nor did the authors in [8]) any stability issues for block-sizes with reasonable size (smaller than or equal to 32).

Regarding the complexity of this algorithm, we notice that we are able to avoid some duplicate computations such as $H\bar{P}^{(j)}$ at line 9 and $HP^{(j+1)}$ at line 17 by storing the latter for

Algorithm 2.4 Stabilized Block-CG acceleration

1: $X^{(0)}$ is arbitrary
2: $R^{(0)} = K - HX^{(0)}$
3: $\gamma_0^T \gamma_0 = chol\left(R^{(0)^T} R^{(0)}\right)$
4: $\bar{R}^{(0)} = R^{(0)} \gamma_0^{-1}$
5: $\beta_0^T \beta_0 = chol\left(\bar{R}^{(0)^T} H \bar{R}^{(0)}\right)$
6: $\bar{P}^{(0)} = \bar{R}^{(0)} \beta_0^{-1}$
7: **for** $j = 0, 1, 2..$ until convergence **do**
8: $\quad \lambda_j = \beta_j^{-T}$
9: $\quad R^{(j+1)} = \bar{R}^{(j)} - H\bar{P}^{(j)} \lambda_j$
10: $\quad$ /* Starting stabilization */
11: $\quad \gamma_{j+1}^T \gamma_{j+1} = chol\left(R^{(j+1)^T} R^{(j+1)}\right)$
12: $\quad \bar{R}^{(j+1)} = R^{(j+1)} \gamma_{j+1}^{-1}$
13: $\quad$ /* Ending stabilization */
14: $\quad \alpha_j = \beta_j \gamma_{j+1}^T$
15: $\quad P^{(j+1)} = \bar{R}^{(j+1)} + \bar{P}^{(j)} \alpha_j$
16: $\quad$ /* Starting stabilization */
17: $\quad \beta_{j+1}^T \beta_{j+1} = chol\left(P^{(j+1)^T} H P^{(j+1)}\right)$
18: $\quad \bar{P}^{(j+1)} = P^{(j+1)} \beta_{j+1}^{-1}$
19: $\quad$ /* Ending stabilization */
20: $\quad X^{(j+1)} = X^{(j)} + \bar{P}^{(j)} \lambda_j \left(\prod_{i=j}^{0} \gamma_i\right)$
21: **end for**

the next iteration. Indeed, in the next iteration, $H\bar{P}^{(j)}$ is equivalent to the current iteration's $\left(HP^{(j+1)}\right)\beta_{(j+1)}^{-1}$.

We describe in Algorithm 2.5 the function that computes the stabilized matrices that takes the matrices D and W and returns the stabilization matrix U, the stabilized matrices $\bar{D}$ and $\bar{W}$ corresponding to $\bar{P}^{(j+1)}$ and $H\bar{P}^{(j+1)}$ in the case of $P^T HP$ and $\bar{R}^{(j+1)}$ in the case of $R^T R$.

At lines 2 and 4, we compute the dense matrix-matrix product depending on the case, then we factorize the resulting matrix at line 6 and stabilize the matrices on lines 7 and 9.

Algorithm 2.5 Stabilization: stab()

Input: D and W
Output: $U, \bar{D}$ and $\bar{W}$
1: **if** $D \neq W$ **then**
2: $\quad C \leftarrow D^T W$
3: **else**/* Handle the case of $D^T D$ */
4: $\quad C \leftarrow D^T D$
5: **end if**
6: $U \leftarrow chol(C)$
7: $\bar{D} \leftarrow DU^{-1}$
8: **if** $D \neq W$ **then**
9: $\quad \bar{W} \leftarrow WU^{-1}$
10: **end if**

If we use this algorithm for $P^T HP$, with inputs $P^{(j+1)}$ and $HP^{(j+1)}$, we will obtain $\bar{P}^{(j+1)}$ and $H\bar{P}^{(j+1)}$ as results. The latter, as said previously, is the matrix $H\bar{P}^{(j)}$ of the next itera-

tion, therefore by storing the matrix temporarily we avoid the computation of this product (which requires a call to the direct solver).

Algorithm 2.6 Stabilized Block-CG acceleration

1: $X^{(0)}$ is arbitrary
2: $R^{(0)} = K - HX^{(0)}$
3: $\left[\gamma_0, \bar{R}^{(0)}\right] = stab\left(R^{(0)}\right)$
4: $P^{(0)} = R^{(0)}$
5: $k = 0$
6: **while** $k < it\_max$ **do**
7: $\quad\left[\beta_j, \bar{P}^{(j)}, H\bar{P}^{(j)}\right] = stab\left(P^{(j)}, HP^{(j)}\right)$
8: $\quad\lambda_j = \beta_j^{-T}$
9: $\quad X^{(j+1)} = X^{(j)} + \bar{P}^{(j)}\lambda_j\left(\prod_{i=j}^{0}\gamma_i\right)$
10: $\quad R^{(j+1)} = \bar{R}^{(j)} - H\bar{P}^{(j)}\lambda_j$
11: $\quad$**if** $converged$ **then**
12: $\quad\quad break$
13: $\quad$**end if**
14: $\quad\left[\gamma_{j+1}, \bar{R}^{(j+1)}\right] = stab\left(R^{(j+1)}\right)$
15: $\quad\alpha_j = \beta_j\gamma_{j+1}^T$
16: $\quad P^{(j+1)} = \bar{R}^{(j+1)} + \bar{P}^{(j)}\alpha_j$
17: $\quad k = k + 1$
18: **end while**

We update the previous version of the stabilized block-CG algorithm using these results to obtain Algorithm 2.6. We will use the function $stab()$ defined in Algorithm 2.5 with only one parameter in the case when D and W are equal. We also use the Matlab notation $[\dots]$ to extract multiple return values from the function $stab$.

Algorithm 2.6 presents aesthetic simplifications compared to the former one. The stabilization of $P^{(0)}$ is now done inside the loop at line 7 followed by an immediate update of the iterate. This update used to be at the end of the loop even if the data required was available beforehand. We also extracted the $H\bar{P}$ matrix from the stabilization process to be used at line 10 as described previously.

Classical vs. Block version. The Block-CG follows the same steps as the classical CG and improves it by adding orthogonalization steps. In addition to that, it does matrix-matrix operations rather than matrix-vector operations because it uses multiple right-hand sides. Therefore, the complexity of the Block-CG is increased and so is the floating-point operations count (FLOP) per iteration. We look now at where this increase of FLOP happens and if it can be prohibitive.

Orthogonalization. During the orthogonalization, we do two kind of operations:

- First, a matrix-matrix product corresponding to D^TW is done instead of a dot product in the case of classical CG. For that purpose we use the BLAS kernels `GEMM` instead of the `DOT` routine. This offers an increase in performance due to the fact that `GEMM` is a level 3 BLAS kernel compared to the level 1 BLAS kernel `DOT`. Depending on the block-size, the matrix-matrix product might out-perform the simple dot product due to the level 3 BLAS kernel efficiency. However, increasing the block-size further can and will decrease the performance. These parameters are machine dependent as is the BLAS implementation,

therefore some experiments have to be done to find the best block-size with respect to time per iteration and the gains resulting from the possible reduction in the number of iterations for convergence.

- The second operation is the Cholesky factorization of the matrix obtained from the previous matrix-matrix product. It is safe to say that due to the small size of the block under consideration (because it is not useful in general to use too large block-sizes), the time for factorizing the resulting matrix does not vary so much on a modern architecture for the range of block sizes under consideration (i.e. 1 to 32). Therefore, this step is not really penalizing.

Sum of Projections. The other change concerns the computation of the HP product. This product, as stated previously, is a succession of three steps:

- First we compute the product A_iP where i is the partition number. Increasing the block size will also change the operations to be performed, and in this case we use level 3 SpBLAS kernels (SpBLAS stands for Sparse BLAS) that will benefit from the block-size compared to a level 2 SpBLAS kernel as has been shown in [35].

- The second step is the solution of multiple right-hand sides within the direct solver. The performance is usually increased with larger block-sizes during the forward and backward substitutions, where level 3 BLAS kernels are used. Moreover, modern architectures have a high rate of FLOP per seconds, therefore exploiting that rate at a maximum will bring the most benefit from increasing the block-size.

- The last step is the sum of the solutions of the augmented systems, and it is related to the total number of elements to be summed. It can be penalized by the increase in block-size especially in the parallel case where we have to send parts of the solution between different processes. We discuss this part in more detail in Chapter 4 when we study the implementation of the $PS_A()$ algorithm.

2.5 Convergence behaviour

We assume that the partitions of the system (2.1) are of similar size. We study in this section how the ill-conditioning of the matrix A can affect the corresponding iteration matrix of block Cimmino and therefore affect the convergence.

To illustrate this behaviour we use a realistic test matrix `bayer01` obtained from Bayer AG by Friedrich Grund and available from the sparse matrix collection of the University of Florida [36]. It is of order 57735 and has 277774 nonzero entries. We partition it into 16 uniform partitions, and we show its pattern in Figure 2.7.

In Figure 2.8, we show the spectrum of the iteration matrix for the `bayer01` matrix when using block Cimmino with these 16 uniform partitions.

We will use figures like Figure 2.8 in the following to show the spectrum of iteration matrices. The eigenvalues are distributed from the largest at the right to the smallest at the left. Colour wise, the deep blue is the largest and the red is the smallest. Two red lines are used as markers to give an idea how many small eigenvalues there are, the left marker is for eigenvalues smaller than $\lambda_{max}/100$ and the right one for eigenvalues smaller than $\lambda_{max}/10$.

In Figure 2.8 we notice that there is a good clustering of eigenvalues around the value 1. This property is common among the iteration matrices of block Cimmino. However, we notice that there is a subset of small eigenvalues making it badly conditioned. This, combined with the spread of intermediate eigenvalues, more or less large depending on the size of the system and the

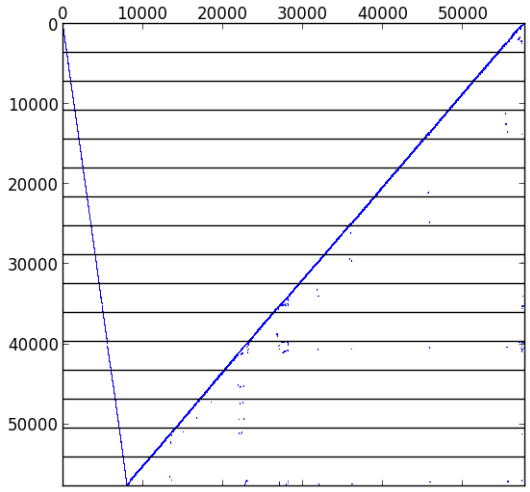


Figure 2.7: Nonzero pattern of the matrix `bayer01`.

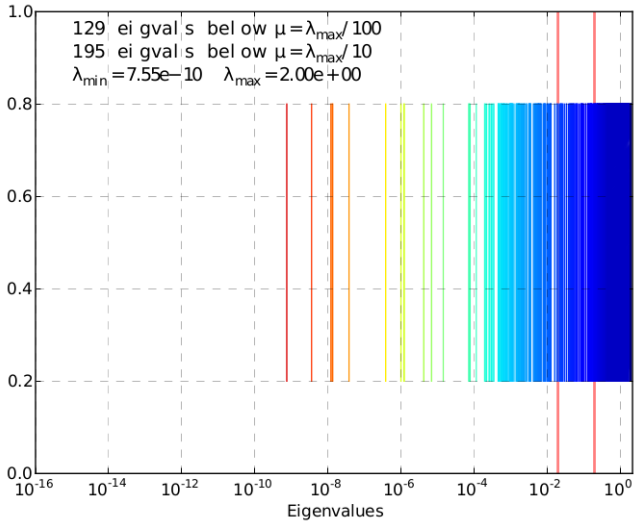


Figure 2.8: Spectrum of block Cimmino iteration matrix for `bayer01` with 16 uniform partitions.

partitioning, make it hard for the CG to converge quickly. We will discuss several ways to reduce this ill-conditioning in Chapter 3.

We shall next compare the convergence history for different block sizes for the `bayer01` matrix. To monitor convergence, we use in the following

$$\omega_k = \frac{\|r^{(k)}\|_\infty}{\|A\|_\infty \|x^{(k)}\|_1 + \|b\|_\infty},$$

where $r^{(k)} = Ax^{(k)} - b$ is the residual of the original system at the k-th iteration. We are therefore checking for convergence of the original system using the iterate $x^{(k)}$ resulting from the CG or the Block-CG acceleration.

We show in Figure 2.9 the number of iterations required to converge for the matrix `bayer01`. The main issue mentioned previously with the eigenvalue distribution, influences the convergence behaviour and gives it the plateaux-oriented convergence that we see in the classical CG case.

However, by using the stabilized Block-CG we notice that the plateaux have been reduced. The more we increase the block-size, the smaller the plateaux become and the faster we converge. In fact, a reasonably large block-size can reduce the iteration count substantially from 4473 iterations for the CG acceleration to 68 iterations with block-size 32. Results are summarized in Table 2.1.

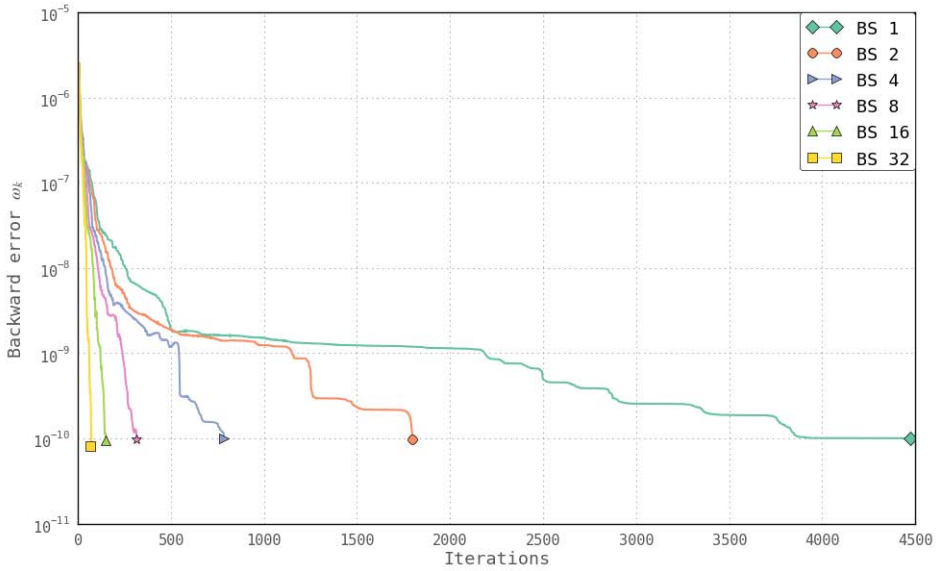


Figure 2.9: Number of iterations needed to converge for different block sizes for the matrix `bayer01` with 16 uniform partitions.

The number of iterations is important but so is the amount of work needed per iteration. We show in Figure 2.10 the number of matrix-vector operations needed to converge. Indeed, each iteration requires as many matrix-vector products as the block-size in use (as said earlier, these matrix-vector products are transformed into a matrix-matrix product that exploits level 3 BLAS kernels). We notice that increasing the block-size reduces the amount of work until it is halved and is more or less stabilized for a block-size of 32.

Block-size	**Nb. iter.**	**M-V ops.**	**Work ratio to 1**	**Total Time**	**Timing ratio to 1**
CG	4473	4473	1.00	567s.	1.00
2	1799	3598	0.80	395s.	0.70
4	784	3136	0.70	221s.	0.39
8	315	2520	0.56	128s.	0.23
16	148	2368	0.53	127s.	0.22
32	68	2176	0.49	120s.	0.21
64	29	1856	0.41	123s.	0.22
128	16	2048	0.45	140s.	0.25

Table 2.1: Convergence results for the `bayer01` matrix with the number of iterations, the work corresponding to the number of matrix-vector operations, the ratio of work with respect to CG, the total time to convergence and finally the ratio of timings with respect to the time for the CG to converge.

To get more details on how the block-size affects both the convergence and the timing to achieve it, we show in Table 2.1 the results when changing the block-size from 1 (classical CG) to 128. Notice that, for this problem, increasing the block-size decreases the amount of matrix-vector operations needed for convergence until block-sizes 32/64 but then goes in the reverse direction

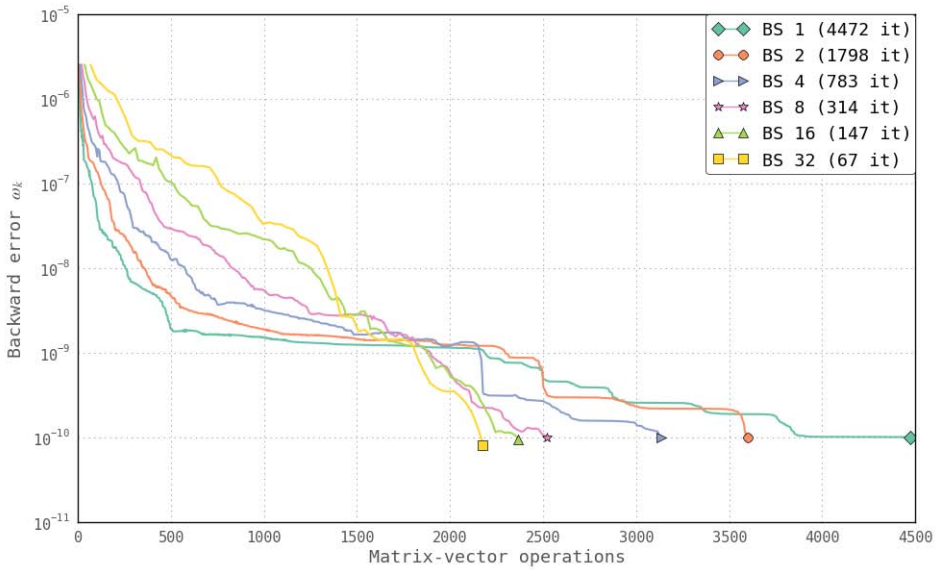


Figure 2.10: Number of matrix-vector operations needed to converge for different block sizes for the matrix `bayer01` with 16 uniform partitions.

for larger block-sizes. We see from the 4th column in this table that the corresponding reduction ratio of the amount of work is quite stable with few reductions at the end.

The timings are presented in the last two columns. We notice that they follow the decrease in amount of work until block-size 32 then rise afterwards. The most important part in that decrease is that it is faster in the beginning due to level 3 BLAS effects that reduce the matrix-vector timings.

We show in Table 2.2 the time per iteration when increasing the block-size, the ratio to the previous block-size and the ratio to classical CG (block-size 1). As long as the ratio to block-size 1 is lower than the block-size tested then results benefit from level 3 BLAS kernels. The *ratio to previous* tested block-size shows the improvements during the increase, if it is lower than 2 then we have improvements, if it is higher then we do not.

Block-size	1	2	4	8	16	32	64	128
Avg. time per iteration	0.13	0.22	0.28	0.41	0.86	1.77	4.22	8.75
Ratio to block-size 1		1.69	2.15	3.15	6.62	13.62	32.46	67.31
Ratio to previous		1.69	1.27	1.46	2.10	2.06	2.38	2.07

Table 2.2: Average time per iteration for `bayer01`.

For this problem, and on this computer, we notice that we have no more gains after a block-size of 16, and the gains noticed previously come only from the reduction in the amount of work. These results are problem dependent and architecture dependent. Having a larger cache and a smaller matrix will give better results on larger block-sizes, but this will always have an upper limit after which the performance will degrade.

Numerical experiments. We look now at a larger set of matrices and see how block Cimmino behaves on different problem orders and applications. The matrices are listed in Table 2.3 which are available from the sparse matrix collection of the University of Florida [36]. The matrices are listed by increasing order. We list the number of entries they contain and the number of partitions

we create from those matrices. The number of partitions is proportional to the order of the matrix. In Chapter 3 we study different strategies to create and to select the partitions. In the case of `cage13` and `cage14`, the number of partitions is high compared to the other matrices, this is because their augmented systems are quite difficult to factorize by the direct solver unless the partitions are small.

Problem	**Order**	**Nonzeros**	**Partitions**	**Application**
`gre_1107`	1107	5664	5	Directed weighted graph
`bayer01`	57 735	277 774	16	Chemical process simulation problem
`lhr71c`	70 304	1 528 092	16	Chemical process simulation problem
`torso3`	259 156	4 429 042	32	Electro-phys, 3D model
`cage13`	445 315	7 479 343	256	Directed weighted graph
`Hamrle3`	1 447 360	5 514 242	64	Circuit simulation
`cage14`	1 505 785	27 130 349	1024	Directed weighted graph

Table 2.3: Matrices to be tested. Coming from different applications, of different sizes and symmetry.

We show in Figures 2.11 to 2.17 the number of matrix-vector operations for each block-size and the time it takes to converge. We stop the block-CG after 5000 iterations. The runs are made in sequential mode on `Conan` (`AMD` Opteron processor) with no threaded BLAS.

From these results, we notice three kinds of behaviour. If the classical CG converges fast enough then increasing the block-size will usually give worse performance as we see for the `torso3` and the `cage` matrices. If the decrease in the number of iterations when increasing the block-size is not enough, the number of matrix-vector operations will increase and this will slow down the convergence. Moreover, if there is a decrease in amount of work due to a large gain in number of iterations, but the time per iteration increases excessively, then the convergence will slow down too.

In the other cases, where classical CG converges slowly, increasing the block-size helps to some extent as is seen for `gre_1107`, `bayer01` and `Hamrle3`. In these cases, we notice that we can achieve a large decrease in the amount of work which will speedup the convergence. However, due to the fact that the time per iteration will increase, once we increase the block-size further, the convergence will slow down. In the case of `lhr71c`, we obtained the best timings with block-size 32 because of smaller block-sizes result in poorer convergence.

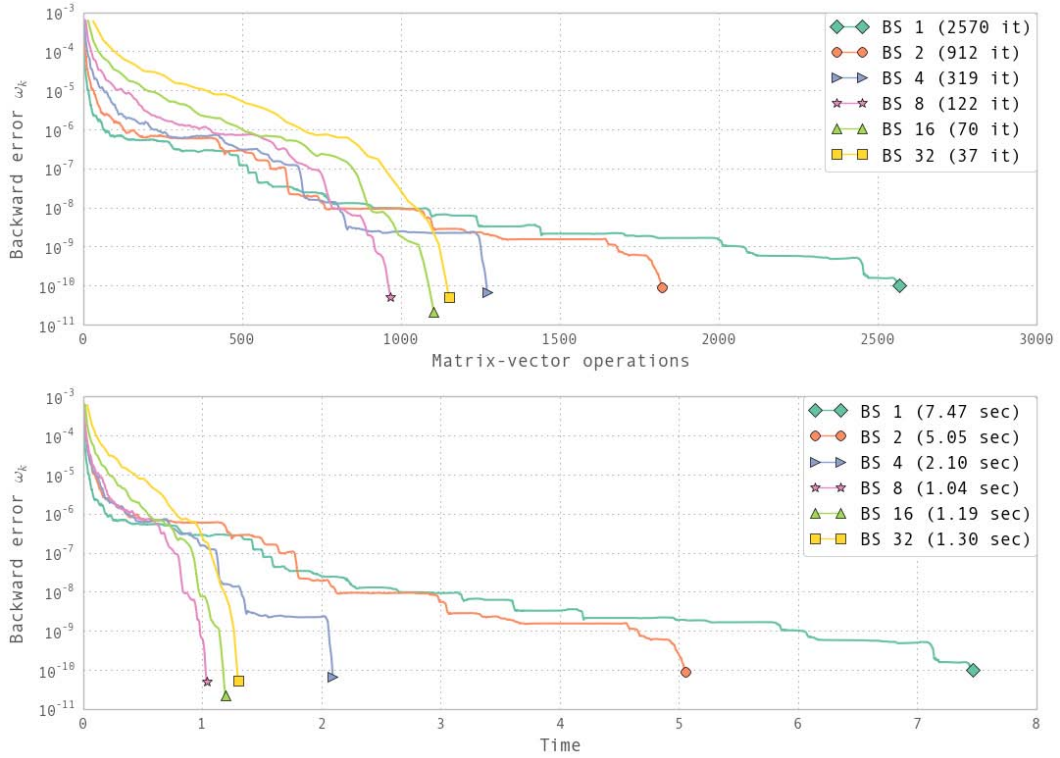


Figure 2.11: Convergence and timing comparison for the matrix `gre_1107` with different block-sizes.

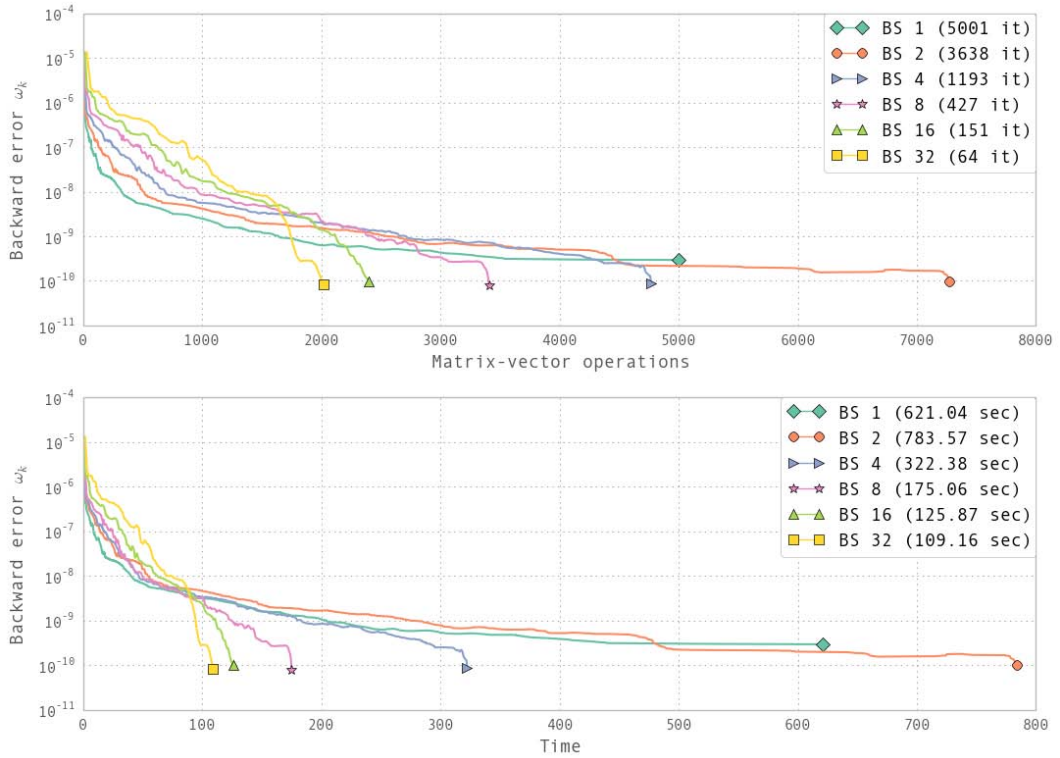


Figure 2.12: Convergence and timing comparison for the matrix `bayer01` with different block-sizes.

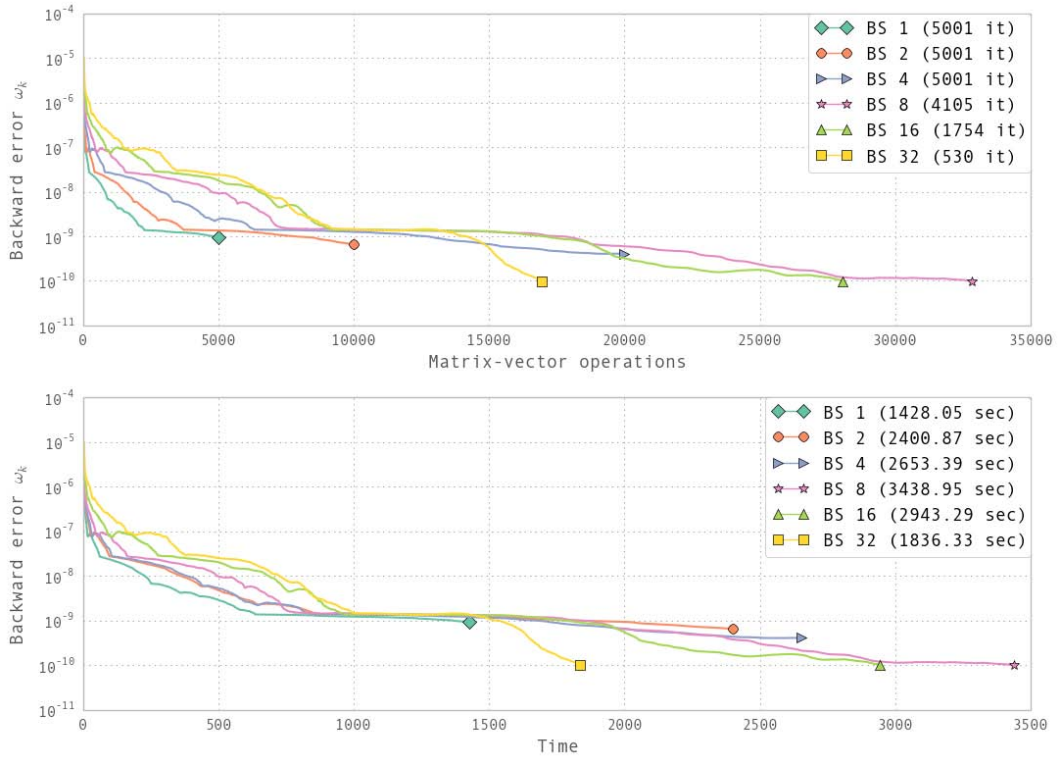


Figure 2.13: Convergence and timing comparison for the matrix `lhr71c` with different block-sizes.

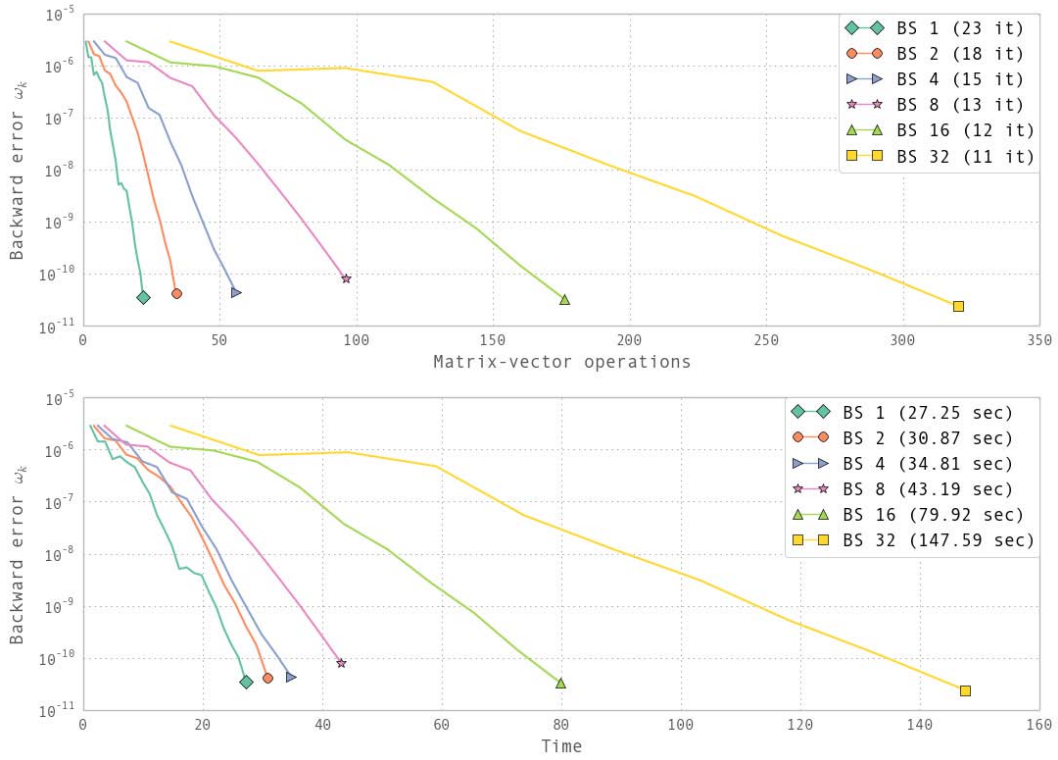


Figure 2.14: Convergence and timing comparison for the matrix `torso3` with different block-sizes.

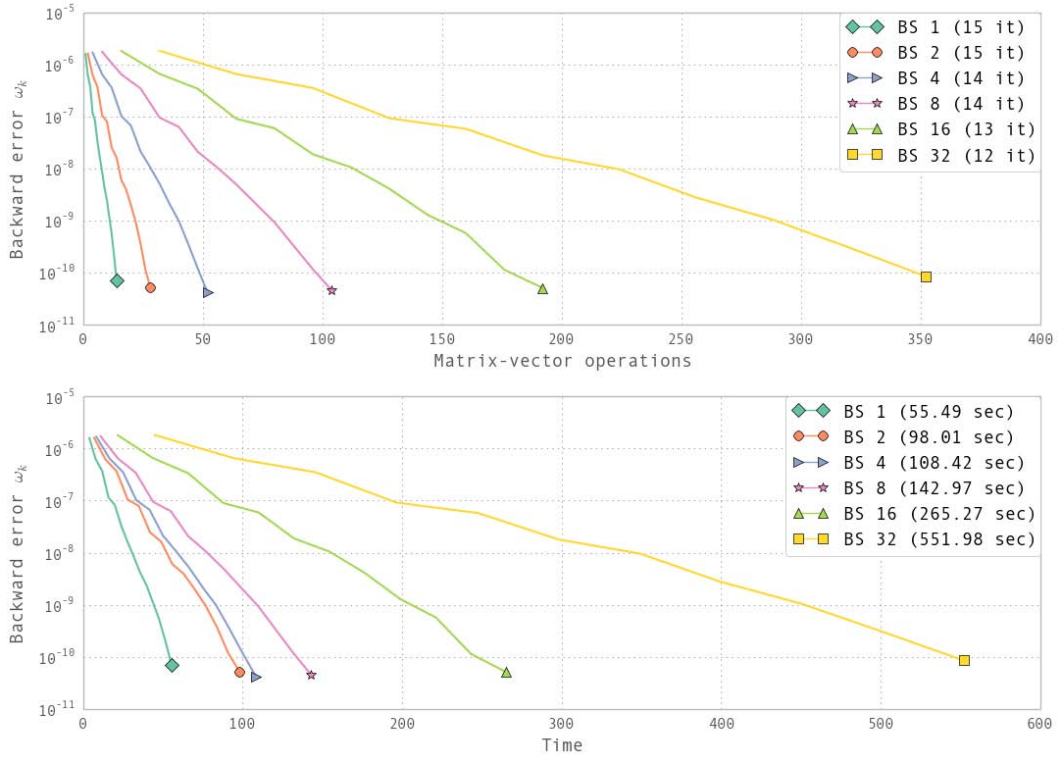


Figure 2.15: Convergence and timing comparison for the matrix `cage13` with different block-sizes.

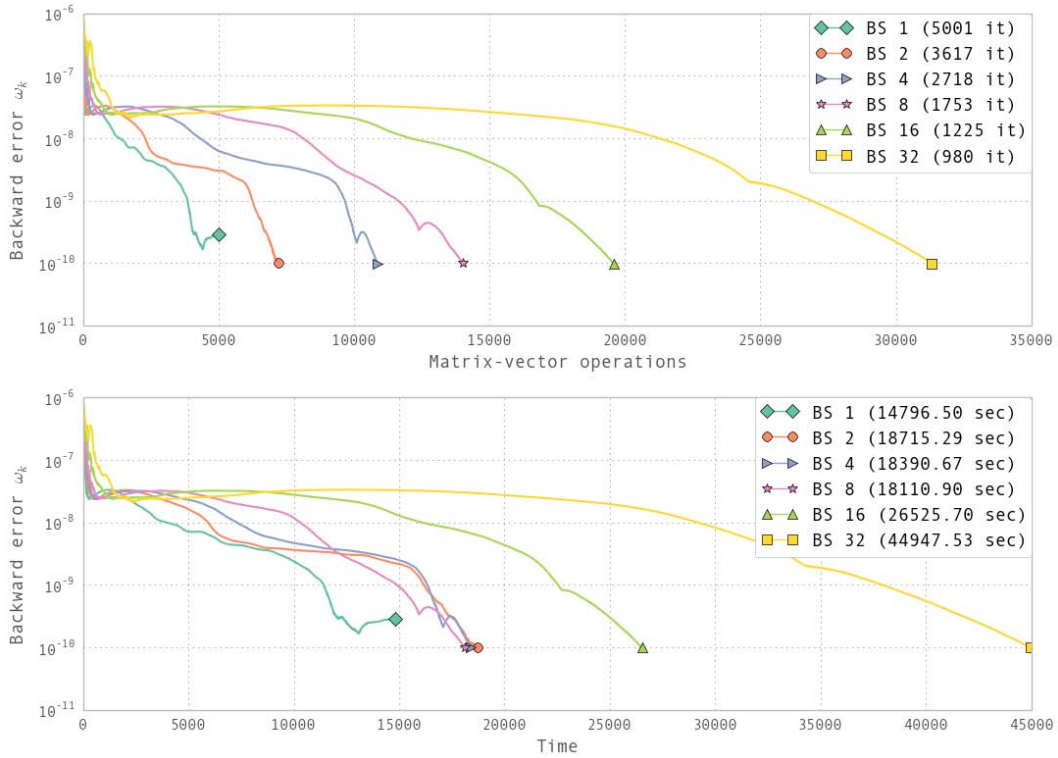


Figure 2.16: Convergence and timing comparison for the matrix `Hamrle3` with different block-sizes.

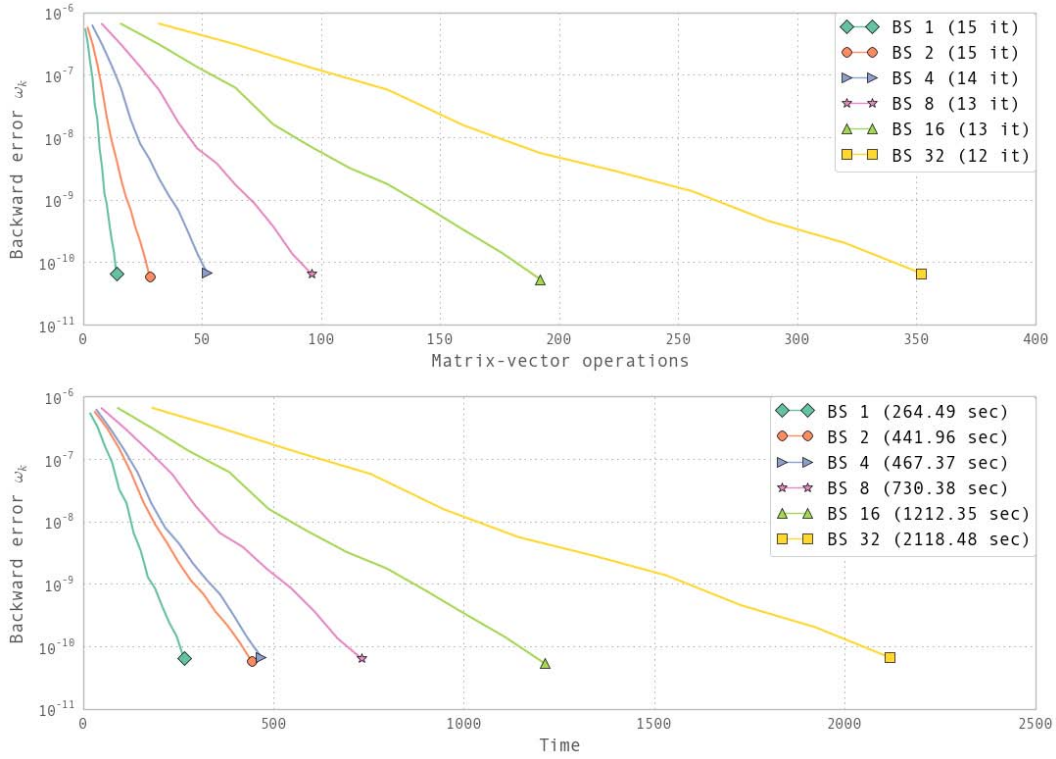


Figure 2.17: Convergence and timing comparison for the matrix `cage14` with different block-sizes.

Chapter 3

Partitioning strategies

The benefit of the block Cimmino algorithm for solving sparse linear systems is that the solution process is reduced to the solution of a sequence of much smaller independent systems within a very simple iterative scheme. The Achilles heel of such an approach is that the iterative method is equivalent to block Jacobi method on the normal equations and so can suffer from slow convergence. This was illustrated at the end of Chapter 2, with some particular examples with very slow and time consuming convergence.

In this chapter, we examine how novel partitioning schemes can be used to accelerate this convergence while, at the same time, improving the capability of the method in exploiting parallelism that we will describe in the next chapter.

After our description of how the ill-conditioning affects the block Cimmino algorithm in Section 3.1, we indicate, in Section 3.2, why we target orderings to block tridiagonal form and how this can be used to reduce the iteration count of block Cimmino. We then discuss two algorithms for obtaining such a form in Section 3.3. In Section 3.4, we describe a different approach to obtaining a partitioning with similar properties.

3.1 Block Cimmino's Iteration Matrix

We recall that our approach to solving the linear system

$$Ax = b \tag{3.1}$$

involves a partitioning into p strips of rows, with $p \leq n$ as in:

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix}. \tag{3.2}$$

As described in Chapter 2, the block Cimmino method projects the current iterate simultaneously onto the manifolds corresponding to the strips and takes a convex combination of all the resulting vectors.

A general study of the convergence of this method and of other related block-row and block-columns methods can be found in [6]. A block SSOR algorithm of this type introduced in [37], is accelerated using conjugate gradients. They study its robustness compared to some preconditioned conjugate gradient methods of the Yale package PCGPACK. Further work on block SSOR as well as comparisons with block Cimmino can be found in [10, 38, 7, 32]. Our aim, in this chapter, is to focus on preprocessing and partitioning strategies of the original system (3.1) to bring it to the form (3.2).

If the matrix A is ill conditioned then there are some linear combinations of rows that are almost equal to zero and, as mentioned in [7], these linear combinations may occur inside the partitions or across the partitions after row partitionings of the form (3.2). Assuming that the projections in the block Cimmino algorithm are computed exactly on the subspaces (by using a direct method), then the rate of convergence of the block Cimmino algorithm depends only on the conditioning across the partitions. If we additionally consider Conjugate Gradient acceleration of the block Cimmino method, as in Chapter 2, the convergence behaviour of the resulting method is directly linked to the spectrum of the $n \times n$ matrix H formed as the sum of the previously mentioned projections, and given by

$$H = \sum_{i=1}^{p} {A_i}^T \left(A_i {A_i}^T \right)^{-1} A_i \ , \tag{3.3}$$

assuming, for simplicity, that the row partitions A_i have full row rank. An efficient implementation of Block Cimmino requires a combination of a robust method for computing the projections and a partitioning strategy that minimizes the ill-conditioning across the partitions.

Let us see now how the ill-conditioning across the partitions can be expressed. Consider that the matrix is partitioned as in (3.2), and let the **QR** decomposition of the partitions ${A_i}^T$ be given by

$$\begin{array}{lll} {A_i}^T = Q_i R_i, & i = 1, \ldots, p \ \text{ where } A_i \text{ is an } m_i \times n \text{ matrix of full row rank} \\ Q_i \quad n \times m_i, & {Q_i}^T Q_i = I_{m_i \times m_i} \\ R_i \quad m_i \times m_i, & R_i \text{ is a nonsingular upper triangular matrix;} \end{array}$$

then:

$$\begin{aligned} H &= \sum_{i=1}^{p} {A_i}^T \left(A_i {A_i}^T \right)^{-1} A_i \\ &= \sum_{i=1}^{p} Q_i {Q_i}^T \\ &= (Q_1 \cdots Q_p)(Q_1 \cdots Q_p)^T \end{aligned} \tag{3.4}$$

But, from the theory of the singular value decomposition (see [39, 40]), the nonzero eigenvalues of $(Q_1 \cdots Q_p)(Q_1 \cdots Q_p)^T$ are also the nonzero eigenvalues of $(Q_1 \cdots Q_p)^T (Q_1 \cdots Q_p)$.

Thus, the spectrum of the matrix H is the same as that of the matrix

$$\begin{pmatrix} I_{m_1 \times m_1} & {Q_1}^T Q_2 & \cdots & \cdots & {Q_1}^T Q_p \\ {Q_2}^T Q_1 & I_{m_2 \times m_2} & {Q_2}^T Q_3 & \cdots & {Q_2}^T Q_p \\ \vdots & & \ddots & & \vdots \\ {Q_p}^T Q_1 & & \cdots & & I_{m_p \times m_p} \end{pmatrix} \tag{3.5}$$

where the ${Q_i}^T Q_j$ are matrices whose singular values represent the cosines of the principal angles between the subspaces $\mathcal{R}({A_i}^T)$ and $\mathcal{R}({A_j}^T)$ (see [41]). These principal angles (see [40, pages

584-585]) are also defined successively by

$$\begin{aligned}\cos(\Psi_k) &= \max_{u\in\mathcal{R}(A_i{}^T)} \max_{v\in\mathcal{R}(A_j{}^T)} \frac{u^T v}{\|u\|\ \|v\|} \\ &= \frac{u_k^T v_k}{\|u_k\|\ \|v_k\|}\end{aligned} \tag{3.6}$$

subject to

$$\begin{aligned} u^T u_p &= 0 \qquad p = 1,\ldots,k-1 \\ v^T v_p &= 0 \qquad p = 1,\ldots,k-1, \end{aligned}$$

with k varying from 1 to $m_{ij} = \min(\dim(\mathcal{R}(A_i{}^T)), \dim(\mathcal{R}(A_j{}^T)))$. The vectors $\left\{u_1,\ldots,u_{m_{ij}}\right\}$ and with $\left\{v_1,\ldots,v_{m_{ij}}\right\}$ are called the principal vectors between the subspaces $\mathcal{R}(A_i{}^T)$ and $\mathcal{R}(A_j{}^T)$.

Note that the principal angles satisfy $0 \leq \Psi_1 \leq \cdots \leq \Psi_{m_{ij}} \leq \pi/2$, and that having $\Psi_k = \pi/2, k = 1,\ldots,\ m_{ij}$, is equivalent to $\mathcal{R}(A_i{}^T)$ being orthogonal to $\mathcal{R}(A_j{}^T)$. Intuitively, the wider the principal angles between the subspaces, the closer H is to the identity matrix, and thus the faster the convergence of the conjugate gradient acceleration.

Nevertheless, even the knowledge of the principal angles between every pair of subspaces would not in general give us any a priori information about the spectrum and the ill-conditioning of the resulting matrix H. We will therefore focus in the following on particular partitionings for which there exists a strong relationship between these principal angles and the spectrum of the iteration matrix.

3.2 Block tridiagonal structures and two-block partitioning

If the block rows A_i are nearly mutually orthogonal, i.e. AA^T is strongly block-diagonally dominant, we can expect that the method will converge very quickly, if the projections are computed accurately. Conversely, the structure of AA^T tells us about the orthogonality of the subspaces represented by block partitions of A. If the block (i,j)th entry of AA^T is zero then the subspaces corresponding to the blocks A_i and A_j are orthogonal. Thus, if AA^T is block tridiagonal, the blocks of A are such that the even numbered blocks are orthogonal to each other as are the odd-numbered blocks. Thus if we solve the projected subproblems accurately (using say a direct method) then we also solve the subproblems corresponding to the odd and even numbered blocks accurately. The partition of A is thus of the form

$$\begin{bmatrix} A \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \tag{3.7}$$

where $B_1 = \left\{\bigcup_i A_i \ / \ i \text{ odd}\right\}$ and $B_2 = \left\{\bigcup_i A_i \ / \ i \text{ even}\right\}$. The partitioning in equation (3.7) is called a two-block partitioning. We denote by m_1 and m_2 the number of rows in B_1 and B_2 respectively. We call the block of smaller size, say B_2 in the above example, the interface block.

With such a partitioning, because of the structural orthogonality between the blocks A_i within B_1 and within B_2 respectively, the matrix H in (3.3) can be considered as $P_1 + P_2$, where $P_1 = P_{\mathcal{R}(B_1{}^T)}$ and $P_2 = P_{\mathcal{R}(B_2{}^T)}$, where each of these two projectors is in fact a sum of independent projectors acting on orthogonal subspaces.

It was shown by [6] that, with such a two-block partitioning, the spectrum of matrix H is

$$\begin{array}{lll} \lambda_k & = 1 + \cos\Psi_k & k = 1, \ldots, m_2 \\ \lambda_k & = 1 - \cos\Psi_{k-m_2} & k = m_2+1, \ldots, 2m_2 \\ \lambda_k & = 1 & k = 2m_2+1, \ldots, n \end{array}$$

where $\{\Psi_k\}_1^{m_2}$ are the principal angles between $\mathcal{R}({B_1}^T)$ and $\mathcal{R}({B_2}^T)$.

On the one hand, matrices in this form can be easily partitioned into sufficient blocks to utilize all the processors of the target machine (provided the matrix A is large enough in comparison with the size of the tridiagonal substructure). On the other hand, the above expression for the spectrum tells us that the block Cimmino algorithm with conjugate gradient acceleration in general works better for small interface block sizes (e.g. small m_2) and, in exact arithmetic, takes not more than $2m_2$ steps for convergence.

The idea of exploiting such sparsity structures appropriately in row-projection methods has already been widely studied and discussed. For instance, [37] have developed a three-block partitioning strategy of this type which they have exploited with their block SSOR algorithm accelerated with conjugate gradients. Further work and discussions can also be found in [38, 42]. In [10], some particular scalings used in conjunction with two-block partitioning strategies are also investigated. These scalings help to open the principal angles between the two subspaces $\mathcal{R}({B_1}^T)$ and $\mathcal{R}({B_2}^T)$ and can be related to oblique projections associated with some ellipsoidal norms. Finally, the potential for parallelism in distributed memory environments of the block-Cimmino method accelerated with the block-Conjugate gradient algorithm is investigated and discussed in detail in [43] (see also [44, 45]).

3.3 Two complementary Preprocessing Strategies based on Cuthill-McKee

As discussed in the previous section, the main idea behind the so called two-block partitioning strategy is to exploit structural orthogonality between the subspaces $\mathcal{R}({A_i}^T)$ defined by the partitioning (3.2). We have indicated that this structural orthogonality can be analysed on the basis of the sparsity pattern of the normal equations matrix AA^T. For example, for two-block partitionings of the type described above, the sparsity pattern of AA^T is block tridiagonal, with diagonal blocks of a size corresponding to the number of rows in each block A_i defined by the partitioning (3.2).

Preprocessing strategy I. This simple remark can be used to define a preprocessing strategy that will enable the construction of two-block partitionings for sparse matrices with any type of sparsity structure. The idea is to permute first the rows of the matrix A, based on permutations that transform the normal equations matrix AA^T into block tridiagonal form. We thus determine a permutation matrix P such that

$$B = PAA^TP^T \tag{3.8}$$

has a block tridiagonal form. To this end, we exploit an implementation of the Cuthill-McKee Algorithm (see for instance [46], [47], [48]) for ordering symmetric matrices. We then solve the row-wise permuted system of equations

$$\hat{A}x = \hat{b} \tag{3.9}$$

with $\hat{A} = PA$, and $\hat{b} = Pb$, using the block Cimmino algorithm. From the block tridiagonal structure of the matrix B, the block row partition (3.2) of $\hat{A}$ is defined with blocks of rows with each block determined by the size of the diagonal blocks in the block tridiagonal structure of B,

or by the number of rows in a contiguous subset of these diagonal blocks. The row partitioning we obtain from this still has the properties of the two-block partitioning described in Section 3.2.

This preprocessing strategy, for general sparse matrices, exploits only the sparsity structure in the normal equations matrix AA^T and not the values contained in this matrix. We should mention and will illustrate with experiments in the following sections that, for very general sparsity patterns, the block tridiagonal structure obtained with the Cuthill-McKee Algorithm has diagonal blocks with very differing sizes leading to a partitioning with a poor degree of parallelism because of unbalanced tasks in the block-row projections. From the discussion in Section 3.1, we recall also that the main objective when defining the partitioning is to minimize the effects of ill-conditioning across the blocks, which simply means keeping the principal angles between every pair of subspaces as open as possible. Strict orthogonality between every second block of rows is only one step in this direction; we may also try to take into account in some way the angles themselves when defining the reorderings and partitionings.

Preprocessing strategy II. This strategy will take into account the values in the matrix AA^T as well as the sparsity structure of that matrix in an attempt to define a reordering/partitioning strategy with numerical properties close to that of two-block partitioning but with more flexibility for building the blocks and potentially much better parallelism.

In this preprocessing strategy, the matrix AA^T, is first normalized through:

$$\begin{aligned} C &= AA^T \\ D &= \mathbf{diag}(C) \\ S &= D^{-\frac{1}{2}} C D^{-\frac{1}{2}}. \end{aligned}$$

The entries in the normalized matrix S correspond to the cosine of the principal angle between every pair of rows (in other words, the degree of collinearity between any pair of rows), and we may expect that if such a cosine is relatively small, then the corresponding pair of rows are almost orthogonal and can be considered so. Our next step in this preprocessing strategy is then to keep only the nonzero entries in S which are above a given tolerance value τ in absolute value, viz

$$F = \mathbf{filter}(|S|, \tau),$$

and to permute the resulting matrix F into block tridiagonal form using the Cuthill-McKee algorithm as before

$$\widehat{B} = PFP^T \tag{3.10}$$

and solve (3.9) by using the row partitioning for $\hat{A}$ defined by the block tridiagonal structure of $\hat{B}$.

In practice, we do not want to form AA^T but we first ensure that the diagonal entries will be one by scaling the original matrix A so that the rows have 2-norm equal to one. This can be accomplished by using the HSL routine `MC77` [21]. Of course, since numerical values have been dropped from the normal equations matrix, we cannot expect that the resulting partition will provide two subsets of structurally orthogonal blocks of rows but, if the values dropped are sufficiently small, we may expect that the numerical properties of the resulting iteration matrix will be relatively close to that of the "*strict*" two-block partitioning case. Additionally, since the filtered matrix F has less entries than the original normal equations matrix C, the resulting block tridiagonal permuted matrix $\hat{B}$ will surely have a smaller bandwidth than B and this may help to define a partitioning on A with more blocks, better balanced projections, and a higher degree of parallelism. In the following section, we will experiment and compare these two preprocessing strategies with another strategy that we will now describe.

3.4 Using a hypergraph partitioning

A main aim of the partitioning strategies that we have just described is to decouple the blocks to reduce the number of block Cimmino iterations. We now look at a way to do this more directly.

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is a generalization of the concept of graphs, where $\mathcal{V}$ are the vertices and $\mathcal{N}$ are the hyperedges, also called nets. In a hypergraph, nets can connect more than two vertices whereas, in a graph, edges connect only two of them.

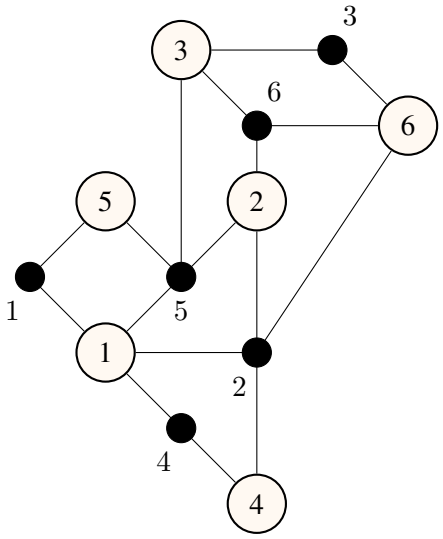


Figure 3.1: A hypergraph representing a sparse matrix. Large circles are the vertices (corresponding to the rows) and small dark circles are the nets (corresponding to the columns).

Hypergraphs can be useful to represent a sparse matrix where the vertices can be linked to the row numbers and the nets to the column numbers. We show in Figure 3.1 a hypergraph example with 6 vertices and 6 nets representing a 6×6 sparse matrix shown in Figure 3.2. Notice how we can easily see from the hypergraph all the interconnections between the different rows in the matrix.

	1	2	3	4	5	6
1	×	×		×	×	
2		×			×	×
3			×		×	×
4		×		×		
5	×				×	
6		×	×			×

Figure 3.2: An illustrative sparse matrix.

If we partition the matrix uniformly in three sets (partitions) as in Figure 3.3, we see that each partition has overlapping columns with at least another partition. This creates an all-to-all interactions between the partitions. Columns 2,4,5 and 6 are shared by all partitions, whereas columns

1 and 3 are shared between only two partitions. This kind of partitioning will be prohibitive for parallel runs particularly because of the communication cost.

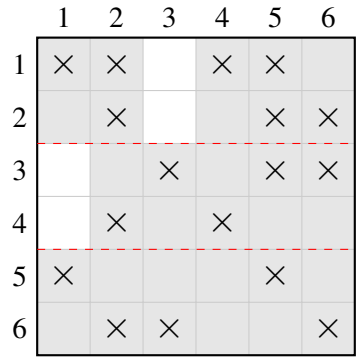


Figure 3.3: The partitioning of the illustrative matrix, with the columns on which the communications will happen shaded in gray.

A k-way hypergraph partitioning creates a partitioning $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_k\}$, where $\mathcal{V}_i$ is a nonempty subset of the vertices $\mathcal{V}$ where $\mathcal{V}_i \cap \mathcal{V}_j = 0, i \neq j$ and $\bigcup_{i=1}^{k} \mathcal{V}_i = \mathcal{V}$, i.e. all the parts in Π are pairwise disjoint and their union is equal to $\mathcal{V}$. The problem of minimizing the interconnections between the partitions is NP-hard [49].

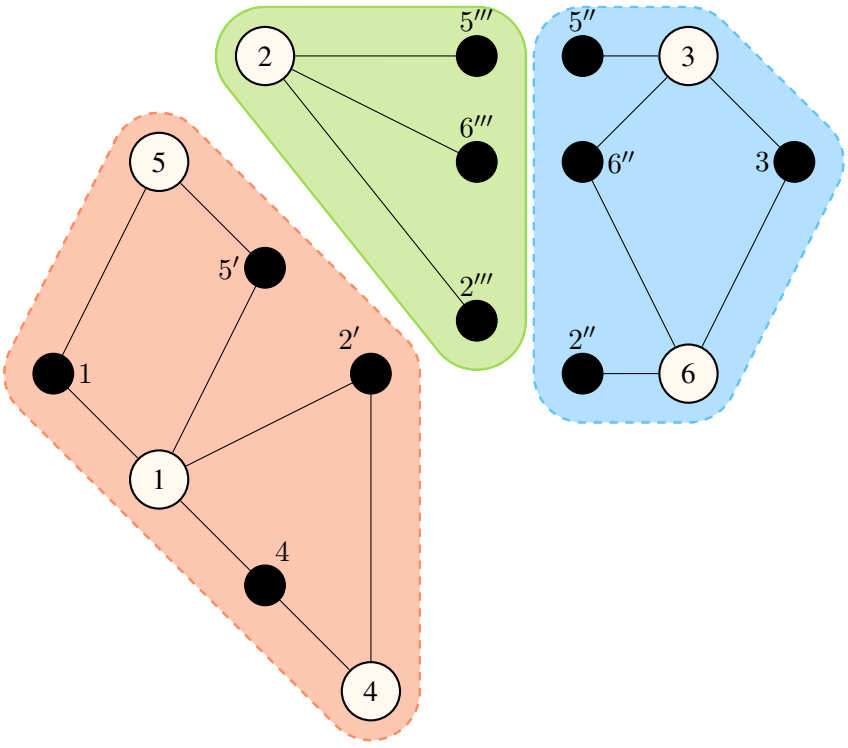


Figure 3.4: A possible partitioning of the previous hypergraph.

A possible partitioning of the previous hypergraph is shown in Figure 3.4 where the three regions represent the three partitions. To easily distinguish the three partitions we duplicated the hyperedges 2, 5 and 6, corresponding to the column overlap (interconnections) between the

partitions. The partition numbers are, from left to right, partition 1, 3 and 2 respectively. The interconnections and the resulting permuted matrix are represented in Figure 3.5.

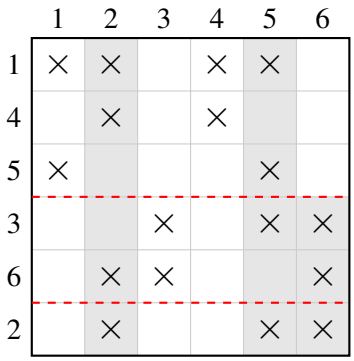


Figure 3.5: The partitioning using a hypergraph partitioner of the illustrative matrix. The parts in gray are the interconnections.

We see that the interconnections in the hypergraph are represented by the shaded overlapping columns in the matrix, with a reduced number of global interconnections from 4 to 2 compared to the former case, and the total number of interconnections reduced from 6 to 3.

In this manuscript, and in our implementation, we use the `PaToH` hypergraph partitioner [50, 51]. This partitioner allows us to do a k-way partitioning of a sparse matrix. It permits us also to define a balancing between the number of rows of each partition. In the following experiments, we will use two levels of balancing. The first is a weak balancing where we allow the partitions to have a large difference in number of rows while reducing greatly the number of interconnections. This helps to group the interconnected rows within the partitions as much as possible. The second level of balancing is strong balancing where the target is to have an almost equal number of rows per partition.

3.5 Partitioning Experiments

In this section, we perform some experiments with the block Cimmino solver using Block-CG acceleration (see [8, 45]) and focus on the effects of the preprocessing strategies on the performance. We have therefore chosen a fixed block size for the Block-CG acceleration and stop the iterations on the basis of a normwise backward error (see [52])

$$\omega_k = \frac{\|Ax^{(k)} - b\|_\infty}{\|A\|_\infty \|x^{(k)}\|_1 + \|b\|_\infty}$$

of less than 10^{-12}. A small value for ω_k means that the algorithm is normwise backward stable (see [53]) in the sense that the solution $x^{(k)}$ is the exact solution of a perturbed problem where the max norm of the error matrix is less than or equal to ω_k.

In Section 3.5.1, block Cimmino is used to solve the `SHERMAN3` linear system from the Harwell-Boeing matrix collection [54]. Section 3.5.2 contains the results from experiments of runs of block Cimmino on the `bayer01` problem from the sparse matrix collection at the University of Florida [36], we then perform some experiments with a larger set of matrices.

3.5.1 A first illustrative example: SHERMAN3 problem

The matrix SHERMAN3 is a symmetric matrix of order 5005. This matrix comes from the discretization of partial differential equations extracted from a three dimensional oil reservoir simulation model on a $35 \times 11 \times 13$ grid using a seven-point finite-difference approximation.

The original pattern of matrix SHERMAN3 is shown in Figure 3.6. In the first experiment, a trivial block-row partition of the linear system is used, with eight equal-sized blocks of rows. This results in blocks of 625 rows except for the last one of 630.

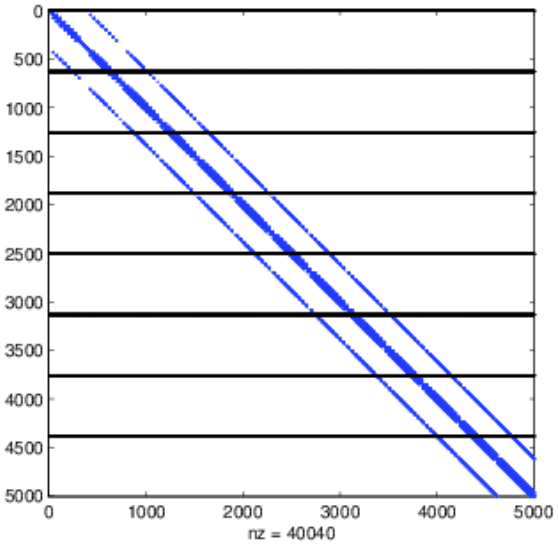


Figure 3.6: Sparsity pattern of the SHERMAN3 matrix. The matrix has been partitioned into 8 blocks of rows.

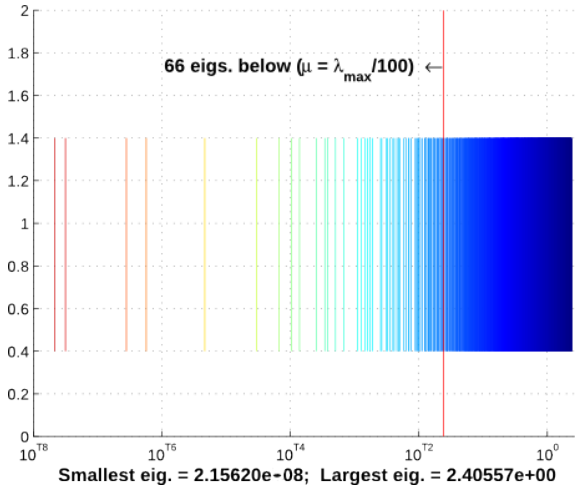
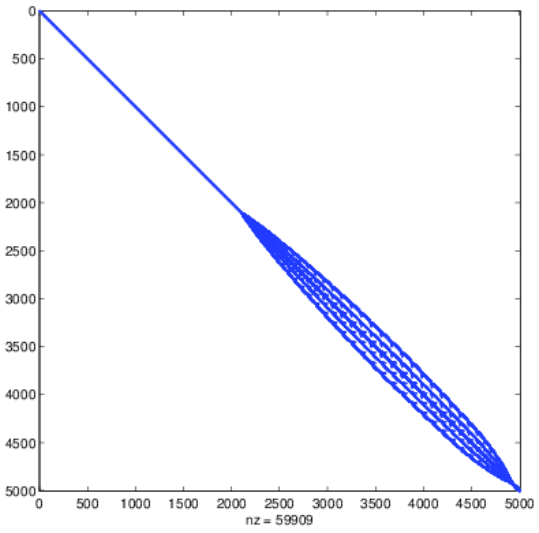


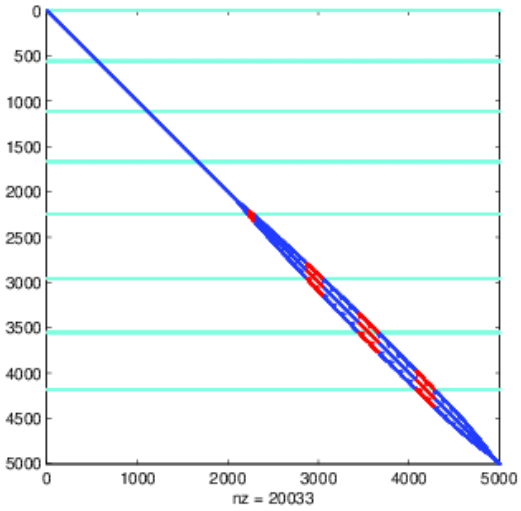
Figure 3.7: Eigenvalue distribution spectrum of H for the SHERMAN3 problem.

The spectrum of H from the original matrix with a straightforward partitioning as shown in

Figure 3.6 is given in Figure 3.7. This shows a large cluster of eigenvalues around 1 but a few trailing eigenvalues associated with bad conditioning (smallest eigenvalue of the order of 10^{-8}).



(a)



(b)

Figure 3.8: (a) Permuted normal equations matrix from SHERMAN3 using the Cuthill-McKee algorithm. (b) The SHERMAN3 matrix after row permutations following the preprocessing *strategy I* and with additional column grouping.

Since the block Cimmino method is numerically independent of any column permutations, after the row-partitioning of the system, we also perform in all cases some column permutations to group together columns belonging to the same subsets of row-partitions in order to ease the communication phase in the algorithm when merging the results from the different projections. By doing so, we improve the parallel execution of the block Cimmino method. We do not illustrate this in the following because it is not very important for the discussion when comparing the different preprocessing and partitioning strategies, and we refer to [43] for more technical details on the distributed memory implementation of the block Cimmino method accelerated with a block-CG algorithm.

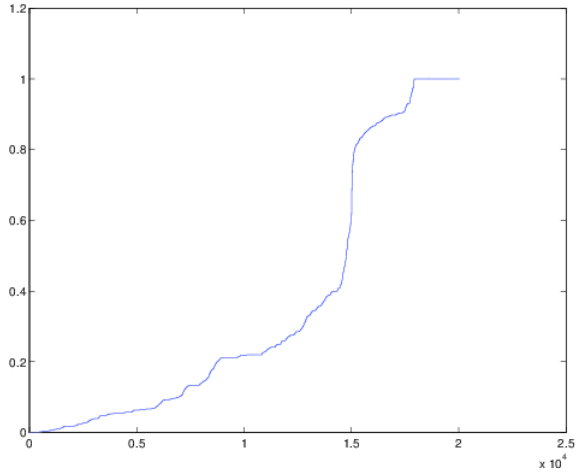


Figure 3.9: Normalized nonzero entries in the normal equations matrix from SHERMAN3. On the x-axis, the entries are displayed in increasing order of their absolute numerical value (given on the y-axis).

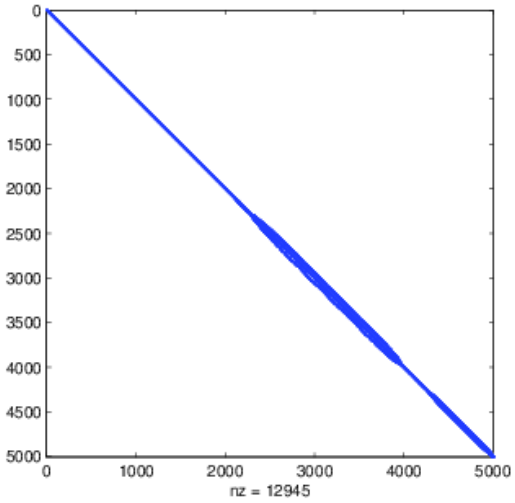
In the second round of experiments, the preprocessing *strategy I* from Section 3.3 is used. From the block tridiagonal structure of the permuted normal equations matrix of SHERMAN3, the matrix is partitioned into blocks of rows. Figure 3.8-b shows the pattern of matrix `SHERMAN3` after row permutation following the preprocessing *strategy I*, using a level set algorithm, Cuthill-McKee, to permute the normal equations matrix into block tridiagonal form, as indicated in Figure 3.8-a. The partitioning shown is a two-block partitioning, and columns belonging to the same subsets of blocks have been grouped together as mentioned before. The block-row partitions have been defined using the block tridiagonal structure in the normal equations matrix. As in the first set of experiments described above, we again aim at obtaining 8 blocks of about 625 rows each.

The sparsity pattern of the permuted `SHERMAN3` matrix after completion of preprocessing *strategy II* is shown in Figure 3.10-b. In this case we dropped the normalized entries of the normal equations matrix that are below 0.2. We plot in Figure 3.9 the distribution of all the entries of the normal equations matrix, where we see that dropping at 0.2 will remove less than the half of entries. The matrix with the entries dropped was permuted using the ordering from the Cuthill-McKee algorithm. The associated block-row partition, indicated in Table 3.1, has been defined from the block tridiagonal structure of the matrix $\hat{B}$ in (3.10), as described in Section 3.3, with the aim of again obtaining 8 blocks of about 625 rows each whenever possible.

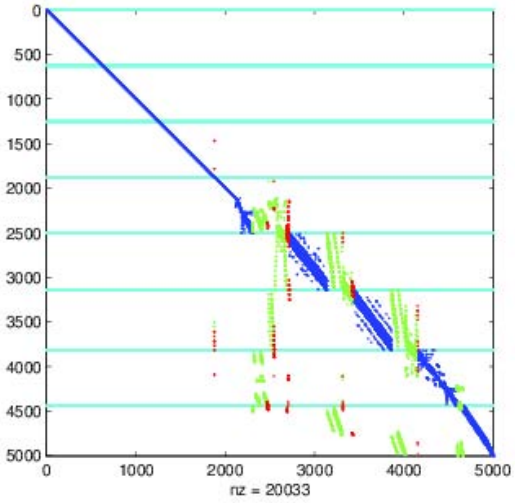
In the case of the hypergraph partitioning we use two different imbalance parameters, a weak balancing which tolerates partitions up to 8 times larger than other partitions, and a strong balancing which tolerates up to 50% imbalance.

Also, we notice that the matrix $\hat{B}$ in Figure 3.10-a has a smaller bandwidth than the matrix in Figure 3.8-a. Thus the preprocessing *strategy II* offers more degrees of freedom to define the row partitions than *strategy I*, since they are of smaller size. Therefore, we can define more partitions while maintaining a good balance between the number of rows in each partition.

The spectrum of H for strategies I and II are shown in Figures 3.11 and 3.12, respectively. Notice that compared to the spectrum of H arising from the original matrix the spectrum presents a better clustering of the eigenvalues. In *strategy I* the largest eigenvalue is 2 which confirms a two-block partitioning. However in *strategy II* we get a largest eigenvalue only slightly larger than 2 and thus we call *strategy II* a **near two block partitioning**. The convergence results in Table 3.2 illustrate the effect of the better clustering of eigenvalues obtained with *strategy II*.



(a)



(b)

Figure 3.10: (a) Sparsity pattern of matrix $\hat{B}$ in (3.10), obtained after removing nonzero entries less than 0.2 from the normalized SHERMAN3 normal equations matrix, and using the Cuthill-McKee algorithm to permute the resulting matrix into block tridiagonal form. (b) Matrix SHERMAN3 after row permutations following the preprocessing *strategy II* and with additional column grouping.

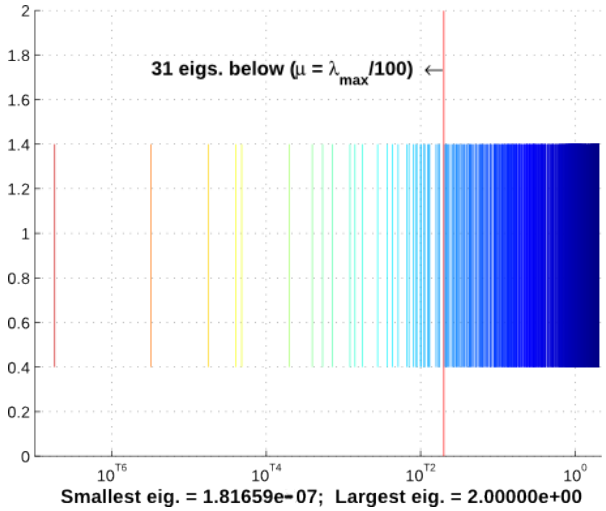


Figure 3.11: Eigenvalue spectrum of H for the permuted SHERMAN3 problem arising from using **strategy I.**

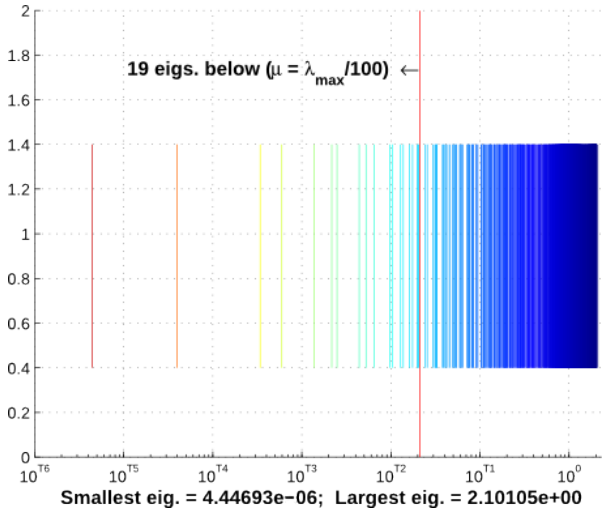


Figure 3.12: Eigenvalue spectrum of H for the permuted SHERMAN3 problem arising from using **strategy II** with a drop at 0.2.

	Number of rows in each partition							
Original Matrix A	625	625	625	635	629	628	784	454
Strategy I	625	625	625	635	629	628	784	454
Strategy II (drop 0.1)	625	625	625	744	707	669	628	382
Strategy II (drop 0.2)	625	625	625	627	641	671	630	561
strategy II (drop 0.4)	637	637	637	637	637	637	637	546
`PaToH` (weak balancing)	1019	396	1015	400	544	544	544	543
`PaToH` (strong balancing)	626	626	626	625	626	625	626	625

Table 3.1: Description of the 8 block-row partitions obtained for matrix SHERMAN3.

The second column of Table 3.2 shows convergence results from the runs of the block Cimmino method using different partitioning strategies. These results should be multiplied by the block-size – 8 in our case – to obtain the number of matrix-vector operations, we show them in the last column to easily compare the amount of work. We note that the uniform partitioning requires the highest number of iterations for convergence since neither the structure of the matrix nor the value of the matrix entries were taken into account.

However, when we take into account both the value of the entries and the structure of the matrix as in *strategy II*, we are able to reduce the iteration count to 68 when dropping at 0.2 compared to 102 when using *strategy I*. We examine further the effect of dropping on the convergence by showing counts for dropping at 0.1 and 0.4. We see that, as we increase the dropping value from 0.1 to 0.2, we get a finer partition with the higher values in the blocks. However, as we increase the dropping value more (to 0.4) we start to lose these connections resulting in an increase in iteration count.

Using `PaToH`, we can reduce the iteration count to 52 when we use loose balancing. By doing this, we have larger partitions and are able to reduce the interconnections between them. We notice that effect when enforcing stronger balancing where the iteration count rises to 74.

	Nb. of iterations	M-V operations
Original Matrix A	190	1520
Strategy I	102	816
Strategy II (drop 0.1)	73	584
Strategy II (drop 0.2)	68	544
Strategy II (drop 0.4)	89	712
`PaToH` (weak balancing)	52	416
`PaToH` (strong balancing)	74	592

Table 3.2: Convergence of SHERMAN3 in the three different sets of experiments.

3.5.2 Second illustrative example: `bayer01` problem

We now look at these strategies on a larger problem, `bayer01`. This matrix was obtained from Bayer AG by Friedrich Grund and is available from the sparse matrix collection at the University of Florida [36]. It is of order 57735 and has 277774 nonzero entries. We show the pattern of the matrix in Figure 3.13 where we have superimposed 16 uniform partitions.

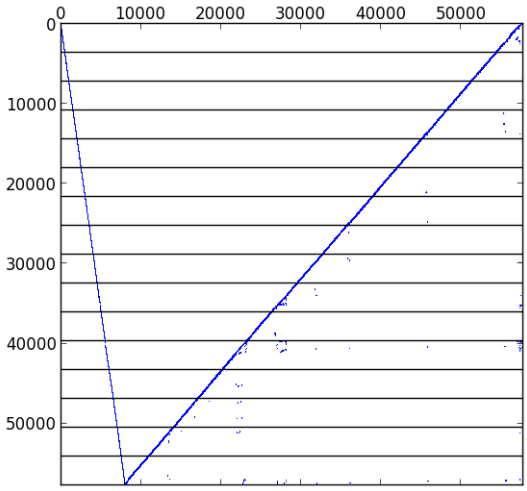


Figure 3.13: Nonzero pattern of the matrix `bayer01`.

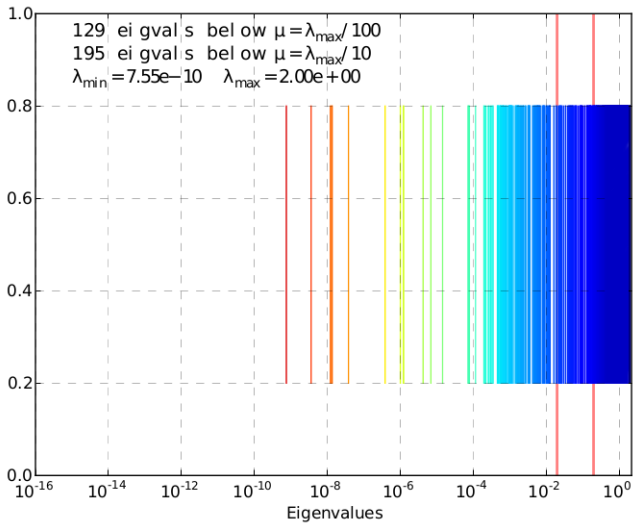


Figure 3.14: Spectrum of block Cimmino iteration matrix for `bayer01` with 16 uniform partitions.

If we run the block Cimmino algorithm on the matrix partitioned as in Figure 3.13 then the resulting spectrum of the iteration matrix is shown in Figure 3.14. We see that, while there is a good clustering of the eigenvalues around the value 1, just as we have noticed with the matrix `SHERMAN3`, the matrix is still quite badly conditioned. Indeed, many small eigenvalues are present and these eigenvalues will increase the iteration count when using the conjugate gradient acceleration.

If, however, we first reorder the matrix using PaToH and then partition it, we get the spectrum for the iteration matrix shown in Figure 3.15 where we note that the intermediate eigenvalues have been shifted towards 1 thus improving the clustering of eigenvalues in the iteration matrix. The reduction in the number of small eigenvalues is expected to improve the convergence.

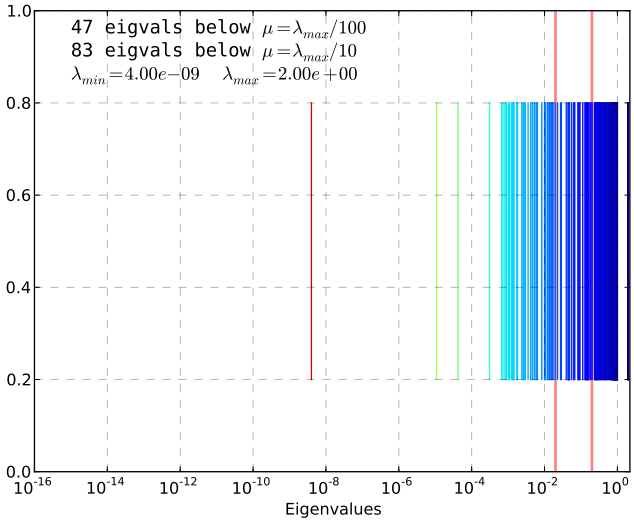


Figure 3.15: Spectrum of block Cimmino iteration matrix for bayer01 with 16 partitions obtained using the hypergraph partitioner PaToH.

	Smallest partition	Largest partition	Ratio
Original Matrix A	3608	3615	1.002
Strategy I	1673	4342	2.595
Strategy II (drop 0.05)	1448	4017	2.774
Strategy II (drop 0.1)	3318	3660	1.103
Strategy II (drop 0.2)	3522	3634	1.032
PaToH (weak balancing)	952	7597	7.98
PaToH (strong balancing)	3608	3609	1

Table 3.3: Information on the 16 block-row partitions obtained for matrix bayer01.

Table 3.3 summarizes the partitioning information after applying the different strategies. As there are 16 partitions, we show only the smallest, the largest and the ratio between the largest and the smallest partition. We notice that when using *strategy I* we are constrained by the size of the level sets so that size balancing of the partitions is quite difficult, and the largest partition is more than 2.5 times the size of the smallest partition. As expected, by using *strategy II* we obtain better load balancing between partitions and this improves as we increase the dropping parameter. In the case of PaToH, weak balancing gives the greatest freedom for partitioning, and our largest partition is nearly 8 times larger than the smallest. If we constrain this freedom by strengthening the balancing, we obtain a partition with almost equal-sized blocks.

	Nb. of iterations	M-V operations
Original Matrix A	426	3408
Strategy I	459	3672
Strategy II (drop 0.05)	270	2160
Strategy II (drop 0.1)	204	1632
Strategy II (drop 0.2)	343	2744
PaToH (weak balancing)	76	608
PaToH (strong balancing)	205	1640

Table 3.4: Convergence of bayer01 with the different partitioning strategies using block-CG with block-size 8.

We show in Table 3.4 the convergence results for the matrix bayer01 with a block-size 8. We notice a similar behaviour as with the SHERMAN3 matrix where the preprocessing *strategy II* improves the iteration count when dropping until a certain threshold. However, the preprocessing *strategy* I is worse than just partitioning the original matrix uniformly. This behaviour was seen on several problems which makes this strategy unreliable. Finally, using PaToH both with weak and strong balancing gives good results for these two problems.

More experiments. The results above are not universal, as we see from runs on some of our test problems in Table 3.5, uniform partitioning can give the best result. We take the results presented in the previous chapter and select the block-size that gives the best timings with uniform partitioning and compare it with PaToH partitioning with 50%, 150% and 200% imbalance.

Problem	Block-size	Uniform	P (50%)	P (150%)	P (200%)
gre_1107	8	121	72	**64**	65
SHERMAN3	8	190	74	60	**52**
bayer01	8	426	205	315	**76**
lhr71c	32	529	2505	**518**	Failed
torso3	1	**22**	32	28	29
Hamrle3	4	2717	759	759	**683**

Table 3.5: Number of iterations when using uniform and hypergraph partitioning for some test problems. Failed means that over 5000 iterations were performed without convergence. Best results are shown in bold.

We notice that increasing the partitioning freedom by increasing the imbalance usually improves the convergence as seen for SHERMAN3 and bayer01. Hamrle3 behaves similarly except for 50% and 150% where it gives the same partitioning and thus the same number of iterations. Convergence in the case of gre_1107 improves slightly at 150% then worsen afterwards. This happens also with torso3 showing that increasing the imbalance does not always mean better convergence. torso3 on the other hand gives better results with uniform partitioning.

The lhr71c is an interesting case where strong balancing makes the method converge slowly and fails to converge after more than 5000 iterations. An other interesting aspect for this problem is that the uniform partitioning gives comparable results to a medium imbalancing, we prefer in

such a case to use uniform partitioning. This shows that different partitionings will lead to different iteration matrices, and therefore we cannot predict in advance how it will behave.

In addition to the fact that some test problems work better with uniform partitioning, we have to take into account that if we obtain only small iteration count improvements using hypergraph partitioning it may not always be the best way to go. Indeed, using libraries such as `PaToH` will generate an overhead which, if not covered by the gains in block-CG, will degrade the overall performance, we noticed this situation with the matrices `cage13` and `cage14`.

3.6 Concluding remarks

The results show that preprocessing the original system of equations to obtain a different partitioning of the system can have a profound effect on the convergence of the block Cimmino algorithm resulting in a more robust and efficient implementation. Indeed, a good preprocessing strategy can minimize the ill-conditioning between the different blocks.

Our *Strategy I* did this by enforcing a two-block partitioning but it was not very good at obtaining a good blocking of the original matrix. We thus tried *Strategy II* which did much better with respect to a regular partitioning but at the cost of losing, to a controlled extent, the two-block partitioning. Although this did better than *Strategy I*, it was still not a robust approach and did not always provide an improvement over a straight partitioning of the original system. It also required the selection of a dropping parameter that was dependent on the problem being solved and so was hard to choose in advance. We thus examined a strategy that uses a hypergraph model to partition the original system with the aim of directly reducing the interconnection between block partitions. We use the `PaToH` library to effect this partition and found that it compared very favourably to our strategies based on the normal equations and was also more robust in that it produced partitions better than a uniform partition of the initial matrix in most cases. It was also possible to use `PaToH` to obtain a well balanced partitions with roughly the same number of rows.

Chapter 4

Hybrid implementation of the parallel block Cimmino method

As in many solvers, the solution phase is always preceded by a preprocessing phase that prepares the data structures and tries to make the solution phase as efficient as possible.

For the following, consider the illustrative linear system $Ax = b$ shown in the Figure 4.1 partitioned uniformly into 3 partitions, we use this system solely for description purposes.

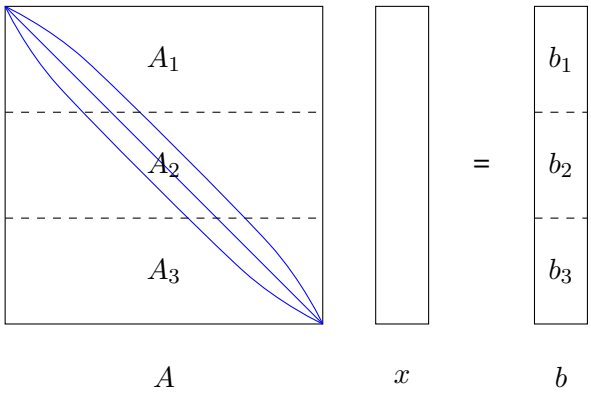


Figure 4.1: Illustrative example of a uniform partitioning of a linear system of equations.

This chapter is divided in two main parts, the preprocessing phase in which we prepare the data for the second part which is the solution phase. We talk about the preprocessing phase in Section 4.1, we show the different algorithms used during that phase and how the data structures are handled to facilitate the solution phase. Next, we describe the different steps for the solution phase, starting with the first master-slave parallel version described in Section 4.2. Then we introduce the new fully distributed hybrid scheme in Section 4.3. Throughout this description, we take a look at the different pieces contributing to the design of the whole block-Cimmino distributed solver.

4.1 Preprocessing

In the preprocessing phase the matrix and the right-hand side go through different manipulations that help the solution phase to go faster. In this section, we describe these steps in order of their execution.

4.1.1 Scaling

It is shown (see [32] and [6]) that the convergence of the block-Cimmino is unaffected by the row scaling but depends only on the principal angles between the partitions of the matrix. However, the column scaling affects this convergence and can be considered as a diagonal preconditioner of the block-Cimmino iterative scheme. The only gain we will obtain from row scaling is the improvements in the quality of the factorization of the augmented systems, and in particular on the numerical pivoting.

4.1.2 Partitioning

We have shown in Chapter 3 the different partitioning strategies we have implemented. We have seen that we can either partition the matrix uniformly without any permutations, or we can reorder the rows so that the partitions create good properties for the iteration matrix which will help to speedup the convergence.

One effect the partitioning has on the parallel version is the number of interconnections (overlapping columns) between the partitions. A partitioning that targets a small number of interconnections will reduce the communications during the different steps. We can also use a partitioning that tries to build two sets of partitions where the inner partitions of each set are mutually orthogonal to each other.

4.1.3 Augmented Systems

We have seen in Chapter 2 that the block-Cimmino method needs to compute a set of projections and to do a sum afterwards. We have seen that our preferred way of computing these projections is to use an augmented system approach.

In Section 2.4, in particular in Algorithm 2.3, we described the algorithm PS_A used to compute the sum of projections. In this algorithm, we sequentially compute the solution to each of the augmented systems

$$\begin{pmatrix} I & A_i^T \\ A_i & 0 \end{pmatrix} \begin{pmatrix} u_i \\ v_i \end{pmatrix} = \begin{pmatrix} 0 \\ A_i p^{(j)} \end{pmatrix}.$$

We also indicate that the sum of the vectors u_i, the projections, is actually the matrix-vector operation needed in the block Cimmino iteration.

Rather than building the augmented systems with the full partition, we decided to compress each partition by reducing its number of columns, by removing null columns when building the augmented system. For the case of the previous illustrative example, we show the parts to be kept in Figure 4.2, where $\hat{A}_i$ is the non-null part of the partition A_i.

For the linear system in Figure 4.1, the augmented system for the second partition is as shown in Figure 4.3.

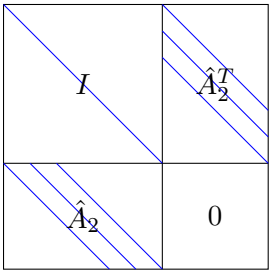


Figure 4.3: An example of a compressed augmented matrix.

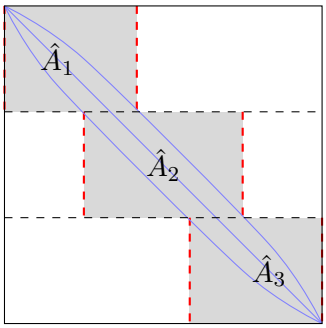


Figure 4.2: Illustration of the important parts to be kept for each partition.

Notice that we removed all the null columns from the partition which made the augmented system smaller. This compression, depending on the sparsity of A and on the partitioning, can also happen anywhere within the columns of each block independently.

The issue that arises is the fact that the solution is compressed as well. To recover the original solution –more precisely, to be able to sum the projections correctly– we keep track of the compression and create a vector that we call $ColumnIndex_i$. This vector indexes each column of the compressed partitions to its original column. We show in Figure 4.4 an example of how this compression works on a vector. In this case the 3rd item in the $ColumnIndex_i$, shown in the middle, has the value 5. We take the 5th element from the original vector and put it into the 3rd position in the compressed one, this process is easily reversible.

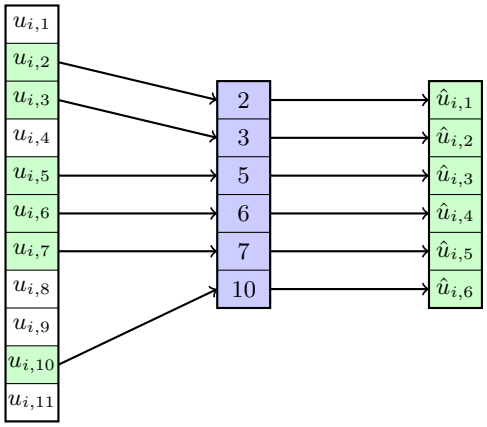


Figure 4.4: The compression process of a vector, left to right, using the $ColumnIndex_i$ vector, in the middle.

4.2 Master-Slave Block Cimmino Implementation

In this section we assume the conjugate gradient method runs on a single process and has all the matrix available in memory. It is important to first describe this implementation in detail as it will be generalized in the fully parallel implementation described in Section 4.3. The following strategy is architecture-independent, therefore we suppose that we have a generic hybrid architecture containing processes with both shared and distributed memory.

We know that the most computationally heavy operation in a conjugate gradient iterative scheme is the matrix-vector operation, especially if we are using a stabilized block-CG acceleration where the vector becomes a matrix (a set of vectors).

Computing the projections. When all the partitions are on the same process we have to avoid a sequential one-by-one factorization of the augmented matrices; the same thing goes for computing the solutions. Also, the factorization of all these augmented matrices needs computational power, therefore we aim to factorize the augmented matrices using all the processes available to us.

The idea is to do all the factorization in a single shot and this can be done by combining all the augmented systems into a single large system with a block-diagonal structure where each diagonal block is one of the augmented matrices, coming from the partitions, as shown in Figure 4.5.

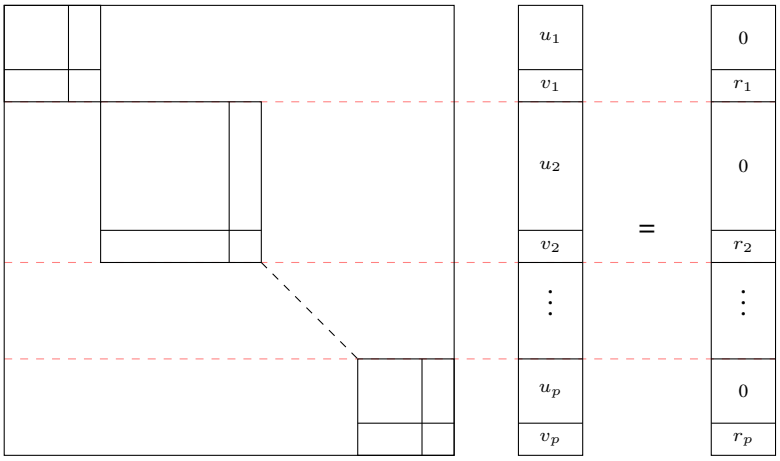


Figure 4.5: Linear system with a block diagonal matrix grouping all the augmented matrices.

Notice also from this figure that we are able to specify larger blocks, corresponding either to larger partitions or less compressible ones. This freedom in size should not overly perturb the performance of the direct solver and it is important for convergence purposes as we have seen in Chapter 3. We will call this matrix, temporarily, the B matrix.

Both the factorization and the solution of this linear system can be done in parallel. Indeed, the slave processes in this strategy are used only to help with the factorization and the solution within the direct solver.

The projection sum algorithm has to be adapted to handle this structure. We propose here a rewriting of the PS_A algorithm as shown in Algorithm 4.1. The first modification is in the introduction of two input variables α and β which will help to identify if we want to compute the residual (when both variables are 1), or a matrix-vector product (when $\alpha = 0$ and $\beta = -1$). The second change is the use of the hat symbol to describe the compressed vector $\hat{z}$ or compressed partitions $\hat{A}_i$. However, the output is guaranteed to be exactly the same as the result obtained with the non-compressed case.

In this algorithm, we start by computing r_i for each partition ($\alpha = 1$ and $\beta = 1$). Then we stack them piecewise into the sparse vector v by respecting the sparse structure of the right-hand side described in Figure 4.5. This vector is then used to compute the solution s to the block diagonal system, the $direct\_solve()$ call supposes that the B matrix has already been factorized. Next, we extract the vectors $\hat{u}_i$ from the solution at line 10, the length of which corresponds to the number of columns in the compressed partition. To extract the next vector we advance by the size of each augmented system which corresponds to the sum of the number of rows and columns of the partition.

Algorithm 4.1 Projections Sum (PS_A)

Input: w, $\hat{z}$, α and β
Output: $\sum_{i=1}^{p} A_i^+ (\alpha \times w_i - \beta \times A_i z)$

1: $\delta \leftarrow \begin{bmatrix} 0 & \dots & 0 \end{bmatrix}^T$
2: **for** $i = 1 \rightarrow p$ **do**
3: $\quad r_i \leftarrow \alpha \times w_i - \beta \times \hat{A}_i \hat{z}$
4: **end for**
5: $v \leftarrow \begin{bmatrix} 0 & r_1 & \dots & 0 & r_p \end{bmatrix}^T$
6: $s \leftarrow direct\_solve\,(Bs = v)$
7: $pos = 0$
8: **for** $i = 1 \rightarrow p$ **do**
9: $\quad size\_as_i \leftarrow nbrows(\hat{A}_i) + nbcols(\hat{A}_i)$
10: $\quad \hat{u}_i \leftarrow s(pos : pos + nbcols(\hat{A}_i) - 1)$
11: $\quad pos \leftarrow pos + size\_as_i$
12: $\quad \delta \leftarrow \delta + expand(u_i, ColumnIndex_i)$
13: **end for**
14: **return** δ

The result is obtained through the sum of the decompressed $\hat{u}_i$. The $expand()$ function uses the $ColumnIndex_i$ vector to do this. In reality, the vectors are not uncompressed, the values of $\hat{u}_i$ are summed with their corresponding values in the compressed vector δ. This way, no extra memory is required. One drawback of this approach is the double indirect reference, to get the column number and then to access the actual entry.

In the case of our illustrative example, the columns of each partition are contiguous therefore it becomes easy to do the sum. We show in Figure 4.6 the columns of each partition that are interconnected to other partitions in red (dotted) and in green (non-dotted) the columns that are not interconnected.

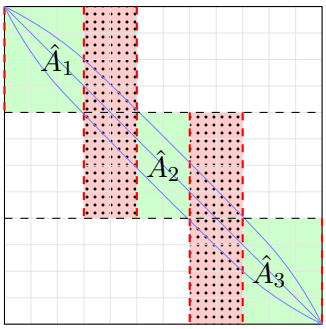


Figure 4.6: Illustration of the interconnections between the partitions in red and the columns that are not interconnected in green.

The interconnections between the partitions, shown in red (dotted pattern), are to the parts in the u_i (corresponding to the partition A_i) to be summed and the rest, shown in green, corresponds to the columns linked to only this partition and that are to be copied into the δ vector directly. We reproduce this colour scheme in Figure 4.7 to illustrate how this sum is performed in the contiguous case.

In a more generic case, where columns are not contiguous, we have to use the indirections

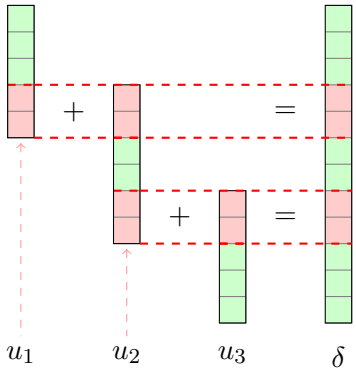


Figure 4.7: The sum of projections in the case of our illustrative example where the columns are contiguous.

defined by the vectors $ColumnIndex_i$ and $ColumnIndex_j$, an illustration of this case is shown in Figure 4.8, where we do a sum of two vectors u_i and u_j, shown on the sides, into a large vector in the middle corresponding to δ. Notice that the indirections obtained by $ColumnIndex_i$ permit us to find the elements that need to be summed and those to copy into δ when no sum is needed, the summed elements are represented in red (hatched pattern) and the copied ones are in green (dotted pattern).

Notice that the δ vector, that is going to be used to update the solution vector later on, is not compressed and its length corresponds to the number of columns of the original matrix. If we compress it then we suppose that the corresponding column in the matrix is null which contradicts our initial supposition that the problem is not rank deficient. However, this limit is only present in the master-slave case, where the master handles the whole matrix.

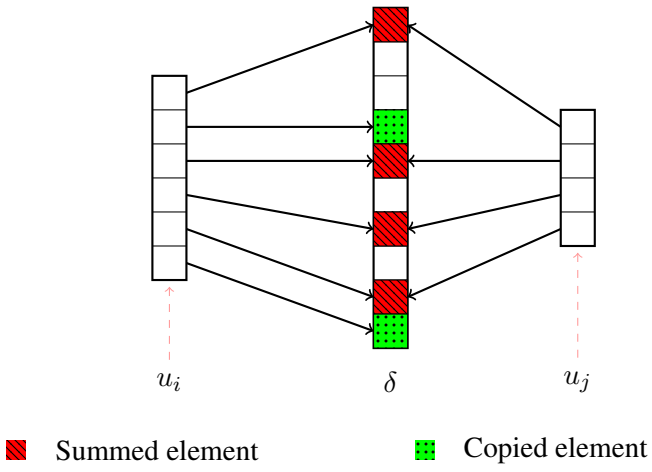


Figure 4.8: Sum of compressed projections.

One can permute the columns of each partition to create contiguous sets of columns so that the sum becomes easier and faster. This is not implemented yet, and should greatly improve the performance as it is cache-friendly.

Work-flow summary. Once the master has created the augmented systems and concatenated them into a block-diagonal matrix that we have called B, it sends it to the direct solver for analysis and factorization as shown in Figure 4.9. Once it is factorized, we launch a sequential Block-CG

in which at each iteration, as we have seen in Chapter 2.4, we compute either a matrix-vector or matrix-matrix product depending on whether the block-size in Algorithm 4.1 is 1 or larger.

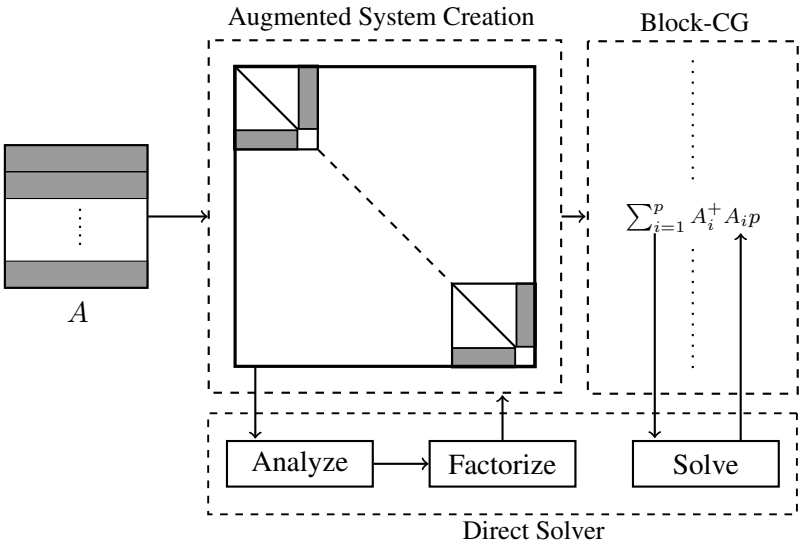


Figure 4.9: The master-slave strategy work-flow

The fact that the bottom box, the direct solver, is independent from the rest of the code makes it possible to launch it in parallel mode. The slaves in our work-flow are all to be used within the direct solver and contribute to the factorization and solution phase.

4.3 Distributed block Cimmino

There are two main issues with the master-slave scheme. Firstly, it may create a bottleneck during the sum due to the fact that all the slaves have to send their results back to the master and the master has to do the sum by itself while the slaves are idling. The second one is that increasing the number of processes will not help during the projection sum due to the more limited scalability of the direct solver's solution phase.

The solution to this problem is simple. Rather than delegating the whole block-CG to a single process, we create a distributed version of it and exploit the available processes to do that. That way, we will reduce the amount of work to be done during the block-CG (during the matrix-matrix products and sum of projections) and unclog the processes during the recovery of the solutions of the augmented systems.

The issue that arises now is that the data structures about the matrix, i.e. the partitions and the compression of columns, no longer resides on a single process (the former master process) but have to be distributed over a few processes that will take part in the distributed block-CG.

4.3.1 Mapping data

There are three cases to consider when running block Cimmino in parallel. Either we have as many processes as partitions, more processes than partitions or fewer processes than partitions.

Similar numbers. In this case, the basic idea is to distribute one partition per process and all processes become part of the distributed block-CG. Therefore each process will build a single augmented system and will call a direct solver instance to solve it sequentially. The communications

between processes will be limited to the interconnections between the partitions belonging to each of these processes.

More processes than partitions. This case is quite similar to the previous one, the only difference is that the remaining processes that did not get a partition will become slaves to the processes participating in the distributed block-CG. The solution of the augmented systems can be performed in parallel if a slave is assigned to it, or in sequential if no slave is assigned to it. We will call the process assigned to a block-CG as **masters** and the rest as slaves.

Less processes than partitions. This situation is a hybrid of the master-slave scheme and the distributed scheme. Indeed, as we are not able to handle a single partition per process, some of them will get more partitions. If they do, they will handle these partitions in a similar way as the master-slave case, i.e. they will build a local block diagonal matrix containing the augmented systems of the partitions they got and they will solve them sequentially. Those processes that got a single partition will behave just like the first case described above.

We show in Figure 4.10 how the processes are distributed in a generic case. The circles represent the processes, the Mk circles are the masters assigned to a block-CG and the empty ones are the slaves. The central, blue, area represents the MPI (Message Passing Interface) communicator to be used by the block-CG masters during their exchanges. The triangular shaped areas, in red, are the communicators that each master will use with its slaves to solve the augmented systems. In this figure we suppose that we have 4 partitions meaning that we cannot (following the scheme we have defined previously) have more masters in the distributed block-CG, and that we have many more processes available to us that are assigned as slaves. The numbering starts from 0 to respect the MPI standard.

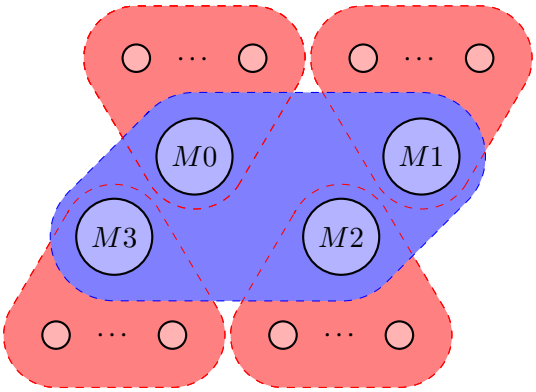


Figure 4.10: Representation of how the processes are organized. The Mk circles are the masters, the other circles are the slaves.

The third case, where we have less processes than partitions, requires a special treatment when distributing the partitions. Indeed, in the first two cases we can simply distribute the partitions successively. However, in the last case we have to do the distribution so that each master has similar work to do and data to store as the others. The simplest way is to use a greedy algorithm that will go through all the partitions, using a weight criteria that determines which partition goes where.

We show in Algorithm 4.2 how this greedy algorithm works. It simply tries to group partitions into sets so that the sets have similar weights. In our case, this algorithm receives as input the weights of the partitions and partitions them into sets that correspond to the numbers of partitions

for each master. The weights can be either the number of rows, the number of entries, or any other criteria related to the partitions that might improve the performance.

Algorithm 4.2 Greedy partitioning

Input: $weights$ and $nb\_masters$
Output: $partitions$

```
for i = 1 → nb_masters do /* Initialize partitions with empty sets and their weights to 0 */
    partitions[i] ← [ ]
    set_weight[i] ← 0
end for
w ← 1
s ← 1
while w ≠ size(weights) do
    for i = 1 → nb_masters do /* Try to find a set of partitions with less weight than the last one */
        if set_weight[i] ≤ set_weight[s] then
            s ← i;
        end if
    end for
    set_weight[s] ← set_weight[s] + weights[w] /* Increase the weight of the current set */
    partitions[s].push(w)
    w ← w + 1
end while
return partitions
```

This version of the algorithm has a major drawback, it distributes the first partitions in a cyclic manner. Therefore, we may finish by having partitions that are not close to each other on the same master. Depending on the structure of the matrix, this can reduce the amount of work needed during the local sums (by having local partitions with few interconnections) but increases the communications with other masters as they might have the closest partitions.

Another way to do this partitioning can be by using a graph partitioner by putting weights onto the vertices and partitioning the graph to get equilibrated sets. A linear graph, as shown in Figure 4.11 where each vertex is connected to its successive vertex will give us the ability to create partition sets $part_k$ containing matrix partitions that are contiguous, which can sometimes be good for communications especially when the interconnected partitions are neighbours. The w_i value is the weight of the matrix partition A_i. As stated previously this can be any property of the partition that the user decides but it is set by default to the number of rows in the partition.

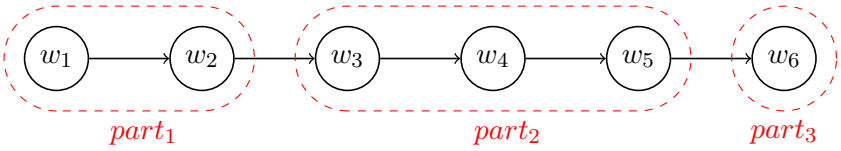


Figure 4.11: Partitioning using a linear graph.

A more complex way to create the partition sets would be to create a graph where the vertices are the weights of each partition and the edges represent the weight of the interconnection (this can be for example the number of columns in the interconnection) represented here by $v_{i,j}$ being the weight of the interconnection between the partition A_i and A_j. To create the partition sets in this case, we will use a multi-criteria graph partitioner so that the weights of each of the vertices

are distributed among the sets and the weights of the edges are maximized inside the set. This would reduce the communication footprint during the sum of projections.

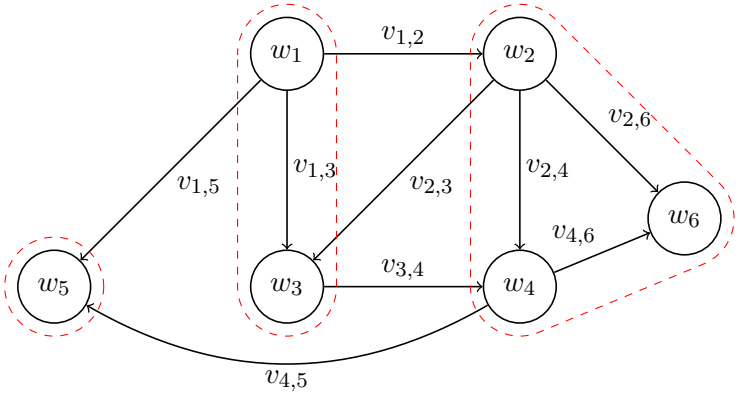


Figure 4.12: Partitioning using a graph defining the interconnections.

After the mapping is done, the partitions are sent to each master with the $ColumnIndex_i$ vector corresponding to each partition. Each master defines m_k as being the total number of rows (sum of the number of rows of all the partitions it has) and n_k as being the number of distinct columns in all its partitions. The value n_k is not the sum of the number of columns due to the simple fact that the partitions can contain overlapping columns. We will see how it works in Section 4.3.3.

4.3.2 Determining the slaves

Once each master has its partition, it creates its augmented system and gives it to the direct solver for an initial analyses, the different analyses are done simultaneously between the masters but each of them runs sequentially. This analysis will determine how many FLOPs are needed for the factorization of the augmented system. The selection of slaves is done sequentially to avoid any conflict between the masters.

The selection is simple, we assign to each master a priority corresponding to the relative local FLOP count. Once a slave is assigned to the top priority master, we reduce this master's priority by 10% and repeat the process.

Once all the slaves are assigned to their masters, we create for each set of master/slaves a communicator that we shall call $intra\_communicator$. This communicator is the red (triangularly shaped) set that we have seen in Figure 4.10. The analysis is launched one more time if the size of the communicator is larger than one so that the direct solver does the extra work needed in the parallel case (the mapping). To avoid having too long an analysis phase, we extract from the first analysis the ordering obtained and give it as input to the direct solver for the second analysis.

4.3.3 Sum of projections

The sum of projections goes through three main steps. The first two steps are similar to Algorithm 4.1 and are applied locally on each master, except if this master has a single partition. Indeed, if we have one partition on a master we do not need to do any sum between the u_i as we have only one of them. The third step is the combination of the δ_k, where k is the current master's number.

We show in Figure 4.13 how the distributed sum of projections work. Each Mk master, computes its u_i, if it has many then it sums them into δ_k; if not, it expands u_1 (as there is only one) into

δ_k. Next, each Mk master communicates with the other masters that it has interconnections with, then sends the elements that are interconnected and receives at the same time its interlocutors' elements. This way every master gets the data it needs and does the sum locally without waiting for others to finish the communications with their other interlocutors. We can of course do the sum on one of two communicating masters. However, it would create a hard synchronization between them.

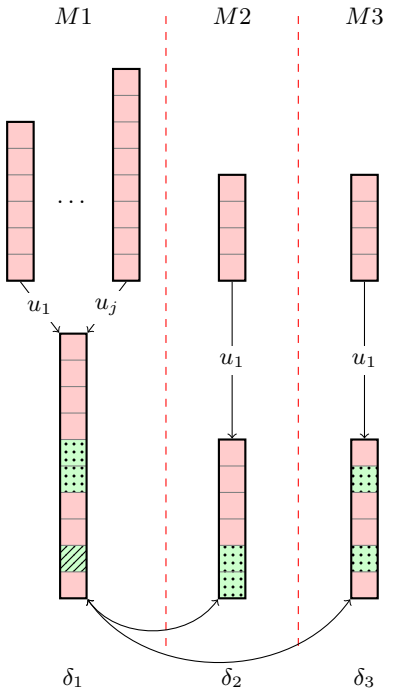


Figure 4.13: Sum of compressed and distributed projections. Red (pattern less) entries are locally computed, green ones (dotted) have been exchanged then summed. The hashed green entries are the entries that have been exchanged with more than one master.

We observe from the previous figure that if two masters do not have interconnections, they do not need to communicate. Moreover, the size of $delta_k$ can be different from a master to another, which means that in the Block-CG iterations, the different matrices R, P, and X have different sizes from one master to another.

4.3.4 Distributed block-CG

The distributed version of the block-CG algorithm uses the same stabilized Algorithm 2.4, shown in Section 2, with one main difference, the matrix-matrix product HP is done using the process described in Section 4.3.3 and the dense matrix-matrix product done during the stabilization processes is done in a distributed manner.

The dense matrix-matrix product is performed using `PBLAS`'s `PDGEMM` routine that will compute it in a distributed manner. However, two issues arise for our case:

- The fact that some entries in P and R are duplicated between the different masters, requires an additional step before launching the product. For each couple of communicating masters, we define which master will handle the duplicate entries, this selection can be done by a simple criteria that can be either the number of local entries or just on a first-arrived first-served basis.

- A redistribution is needed to arrange the data in a block manner. Indeed, `PBLAS` requires either a block, a cyclic or a block-cyclic distribution of data.

This solution requires additional communication and has some a priori requirements such as the distribution of data in a grid manner which is not true in our case.

Our solution uses a simple algorithm, defined in Algorithm 4.3 where we handle both the case of $P^T HP$ and $R^T R$. In this algorithm, the matrix to be transposed is called D and the other W, if we are in the case of $R^T R$, we work only on D and suppose that W is empty.

Algorithm 4.3 Parallel Stabilization: pstab()

Input: D and W
Output: $\bar{D}$, $\bar{W}$ and U

1: **if** $D \neq W$ **then**
2: $\quad \widehat{D} \leftarrow local(D)$
3: $\quad \widehat{W} \leftarrow local(W)$
4: $\quad \widehat{C} \leftarrow \hat{D}^T \hat{W}$
5: **else**/* Handle the case of $D^T D$ */
6: $\quad \widehat{D} \leftarrow local(D)$
7: $\quad \widehat{C} \leftarrow \widehat{D}^T \widehat{D}$
8: **end if**
9: $C \leftarrow all\_reduce\left(\widehat{C}\right)$
10: $U \leftarrow chol(C)$
11: $\bar{D} \leftarrow DU^{-1}$
12: **if** $D \neq W$ **then**
13: $\quad \bar{W} \leftarrow WU^{-1}$
14: **end if**

In the lines 2,3 and 6, we use the function $local()$ that uses a local vector called $CommMap_k$ that defines which entries the current master k will handle. The i-th position of this vector contains 1 if the i-th value in P, R or X is used in the computation and hence it is kept, and contains -1 if

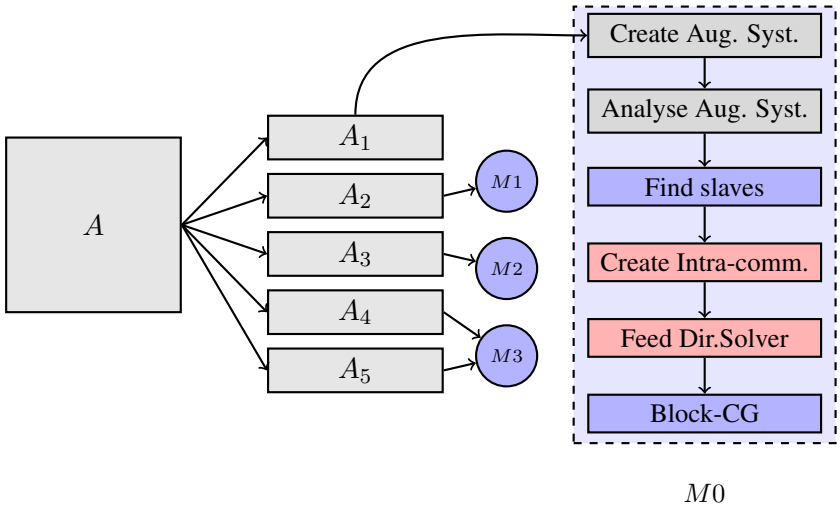


Figure 4.14: The distributed work-flow of the Block-Cimmino method. In blue, the steps that have inter-master communications. In red, the steps that have communications inside the $intra\_communicator$. In gray, the steps that are done sequentially.

it is not. The resulting vectors $\hat{D}$ and $\hat{W}$ are smaller and contain less rows than the original D and W respectively. We then compute in lines 4 and 7 the local dense matrix-matrix product using the `DGEMM` routine and then do a global sum in line 9. This global sum, using `MPI`'s $all\_reduce()$ ensures that all the masters have the same data and avoids a reduce followed by a broadcast implementation. All the masters then do the same factorization at line 10 of an $s \times s$ matrix, where s is the block-size. The block-size is generally small, we do not need a parallel implementation for this factorization. We then return the stabilized vectors $\bar{D}$ and $\bar{W}$ corresponding to $\bar{P}$ and $H\bar{P}$ respectively or $\bar{R}$ only depending on the case.

Workflow. We summarize in Figure 4.14 the workflow of our distributed block-Cimmino implementation. This implementation accepts different levels of parallelism: distributed block-CG only, if we have fewer or as many processes as partitions, and distributed block-CG combined with a parallel direct solver if we have more processes than partitions.

4.4 Numerical experiments

We run the test problems listed in Table 4.1, using our distributed solver and using the block-size that gives the best timings. For the partitioning, we use hypergraph partitioning with medium imbalance. The experiments are performed on `Oakleaf` with 2 MPI processes per node to avoid any memory congestion.

Problem	Order	Nonzeros	Partitions	Block-size
N1 - EDF/R6	132 106	2 103 332	16	16
N2 - lhr71c	70 304	1 528 092	16	32
N3 - torso3	259 156	4 429 042	32	1
N4 - Hamrle3	1 447 360	5 514 242	64	4
N5 - Hamrle3	1 447 360	5 514 242	128	4
N6 - cage13	445 315	7 479 343	256	1
N7 - cage14	1 505 785	27 130 349	1024	1
N8 - nlpkkt80	1 062 400	28 192 672	64	4

Table 4.1: Matrices to be tested. We show their orders, number of partitions and block-size used in block-CG.

Problem	N1	N2	N3	N4	N5	N6	N7	N8
Factorization	1.61	1.02	5.72	16.85	13.99	16.81	60.68	30.93
B-CG iteration	0.90	0.99	0.35	6.73	6.77	3.51	15.35	5.03

Table 4.2: Timings for factorization and for a single Block-CG iteration on 4 MPI processes.

We are interested in the time for factorization of the augmented systems (Figure 4.15) , the average time per iteration (Figure 4.17) and the average memory consumption during the factorization (Figure 4.16). We run with 4 MPI processes, for which timings of factorization and a single block-CG iteration are shown in Table 4.2. We show the time for a single iteration because the number of iterations can have small variations when changing the number of processes because the order of operations may be different and have different rounding properties.

For the factorization we see that the speedups depend greatly on the number of augmented systems (number of partitions) but also on their order and density. We notice the effect of the density of augmented systems when comparing `N4` and `N8`; both have the same number of partitions, and although `N4` is larger, `N8` is denser and so are its augmented systems. Hence, the speedups of the latter problem are better than the former. The same happens with `N1` and `N2`, the former is denser and has better speedups with the same number of partitions. We notice also that for small augmented systems, increasing the number of cores further will result in the reverse situation. We discuss this behaviour (especially on this machine) at the end of Chapter 5.

The number of augmented systems has an important effect on the speedup, indeed we notice that when having more partitions, speedups are better until we have a higher number of processes. To illustrate this effect, we took the same matrix `Hamrle3` and partitioned it into 64 (problem `N4`) and 128 partitions (problem `N5`). We notice that with 128 augmented systems, we do not see a slow down as we do with 64 augmented systems on 64 processes. However, starting from 128 processes, `N4` has better speedup.

To understand this behaviour we have to look at the two phases in these speedups. Firstly, when we have less processes than partitions, the factorization on each master is sequential, thus increasing the number of processes will increase the number of masters, hence decreasing the workload for the factorization. We notice that until we have one partition per process, the speedups are good for all problems with little slow down. Secondly, when we go over this ratio of one partition per master, the factorization becomes parallel and the speedups are related to how parallel the factorization of these augmented systems is. If they are large, the speedups keep a good slope; otherwise we see a lower speedup which explains why `N4` has better speedups at higher process count than `N5`.

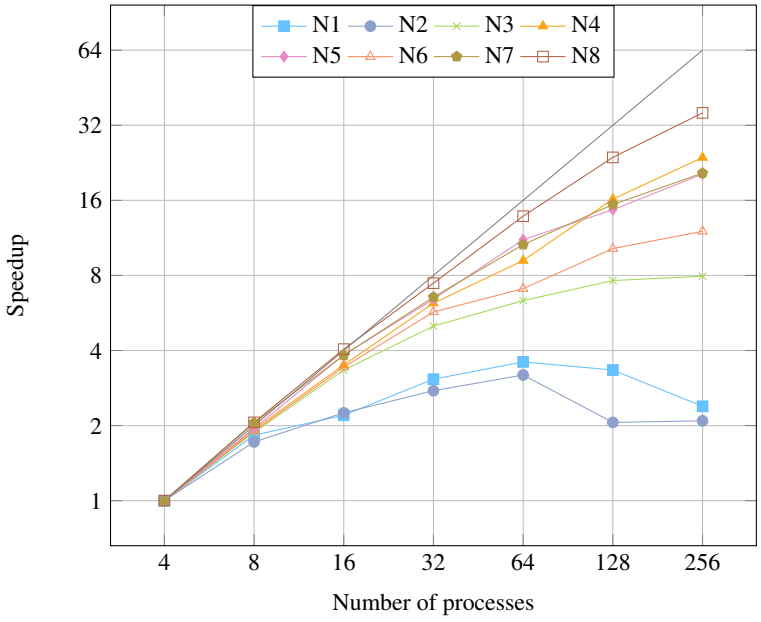


Figure 4.15: Speedups of the factorization of augmented systems. The gray (symbol-less) line represents the ideal speedup.

The direct solver needs a lot of memory to factorize the augmented systems. We see in Figure 4.16 that, as expected, increasing the number of processes decreases greatly the memory usage per process. It is linear in most cases until we have one partition per process, and then becomes super-linear when we have two processes per partition. On some of them, a slowdown is noticed afterwards as expected. These results are an important aspect of the block-Cimmino method, split-

ting the problem more breaks the problem complexity and hence its memory requirement during the factorization.

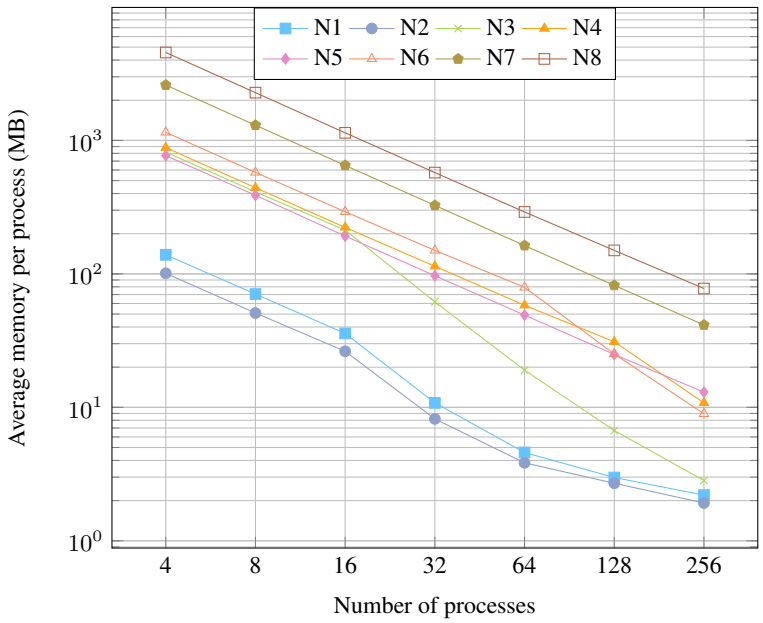


Figure 4.16: The evolution of the average memory (MB) per process when increasing their number.

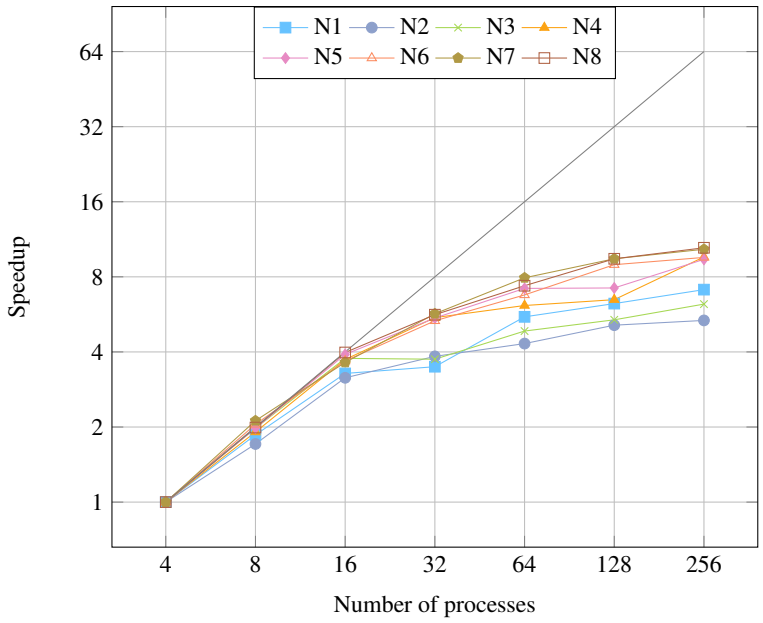


Figure 4.17: Speedups of the block-cg acceleration. The gray (symbol-less) line represents the ideal speedup.

Now, we look at the speedups obtained during block-CG acceleration. Just as for the factorization we have two phases, one when there are less processes than partitions or an equal number and another with more processes than partitions. At the beginning, each master computes the sum of projections sequentially. Therefore reducing the amount of work to be done by increasing

the number of processes will speedup the sum of projections. This gain is not sustained as the communication between the different masters, to update the iterate and during the dot product (or matrix-matrix, depending on the block-size) for stabilization, will slow down the iterations.

A deeper analysis of the different steps in block-CG will reveal possible improvements to our implementation such as computing in parallel the matrix-vector product $\hat{A}_i\hat{z}$, in the projection sum algorithm, adding a higher level of distribution of block-CG by using the available slaves during the other steps of each iteration.

We compare now the results obtained using the direct solver `MUMPS` with the results obtained using block-Cimmino. The runs are done on `Conan` as this machine has a large enough memory. We show in Table 4.3 the timings for *Analysis* and *Factorization* and memory consumption during the factorization in the direct solver and compare them to *preprocessing*, *augmented systems factorization*, *block-CG* timings and average memory consumption per process.

Problem	N1	N2	N3	N5	N6	N7	N8	N8 (256)
[MUMPS] Analysis	1.0	3.10	4.54	171	31		24	
[MUMPS] Fact.	0.56	0.39	6.39	1112	1716	Failed	1802	
[MUMPS] MEM.	21	13	100	1140	5158	38 386	927	
[BC] Preprocess	4.9	2.05	1.32	18.29	11.23	142.81	10	57
[BC] Aug.Sys. Fact.	0.27	1.23	4.10	3.69	2.48	3.79	303	33
[BC] Block-CG	162	173	3.15	346	4.46	19.03	2194	2149
[BC] MEM.	57	59	189	247	186	666	3200	1310

Table 4.3: Comparison of `MUMPS` and distributed implementation of block-Cimmino (`BC`). We use the same problems used earlier and we test with a different partitioning of the matrix in problem `N8` with 256 partitions.

We notice from these results that on small problems such as `N1` and `N2`, the direct solver consumes less memory than our approach and is able to analyse and factorize the linear system faster. On `N3`, we are able to get comparable timing with slightly more memory usage. The larger problem `N8` gives extremely slow convergence with 64 partitions and consumes more memory than the direct solver, however increasing the number of partitions decreases the factorization time of augmented systems, the memory usage per core and slightly the time to converge.

On the other hand, on the other problems we are able to achieve better results in term of timing and memory usage. The factorization of the augmented systems becomes negligible compared to factorizing the original linear system. The most time consuming part in our method is the block-CG and in the direct solver it is the factorization. On the problems `N5` and `N6`, our method performs better than the direct solver in term of time and memory usage. Indeed, we use only 20% and 4% respectively of the memory required by the direct solver. For problem `N7`, we did not put factorization results as it failed for lack of memory. The analysis estimated an average of 38GB of memory per core while our method used only 666MB per core to solve this problem.

4.5 Concluding remarks

We have described in this chapter the hybrid parallel implementation of the block Cimmino method. We started by describing our earliest attempt to build a parallel solver using the master-slave model. Then we migrated into a fully distributed model for both the block-CG and the direct solver. We discussed the different algorithms involved in that implementation and suggested possible improvements to reach better speedups.

We performed runs on a variety of problems with different sizes and numerical properties that showed the parallel potential of our implementation. Moreover, we compared it against a direct solver and have seen that, on some large problems our implementation can be faster and that it consumes much less memory in general than the direct solver. This is thanks to the splitting of the original problem and the consequent reduction of its complexity during the factorization which induces lower fill-in and hence the lower memory usage. The large memory usage by the direct solver `MUMPS` is also being investigated in [5] by using low-rank approximation techniques.

Chapter 5

The Augmented Block Cimmino

We introduce and study a novel way of accelerating the convergence of the block Cimmino method by augmenting the matrix so that the subspaces corresponding to the partitions are orthogonal. This results in building and solving a relatively smaller symmetric positive definite system.

In our discussion on the theoretical properties of our algorithm, we will use a realistic test matrix to illustrate this behaviour. This is the matrix `bayer01` obtained from Bayer AG by Friedrich Grund and available from the sparse matrix collection at the University of Florida [36]. It is of order 57735 and has 277774 nonzero entries. We partition it into 16 uniform partitions, and we show its pattern in Figure 5.1. In Figure 3.15, we have shown the spectrum of the iteration matrix, H, for the `bayer01` matrix when using block Cimmino with these 16 uniform partitions. This matrix has been studied in Chapter 2 and Chapter 3. Later, in Section 5.4, we extend the study by testing the method on larger problems.

Although we see that there is a good clustering of eigenvalues around the value 1 (and this property is true in general) the matrix may still be badly conditioned. One way to increase the clustering while reducing the ill-conditioning is to open the angles between subspaces corresponding to the different partitions. One method for doing this is to reorder the matrix and partition it following the level sets obtained from the use of the Cuthill-McKee algorithm on AA^T [55]. Although this helps to open the angles, it is costly, difficult to implement in a distributed memory environment, and does not always give better results.

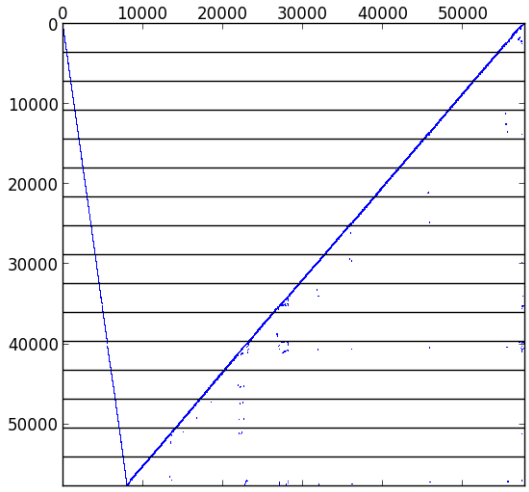


Figure 5.1: Nonzero pattern of the matrix `bayer01`.

As an alternative, we use a hypergraph partitioner `PaToH` [56] to find a row permutation

that will make the partitions less interconnected. PaToH provides permutations so that the reordered matrix is in bordered block diagonal form where the blocks on the diagonal will usually be underdetermined. We notice that, with the matrix in this form, the overlap is only within the boundary columns that PaToH is trying to minimize. One can input to PaToH the desired number of partitions and, in this present comparison, we use the value 16, the same number as for our uniform partition. A deeper comparison of the benefits of using a hypergraph partitioner with block-Cimmino was discussed in Chapter 3.

As we have seen in the previous chapters, a better partitioning combined with stabilized block-CG may improve the convergence of the method. Further techniques, such as the combination of Chebyshev preconditioning with CG can be beneficial to extract some near-invariant subspace (corresponding to all eigenvalues below some threshold, typically $\lambda_{max}/100$) from the Krylov subspace and to reuse this spectral information to speed up further solutions with the same matrix but new right-hand sides [31]. Such techniques are, however, limited because there may be many eigenvalues in such fixed intervals especially for very large size linear systems, and so the memory requirements may become prohibitive in some cases.

5.1 The augmented block Cimmino method

For simplicity, assume that we have a matrix with a block tridiagonal structure as shown in Figure 5.2(a). Note that the block Cimmino method can work with any matrix structure.

In this figure we have defined four partitions A_1 to A_4. As we can see in this practical example, each partition is interconnected only with its neighbours. Thus, the product of these partitions can be represented by the following :

$$A_i A_j^T = \begin{cases} 0 & \text{if } j \neq i \pm 1 \\ A_{ij} A_{ji}^T & \text{if } j = i \pm 1 \end{cases}$$

where A_{ji} and A_{ij} are the submatrices of A_i and A_j respectively that overlap column-wise with each other, see Figure 5.2(a).

We then augment the matrix A to generate a matrix $\bar{A}$ with partitions $\bar{A}_i$ so that the inner products $\bar{A}_i \bar{A}_j^T$ are zero. We consider three different ways to augment the matrix to obtain these zero matrix products.

- One can repeat the submatrices A_{ij} and A_{ji}, reversing the signs of one of them as in the following

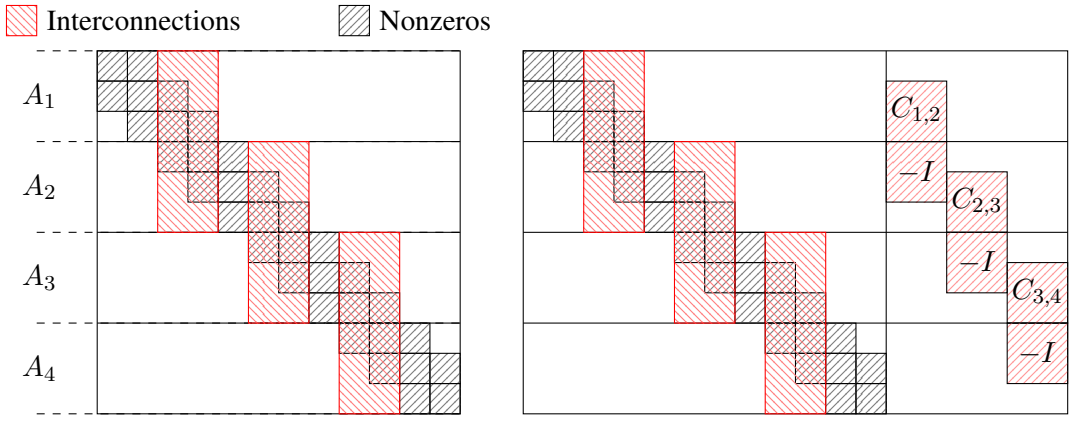
$$\left[\begin{array}{ccccc|cc} A_{1,1} & A_{1,2} & & & & A_{1,2} & \\ & A_{2,1} & A_{2,2} & A_{2,3} & & -A_{2,1} & A_{2,3} \\ & & & A_{3,2} & A_{3,3} & & -A_{3,2} \end{array}\right] \tag{5.1}$$

- We can also use the normal equations so that, for each pair of neighbouring partitions A_i and A_j, we expand them on the right with $C_{ij} = A_{ij} A_{ji}^T$ and $-I$ respectively as in the following

$$\left[\begin{array}{ccccc|cc} A_{1,1} & A_{1,2} & & & & C_{1,2} & \\ & A_{2,1} & A_{2,2} & A_{2,3} & & -I & C_{2,3} \\ & & & A_{3,2} & A_{3,3} & & -I \end{array}\right] \tag{5.2}$$

  It would also be possible to expand with $-I$ and C_{ij}^T which will reduce the number of augmenting columns if C_{ij} has more columns than rows.

- Finally, we can use an SVD decomposition of C_{ij}, viz. $C_{ij} = U_{ij} \Sigma_{ij} V_{ij}$. We then replace C_{ij} by the product $U_{ij}\Sigma_{ij}$ and $-I$ by $-V_{ij}$. We will discuss the possible benefits of this



(a) Block tridiagonal structure (b) Normal equations based augmentation

Figure 5.2: Illustrative example of an augmentation process.

approach in Chapter 6 when investigating ways to compress information in these various augmentation approaches.

Once we build these submatrices, we place them so that they start in the same column as shown in Figure 5.2(b).

In the case of augmentation by repeating the submatrices A_{ij} and A_{ji}, the product of each pair of partitions $\bar{A}_i$ and $\bar{A}_j$ gives

$$\bar{A}_i \bar{A}_j^T = A_{ij} A_{ji}^T - A_{ij} A_{ji}^T,$$

and in the case of the augmentation using normal equations C_{ij} we obtain

$$\bar{A}_i \bar{A}_j^T = A_{ij} A_{ji}^T - C_{ij}.$$

Notice that in both cases we get $\bar{A}_i \bar{A}_j^T = 0$.

Which of these alternatives to use will depend on the structure of the overlaps and we want to choose the one that minimizes the number of augmenting columns. In the following, we first concentrate on the C_{ij} and $-I$ augmentation as shown in Figure 5.2(b) before we compare, in Table 5.2, its behaviour with the approach (5.1). To prevent the C_{ij} blocks from different neighbours creating new interconnections, we shift them side by side columnwise as shown in the illustrative example. We note that in cases where there is a largely full column in the submatrices A_{ij}, the second formulation (5.2) might involve a full C_{ij} and so become too expensive with respect to the first formulation (5.1).

The resulting augmented matrix is in all cases of the generic form $\bar{A} = \begin{bmatrix} A & C \end{bmatrix}$. However, to ensure that our new system $\bar{A} \begin{bmatrix} x \\ y \end{bmatrix} = b$ has the same solution x, we add extra constraints to the system to force y to be equal to 0, resulting in the new system

$$\begin{bmatrix} A & C \\ 0 & I \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

$$\begin{array}{lccc} \text{hence} & Ax + Cy & = & b \\ \text{and} & y & = & 0 \\ \text{thus} & Ax & = & b. \end{array}$$

Now that we have shown that we have the same solution from our new system, we have one remaining problem. The partitions in $\begin{bmatrix} A & C \end{bmatrix}$ are mutually orthogonal, but they are not orthogonal

with the extra rows in the bottom partition $Y = \begin{bmatrix} 0 & I \end{bmatrix}$, where I has the same size as the number of columns of C. Therefore, we project Y^T onto the orthogonal complement of the upper part using the orthogonal projector

$$P = P_{\mathcal{R}(\bar{A}^T)} = P_{\bigoplus_{i=1}^{p} \mathcal{R}(\bar{A}_i^T)} = \sum_{i=1}^{p} P_{\mathcal{R}(\bar{A}_i^T)}$$

which holds as a sum because of the enforced numerical orthogonality between the blocks $\bar{A}_i$ of $\bar{A}$.

The resulting projected set of rows

$$\begin{bmatrix} B & S \end{bmatrix} = W = Y\,(I - P) \tag{5.3}$$

is orthogonal to all other partitions of $\bar{A}$. However, the right-hand side has to change to maintain the same solution, viz

$$\begin{aligned} f = \begin{bmatrix} B & S \end{bmatrix} \begin{bmatrix} x \\ 0 \end{bmatrix} &= Y\,(I - P) \begin{bmatrix} x \\ 0 \end{bmatrix} \\ &= -YP \begin{bmatrix} x \\ 0 \end{bmatrix} \\ &= -Y\bar{A}^{+}\bar{A} \begin{bmatrix} x \\ 0 \end{bmatrix} \\ &= -Y\bar{A}^{+}b \end{aligned}$$

resulting in the new linear system

$$\begin{bmatrix} A & C \\ B & S \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ f \end{bmatrix} \tag{5.4}$$

which, if $f = -Y\bar{A}^{+}b$, has the same solution as the previous one, with x corresponding to the solution of the original system (2.1).

Once the new augmented matrix has been built, we can apply block Cimmino to it by keeping the partitions that were defined for $\bar{A}$ and including W as a single partition. The eigenvalues of the iteration matrix in this case are all 1. This contrasts with the spectrum of the non-augmented system shown in Figures 3.14 and 3.15. Since all eigenvalues of the iteration matrix are 1, the block Cimmino method will converge in one step. That is, because of the orthogonality of $W = [B \;\; S]$ with $\bar{A}$, the solution is given by:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \bar{A}^{+}b + W^{+}f. \tag{5.5}$$

To illustrate what we said previously, we show a picture of the partitioned $\bar{A}$ built from the `bayer01` matrix in Figure 5.3, and the S matrix generated from the augmentation process in Figure 5.4. This is a graphic illustration of the relative sizes of the unaugmented matrix and the matrix S. Although it is relatively much smaller, the dimension of S is still 918 which can be reduced to 804 by using transposes of the submatrices C_{ij} if they are rectangular and have more columns than rows (as mentioned above). Although the order of C is 1.4% of the dimension of the original matrix, the construction and solution of this matrix can still be a significant part of the overall computation which is why we seek to reduce this order in Chapter 6.

We compare in Table 5.1 the size of S for three of our test matrices partitioned either uniformly with p equally sized partitions, or using `PaToH` with the same number of partitions. From this we see that, in the case of uniform partitioning, the dimension of S can be even greater than the

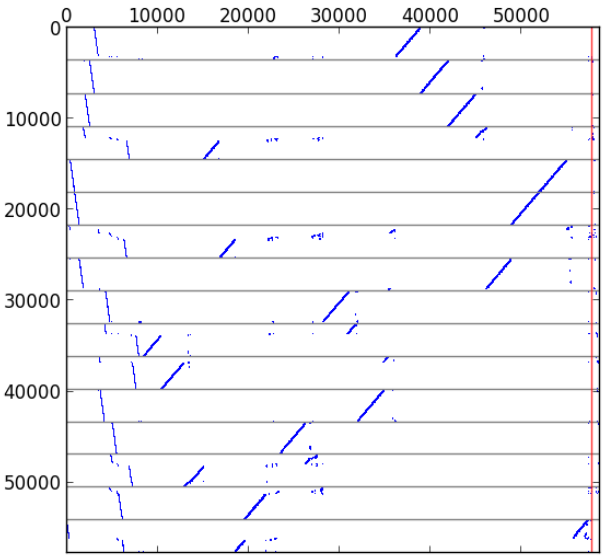


Figure 5.3: Pattern of the $[A\ C]$ part when augmenting the matrix `bayer01`.

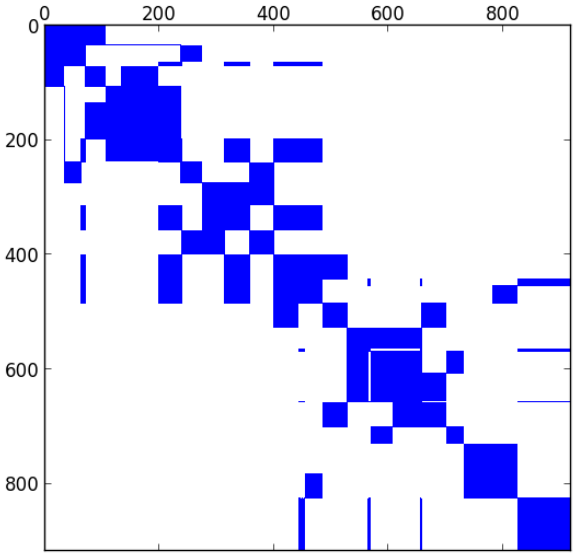


Figure 5.4: Pattern of the S matrix for the augmented system for the matrix `bayer01`.

original dimension and also that the strategy of using C_{ij} or C_{ij}^T, whichever has fewer columns (shown in the column **Reduced** S in the table), can be very beneficial. Similar results are found in all our test cases. Therefore, we will use these parameters (`PaToH` partitioning and S reduction) throughout the rest of this paper.

Keeping in mind the previous assumptions, we compare in Table 5.2 the size of S, on a larger set of matrices, when augmenting the matrix either using the normal equations C_{ij} or using the submatrices $A_{ij}/-A_{ji}$. We notice that most of the time, except for the `gre_1107` and `EDF/R6`, the $A_{ij}/-A_{ji}$ augmentation gives a smaller S.

In addition to all these parameters, the number of partitions has a considerable effect on the size of S. If we partition the matrix `Hamrle3` for instance into 128 partitions, the size of S increases to 67679. However, as we will show in Section 5.4, increasing the number of partitions

Matrix	Size	#Part.	Partitioning	Size of S	Reduced S
bayer01	57735	16	Uniform	2951	2469
			PaToH	918	804
lhr34c	35152	8	Uniform	10 108	1803
			PaToH	1815	1470
lhr71c	70304	16	Uniform	20 283	3671
			PaToH	3203	2452

Table 5.1: Dimension of S for some test matrices with C_{ij} augmentation.

Matrix	Size	#Part.	Augmentation	Size of reduced S
gre_1107	1107	5	C_{ij} based	373
			A_{ij} based	412
lhr34c	35152	8	C_{ij} based	1470
			A_{ij} based	919
bayer01	57735	16	C_{ij} based	804
			A_{ij} based	542
lhr71c	70304	16	C_{ij} based	2452
			A_{ij} based	2044
EDF/R6	132106	16	C_{ij} based	6400
			A_{ij} based	6506
bmw3_2	227362	16	C_{ij} based	19 255
			A_{ij} based	16 695
Hamrle3	1447360	64	C_{ij} based	64 130
			A_{ij} based	46 564

Table 5.2: Comparison of the size of S with different augmentation approaches.

can be helpful when exploiting parallelism.

5.2 The matrices W and S

In this subsection, we examine some properties of the matrices W and S. The size of S depends directly on the number of columns in the C block. Thus, fewer interconnections between the partitions imply a reduced size of C and S.

Since $\begin{bmatrix} B & S \end{bmatrix} = Y(I - P)$, we see that

$$S = Y(I - P)Y^T \tag{5.6}$$

from which, as P is an orthogonal projection matrix, we can immediately see that S is symmetric.

Since S is a restriction of the orthogonal projector $(I - P)$ whose eigenvalues are in the range $[0, 1]$, the eigenvalues of S are in the range $[0, 1]$. We see this in Figure 5.5.

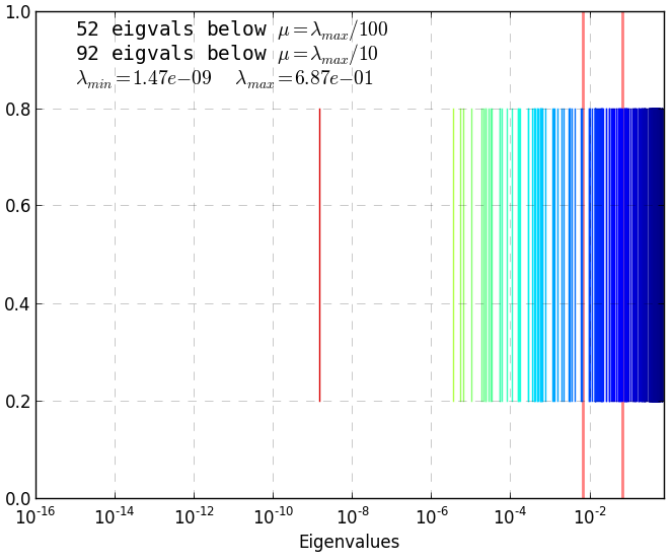


Figure 5.5: Eigenvalues of S matrix for `bayer01`.

From the definition of W,

$$\begin{aligned} WW^T &= [B\ S][B\ S]^T \\ &= BB^T + SS^T \\ &= BB^T + S^2 \end{aligned} \tag{5.7}$$

but also from equation (5.3) and because $(I - P)$ is an orthogonal projector, we can compute WW^T as

$$\begin{aligned} WW^T &= Y\,(I-P)\,(I-P)^T\,Y^T \\ &= Y\,(I-P)^2\,Y^T \\ &= Y\,(I-P)\,Y^T \\ &= S \end{aligned} \tag{5.8}$$

and thus

$$BB^T = S - S^2 \tag{5.9}$$

(which also shows that the eigenvalues of S lie between 0 and 1).

The matrix $S = Y(I - P)Y^T$ reflects the bad conditioning of the Cimmino iteration matrix. Hopefully, the size of S is small enough for this approach to be better than the original Cimmino iteration. This helps to reduce the length of plateaux in the convergence of conjugate gradients while having Krylov spaces of smaller dimensions. This aspect is also seen in domain decomposition methods where the problem is condensed into a smaller matrix called the Schur complement that is usually denoted by S (we have chosen the S notation as an analogy with this).

As W is a partition in our augmented block Cimmino algorithm, the solution obtained using equation (5.5) involves W^+. Since W^+ can be expressed as $W^T(WW^T)^{-1}$, we have from equation (5.8) that

$$W^+ = W^T S^{-1} = (I - P)Y^T S^{-1}. \tag{5.10}$$

This involves only S and P. Therefore, the computation using W^+ can be easily performed.

5.3 Solving the augmented system

Due to the orthogonality between the partitions of $\bar{A}$, we have

$$\bar{A}^{+}b = \sum_{i=1}^{p} \bar{A}_i^{+} b_i \quad \text{and} \quad P = \sum_{i=1}^{p} P_{\mathcal{R}(\bar{A}_i^T)} \tag{5.11}$$

which can be used jointly with equation (5.10) to build a solution of the augmented system

$$\begin{bmatrix} A & C \\ B & S \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ f \end{bmatrix},$$

through equations (5.5) and (5.3). This solution can be expressed as

$$\begin{aligned} \begin{bmatrix} x \\ y \end{bmatrix} &= \bar{A}^{+}b + W^{+}f \\ &= \sum_{i=1}^{p} \bar{A}_i^{+} b_i - (I - P)\, Y^T S^{-1} Y \sum_{i=1}^{p} \bar{A}_i^{+} b_i. \end{aligned}$$

Solving this system becomes quite simple to do and can be done by following the steps described in Algorithm 5.1.

Algorithm 5.1 Solve using **ABCD**

Input: $\bar{A}$ and b
Output: x

1: Build $w = \bar{A}^{+}b$, using equation (5.11), and then by simple restriction set $f = -Yw$
2: Solve $Sz = f$ (using, for instance, a direct solver as S should be small)
3: Expand $\bar{z} = Y^T z$ and then project it onto the null space of $\bar{A}$ viz. $u = (I - P)\,\bar{z}$
4: Then sum $w + u$ to obtain the solution $\begin{bmatrix} x \\ y \end{bmatrix}$, where $y = 0$

Note that we don't need to build B explicitly, only S is used. In that respect, since S is of smaller size and symmetric positive definite, we can either build S and factorize it or use it implicitly in a conjugate gradient procedure through matrix-vector products with $Y(I - P)Y^T$ which implies a sequence of contraction, projections and expansion of the vector.

In this section we have introduced a new technique to solve linear systems based on this augmentation approach for the block-Cimmino method, and we shall call it the **ABCD** method, that stands for *Augmented Block Cimmino Distributed method*. Next, we will investigate the performance of the **ABCD** method on the set of problems listed previously.

5.4 Parallel ABCD and Numerical results

We shall now investigate in detail the building of S in parallel. We recall the three main postulates that we have discussed in Chapter 4 about the parallel implementation of the method.

- If we have more partitions than processes, we will distribute one or more partitions per process so that each process has similar work to do.
- If we have as many partitions as processes, we distribute one partition per process.
- If we have more processes than partitions, we distribute the partitions just like in the previous case, and the remaining processes are incorporated as slaves within the different MPI communicators created for parallel factorization and solution of the augmented systems.

A process containing a partition is called a master and is denoted by M_k where k is the process number. Figure 5.6 illustrates how the processes are distributed, large circles are the masters and small ones are the slaves. This figure assumes that there are 4 partitions and many more processes. Each M_k process has its own partition (or set of partitions) and works with the slaves in its communicator (triangularly shaped area).

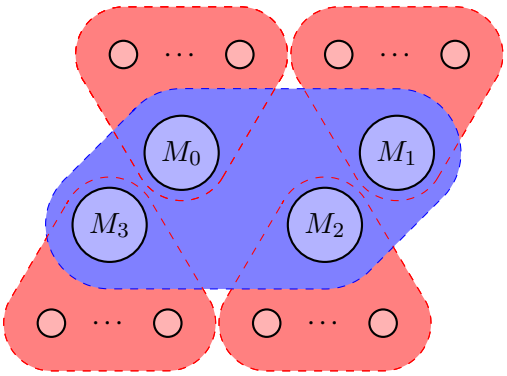


Figure 5.6: Representation of how the processes are organized, the M_k circles are the masters, the other circles are the slaves.

We know that each column of S is a sum of projections applied to the corresponding canonical vectors. By rewriting S as being

$$S = Y\,(I - P)\,Y^T = YD,$$

we see that each column of D can be defined as

$$D(:, l) = e_l - \sum_{i=1}^{p} \bar{A}_i^+ \bar{A}_i e_l.$$

The vector resulting from the product $\bar{A}_i e_l$ is sparse, therefore we can use the direct solver's feature of sparse right-hand side inputs. This feature in `MUMPS` improves greatly the speed of the solution phase as we will see later.

What we also know is that each column of S, hence D, corresponds to one and only one pair of partitions that are interconnected. Indeed, as we have seen in the previous section, the augmentation is done by pairs of partitions, therefore a canonical vector will trigger only one particular couple of partitions and the other partitions will yield a zero vector during the projection simply because $\bar{A}_i e_l = 0$ in most cases (in the summation above) . Hence, we are able to rewrite the previous equation as

$$D(:, l) = e_l - \left(\bar{A}_i^+ \bar{A}_i e_l + \bar{A}_j^+ \bar{A}_j e_l\right), \tag{5.12}$$

where the column l, in this case, is related to the pair of partitions $\bar{A}_i$ and $\bar{A}_j$. This aspect opens an opportunity to compute the columns of S in parallel in a very cheap way.

Distributed computation. Suppose that we have many processes available to us, each of those holding one or more partitions. We build D_k, a partial computation of D, where k is the k-th master. The l-th column of D_k satisfies equation (5.12) and is written as:

$$D_k(:, l) = \left(\frac{1}{2} e_l - \bar{A}_i^+ \bar{A}_i e_l\right) + \left(\frac{1}{2} e_l - \bar{A}_j^+ \bar{A}_j e_l\right), \tag{5.13}$$

and we can easily see that D is the sum of all the D_k partial contributions.

To illustrate this process we introduce a simple example in Figure 5.7 with four partitions and three masters sharing these partitions. We can illustrate all the possibilities related to the pair of partitions A_i and A_j using process M_0:

- The pair of partitions are on this process.
- Only one partition of the pair is on this process.
- The pair of partitions are on other processes.

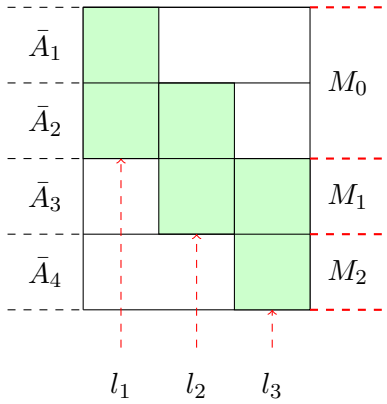


Figure 5.7: A representation of the different cases when computing a column of D. M_k is the k-th master, M_0 holds two partitions while the others have only one.

This affects the computation of the columns of D_0 as the column l is also one of the three cases l_1, l_2 and l_3. The corresponding computed column $D_0(:, l)$ is handled differently depending on the position of the column l as follows

- **As l_1 involves $\bar{A}_1$ and $\bar{A}_2$** the computed column $D_0(:, l_1)$ corresponds directly to the fully summed column $D(:, l_1)$ (performed locally on M_0).
- **As l_2 involves $\bar{A}_2$ and not $\bar{A}_1$** the computed column $D_0(:, l_2)$ contains only part of the information in the corresponding column in D (shared by M_0 and M_1). It should be summed later with the other partial computation of the column l_2 computed by the other master holding it as a local column. This can be done by doing a distributed sum between the two related processes, we will see later that we can avoid this explicit sum and perform it at the factorization time as the matrix S is being factorized.
- **As l_3 does not involve $\bar{A}_1$ or $\bar{A}_2$** this column is not used in the computation of D_0 and is considered empty. Therefore, there is no need to do the projection.

By following these rules, we are able to reduce the amount of work each process has to do to compute its part D_k of the final D. Notice also that we do not need to store the whole D_k but we can store only the bottom part corresponding to partials of the matrix S that we shall denote as S_k.

Assembly of S. Building the final S before giving it to the direct solver can be done either by summing the local parts S_k held by each master and building the fully summed matrix in a centralized way, or by exchanging the corresponding columns to do the sum in a distributed manner, then giving the direct solver a distributed matrix as input. We see that the later solution

is more appealing, as S will be kept distributed, and the only communication to be done is related to the columns in the second case –the l_2 one.

One way to do this distributed sum is by asking the masters to send their interconnected columns to the master of their paired partition. The master that has less work will do the sum of duplicate entries (entries with same row/column indices), and ignores the remaining entries. The other master will just nullify the duplicate entries in its local S_k. In this way we avoid having the same entries (duplicates) on both masters.

Another way to do the sum is by relying on the direct solver to handle it. Indeed, the direct solver `MUMPS` detects duplicate entries and does the sum during the factorization when assembling the fully summed variables. The matrix S can be given both in a distributed manner and with duplicate entries to the direct solver, therefore no modification is done to the S_k matrices before giving them to the direct solver. This method is expected to be more reliable than doing the sum explicitly (error prone and communication hog), and this contributes even better to the hybrid scheme we are trying to settle.

Problem	**Size**	**Nonzeros**	**Partitions**	**Application**
N_1: lhr71c	70 304	1 528 092	16	Chemical Process simulation
N_2: EDF/R6	132 106	2 103 332	16	Structural Problem
N_3: bmw3_2	227 362	5 757 996	16	Structural Problem
N_4: Hamrle3	1 447 360	5 514 242	64	Circuit Simulation

Table 5.3: The list of problems tested with ABCD.

Parallel experiments. We first look at the effect of the blocking factor (the number of columns of S_k) to be computed at once by the direct solver. We consider matrix $Hamrle3$, partitioned into 64 partitions, and we compare the building of S with a blocking factor of 64, 128 and 256. The runs are made on a 32 core (8 sockets) AMD machine with 500GB of shared memory. Results are presented in Table 5.4 where we list the time to build S and the average memory consumption. We show the average total memory used by ABCD while it is building S, it contains the memory used by the direct solver to factorize the augmented systems, the memory to store the multiple right-hand sides, and the memory used by ABCD to store the partitions, the vectors and the S_k matrix.

	32	64	128	256
Time to build S (seconds)	66.2	54.3	51.9	53.3
Average memory per core (MB)	316	385	441	618

Table 5.4: Comparison of the time it takes to build S and the average memory needed when changing the blocking factor.

We see from these results that increasing the blocking factor can help to speedup the building of S but will get worse at higher levels. This effect is architecture dependent (size of cache) and the optimal value could be significantly different on other machines. Moreover, our current implementation can be improved to avoid the conversion from dense solution vectors obtained from the direct solver to sparse vectors to be stored in S_k. If the direct solver supports sparse solution output, this operation will be seamless.

In the following, we test **ABCD** on some of the matrices considered previously. They are all coming from the sparse matrix collection at the University of Florida [36], except for the `EDF/R6` from `EDF`. The matrices are listed in Table 5.3 in which we show the matrices name, their sizes, the partitioning applied and the application category.

The number of partitions is chosen based on the results obtained in the previous chapters. A smaller number of partitions will increase the time to factorize the augmented systems and, as we will see later, can decrease the performance of ABCD.

The runs are performed on the `Hyperion` supercomputer. For each problem we will test three cases: $p/2$, p and $2 \times p$ cores, where p is the number of partitions. This enables us to test all the possible situations that our algorithm might encounter. We compare in Tables 5.5, 5.6, 5.7 and 5.8 the results of running block-Cimmino and **ABCD** on each of the problems listed in Table 5.3. The results shown in these tables are split into two sets of columns, the first set lists the block-size used for the block-CG acceleration, the time to factorize the augmented systems resulting from the partitions of the matrix A and the time it takes for the block-CG to converge. In the second set, we show the size of S, the time it takes to augment the matrix A and build $\bar{A}$, the time it takes to factorize the partitions of the augmented matrix $\bar{A}$, the time it takes to build S in parallel and the time to factorize S.

We use a developer version of the direct solver `MUMPS` in these runs. This version has an improved solution phase with multiple right-hand sides and is optimized for sparse right-hand sides. We will discuss later how the direct solver, and its parameters, affect the results.

`lhr71c` (16 parts)	**Block-Cimmino** (blk. size = 32)			**ABCD** (size S = **2044**)		
Nb. processes	8	16	32	8	16	32
Building $\bar{A}$	-	-	-		0.04 s.	
Augmented Sys. factorization	3.62 s.	3.12 s.	2.22 s.	4.61 s.	4.33 s.	2.77 s.
Block-CG	340 s.	175 s.	172 s.	-	-	-
Building S	-	-	-	10.14 s.	7.08 s.	6.73 s.
Factorization S	-	-	-	0.08 s.	0.08 s.	0.08 s.

Table 5.5: Comparison of timings of the different steps involved in block-Cimmino and **ABCD** on the problem `lhr71c`.

`R6` (16 parts)	**Block-Cimmino** (blk. size = 16)			**ABCD** (size S = **6506**)		
Nb. processes	8	16	32	8	16	32
Building $\bar{A}$	-	-	-		0.08 s.	
Augmented Sys. factorization	0.56 s.	0.34 s.	0.26 s.	0.57 s.	0.37 s.	0.33 s.
Block-CG	119 s.	95 s.	84 s.	-	-	-
Building S	-	-	-	7.31 s.	2.65 s.	2.44 s.
Factorization S	-	-	-	1.01 s.	0.79 s.	1.14 s.

Table 5.6: Comparison of timings of the different steps involved in block-Cimmino and **ABCD** on the problem `R6`.

The first thing that we notice by comparing the **ABCD** approach to the block-Cimmino one is that the augmentation does not really penalize the factorization of the augmented systems. The highest increase in time was in the case of the `lhr71c` matrix where we notice a 27% increase with 8 processes. We notice also that the factorization of the augmented systems presents nice speedups in general and especially for large problems.

Another important aspect is the generally low cost of the augmentation (building $\bar{A}$). The time presented in the tables is common to all parallel cases as this augmentation is sequential (the

bmw3_2 (16 parts)	**Block-Cimmino**			**ABCD** (size S = **16695**)		
Nb. processes	8	16	32	8	16	32
Building $\bar{A}$	-	-	-		0.04 s.	
Augmented Sys. factorization				4.61 s.	4.33 s.	2.77 s.
Block-CG		Failed		-	-	-
Building S	-	-	-	10.14 s.	7.08 s.	6.73 s.
Factorization S	-	-	-	0.08 s.	0.08 s.	0.08 s.

Table 5.7: Comparison of timings of the different steps involved in block-Cimmino and **ABCD** on the problem bmw3_2.

Hamrle3 (64 parts)	**Block-Cimmino** (blk. size = 4)			**ABCD** (size S = **46564**)		
Nb. processes	32	64	128	32	64	128
Building $\bar{A}$	-	-	-		2.88 s.	
Augmented Sys. factorization	2.54 s.	2.08 s.	1.17 s.	2.96 s.	2.25 s.	1.22 s.
Block-CG	177 s.	152 s.	139 s.	-	-	-
Building S	-	-	-	32.91 s.	21.28 s.	20.83 s.
Factorization S	-	-	-	10.18 s.	14.39 s.	26.15 s.

Table 5.8: Comparison of timings of the different steps involved in block-Cimmino and **ABCD** on the problem Hamrle3.

distributed version has not been implemented yet). This timing includes the detection of the A_{ij} and A_{ji} blocks for all partitions and the augmentation process (the duplication of these columns).

Building the columns of S is the most time consuming part. We notice also that in most cases the speedups are good until we reach as many processes as there are partitions (the number is given in Table 5.3). This is explained by the fact that under that ratio each process has a larger amount of columns to compute and a lot of explicit sums to handle too. Once we have as many processes as there are partitions we reach the minimum number of columns to be handled per process (first level of parallelism). Hence, when increasing the number of processes we are adding slaves that should help during the solution phase and the factorization of S (second level of parallelism).

The results we obtain during the building of S are not satisfactory when increasing the number of processes above the number of partitions. A deeper analysis of the behaviour of our solver and the direct solver used to compute the columns might reveal the problem. However, we know that two factors are involved in building the columns of S. The first is the number of columns to be computed per process that stays unchanged when the number of processes increases. The second is that except for the computation of the dense solutions in parallel by the direct solver the rest is sequential. This is spent preparing the input for the direct solver and creating a sparse local S_k matrix from the solutions.

Before we analyse the factorization of S (at the end of the chapter), let us look at a possible solution, but a temporary one. We can try to decrease the average number of columns to be handled per process. We can achieve that by increasing the number of partitions, which will increase the size of S but at the same time decrease the size of the interactions between partitions, hence the number of columns to be computed per process.

To illustrate that, consider the Hamrle3 case, if we partition it into 128 partitions the size of S increases from $46,564$ (in the case of 64 partitions) to $67,679$. In this case, the average number of columns to be handled per process (in the partition per process ratio) is 1057 which is less than what we had to compute when we partitioned the matrix into 64 partitions which was 1455 columns per process on average. This reduction should, with the same number of processes, give

Hamrle3	**64 parts** (size $S = 46,564$)			**128 parts** (size $S = 67,679$)		
Cores	64	128	256	64	128	256
Build S	21.98 s.	20.83 s.	18.31 s.	13.83 s.	8.35 s.	7.71 s.
Fact. S	14.39 s.	26.15 s.	26.62 s.	9.65 s.	14.84 s.	14.27 s.

Table 5.9: Comparison of timings to build and factorize S resulting from Hamrle3 partitioned into 64 and 128 partitions respectively.

better results in the case of 128 partitions. The results when we partition the matrix Hamrle3 into 128 partitions are shown in Table 5.9.

Notice that 128 partitions on 128 processes brings a dramatic improvement over the 64 partitions case when building S. However, just as in the previous cases, we did not get good improvements when increasing the number of processes over the number of partitions (going from 128 to 256 processes). However, these results show that for ABCD, the first level of parallelism is really efficient as on the same number of processes increasing the number of partitions decreased the time to build S from 26.15 seconds to 14.84 seconds.

On the other hand, building and solving a linear system with the matrix S will consume more memory especially during the factorization. We show in Table 5.10 the amount of memory required for the different problems with the three test cases ($p/2$, p and $2 \times p$ processes). We notice that the amount of memory per process, required to factorize S is not excessive.

	lhr71c	EDF/R6	bmw3_2	Hamrle-64	Hamrle-128
Nb. entries	397 346	4 800 395	35 102 290	65 132 957	169 649 936
Avg. memory/process	0.4	4.6	33.5	15.5	20.2
$p/2$ processes	45	89	318	378	340
p processes	85	116	270	437	312
$2 \times p$ processes	83	109	231	424	312

Table 5.10: Comparison of the average amount of memory per process in MB to store S over the different processes that build S , and the average amount of memory used during the factorization of S in the different cases.

Another performance issue that we noticed in these results is that the factorization timing increases with the number of processes. This behaviour can be easily understood if we break down the factorization into computational and data distribution steps as shown in Table 5.11. We forcus on using 64 and 128 processes since the matrix S is distributed exactly the same way. Timings were obtained on a different computer (Oakleaf) which has slower network infrastructure and will thus illustrate better the extra-cost of communication.

Cores	64	128
Computational timing	6.82s.	4.47s.
Data distribution timing	57.53s.	125.56 s.
Total time to factorize	64.35s.	130.03 s.

Table 5.11: Detailed timing of the factorization of S from the matrix Hamrle3 partitioned into 64 partitions.

Table 5.11 clearly shows that the time spent in data distribution doubles when doubling the

number of processes. This excessive communication time is due to a data duplication related to dynamic scheduling that we explain in the following.

To explain it let us first recall how the distributed multifrontal solver MUMPS handles factorization. During the analysis phase, an assembly tree that defines a partial order of the factorization's tasks is built. This assembly tree is processed from the leaf nodes to the root nodes and corresponds to the dependency graph of the factorization phase. A so called master process is assigned to each node during the analysis phase. For each node, the master process can then dynamically assign processes (called slaves), among candidates computed during analysis phase, to help the master process handling the current task. The nodes in the task tree that exploit this parallelism are called *type 2* nodes (*type 1* nodes are those handled by a single process). The wider the tree is, the more tree parallelism can be exploited. Near the root node, tree parallelism is less and more processes are then dynamically assigned to collaborate in processing the node.

In the context of *type 2* nodes, since the slaves are not known in advance, the data distribution related to part of the original matrix is only known by the slave processes during the factorization. To overcome this issue, MUMPS duplicates corresponding data among all candidates participating in each node.

On sparse matrices (with few entries per column) and with a relatively large number of nodes, this extra volume of communication is very small. Our matrix S is less sparse, with very few almost dense nodes that are all processed as *type 2* nodes. Hence, the volume of communication to duplicate data over all candidates will thus significantly increase when increasing the number of participating processes. This increase is higher than the gain we will get from the parallelism expected from *type 2* nodes (that we indeed notice in our results).

Cores	64	128
Computational timing	6.79s.	9.49s.
Data distribution timing	1.70s.	1.94s.
Total time to factorize	8.49s.	11.43s.

Table 5.12: Detailed timing of the factorization of S from the matrix Hamrle3, partitioned into 64 partitions, when disabling data duplication prior to factorization.

MUMPS offers the possibility to force all nodes to be of *type 1* and thus suppresses data duplication. Results when this option is enabled are shown in Table 5.12. We notice a clear reduction in the data distribution time in both cases as there is no data duplication. The drawback of this option lies in the fact that we only exploit tree parallelism and thus the parallelism of the factorization phase is degraded. On such matrices and probably on all block structured matrices, the data distribution of the original matrix should be redesigned. One simple solution would be to have only the master process of a *type 2* node storing all the data associated with the original matrix. Then, once the master process has selected the candidates that will collaborate in processing the node, it sends without duplication the correct data to the correct slave processes. Such a scheme, not available in the current version of the MUMPS solver, would completely suppress the extra cost of duplication while ensuring a high level of parallelism.

All the results we have seen previously, show clearly that for these matrices the **ABCD** method offer a faster solution than the block-Cimmino method. However, the common thing about all these problems is the difficulty for block-Cimmino to converge fast enough. Indeed, we had to use large block-sizes to attain convergence which increased the timings. In the case where block-Cimmino converges fast enough (such as cage13, cage14 and torso3), it is more appropriate to use it than the **ABCD** method.

5.5 Conclusions

The block Cimmino method for solving sets of sparse linear equations has the merit of being embarrassingly parallel, but its convergence is equivalent to block Jacobi on the normal equations and can be slow. Preprocessing techniques, partitioning strategies and the use of block conjugate gradients can improve its convergence. However, all of these have their drawbacks or limitations.

We have proposed a novel technique to augment the systems to force the block Cimmino subspaces to be orthogonal so that only one iteration is required. This involves the solution of a relatively smaller positive definite system that we solve by forming the matrix and using a direct solver. When this system is small, the method works well. However, when it is large, the memory costs can become prohibitive. We study in the next chapter two possible solutions for this. We see in the next chapter possible improvements to this approach and analyse the gains we might get when solving successive right-hand sides.

Chapter 6

Possible improvements to ABCD

One of the main issues with ABCD is the size of S and its number of entries. On some larger problems it can become prohibitive to build, store and factorize such a large matrix. In this chapter we study two approaches to overcome this issue; in the first we try to reduce the size of S and in the second we try to avoid building S.

6.1 Filtered augmented block Cimmino

In this section we study an approach that permits us to build a smaller S and therefore helps in the cases where memory is a strong constraint. We do this by reducing the number of columns of C which exactly determines the order of the matrix S. Of course, one way of doing this is to use a partitioning where there is already good orthogonality between subspaces so that there are only a few columns in C [45]. We discuss, in this section, the theory involved in this approach and what benefits it might offer and how this idea can be exploited later.

We study three dropping strategies to reduce the order of S:

- We drop columns from C after it is generated, removing from C_{ij} the columns where the maximum element is smaller than some predefined threshold. Thus, the columns that have at least one element larger than the threshold are kept in C_{ij}, and we keep only the corresponding columns in the matrix $-I$. We illustrate the results of this dropping strategy in Section 6.1.1.

- In the case where we augment the matrix using the submatrices A_{ij}, a drop based on a straight threshold for entries in each column is not the correct way to go as it can destroy useful information present in these columns. We propose to drop a column depending on a relative scaled norm of the column. We discuss this in more detail in Section 6.1.2.

- We finish by studying another way to drop columns in C by building an SVD $U_{ij}\Sigma_{ij}V_{ij}^T$ of each C_{ij}. From usual data compression techniques, we then keep only the k largest singular values and the corresponding columns in U_{ij} and V_{ij}. We thus obtain a compressed matrix $U_{ij_{1,k}}\Sigma_{ij_{1,k}}V_{ij_{1,k}}^T$ of k columns. Rather than taking a fixed number of the largest columns, we can can just select those columns with singular values above a given threshold. We study both cases in Section 6.1.3.

The main problem, when filtering out columns in C, is that the partitions of the augmented matrix $\bar{A}$ will no longer be mutually orthogonal, so that the matrix $\sum_{i=1}^{p} P_{\mathcal{R}(\bar{A}_i^T)}$ will no longer be an orthogonal projector. Thus, when solving systems involving $\sum_{i=1}^{p} P_{\mathcal{R}(\bar{A}_i^T)}$, we will no longer get the projector $P_{\bigoplus_{i=1}^{p}\mathcal{R}(\bar{A}_i^T)}$directly. However, we can recover this, based on intrinsic properties

of both the block Cimmino iterative scheme and of the conjugate gradient method on semi-positive definite systems.

Block Cimmino gives the minimum norm solution $u = \bar{A}^+ y$ to the system $\bar{A}u = y$ as shown by Elfving [6]. The iteration matrix of block Cimmino, $\sum_{i=1}^{p} \bar{A}_i^+ \bar{A}_i$, is symmetric semi-positive definite, and solving the consistent linear system

$$\sum_{i=1}^{p} \bar{A}_i^+ \bar{A}_i u = \sum_{i=1}^{p} \bar{A}_i^+ y_i$$

with conjugate gradients will obtain the minimum norm solution as shown by Kaasschieter [57]. Using these properties, we are able to recover $w = \bar{A}^+ b$, needed in step 1 of our solution scheme in Section 5.3, through using conjugate gradient iterations to solve the system

$$\sum_{i=1}^{p} \bar{A}_i^+ \bar{A}_i w = \sum_{i=1}^{p} \bar{A}_i^+ b_i,$$

in which the matrix is the iteration matrix of block Cimmino applied to the system

$$\bar{A}w = b \tag{6.1}$$

with the same partitioning as originally used on A.

The same issue arises in the construction of the matrix $S = Y(I - P)Y^T$ which involves the projector $P = \bar{A}^+ \bar{A}$. We can use the same approach as above simply replacing the right-hand side b in (6.1) by $\bar{A}e_j$ where e_j are canonical vectors from the columns of the matrix Y^T.

Solving the system

$$\bar{A}z = \bar{A}e_j$$

gives the minimum norm solution

$$\begin{aligned} z &= \bar{A}^+ \bar{A} e_j \\ &= P e_j \end{aligned}$$

where e_j is a canonical vector from the matrix Y^T. Accelerating this solution using conjugate gradients amounts to solving the system

$$\sum_{i=1}^{p} \bar{A}_i^+ \bar{A}_i z = \sum_{i=1}^{p} \bar{A}_i^+ \bar{A}_i e_j.$$

We solve the previous system for all e_j in Y^T to obtain the matrix PY^T that can be used to build $S = Y(I-P)Y^T$ explicitly. As the solutions associated with these columns are independent from each other we may also exploit an extra level of parallelism.

If we can keep the dimension of S small enough and easy to handle by a direct solver, then it is attractive to build it explicitly and factorize it using a direct method. This is the approach we will consider in the following.

Just like we have done in the previous chapters, we will illustrate the different algorithms with the matrix `bayer01`.

6.1.1 Filtering C_{ij}

We now examine the effect of dropping entries in the contributing subblocks C_{ij} on the dimension of S and on the subsequent solution of the problem.

We show in Figure 6.1 the reduction in the order of S for `bayer01` as we increase the relative dropping tolerance τ for filtering out entries in the C_{ij}. Since the structure of S has large dense

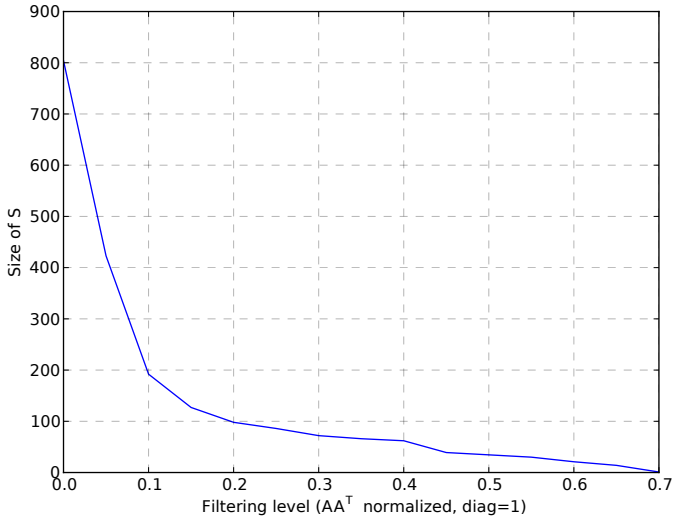


Figure 6.1: Number of columns of S against filtering level on C, for `bayer01`.

Filtering threshold τ	0	0.1	0.3	0.4	0.45
Size of S	804	192	72	62	39
Min. Iterations to build S	1	4	11	13	13
Avg. Iterations to build S	1	15	26	34	62
Max. Iterations to build S	1	48	55	60	101
Nb. Iter. $w = \bar{A}^+ b$	1	58	91	281	402
Nb. Iter. $u = (I - P)\,\bar{z}$	1	51	71	212	202
$\|r\|/\left(\|A\|\|x\| + \|b\|\right)$	$3e-16$	$1e-11$	$1e-11$	$1e-11$	$1e-11$
$\|r\|/\|b\|$	$2e-15$	$6e-11$	$6e-11$	$4e-11$	$5e-11$
$\|x^* - x\|_\infty / \|x^*\|_\infty$	$3e-11$	$4e-10$	$1e-09$	$4e-09$	$9e-09$

Table 6.1: Filtering columns of $C_{ij}/-I$ in ABCD on `bayer01`.

blocks, the value of reducing its dimension from nearly 804 to less than 192 when dropping at 0.1 is obvious. The effect of this dropping on the convergence of the block Cimmino iteration used to build $w = \bar{A}^+ b$ is shown in Figure 6.3.

We notice that as one might expect, the more columns that we drop the slower the convergence. This is due to the fact that the more columns of C that we drop, the more we destroy the orthogonality between partitions and, in the limit, we may just be left with the original Cimmino iteration matrix and its bad conditioning. Looking at the eigenvalue distribution of the iteration matrix in each case, as shown in Figure 6.2, we see that the more we drop, the more intermediate eigenvalues appear. So when the smallest eigenvalue changes from 4.59×10^{-3} to 2.44×10^{-3} (for a filtering threshold of 0.1 and 0.3 respectively), we require an additional 43 iterations. Moreover, when dropping more columns, two very small eigenvalues appear, one at 4×10^{-9} and another at 1×10^{-5}. This explains the appearance of plateaux in the convergence profile in Figure 6.3 when filtering at 0.4 and 0.45.

We summarize the results of filtering C for the `bayer01` matrix in Table 6.1. The order of S decreases quite significantly as we increase the filtering level although, after a certain point, the gains are less noticeable. As we expect, at larger filtering thresholds, when the partitions of $\bar{A}$ depart more and more from being orthogonal, we observe that CG requires more iterations. The same goes for the construction of S which also requires more iterations. The resulting accuracy is

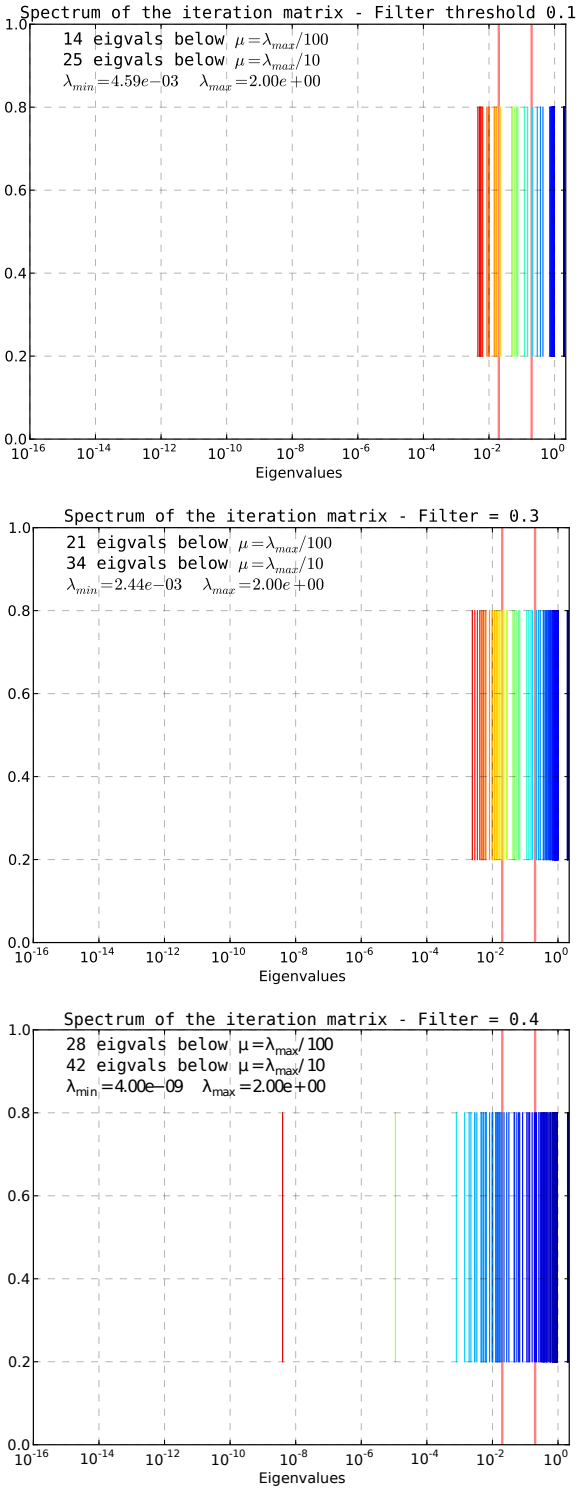


Figure 6.2: Eigenvalues of the iteration matrix after filtering C columns with different filtering thresholds.

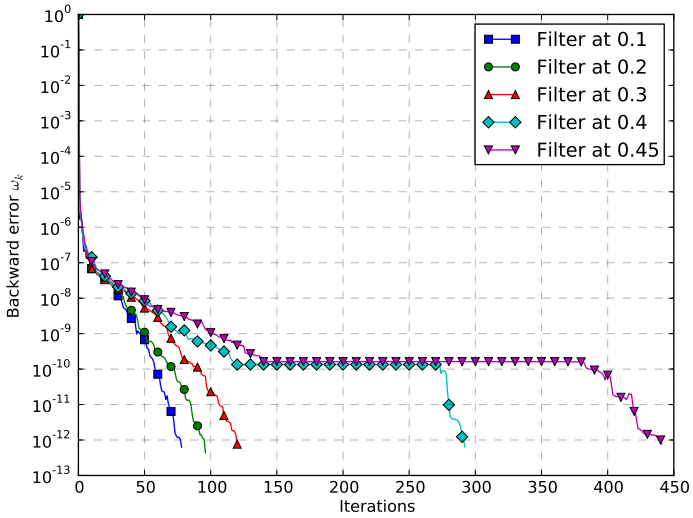


Figure 6.3: The effect of dropping columns of C on the convergence of the block Cimmino process.

still good, even if somewhat degraded from the unfiltered case.

6.1.2 Filtering A_{ij}

We examine now the effect of dropping entries in the submatrices A_{ij} and its corresponding $-A_{ji}$ when using the first variant (5.1) for the augmentation process. To avoid losing important information when dropping, we look at the norm of the outer product of the k-th column from each submatrix with respect to the number of entries generated by this outer product. This is implemented by applying the following steps on each couple $A_{ij}/-A_{ji}$ in C:

- We compute the Frobenius norm of each rank-one update corresponding to the k-th column $\begin{bmatrix} A_{ij(*,k)} & -A_{ji(*,k)} \end{bmatrix}^T$. Denoting this by $frob_{ij}(k)$, we can compute it by :

$$frob_{ij}(k) = \| A_{ij(*,k)} \|_2 \| A_{ji(*,k)} \|_2.$$

- We denote by d_{ij}, an underestimate of the number of entries in $A_{ij}A_{ji}^T$, by :

$$d_{ij} = \max_k \left(card\{|A_{ij}(*,k)| > 0\} \times card\{|A_{ji}(*,k)| > 0\} \right),$$

  where $card\{y\}$ is the number of entries in y.

- To identify the dominant contributions from the different rank-one updates, we first compute as a reference value the mean distribution of the rank-one contributions :

$$\nu_{ij} = \frac{\text{average}\left(frob_{ij}(k)\right)}{\sqrt{d_{ij}}}.$$

- For each column, we then count the number of influential entries :

$$card_{ij}(k) = card\{\left|A_{ij(*,k)}\right| \geq \frac{\nu_{ij}}{\| A_{ji(*,k)} \|_\infty}\},$$

$$card_{ji}(k) = card\{\left|A_{ji(*,k)}\right| \geq \frac{\nu_{ij}}{\| A_{ij(*,k)} \|_\infty}\}.$$

If $card_{ij}$, respectively $card_{ji}$, is zero, we set it to m_{ij}, respectively m_{ji}, the number of rows in A_{ij}, respectively A_{ji}.

- Next, we define for each of the columns a scaled norm

$$\mu_{ij}(k) = \frac{frob_{ij}(k)}{\sqrt{card_{ij}(k) \times card_{ji}(k)}}. \tag{6.2}$$

- Finally, using a given threshold τ we select the columns to retain in the couple $A_{ij}/-A_{ji}$ satisfying $\mu_{ij}(k) \geq \tau$.

By following these steps, we try to extract some information regarding the numerical density of each rank-one update corresponding to the columns of a given couple $A_{ij}/-A_{ij}$. This numerical density is then used to select the most influential columns in each couple (those above the threshold τ).

We show in Figure 6.4 the reduction in the order of S for `bayer01` as we increase the relative dropping tolerance τ for filtering the entries in the $A_{ij}/-A_{ji}$ couples. We show in Figure 6.5 the effect of this dropping strategy on the block Cimmino iteration cost when computing $w = \bar{A}^+ b$. More detailed results are summarized in Table 6.2.

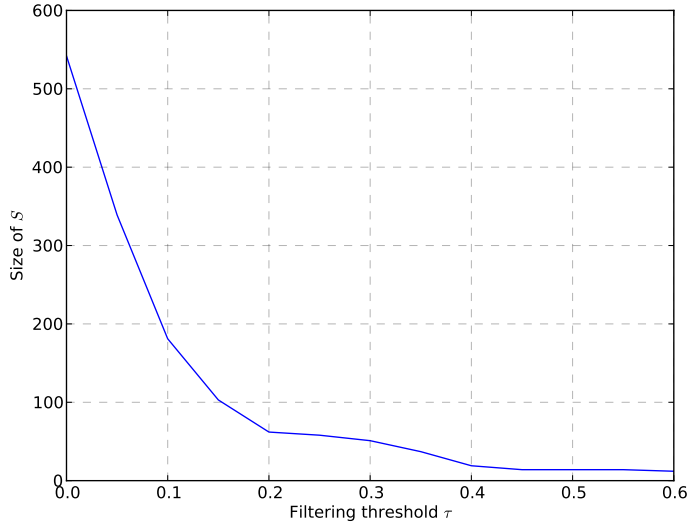


Figure 6.4: Number of columns of S against the filtering level on A_{ij}, for `bayer01`.

We notice the same behaviour as when filtering the C_{ij} normal equations subblocks in the sense that, the more we drop the slower the convergence. We notice that we require 152 iterations to build w with the $A_{ij}/-A_{ji}$ strategy when we have 62 columns in S, while we require 281 iterations using the $C_{ij}/-I$ strategy with the same number of columns in S. We are also able to reduce the number of columns to 19 without seeing too strong a degradation in the convergence while still keeping good numerical properties.

We compare in Figure 6.6 the values of two different scalings of the $frob_{ij}(k)$ norms, $\mu_{ij}(k)$ and $\bar{\mu}_{ij}(k) = \frac{frob_{ij}(k)}{\sqrt{nnz\{A_{ij(:,k)}\} \times nnz\{A_{ji(:,k)}\}}}$. We notice that using the scaled norms $\mu_{ij}(k)$ we retain more columns since nnz is the total number of entries in the column whereas $card_{ji}(k)$ in equation (6.2) is only the number of entries in the column greater than a threshold. We show the effect

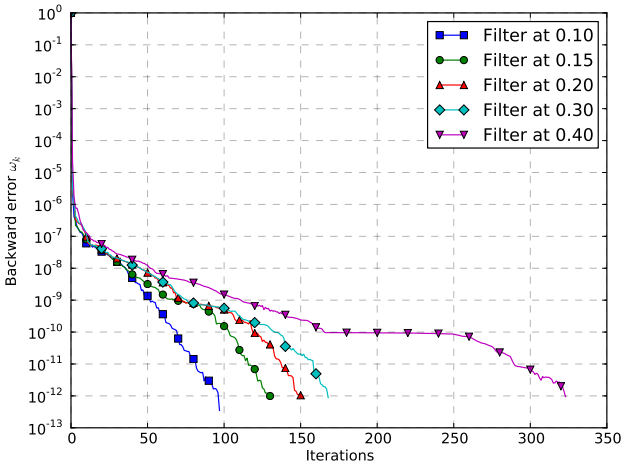


Figure 6.5: The effect of dropping columns on the convergence of the block Cimmino process.

Filtering threshold τ	0	0.1	0.15	0.2	0.3	0.4
Size of S	542	181	103	62	50	19
Min. Iterations to build S	1	4	4	4	6	14
Avg. Iterations to build S	1	18	23	25	29	72
Max. Iterations to build S	1	51	55	63	63	120
Nb. Iter. $w = \bar{A}^+ b$	1	97	130	152	168	323
Nb. Iter. $u = (I - P)\,\bar{z}$	1	91	121	141	160	76
$\|r\|/\left(\|A\|\|x\| + \|b\|\right)$	$6e-16$	$1e-11$	$1e-11$	$1e-11$	$1e-11$	$3e-11$
$\|r\|/\|b\|$	$3e-15$	$5e-11$	$6e-11$	$6e-11$	$6e-11$	$8e-11$
$\|x^* - x\|_\infty/\|x^*\|_\infty$	$2e-11$	$1e-10$	$2e-10$	$5e-10$	$6e-10$	$1e-09$

Table 6.2: Filtering columns of $A_{ij}/-A_{ji}$ in ABCD on `bayer01`.

of using these two measures for the scaled norm of the rank-one updates in Figure 6.6 where we have ordered the columns using the scaled norm $\bar{\mu}_{ij}(k)$. We then see that, when filtering at 0.4 for example, there are four spikes that increase the size of S from 15, when using $\bar{\mu}_{ij}(k)$, to 19. When we look at the number of iterations, we notice that adding those four columns reduces the number of iterations from 498 (not shown in the graphs before), when using $\bar{\mu}_{ij}$, to 323 so clearly the influence of these four columns is a strong one. We thus use the scaled norm $\mu_{ij}(k)$ to determine which columns to retain when using the couples $A_{ij}/-A_{ji}$.

6.1.3 Compressing C_{ij} with SVD

We now study the SVD approach in which we decompose each C_{ij} into $U_{ij}\Sigma_{ij}V_{ij}^T$. We then try to select the minimum number of columns with enough information to maintain good convergence and reduce at the same time the order of S. The resulting augmented matrix is of the following form:

$$\left[\begin{array}{ccccc|cc} A_{1,1} & A_{1,2} & & & & \hat{U}_{1,2}\hat{\Sigma}_{1,2} & \\ & A_{2,1} & A_{2,2} & A_{2,3} & & -\hat{V}_{1,2} & \hat{U}_{2,3}\hat{\Sigma}_{2,3} \\ & & & A_{3,2} & A_{3,3} & & -\hat{V}_{2,3} \end{array}\right] \tag{6.3}$$

where $\hat{U}_{ij}$, $\hat{\Sigma}_{ij}$ and $\hat{V}_{ij}$ are respectively the reduced U_{ij}, Σ_{ij} and V_{ij}, with only those selected columns.

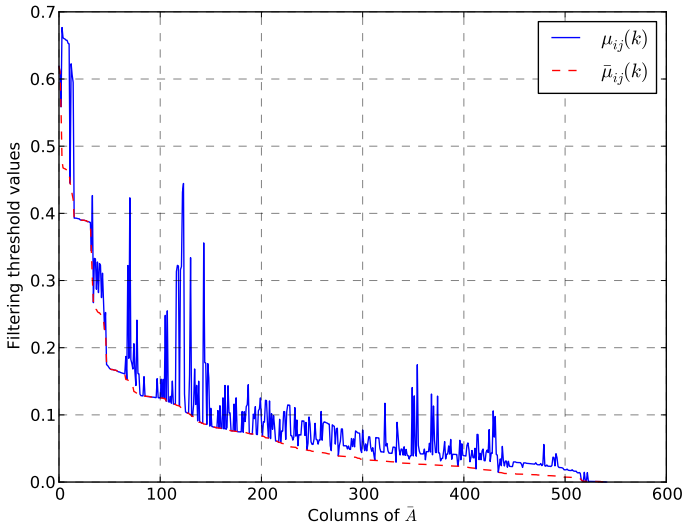


Figure 6.6: Comparison of two different scaled norms for each rank-one update in $A_{ij}/-A_{ji}$.

We look first at a simple case where we compute the k largest singular values and build the augmented matrix as in (6.3). The results from this approach are shown in Table 6.3. We notice that the results are slightly worse than when filtering using $C_{ij}/-I$ or $A_{ij}/-A_{ji}$ for comparable size of S.

Singular values kept per block	5	10	12	15
Size of S	108	209	249	308
Nb. Iter. $w = \bar{A}^{+}b$	188	102	94	46

Table 6.3: Selecting the k largest singular values from C_{ij} in ABCD on `bayer01`.

We try now to improve the previous selection by looking at the smallest singular value within the complete set of selected columns. This is implemented as follows:

- Select k columns from all the SVD decompositions applied to the C_{ij} blocks.
- Find the smallest singular value among them.
- Use it as a threshold to select further columns whose singular value is larger than this value.

We show in Table 6.4, the results obtained on the `bayer01` matrix when using this approach. For instance, by selecting 2 columns initially, we see that the smallest singular value is 0.2. Then, we select all columns whose corresponding singular value is larger than 0.2. The results show that we are able to improve the previous results for comparable sizes of S. However, we lose the ability to control explicitly the number of columns.

6.1.4 Cost analysis

An important aspect of the ABCD method is the reusability of the S matrix for multiple, successive, solves for different right-hand sides. To evaluate the gains obtained using our method, we look at the operation counts needed to solve several, successive, right-hand sides using classical block Cimmino, and using the ABCD method with and without filtering.

Singular values initially selected per block	2	3	4	5
Smallest singular value	0.2	0.11	0.049	0.02
Size of S	96	171	376	481
Nb. Iter. $w = \bar{A}^+ b$	141	107	45	12

Table 6.4: Selecting singular values larger than a threshold from C_{ij} in ABCD on `bayer01`.

We first define the cost of each solution. The costly part in each block Cimmino iteration is the computation of $P_{\bigoplus_{i=1}^{p} \mathcal{R}(A_i^T)} y$. Its cost is basically the number of flops needed by a sparse direct solver to compute a forward-backward substitution. We will denote this cost by $\mathcal{C}_{BC}(A)$. In the case of ABCD, the cost of computing $w = \sum_{i=1}^{p} \bar{A}_i^+ b_i$ is $\mathcal{C}_{BC}(\bar{A})$ per iteration. We can obtain a value for these quantities from MUMPS.

The initial cost of solving $Sz = f$ is a combination of a Cholesky factorization on S with a forward-backward substitution. As we treat the matrix S as a dense matrix, the estimation of the first step is

$$\mathcal{C}_{S,init} = \frac{1}{3} n_s^3 + 2n_s^2,$$

while for the later steps we need only a forward-backward substitution, a bound on this cost is

$$\mathcal{C}_S = 2n_S^2.$$

We show in Table 6.5 the estimations for solving a linear system, with multiple right-hand sides successively, using the classical block Cimmino method, using ABCD without filtering and using ABCD with different filtering thresholds. We denote by it_{bc} the number of iterations needed to solve a single right-hand side using block Cimmino, by it_p the number of iterations to compute both $w = \sum_{i=1}^{p} \bar{A}_i^+ b_i$ and $u = (I - P)\bar{z}$ and by it_s the sum over the number of iterations needed to build all the columns of S. In the case where we use ABCD without filtering, the number of iterations it_p is equal to 2 as we need one to build w and another to build u.

	The cost estimate in floating-point operations	
	First right-hand side	Next right-hand sides
Block Cimmino	$it_{bc} \times \mathcal{C}_{BC}(A)$	$it_{bc} \times \mathcal{C}_{BC}(A)$
ABCD (no filtering)	$(n_s + 2) \times \mathcal{C}_{BC}(\bar{A}) + \mathcal{C}_{S,init}$	$2 \times \mathcal{C}_{BC}(\bar{A}) + \mathcal{C}_S$
ABCD (with filtering)	$(it_s + it_p) \times \mathcal{C}_{BC}(\bar{A}) + \mathcal{C}_{S,init}$	$it_p \times \mathcal{C}_{BC}(\bar{A}) + \mathcal{C}_S$

Table 6.5: Estimation of flops needed per step.

We use the estimations from Table 6.5 to generate the graphs in Figure 6.7 where we show the amount of work needed to solve multiple right-hand sides successively when using the $C_{ij}/-I$ augmentation strategy, where we compare the plain block Cimmino on the original matrix, the augmented version without filtering, and with multiple filtering thresholds. We illustrate in this figure how quickly we may expect to recover the extra work induced by the augmentation process.

We notice that using the full augmentation process without filtering recovers this extra work after just two solutions. Further solutions require very much less work, they require only one block Cimmino iteration plus the cost of a Forward/Backward substitution, making the graph look flat.

Filtering at 0.1 requires more work on the first run. Indeed, the number of columns in S combined with the number of iterations to build a single column of S makes the initial cost very large. This initial cost is less when dropping at 0.3, as the size of S is dramatically reduced while

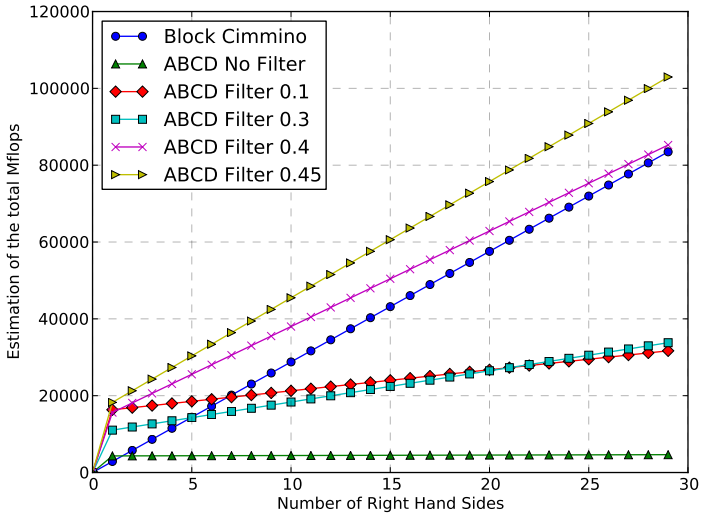


Figure 6.7: Comparison of solving multiple right-hand sides successively when using $C_{ij}/-I$ couples.

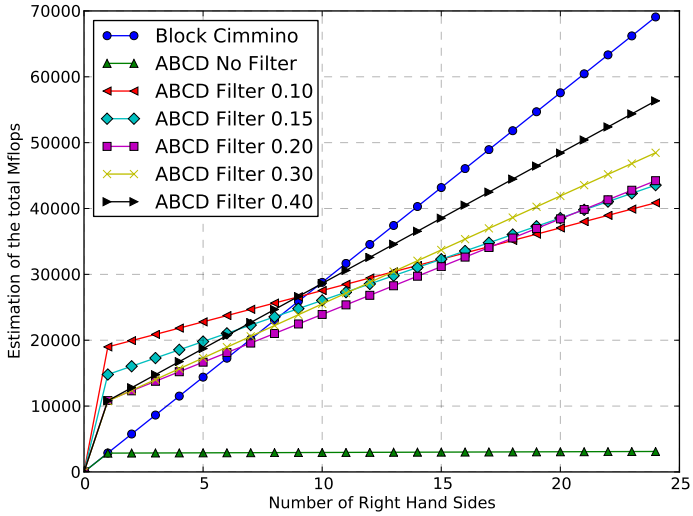


Figure 6.8: Comparison of solving multiple right-hand sides successively when using $A_{ij}/-A_{ji}$ couples.

the number of iterations increases only slightly. But this cost increases again when dropping further; the number of iterations needed for each column increases faster than the reduction in the number of columns in S.

With higher filtering thresholds, it takes more solutions to recover the original cost, and each of these solutions requires more CG iterations, which can be seen in the increased slope of the corresponding straight lines in Figure 6.7. We observe that when the filtering threshold gets too large, the gain per solution is not so effective. When using a filter level of 0.4, we require around 30 solutions to compensate for the extra work which is not unreasonable given the low memory requirements at that level of filtering. Finally, increasing the filtering threshold even further makes the approach useless or counterproductive with respect to the plain block Cimmino solve.

We show in Figure 6.8 the costs when using the $A_{ij}/-A_{ji}$ augmentation strategy. Without

filtering, we perform much better than the original block Cimmino iteration. This is due to the fact that S is very small. However, and contrary to what we have seen with the $C_{ij}/-I$ augmentation strategy, the combination of the small size of S and the small number of iterations to build columns of S makes it possible to use a filter threshold of 0.4, for which the size of S is only 19, and we recover the cost of generating S after only 10 subsequent solves. In this augmentation strategy it is possible to reduce the storage for S while requiring relatively few iterations to build a single column.

6.2 Iterative solution of S

We look in this section at how we can solve the system $Sz = f$ iteratively without building S. Since the matrix S is known to be symmetric positive definite when using a full augmentation (see Section 5.2), we can use the conjugate gradient method to solve the linear system $Sz = f$. We show in Section 6.2.1 how the conjugate gradient method could be used to solve the system. We then discuss a possible preconditioner in Section 6.2.2 that should improve the convergence.

6.2.1 Conjugate gradient on S

We start by recalling the conjugate gradient algorithm in Algorithm 6.1 applied to the system $Sz = f$ when the matrix S is available.

Algorithm 6.1 Conjugate Gradient Algorithm

Input: S and f
Output: z

1: $k \leftarrow 0$
2: $z_k \leftarrow 0$
3: $r_k \leftarrow f$
4: $p_k \leftarrow r_0$
5: **while** not converged **do**
6: $\quad \alpha_k \leftarrow \left(r_k^T r_k\right) / \left(p_k^T S p_k\right)$
7: $\quad z_{k+1} \leftarrow z_k + \alpha_k p_k$
8: $\quad r_{k+1} \leftarrow r_k - \alpha_k p_k$
9: $\quad \beta_k \leftarrow \left(r_{k+1}^T r_{k+1}\right) / \left(r_k^T r_k\right)$
10: $\quad p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
11: $\quad k \leftarrow k + 1$
12: **end while**

In a classical conjugate gradient algorithm, we use S for matrix-vector products as in the 6th line. This product can be formed without first constructing S. We know that

$$S = Y(I - P)Y^T, \tag{6.4}$$

hence, the product Sp_k is defined by

$$\begin{aligned} Sp_k &= Y(I - P)Y^T p_k \\ &= p_k - YPY^T p_k, \end{aligned} \tag{6.5}$$

where P is the orthogonal projector onto the range of $\bar{A}^T$ and $Y = \begin{bmatrix} 0 & I \end{bmatrix}$. In the case of a fully augmented system, the orthogonal projector onto the range of $\bar{A}^T$ is simply the sum of projectors

onto the ranges of $\bar{A}_i^T$. Therefore, we can write P as

$$P = \bar{A}^+ \bar{A} = \sum_{i=1}^{p} \bar{A}_i^+ \bar{A}_i.$$

If we use (6.5) in Algorithm 6.1 we obtain Algorithm 6.2 with implicit computations of the matrix-vector product in the 6th line.

Algorithm 6.2 Conjugate Gradient Algorithm with Implicit S

Input: S and f
Output: z
1: $k \leftarrow 0$
2: $z_k \leftarrow 0$
3: $r_k \leftarrow f$
4: $p_k \leftarrow r_0$
5: **while** not converged **do**
6: $\quad \bar{p}_k \leftarrow p_k - YPY^T p_k$
7: $\quad \alpha_k \leftarrow \left(r_k^T r_k\right) / \left(p_k^T \bar{p}_k\right)$
8: $\quad z_{k+1} \leftarrow z_k + \alpha_k p_k$
9: $\quad r_{k+1} \leftarrow r_k - \alpha_k p_k$
10: $\quad \beta_k \leftarrow \left(r_{k+1}^T r_{k+1}\right) / \left(r_k^T r_k\right)$
11: $\quad p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
12: $\quad k \leftarrow k + 1$
13: **end while**

6.2.2 Preconditioned conjugate gradient on S

We have seen in Section 5.2 that the matrix S is very ill-conditioned and, as we will show later, the classical conjugate gradient would not give good convergence results if used alone. We study in this section one way to improve the convergence by preconditioning the linear system. We have seen in Section 6.1 that we are able to select specific columns of C using different criteria. We can use this process to build an approximation of S containing a subset of columns from S that are selected so that this approximation contains, hopefully, the most important information in S.

In the following, we use `selected_columns` and `skipped_columns` to describe respectively the set of columns used in building our preconditioner and the set of columns not included in this computation. Let W_{sc} and W_{ns} be the restriction matrices for these two respective sets. As an illustrative example we define them as the following:

$$W_{sc} = \begin{bmatrix} I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix} \quad \text{and} \quad W_{ns} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & I & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

In Figure 6.9 we notice the relationship between C, Y and S. Replacing the identity present in $Y = [0\ I]$ by the matrix W_{sc} makes it possible to build only these columns in S, and let $Y_{sc} = [0\ W_{sc}]^T$ be that matrix. The resulting submatrix of S is described by

$$S_{sub} = Y_{sc}(I - P)Y_{sc}^T.$$

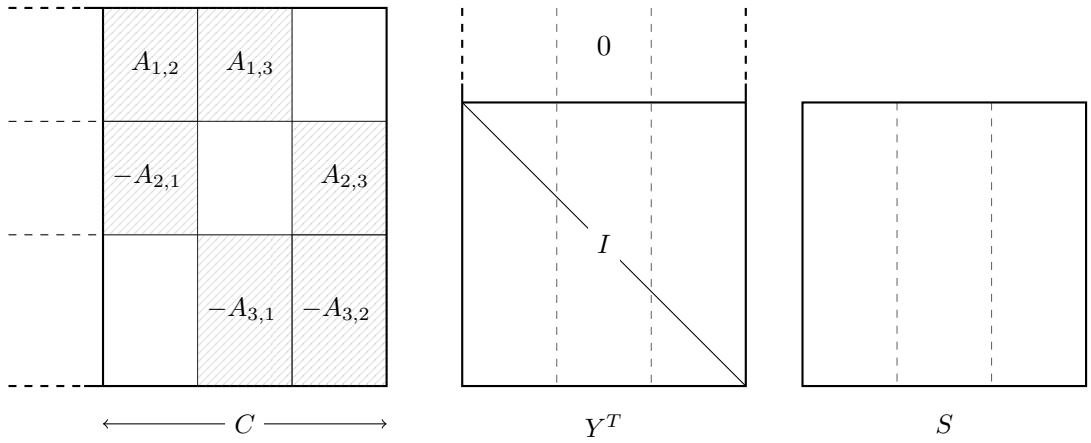


Figure 6.9: The relationship between C, Y and S. The width of the different A_{ij} blocks correspond to the width of the different set of columns (presented by vertical dashed lines) in Y^T and in S.

S_{sub} is a restriction of a symmetric positive definite matrix S which itself is a restriction of an orthogonal projector and hence its diagonal entries are bounded by one. Putting the identity on the empty diagonal entries as follows

$$\widetilde{S} = S_{sub} + W_{ns}, \tag{6.6}$$

allows $\widetilde{S}$ to be symmetric positive definite.

The team behind the direct solver `MUMPS` are developing a feature that allows a user to compute only a set of entries in the solution. We can use this feature to build only the diagonal of the matrix S thus obtaining the preconditioner

$$\widetilde{S} = S_{sub} + diagonal(W_{ns} S W_{ns}^T). \tag{6.7}$$

A representation of the process of building $\widetilde{S}$ is shown in Figure 6.10. Additionally, a sparse approximation of S can also be built (once the feature is available in the direct solver) and, to maintain its definiteness, the diagonal can be shifted.

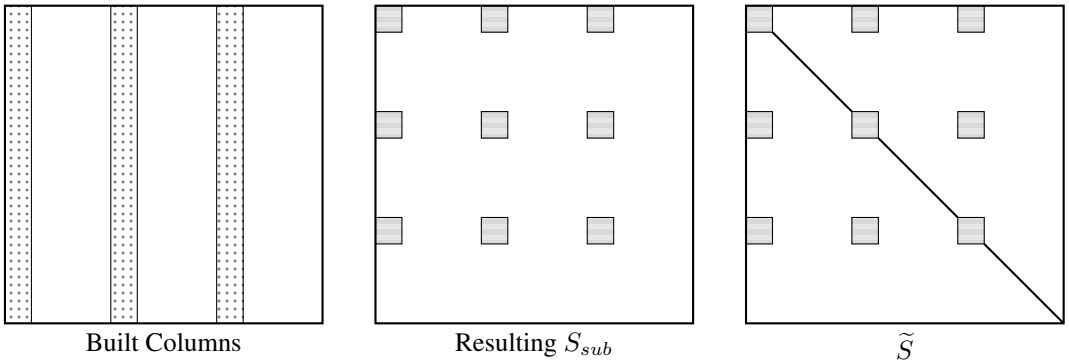


Figure 6.10: Computing parts of S and the resulting $\widetilde{S}$. Starting from left to right, the built columns of S using the matrix Y_{sc}, the resulting restricted matrix corresponding to these built columns and finally the preconditioner $\widetilde{S}$ arising from this restriction.

We show in Algorithm 6.3 the preconditioned version of the conjugate gradient algorithm shown in the previous section. The preconditioning matrix $\widetilde{S}$ is given in entry and computing $\widetilde{S}^{-1} r_k$ is performed with `MUMPS` in parallel.

Algorithm 6.3 Preconditioned Conjugate Gradient Algorithm with Implicit S.

Input: $\widetilde{S}$ and f
Output: z

$k \leftarrow 0$
$z_k \leftarrow 0$
$r_k \leftarrow f$
$u_k \leftarrow \widetilde{S}^{-1} r_k$
$p_k \leftarrow u_k$
while $k <$ `max_iter` **do**
  $\bar{p}_k \leftarrow p_k - YPY^T p_k$
  $\alpha_k \leftarrow \left(r_k^T u_k\right) / \left(p_k^T \bar{p}_k\right)$
  $z_{k+1} \leftarrow z_k + \alpha_k p_k$
  $r_{k+1} \leftarrow r_k - \alpha_k \bar{p}_k$
  if converged **then**
    return z_k
  end if
  $u_{k+1} \leftarrow \widetilde{S}^{-1} r_{k+1}$
  $\beta_k \leftarrow \left(u_{k+1}^T r_{k+1}\right) / \left(u_k^T r_k\right)$
  $p_{k+1} \leftarrow u_{k+1} + \beta_k p_k$
  $k \leftarrow k + 1$
end while

6.2.3 Numerical experiments

Due to the experimental nature of this section, the diagonal of S used for the preconditioner was obtained by also building the remaining columns (`skipped_columns`), which means that we do all the operations needed to build the full matrix S but store only the entries we are interested in. As the augmentation was done using the $A_{ij}/-A_{ji}$ strategy, we use the filtering strategy described in Section 6.1.2 to obtain the selected columns. This strategy requires a filtering threshold (selection threshold) to chose a set of columns in S to be used to build $\widetilde{S}$.

We show in Table 6.6 the results of solving $Sz = f$ iteratively using classical CG, preconditioned CG with diagonal preconditioner, and with an $\widetilde{S}$ obtained with two different selection thresholds. We set 10^{-5} as the stopping criteria and the order of S as a maximum number of iterations. We test the method on the matrix `bayer01` for which the size of the auxiliary system is 865. We notice from these results that the convergence is very slow even when selecting a high number of columns to create the preconditioner.

CG iterations	Selection thresh.	Number of selected cols for $\widetilde{S}$	PCG iterations
	diagonal	0	`Failed` (1.7×10^{-3})
`Failed` (2.8×10^{-3})	0.1	457	401
	0.2	83	461

Table 6.6: Number of iterations in CG and PCG to solve $Sz = f$ arising from ABCD applied to the problem `bayer01`, and the number of columns used to build the preconditioner depending on a given threshold. If the number of iterations reaches the order of S, we mark `Failed` and put the final scaled residual between parentheses.

Another possibility was tested in matlab in which we built a different kind of preconditioner where the columns were selected based on the diagonal entries of S. We used different thresholds

to select columns corresponding to the largest entries on the diagonal and chose the threshold so that we have a similar number of columns as in the previously tested strategy. Using this preconditioner gave worse results than in the previous cases. However, we were able to get better results using **MINRES** with that preconditioner and we show the results in Table 6.7.

MINRES iter.	**Selection thresh.**	**Number of selected cols for $\widetilde{S}$**	**Precond. MINRES iter.**
471	diagonal	0	269
	0.45	478	265
	0.595	84	268

Table 6.7: Number of iterations for MINRES to solve $Sz = f$ arising from ABCD applied to the problem `bayer01`, and the number of columns used to build the preconditioner depending on a given threshold.

We see from these results that MINRES gives better results than preconditioned CG, and a simple diagonal preconditioning already improves the results (even if it is not yet satisfactory). Further analysis has to be done to understand these results and their implications, more tests are also needed to identify better preconditioners.

6.3 Concluding remarks

The augmented block Cimmino method introduced a way to alleviate some of the drawbacks of using block Cimmino, namely slow convergence because of small eigenvalues in the lower part of the spectrum of the iteration matrix. We were indeed able to create a pseudo-direct solver, but that was at a cost of building the auxiliary system.

In this chapter we proposed two possible solutions to handle the auxiliary system. Firstly, we tried to reduce its size by reducing our requirement on the orthogonality generated by the augmentation. To achieve this objective we used three different augmentation models. Each of the different strategies reduced the size of the auxiliary system, but the one applied on the $A_{ij}/-A_{ji}$ augmentation strategy worked better than the others. Yet, compared to a full augmentation, we did not obtain as good a reduction in the size of S as we had hoped. Additionally, the cost of building a smaller auxiliary system is higher than building a full size one because of the embarrassingly parallel method involved in forming the full auxiliary system.

Secondly, we proposed a way to avoid building the auxiliary system and to use it implicitly to obtain the solution. This model is based on two important properties of the augmented block Cimmino method that are the symmetric positive definiteness of the auxiliary system, and the fact that a matrix-vector product using this system amounts to a solution computed using a direct solver. These two properties make it possible to use a conjugate gradient algorithm to obtain the solution without any extra memory cost (that is, without storing the auxiliary system).

We have seen that CG did not give good results on the tested problem as it failed to converge, and that the preconditioners did not give the expected convergence improvements either. Moreover, we have seen that MINRES with a diagonal preconditioner performed better than the preconditioned CG as it converged using all the solution criteria.

Chapter 7

General conclusion

In this manuscript we considered the solution of large sparse linear systems, using a hybrid implementation of a method from the class of row projection techniques. We started, in Chapter 2, by introducing some of the properties of this class of techniques by recalling the theory introduced by Elfving in [6]. We have seen that they offer an opportunity to solve problems with unsymmetric matrices, by implicitly using a symmetric positive definite iteration matrix in a conjugate gradient acceleration process. This acceleration was discussed by Bramley and Sameh in [7] who also investigated numerically the robustness of this acceleration.

In Chapter 2, we introduced and discussed the algorithms lying behind this acceleration process, and we have shown that a classical conjugate gradient acceleration worked poorly on some problems. Indeed, the iteration matrix has small eigenvalues at the end of the spectrum inducing plateaux in the convergence curve. We then introduced the stabilized block conjugate gradient method, studied by Arioli, Duff, Ruiz and Sadkane in [8], to partly resolve these convergence issues. We showed that the block-size used had a great influence on the convergence of some problems, and that the cost of using a moderate block-size is not prohibitive and is able to exploit BLAS kernels to achieve better gains.

In our implementation of this block iterative method, the matrix-vector operations in the conjugate gradient method are computed using a direct solver. Indeed, this operation amounts to solving augmented systems related to the partitions of our linear system. Hence, we introduced different ways in which a direct solver can successfully factorize and solve such augmented systems. This mix of iterative method (conjugate gradients) and direct method creates the **hybrid** aspect of our approach.

We then discussed different ways to obtain the partitions of the linear system in Chapter 3. Rather than uniformly split the linear system into strips of rows, we tried some sophisticated techniques that rely on the structural and numerical properties of the matrix. The first strategy was to apply the Cuthill-McKee algorithm on the normal equations pattern of the original matrix, using the level sets and the row permutations to create partitions that can be grouped into two sets. The partitions in each set are mutually orthogonal to each other; this partitioning is called a Two-Block partitioning and is effective at clustering the eigenvalues of the iteration matrix around 1. As the level sets can be large, we tried to reduce them by introducing a relaxed version where we apply the Cuthill-McKee algorithm on a filtered normal equations matrix. This strategy creates more freedom in partitioning than the previous one as it create much smaller level sets.

Noticing that these two strategies are computationally expensive, we introduced a hypergraph based partitioning of the matrix. This approach regroups the rows of the matrix into subsets so that there are only a few overlapping columns between these sets of rows. Experiments on large matrices showed that in most cases hypergraph partitioning gives better results than the previously studied strategies including uniform partitioning. However, we also showed that occasionally it

is better to uniformly partition the matrix if it then converges quickly, or if the convergence gains by using a hypergraph partitioner do not cover the cost of using the partitioner itself. Sadly, the partitioning of the linear system is problem dependent and thus one cannot predict how the method will behave a priori.

After this preliminary analysis of the numerical properties and algorithmic components of this method we investigated its parallel implementation. In Chapter 4, we discussed in detail the algorithms involved in this implementation. We started by describing an early master-slave implementation where the conjugate gradient acceleration was sequential and the computation of the matrix-vector by the direct solver was parallel. We discussed why this implementation has very low parallel potential and how it introduces many bottlenecks. We then proposed exploiting another level of parallelism by implementing a distributed conjugate gradient acceleration that is combined with a parallel direct solver. We saw how partitions can be mapped appropriately on the processes participating in the distributed conjugate gradient algorithm, and discussed two other approaches to improve this mapping strategy.

Next, we did runs with matrices from different applications in parallel and studied the speedup of the factorization of the augmented systems, the average memory usage and the speedup of the conjugate gradient acceleration. We saw the effect on the speedup of the number and the size of partitions on both the factorization and the conjugate gradient acceleration where we noticed that, on a small number of processes, a larger number of partitions induces a faster factorization and a faster conjugate gradient iteration. However, on a large number of processes, larger partitions will get more speedup as they continue to utilize more slaves in the direct solver. On the other hand, we noticed that the average memory usage when increasing the number of processes decreases linearly and can get superlinear when we have one partition per process.

Afterwards, we made some comparisons with the direct solver `MUMPS`, where we showed that on a few cases we are able to drastically reduce the memory usage. This also allowed us to solve problems that the direct solver was not able to solve due to lack of memory on the testing machine. We also showed that on some problems we were able to obtain the solution faster than the direct solver. This situation happens especially when the factorization suffers from a large amount of numerical pivoting, which slows it down. This opens the way to being able to solve even larger problems and shows that we were able to achieve our initial goal of both breaking the complexity of the problem and at the same time achieving good convergence timings.

In some problems, the number of iterations was however too large because of the ill conditioning of the iteration matrix. In Chapter 5 we introduced a novel approach that tries to reduce the number of iterations by enforcing orthogonality between the partitions of the original matrix. This augmentation process results effectively in an identity iteration matrix which yields convergence in a single iteration. We then introduced some solution steps that do not require any iterative method, giving a pseudo-direct method that we called **ABCD** method (Augmented Block Cimmino Distributed method). In this method we create an auxiliary system that is built using many solutions using the direct solver `MUMPS`. As these solutions are independent, they are done in a distributed fashion which leads to an embarrassingly parallel construction of this auxiliary matrix. Moreover, we avoid explicit communication by exploiting the feature of giving a distributed matrix with duplicated entries as input to the direct solver. This solver then handles implicitly the final assembly (sum of the duplicate entries) in the course of its factorization. This contributes even further to the hybrid aspect we are trying to generate through the ABCD method. We also proposed three possible augmentation schemes for obtaining orthogonality that give auxiliary systems of different sizes. We also showed that using hypergraph partitioner gives usually the best results in terms of size of the auxiliary system. Tests on ABCD showed that this method resolved the iteration matrix problems we had had with straight block Cimmino accelerated using block conjugate gradients. We also draw attention to the fact that when the convergence of the classical conjugate gradient acceleration is fast enough, the overhead of building and factorizing the aux-

iliary system can bring no gains. Hence, our hybrid ABCD method incorporates a parameter to either solve the linear system with classical CG acceleration (iterative mode) or to build and solve the auxiliary system directly (pseudo-direct mode).

Finally, in Chapter 6, we investigated how to reduce the overhead generated by the auxiliary system. Two approaches were discussed, firstly we talked about reducing the size of the matrix by relaxing the augmentation. This reduction exploits properties of the augmentation columns and drops columns that would less disturb the numerical orthogonality. Three strategies were investigated depending on the augmentation scheme. The results of this strategy were not entirely satisfactory as the process of building the reduced auxiliary system loses most of the parallel capabilities of the process of building the full auxiliary system. Moreover, ill conditioning of the iteration matrix quickly increases as we relax more and more the augmentation. Also the computation of the columns of the auxiliary matrix is done via a conjugate gradient process which slows down the construction of this matrix.

Secondly, we proposed a way to use the auxiliary system arising from a full augmentation without building it. This solution exploits the fact that the auxiliary matrix is symmetric positive definite and that a matrix-vector computation amounts to a solution using a direct solver. We tested this approach using the conjugate gradient method on small problems and noticed it was difficult to obtain good convergence without any preconditioning. We proposed a preconditioner that can be built in parallel, following the strategies developed in the previous filtering approach to select the columns to be used in this preconditioner. We also discussed a possible way to create this preconditioner easily using a forthcoming feature in the direct solver `MUMPS` that allows the computation of a few selected entries from the solution. Using this preconditioner in a preconditioned conjugate gradient method allowed the method to converge on some problems, but that was still not satisfactory enough in general. Another test was made using MINRES, and surprisingly this method gave better results than the preconditioned conjugate gradient approach.

Overall. We showed through this work that our method lives up to its promises. It is breaks the complexity of the problems so that memory constraints no longer prohibit us from obtaining the solution. It also offers a parameter that switches the method from an iterative-direct hybrid solver to a pseudo-direct solver. This later, novel, approach solves the convergence problems of the iterative version of our method and also pushes the hybrid approach further by exploiting more features of the direct solver.

Perspectives. This work, and especially the implementation used to obtain the results, are in their early stages. Therefore, much work has still to be done to introduce the improvements we talked about in the manuscript. Among these we have the improved mapping of the partitions on the available processes, using a parallel partitioner and investigating other partitioning libraries. To increase the parallel capabilities of the solver by distributing even more the conjugate gradient acceleration, we can exploit the available slave processes that are currently only used for the direct solver. Finally, we will investigate other preconditioners to better solve the auxiliary system iteratively.

Bibliography

[1] Hui Liu. *The nine chapters on the mathematical art: Companion and commentary*. Oxford University Press, 1999.

[2] Joseph WH Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software (TOMS)*, 12(3):249–264, 1986.

[3] Emmanuel Agullo, Abdou Guermouche, and Jean-Yves LExcellent. A parallel out-of-core multifrontal method: Storage of factors on disk and analysis of models for an out-of-core active memory. *Parallel Computing*, 34(6):296–317, 2008.

[4] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S Li. Superfast multifrontal method for large structured linear systems of equations. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1382–1411, 2009.

[5] PR Amestoy, C Ashcraft, A Buttari, O Boiteau, JY LExcellent, and C Weisbecker. Improving multifrontal methods by means of block low-rank approximation techniques. Technical report, tech. rep., INPT-IRIT, 2012.

[6] T. Elfving. Block-iterative methods for consistent and inconsistent linear equations. *Numerische Mathematik*, 35(1):1–12, 1980.

[7] Randall Bramley and Ahmed Sameh. Row projection methods for large nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(1):168–193, 1992.

[8] Mario Arioli, Iain S. Duff, Daniel Ruiz, and Miloud Sadkane. Block Lanczos techniques for accelerating the Block Cimmino method. *SIAM J. Scientific Computing*, 16(6):1478–1511, 1995.

[9] Stephen L Campbell and Carl D Meyer. *Generalized inverses of linear transformations*, volume 56. Siam, 2009.

[10] Mario Arioli, Iain S. Duff, Joseph Noailles, and Daniel Ruiz. A block projection method for sparse matrices. *SIAM J. Scientific and Statistical Computing*, 13:47–70, 1992.

[11] S. Kaczmarz. Angenaherte Auflosung von Systemen linearer Gleichungen. *Bulletin International de l'Academie Polonaise des Sciences et des Lettres*, 35:355–357, 1937.

[12] G Cimmino. Estensione dell'identità di picone alla più generale equazione differenziale lineare ordinaria autoaggiunta. *Atti della Accad. Nazionanale dei Lincei*, 28:354–364, 1939.

[13] Mario Arioli, Iain S Duff, and Peter PM de Rijk. On the augmented system approach to sparse least-squares problems. *Numerische Mathematik*, 55(6):667–684, 1989.

[14] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Analysis and Applications*, 23(1):15–41, 2001.

[15] Michele Benzi, John C Haws, and Miroslav Tuma. Preconditioning highly indefinite and nonsymmetric matrices. *SIAM Journal on Scientific Computing*, 22(4):1333–1353, 2000.

[16] Iain S Duff and Stéphane Pralet. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM Journal on Matrix Analysis and Applications*, 27(2):313–340, 2005.

[17] S Pralet. *Constrained orderings and scheduling for parallel sparse linear algebra*. PhD thesis, Phd thesis, Institut National Polytechnique de Toulouse. CERFACS Technical Report, TH/PA/04/105, 2004.

[18] Michael T Heath, Esmond Ng, and Barry W Peyton. Parallel algorithms for sparse linear systems. *SIAM review*, 33(3):420–460, 1991.

[19] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, second edition, 1989.

[20] IS Duff and JR Gilbert. Maximum-weighted matching and block pivoting for symmetric indefinite systems. In *Abstract book of Householder Symposium XV*, pages 73–75, 2002.

[21] Daniel Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. *Rutherford Appleton Lab., Oxon, UK, Tech. Rep. RAL-TR-2001-034*, 2001.

[22] Iain S Duff. Developments in matching and scaling algorithms. *PAMM*, 7(1):1010801–1010802, 2007.

[23] Patrick R Amestoy, Iain S Duff, Daniel Ruiz, and Bora Uçar. A parallel matrix scaling algorithm. In *High Performance Computing for Computational Science-VECPAR 2008*, pages 301–313. Springer, 2008.

[24] Alan George and Joseph WH Liu. The evolution of the minimum degree ordering algorithm. *Siam review*, 31(1):1–19, 1989.

[25] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[26] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. Algorithm 837: Amd, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 30(3):381–388, 2004.

[27] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[28] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM journal on numerical analysis*, 16(2):346–358, 1979.

[29] LA Hageman and David M Young. Applied iterative methods academic press. *New York*, pages 18–22, 1981.

[30] Magnus Rudolph Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems, 1952.

[31] Gene H. Golub, Daniel Ruiz, and Ahmed Touhami. A hybrid approach combining Chebyshev filter and conjugate gradient for solving linear systems with multiple right-hand sides. *SIAM J. Matrix Analysis and Applications*, 29(3):774–795, 2007.

[32] Daniel F. Ruiz. *Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment*. Phd thesis, Institut National Polytechnique de Toulouse, 1992. CERFACS Technical Report, TH/PA/92/06.

[33] Dianne P O'Leary. The block conjugate gradient algorithm and related methods. *Linear algebra and its applications*, 29:293–322, 1980.

[34] A. Björck. Solving linear least squares problems by gram-schmidt orthogonalization. *BIT Numerical Mathematics*, 7(1):1–21, 1967.

[35] Karin A Remington and Roldan Pozo. Nist sparse blas: Preliminary performance studies. 1996.

[36] T. A. Davis. University of Florida sparse matrix collection, `http://www.cise.ufl.edu/research/sparse/matrices/`, 2008.

[37] C. Kamath and Ahmed Sameh. A projection method for solving nonsymmetric linear systems on multiprocessors. *Parallel Computing*, 9:291–312, 1988.

[38] Randall Bramley. Row projection methods for linear systems. PhD Thesis 881, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1989.

[39] G. H. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *SIAM J. Numer. Anal.*, 2:205–225, 1965.

[40] Gene H. Golub and Charles F. Van Loan. *Matrix Computations. Fourth Edition*. The Johns Hopkins University Press, Baltimore and London, 2013.

[41] Å. Björck and G. H. Golub. Numerical methods for computing angles between linear subspaces. *Math. Comp.*, 27:579–594, 1973.

[42] Randall Bramley and Ahmed Sameh. Row projection methods for large nonsymmetric linear systems. Technical Report 957, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1990. Also, SISSC, Vol. **13**, January 1992.

[43] Leroy Anthony Drummond Lewis. *Solution of general linear systems of equations using block Krylov based iterative methods on distributed computing environments*. Phd thesis, Institut National Polytechnique de Toulouse, Dec 1995. CERFACS Technical Report, TH/PA/95/40.

[44] M. Arioli, A. Drummond, I. S. Duff, and D. Ruiz. Parallel block iterative solvers for heterogeneous computing environments. In *Algorithms and Parallel VLSI Architectures III, Proceedings of The International Workshop*. Elsevier, 1994.

[45] Mario Arioli, A. Drummond, Iain S. Duff, and Daniel Ruiz. Parallel block iterative solvers for heterogeneous computing environments. In Marc Moonen and Francky Catthoor, editors, *Algorithms and Parallel VLSI Architectures III*, pages 97–108, Amsterdam, 1995. Elsevier.

[46] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings 24th National Conference of the Association for Computing Machinery*, pages 157–172, New Jersey, 1969. Brandon Press.

[47] I. S. Duff, A. M. Erisman, and J.K. Reid. *Direct Methods For Sparse Matrices*. Oxoford University Press, New York, USA, 1989.

[48] A. George. *Computer implementation of the finite-element method*. PhD thesis, Department of Computer Science, Stanford University, Stanford, California, 1971. Report STAN CS-71-208.

[49] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[50] Ümit V. Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, Jul 1999.

[51] Ü. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at http://bmi.os u.edu/ umit/software.htm, 1999.

[52] M. Arioli, I. S. Duff, and D. Ruiz. Stopping criteria for iterative solvers. *SIAM J. Matrix Anal and Applics.*, 13:138–144, 1992. Also, Technical Report TR/PA/91/58 CERFACS, France.

[53] W. Oettli and W. Prager. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides. *Numer. Math.*, 6:405–409, 1964.

[54] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the harwell-boeing sparse matrix collection. Rutherford Appleton Laboratory, Chilton DIDCOT Oxon OX11 0QX, December 1992. RAL-92-086.

[55] L. A. Drummond, I. S. Duff, and D. Ruiz. A parallel distributed implementation of the block conjugate gradient algorithm. Technical Report TR/PA/93/02, CERFACS, Toulouse, France, 1993.

[56] U.V. Catalyürek and C. Aykanat. Patoh: A multilevel hypergraph partitioning tool, version 3.0. *Bilkent University, Department of Computer Engineering, Ankara*, 6533, 1999.

[57] E.F. Kaasschieter. Preconditioned conjugate gradients for solving singular systems. *Journal of Computational and Applied mathematics*, 24(1):265–275, 1988.