Solution of general linear systems of equations using block Krylov based iterative methods on distributed computing environments

Leroy Anthony Drummond Lewis

December 18, 1995

Résolution de systèmes linéaires creux par des méthodes itératives par blocs dans des environnements distribués hétérogènes

Résumé

Nous étudions l'implantation de méthodes itératives par blocs, dans des environnements multiprocesseur à mémoire distribuée, pour la résolution de systèmes linéaires quelconques. Dans un premier temps, nous nous intéressons à l'étude du potentiel de la méthode du gradient conjugué classique en environnement parallèle. Dans un deuxième temps, nous étudions une autre méthode itérative de la même famille que celle du gradient conjugué, qui a été conçue pour résoudre simultanément des systèmes linéaires avec de multiples seconds membres, et communément référencée sous le nom de gradient conjugué par blocs.
La complexité algorithmique de la méthode du gradient conjugué par blocs est supérieure à celle de la méthode du gradient conjugué classique, car elle demande plus de calculs par itération, et nécessite en outre plus de mémoire. Malgré cela, la méthode du gradient conjugué par blocs apparaît comme étant mieux adaptée aux environnements vectoriels et parallèles.
Nous avons étudié trois variantes du gradient conjugué par blocs basées sur différents modèles de programmation parallèle et leur efficacité a été comparée sur divers environnements distribués.
Pour la résolution des systèmes linéaires non symétriques, nous considérons l'utilisation de méthodes itératives de projection par lignes, accélérées par la méthode du gradient conjugué par blocs. Les différentes versions de gradient conjugué par blocs précédemment étudiées sont utilisées en vue de cette accélération. En particulier, nous étudions l'implantation dans des environnements distribués de la méthode de Cimmino par blocs accélérée par la méthode du gradient conjugué par blocs. La combinaison de ces deux techniques présente en effet un bon potentiel de parallélisme.
Pour une bonne performance de l'implantation de cette dernière méthode dans des environnements distribués hétérogènes, nous avons étudié différentes stratégies de répartition des tâches aux divers processeurs, et nous comparons deux séquencements statiques réalisant cette répartition. Le premier a pour objectif de maintenir l'équilibre des charges et le second a pour but de réduire en premier les communications entre les différents processeurs tout en essayant d'équilibrer aux mieux la charge des processeurs.
Finalement, nous étudions des stratégies de prétraitement des systèmes linéaires pour améliorer la performance de la méthode itérative basée sur la méthode de Cimmino.

Mots clefs.
La méthode de Cimmino, gradient conjugué, calcul distribué, calcul hétérogène, méthodes itératives, calcul parallèle, prétraitement des systèmes linéaires, séquencement, systèmes linéaires creux et symétriques, matrices creuses, systèmes linéaires creux non symétriques.

Solution of general linear systems of equations using block Krylov based iterative methods on distributed computing environments

Abstract

We study the implementation of block Krylov based iterative methods on distributed computing environments for the solution of general linear systems of equations. First, we study potential implementation of the classical conjugate gradient (CG) method on parallel environments. From the family of conjugate direction methods, we study the Block Conjugate Gradient (Block-CG) method which is based on the classical CG method. The Block-CG works on a block of s right-hand side vectors instead of a single one as is the case of the Classical CG, and we study the implementation of the Block-CG on distributed environments.
The complexity of the Block-CG method is higher than the complexity of the classical CG in terms of computations and memory requirements. We show that the fact that an iteration of the Block-CG requires more computations than the classical CG makes the Block-CG more suitable for vector and parallel environments. Additionally, the increase in memory requirements is only a multiple of s which generally is much smaller than the size of the linear system being solved. We present three models of distributed implementations of the Block-CG and discuss the advantages and disadvantages from each of these model implementations.
The Classical CG and Block-CG are suitable for the solution of symmetric and positive definite systems of equations. Furthermore, both methods guarantee, in exact arithmetic, to find a solution to positive definite systems in a finite number of iterations.
For non symmetric linear systems, we study block row projection iterative methods for solving general linear systems of equations, and we are particularly interested in showing that the rate of convergence of some row projection methods can be accelerated with the Block-CG method. We review two block projection methods, the block Cimmino and the block Kaczmarz method. Afterwards, we study the implementation of an iterative procedure based on the block Cimmino method using the Block-CG method to accelerate its rate of convergence on distributed computing environments. The complexity of the new iterative procedure is higher than the one of the Block-CG method in terms of computations and memory requirements. In this case, the main advantage is the extension of the application of the CG based methods to general linear systems of equations. We present a parallel implementation of the block Cimmino method with Block-CG acceleration that performs well on distributed computing environments.
This last parallel implementation opens a study of potential scheduling strategies for distributing tasks to a set of computing elements. We present a scheduler for heterogeneous computing environments which is currently implemented inside the block Cimmino based solver and can be reused inside parallel implementations of several other iterative methods.
Lastly, we combine all of the above efforts for parallelizing iterative methods with preprocessing strategies to improve the numerical behavior of the block Cimmino based iterative solver.

Keywords.
Cimmino method, conjugate gradient, distributed computing, heterogeneous computing, iterative methods, parallel computing, preprocessing linear systems, scheduling, symmetric sparse linear systems, sparse matrices, unsymmetric sparse linear systems.

Acknowledgements

The author gratefully acknowledges the following people and organizations for their immeasurable support,

Iain S. Duff
For his professional support and research guidance,

Daniel Ruiz
For the time and efforts he has dedicated to my thesis, and the genuine pleasure that has been working with him,

Joseph Noailles
Who was responsible for my Ph.D. studies,

Mario Arioli
For his remarkable contributions to my work,

Craig Douglas
For all his insights, and valuable comments. Proof-reading my thesis during the process of preparation, and great advice; "you are supposed to have fun while writing your thesis",

J. C. Diaz
For his constant support in my career,

Dominique Bennett
For all the help settling in Toulouse and CERFACS, which is an indirect but substantial support for four years of work in France,

CERFACS
For providing the scientific resources and pleasant working environment. To all my colleagues, and specially to Osni Marques for helping me plotting and finding eigenvalues,

ENSEEIHT - IRIT
The APO team for hosting me in their group, and sharing with me their research experiences, and great resources. Brigitte Sor and Frederic Noailles for their outstanding computing support,

MCS Argonne National Laboratory
For letting me use their parallel computing facilities,

IAN-CNR Pavia Italy
For hosting me in Summer 1994,

CNUSC
For the use of SP2 Thin node,

My family
For always being with me. To Sandra Drummond for her help and company. My friends for all that I have found, learned and shared in our friendship.

Contents

Chapter 1

Introduction

The class of iterative methods comprises a wide range of procedures for solving various types of linear systems. Iterative procedures use successive approximations to the solution of a linear system of equations. A few references to general overviews of iterative methods are Hageman and Young (1981), Barrett, Berry, Chan et al. (1993), van der Vorst (1992) and Stoer and Bulirsch (1980).
The class of iterative methods is subdivided into stationary and nonstationary iterative methods. The former methods are easier to implement and understand but they are less efficient, in terms of convergence, than the latter ones. An iterative method is stationary if it can be expressed in the form

$$x^{(j)} = Bx^{(j-1)} + c$$

where B and c are independent of the iteration count j. Examples of these methods are the Jacobi method, the Gauss-Seidel method, the Successive Over Relaxations method (SOR), Symmetric SOR method (SSOR), the Cimmino method, and the Kaczmarz method.
On the contrary, the computations in a nonstationary iterative methods involve information that changes at each iteration. Examples of these methods are the Krylov projection methods, Classical CG, Chebyshev iteration, Minimal Residual (MINRES), and Generalized Minimal Residual (GMRES).
In general, the Krylov projection methods find a sequence of approximations to the solution of a linear system by constructing an orthogonal basis for the Krylov subspaces. Because these Krylov subspaces are finite, an adequate approximation to the solution is found in a finite number of steps in absence of roundoff errors.
The finite termination property makes the Krylov projection methods very attractive, even when used with stationary methods to accelerate their rate of convergence.

The Block Conjugate Gradient algorithm (Block-CG) also belongs to the Krylov projection methods and is based on the Classical Conjugate Gradient method (CG). Class of Block-CG can be used in the solution of linear systems with multiple right-hand sides. In this case, the Block-CG algorithm is used to concurrently solve linear systems of equations with s right-hand sides instead of solving the linear systems using s applications of the classical CG algorithm.
We propose developing a parallel iterative solver for distributed environments based on a stationary method, the block Cimmino method, that uses a nonstationary method, the block conjugate gradient, to accelerate its rate of convergence. During the development of the parallel iterative

solver, we have built a set of modules that can be reused in other parallel iterative procedures, and have focused on answers to the following statements.

E–1.1 Are there advantages in implementing the classical CG method and Block-CG method on distributed computing environments, without perturbing the robustness of these methods?

E–1.2 Can the efforts of parallelizing the Classical CG and Block-CG be reused inside other iterative procedures? For instance, in the parallel implementation of an iterative solver based on the block Cimmino method with Block-CG acceleration.

E–1.3 What are the parameters that influence the parallel performance of the block Cimmino based solver?

E–1.4 Is there a need to design a scheduler to evenly distribute the workload from iterations of the block Cimmino based solver? If the need is affirmed, can the scheduler be reused in the parallel implementation of other iterative procedures?

E–1.5 Preprocessing a linear system can reduce the number of iterations, but is there also an influence on the parallel performance of the block Cimmino based solver?

In Chapter 2, we study the formulations of the Classical CG method. We review the efforts of several authors that have studied the implementation of the Classical CG on distributed environments. From the Classical CG formulations, a stabilized Block-CG algorithm can be derived. The algorithm is described as stable because it corrects some instability problems during the Block-CG computations that are due to solution vectors that converge at different rates and ill-conditioning that may appear inside the residual matrix $R^{(j)}$.
The order of complexity of the Block-CG algorithm is higher than Classical CG in terms of computations and memory requirements. However, in the absence of roundoff errors, Block-CG promises a faster rate of convergence than Classical CG, and the matrix-matrix computations inside a Block-CG iteration can benefit from faster numerical kernels that improve data locality and increase the Mflop rate. These are desirable features from an algorithm to be implemented in vector and parallel environments. Chapter 3 is devoted to an analysis of the potential parallelization of the Classical CG and Block-CG algorithms to answer statement E–1.1.
The parallelization of the Block-CG algorithm opens more questions about which sections of the algorithm should be parallelized, and which parallel programming model should be used. Three parallel implementations of the stabilized Block-CG algorithm are presented in Chapter 4.
When parallelizing many sequential algorithms, there is a tendency to identify the sections of the algorithm with heavy computations and only parallelize these sections. In the Classical CG and Block-CG the heavy computational sections have been identified by other authors as the matrix-vector products $Ax^{(j)}$ in the Classical CG, or the matrix-matrix products $AP^{(j)}$ in the Block-CG. The results from our analysis of the Block-CG in Section 2.3 prove that parallelizing the $AP^{(j)}$ is not always efficient nor sufficient. Furthermore, we demonstrate this effect with a parallel Block-CG implementation described in Section 4.2.3.
In contrast, if the whole Classical CG algorithm or Block-CG algorithms is parallelized, then the parallel efficiency is proportional to the rates of computation speed to communication speed. Parallelizing the whole Block-CG algorithm leads to a decision of which programming model should be used. In Section 4.2.2 we propose a $Master - Slave$ programming model for parallelizing the whole Block-CG algorithm. And in Section 4.2.1 a different programming model

is used in which the master role is removed and all the parallel processes work at the same level. Results from comparisons of the three parallel Block-CG implementations are presented in Chapter 5.
The Block-CG algorithm only guarantees convergence when the system is symmetric positive definite, although, with the use of a preconditioner, the algorithm can be extended to solve general unsymmetric systems. Block-CG can be used as an acceleration procedure inside another basic iterative method.
In Chapter 6, we study the block Cimmino and block Kaczmarz methods that are regarded as iterative row projection methods for solving general systems of equations. In the same chapter, we review an example of the Block-CG acceleration inside the block Cimmino method.
To answer statement E–1.2, an iterative procedure of the block Cimmino method accelerated with the stabilized Block-CG Algorithm is presented in Section 6.3. And the parallel implementation of this procedure is discussed in Section 6.4 and Section 6.5.
In Section 6.5, we identify that there is a need for a task scheduler that distributes the workload among the different computing elements (CE) in the system, therefore, we study different scheduling strategies in Chapter 7.

One of the advantages of working in heterogeneous computing environments is the ability to provide certain level of computing performance proportional to the resources available in the system and their different computing capabilities. A scheduler is regarded as a strategy or policy to efficiently manage the use of these resources. In parallel distributed environments with homogeneous resources, the level of performance is commensurate with the number of resources present in the system.
A scheduler for parallel iterative methods in heterogeneous computing environments is presented in Chapter 7. The scheduler responds to statement E–1.4. Design issues from the scheduler are discussed in Chapter 7, where we see that the scheduler can easily be modified to suit other parallel iterative procedures.
In heterogeneous computing environments, the scheduler not only considers information from the tasks to be executed in parallel but must also consider information about the capabilities of the CE's. In Section 7.3, a syntax for specifying heterogeneous environments is defined.

The effects of using Block-CG acceleration inside the parallel block Cimmino solver and the effects from different scheduling strategies are separately studied in Chapters 8 and 9. In Chapter 8, the effects of the scheduler on the performance of the parallel block Cimmino solver are studied. And in Chapter 9, the effects of the Block-CG acceleration on the parallel block Cimmino solver are studied, together with an analysis of the parallel performance of the block Cimmino solver.
Also, from the results obtained in Chapters 8 and 9, we identify a need for studying natural preprocessing and partitioning strategies of the linear system of equations.
Preprocessing a linear system of equations improves the performance of most linear solvers and in some cases a more accurate approximation to the real solution is found. For instance, numerical instabilities are encountered in the solution of some linear systems and it is necessary to scale the systems before they are solved. These instabilities occur even when the most robust computer implementations of direct or iterative solvers are used.
Also, performing some permutations of the elements of the original system can substantially reduce the required computing time for the solution of large sparse linear systems by improving the rate of convergence of some iterative methods (e.g., SOR, and Kaczmarz methods).

Preprocessing and partitioning strategies are studied in Chapter 10. The effects of the preprocessing and partitioning strategies on the parallel block Cimmino solver are studied in Chapter 11, and this chapter closes with remarks that answer statement E–1.5.
Lastly, general conclusions and future related work are proposed in Chapter 12.

Chapter 2

Block conjugate gradient

The Block Conjugate Gradient algorithm (Block-CG) belongs to a family of conjugate direction methods and the study of these methods is appealing because they guarantee convergence in a finite number of steps in the absence of round off errors. The Block-CG is based on the classical Conjugate Gradient method (CG) for solving linear systems of equations. The Block-CG can be used in the solution of linear systems with multiple right-hand sides. In this case, the Block-CG algorithm is used to concurrently solve the linear system of equations with s right-hand sides instead of solving the linear systems using s applications of the classical CG algorithm. In addition, the Block-CG algorithm is expected to converge faster than classical CG for linear system of equations with clusters of eigenvalues that are separated. In this chapter, we will present some examples of this effect.
First, we study some relevant properties of the classical CG method and review some of the main issues involved in parallelizing different versions of the CG algorithm. Then, we describe a stabilized Block-CG algorithm and compare it with classical CG.

2.1 Conjugate gradient algorithm

Given the linear system of equations

$$Ax = b \tag{2.1.1}$$

where A is an $n \times n$ symmetric matrix, we write the following Richardson iteration after splitting the matrix A as $I - (I - A)$:

$$\begin{aligned} (I - (I - A))x &= b \\ x^{(j+1)} &= b + (I - A)x^{(j)} \\ x^{(j+1)} &= r^{(j)} + x^{(j)} \end{aligned} \tag{2.1.2}$$

If we multiple by $-A$ and add b to (2.1.2):

$$b - Ax^{(j+1)} = -Ar^{(j)} + b - Ax^{(j)}$$

$$\begin{aligned} r^{(j+1)} &= r^{(j)} - Ar^{(j)} \\ &= (I - A)r^{(j)} \\ &= (I - A)^{(j+1)}r^{(0)} \\ &= \mathcal{P}^{(j+1)}(A)r^{(0)} \end{aligned}$$

where $\mathcal{P}^{(j+1)}$ is a polynomial of degree $j + 1$ with $\mathcal{P}^{(j+1)}(0) = 1$.

Using this polynomial we can rewrite (2.1.2) in the following way:

$$\begin{aligned} x^{(j+1)} &= r^{(0)} + r^{(1)} + \ldots + r^{(j)} + x^{(0)} \\ &= \sum_{k=0}^{j}(I - A)^k r^{(0)} + x^{(0)} \end{aligned} \tag{2.1.3}$$

Without loss of generality, we will assume $x^{(0)} = 0$, and if $x^{(0)} \neq 0$ then a linear transformation of the form $z = x - x^{(0)}$ can be considered, and the system of equations will look like:

$$\begin{aligned} Az &= b - Ax^{(0)} \\ &= \bar{b}, \end{aligned}$$

where $z^{(0)} = 0$.
Removing $x^{(0)}$ from (2.1.3) we get:

$$x^{(j+1)} = \sum_{k=0}^{j}(I - A)^k r^{(0)} \tag{2.1.4}$$

From (2.1.4), it is inferred that the new $x^{(j+1)}$ is in the subspace $\{r^{(0)}, Ar^{(0)}, \ldots, A^j r^{(0)}\}$ which is the Krylov subspace $\mathcal{K}^{j+1}(A; r^{(0)})$.
If x^* is the exact solution to (2.1.1), then we build an iterative procedure that finds at each iteration an approximation $x^{(j)}$ to x^* that satisfies:

$$\min \|x^* - x^{(j)}\|^2$$

for $x^{(j)} \in \mathcal{K}^j(A; r^{(0)})$ and a given norm. Furthermore, if the matrix A is symmetric positive definite (SPD), then we can use the following proper inner product:

$$\begin{aligned} (x, y)_A &= (y, x)_A = (x, Ay), \\ (x, x)_A &= 0 \Leftrightarrow x = 0 \end{aligned} \tag{2.1.5}$$

For instance, if $x^{(1)} \in \text{Span}\{r^{(0)}\}$ then $x^{(1)} = \alpha r^{(0)}$,

$$\min \|x^* - x^{(1)}\|_A^2 = \left(x^* - \alpha r^{(0)}, x^* - \alpha r^{(0)}\right)_A,$$

and has a minimum with respect to α at

$$\alpha = \frac{(r^{(0)}, x^*)_A}{(r^{(0)}, r^{(0)})_A} = \frac{(r^{(0)}, Ax^*)}{(r^{(0)}, Ar^{(0)})}.$$

In general, we want to find $\min \|x^* - x^{(j)}\|_A$ for $x^{(j)} \in \mathcal{K}^j(A; r^{(0)})$

$$\begin{aligned} &\Rightarrow \quad x^* - x^{(j)} \perp_A \mathcal{K}^j(A; r^{(0)}) \\ &\Rightarrow \quad r^{(j)} \perp \mathcal{K}^j(A; r^{(0)}). \end{aligned}$$

If we repeat the search for an approximation j times, we build a set of orthogonal residual vectors $\{r^{(0)}, r^{(1)}, \ldots, r^{(j)}\}$ that is an orthogonal basis for the Krylov subspace $\mathcal{K}^j(A; r^{(0)})$. Further,

$$\mathcal{K}^j(A; r^{(0)}) = \text{Span}\left\{r^{(0)}, Ar^{(0)}, \ldots, A^j r^{(0)}\right\}. \tag{2.1.6}$$

We continue to generate orthogonal residual vectors until a full basis for the subspace $\mathcal{K}^j(A; r^{(0)})$ is found. Beyond this point, we cannot generate more orthogonal vectors and have a solution to (2.1.1).

Inside the iterative procedure, a new $x^{(j)}$ is computed after projecting the residual $r^{(j)}$ onto the Krylov subspace $\mathcal{K}^j(A; r^{(0)})$ with respect to A.

Theorem 2.1 *The orthogonal basis satisfies the following 3-term recurrence:*

$$\alpha^{(j+1)} r^{(j+1)} = Ar^{(j)} - \beta^{(j)} r^{(j)} - \gamma^{(j)} r^{(j-1)} \tag{2.1.7}$$

Proof: The proof is by induction. We start with

$$\begin{aligned} r^{(1)} &\in \text{Span}\{r^{(0)}, Ar^{(0)}\}, \\ \alpha^{(1)} r^{(1)} &= Ar^{(0)} - \beta^{(0)} r^{(0)} \end{aligned}$$

and

$$\begin{aligned} r^{(2)} &\in \text{Span}\{r^{(0)}, Ar^{(0)}, A^2 r^{(0)}\} \\ &\Rightarrow \text{Span}\{r^{(0)}, r^{(1)}, Ar^{(1)}\} \\ \alpha^{(2)} r^{(2)} &= Ar^{(1)} - \beta^{(1)} r^{(1)} - \gamma^{(1)} r^{(0)}. \end{aligned}$$

By induction,

$$r^{(j-1)} \in \mathcal{K}^j(A; r^{(0)}) \Rightarrow r^{(j)} \in \mathcal{K}^{j+1}(A; r^{(0)}),$$

where

$$\mathcal{K}^{(j+1)}(A; r^{(0)}) = \text{Span}\{r^{(0)}, r^{(1)}, \ldots r^{(j)}\}.$$

Thus,

$$\alpha^{(j)} r^{(j)} = A r^{(j-1)} - \sum_{i=0}^{j-1} \delta^{(i)} r^{(i)}. \tag{2.1.8}$$

Since $r^{(j)}$ must be orthogonal to $\{r^{(0)}, r^{(1)}, \ldots, r^{(j-1)}\}$, we must find the values of $\delta^{(i)}$ that satisfy this condition. In other words,

$$(r^{(j)}, r^{(i)}) = 0.$$

Taking the inner product of (2.1.8) with $r^{(i)}$ yields

$$(A r^{(j-1)}, r^{(i)}) - \delta^{(i)} (r^{(i)}, r^{(i)}) = 0.$$

Using the property of inner products (2.1.5) gives us

$$\begin{aligned} (r^{(j-1)}, r^{(i)})_A - \delta^{(i)} (r^{(i)}, r^{(i)}) &= 0 \qquad \text{and} \\ (r^{(j-1)}, A r^{(i)}) - \delta^{(i)} (r^{(i)}, r^{(i)}) &= 0. \end{aligned}$$

Using the induction argument for $r^{(j-1)}$

$$\frac{\left(r^{(j-1)}, \alpha^{(i+1)} r^{(i+1)} + \beta^{(i)} r^{(i)} + \gamma^{(i)} r^{(i-1)}\right)}{(r^{(i)}, r^{(i)})} = \delta^{(i)} \Big|_{i=0}^{j-1},$$

were $\delta^{(i)} = 0$ for $i = 0, \ldots, j-3$, (2.1.8) is reduced to

$$\alpha^{(j)} r^{(j)} = A r^{(j-1)} - \delta^{j-2} r^{(j-2)} - \delta^{j-1} r^{(j-1)}$$

or

$$\alpha^{(j+1)} r^{(j+1)} = A r^{(j)} - \beta^{(j)} r^{(j)} - \gamma^{(j)} r^{(j-1)}.$$

This completes the proof of (2.1.7).

The 3-term recurrence relation (2.1.7) can be rearranged as

$$A r^{(j)} = \alpha^{(j+1)} r^{(j+1)} + \beta^{(j)} r^{(j)} + \gamma^{(j)} r^{(j-1)}. \tag{2.1.9}$$

Let $\{r^{(0)}, r^{(1)}, \ldots, r^{(j-1)}\}$ be the columns of a matrix $R^{(j)}$. From (2.1.9) we get

$$AR^{(j)} = R^{(j)} \begin{bmatrix} \ddots & \ddots & & & \\ \ddots & \ddots & \gamma^{(j)} & & \\ & \ddots & \beta^{(j)} & \ddots & \\ & & \alpha^{(j+1)} & \ddots & \ddots \\ & & & \ddots & \ddots \end{bmatrix} + \alpha^{(j)} \left(0, 0, \ldots, r^{(j)}\right)$$

or

$$AR^{(j)} = R^{(j)}T^{(j)} + \alpha^{(j)}r^{(j)}e_1^T \tag{2.1.10}$$

where $T^{(j)}$ is an $j \times j$ tridiagonal matrix, and e_1 is a canonical vector. We can write the new approximation as a linear combination of $R^{(j)}$, namely,

$$x^{(j)} = R^{(j)}y. \tag{2.1.11}$$

The $x^{(j)}$ that minimizes

$$\min \|x^* - x^{(j)}\|_A^2$$

is derived from

$$\begin{aligned} R^{(j)^T}(Ax^{(j)} - b) &= 0 \\ R^{(j)^T}Ax^{(j)} &= R^{(j)^T}b \\ R^{(j)^T}AR^{(j)}y &= R^{(j)^T}b \end{aligned}$$

From (2.1.10),

$$R^{(j)^T}(R^{(j)}T^{(j)} + \alpha^{(j)}r^{(j)}e_1^T)y = R^{(j)^T}b.$$

Since $r^{(j)}$ is orthogonal with respect to the columns of $R^{(j)}$, we have

$$R^{(j)^T}R^{(j)}T^{(j)}y = \|r^{(0)}\|_2^2 e_1,$$

where

$$R^{(j)^T}R^{(j)} = \begin{bmatrix} \|r^{(0)}\|_2^2 & & \\ & \ddots & \\ & & \|r^{(j-1)}\|_2^2 \end{bmatrix}.$$

Thus,

$$T^{(j)}y = e_1 \Rightarrow y = T^{(j)^{-1}}e_1. \tag{2.1.12}$$

If A is an SPD matrix, then in the relation

$$R^{(j)^T}AR^{(j)} = R^{(j)^T}R^{(j)}T^{(j)},$$

from 2.1.10, $T^{(j)}$ can be transformed into an SPD matrix (e.g., using some row-scaling) and LU decomposed without any pivoting

$$T^{(j)} = L^{(j)}U^{(j)},$$

then, from (2.1.11) and (2.1.12), we have

$$\begin{aligned} x^{(j)} &= R^{(j)}y \\ &= R^{(j)}T^{(j)^{-1}}e_1 \\ &= (R^{(j)}U^{(j)^{-1}})(L^{(j)^{-1}}e_1) \end{aligned}$$

Labeling the terms in parenthesis

$$P^{(j)} \equiv R^{(j)}U^{(j)^{-1}} \text{ and } Q^{(j)} \equiv L^{(j)^{-1}}e_1,$$

after algebraic manipulation, we obtain an equation for the new approximation of

$$x^{(j)} = x^{(j-1)} + q^{(j-1)}p^{(j-1)}.$$

The columns of the matrix P are A-conjugate hence the name of the Conjugate Gradient method. Algorithm 2.1.1 is a version of the classical CG algorithm from Hestenes and Stiefel (1954). On the other hand, if we do not assume the positive definitiveness of A in (2.1.12) we can only hope that $T^{(j)}$ is not singular and the procedure is known as the Lanczos method for symmetric systems (see Lanczos (1952)).

Algorithm 2.1.1 (Conjugate Gradient)

(1) $x^{(0)}$ is arbitrary and $r^{(0)} = b - Ax^{(0)}$

(2) $p^{(0)} = 0$, $\beta^{(-1)} = 0, \alpha^{(-1)} = 0$

(3) For $j = 0, 1, \ldots$, until convergence do:

- (3.1) $p^{(j)} = r^{(j)} + \beta^{(j-1)}p^{(j-1)}$
- (3.2) $q^{(j)} = Ap^{(j)}$
- (3.3) $\lambda = (p^{(j)}, q^{(j)})$
- (3.4) $x^{(j)} = x^{(j-1)} + \alpha^{(-1)}p^{(j-1)}$
- (3.5) $\alpha^{(j)} = \frac{\gamma^{(j)}}{\lambda}$
- (3.6) $r^{(j+1)} = r^{(j)} - \alpha^{(j)}q^{(j)}$
- (3.7) $\gamma^{(j+1)} = (r^{(j+1)}, r^{(j+1)})$
- (3.8) If $(r^{(j+1)})$ is small enough, then
  - (3.8.1) $x^{(j+1)} = x^{(j)} + \alpha^{(j)}p^{(j)}$
  - (3.8.2) stop
- (3.9) $\beta^{(j)} = \frac{\gamma^{(j+1)}}{\gamma^{(j)}}$

(4) Stop

From Algorithm 2.1.1 we see that the CG algorithm requires special consideration for its parallelization. We notice that the computational weight of the algorithm is on step (3.2). Apart from this computation, as shown in Demmel, Heath, and van der Vorst (1993), there are 7 arrays that need to be loaded in cache memory (or memory registers) at each iteration to perform only 10 floating point operations on these arrays which results in poor data locality. Independent of the computer architecture, poor data locality always implies a high number of data transfers which happens between different memory hierarchies, or in the worst case between processors.
Another bottleneck in the algorithm is the computation of the steps (3.3) and (3.7). In the parallel version of the algorithm, the inner products performed in these two steps require that the different processors exchange local data to complete them. Clearly, these two steps are the synchronization points in the algorithm. Moreover, steps (3.4), (3.5), and (3.6) depend on the results from step (3.3). Similarly, step (3.1) depends on the results from step (3.7). These data dependencies prevent us from overlapping communication with useful computations.
In order to reduce the number of synchronization points we review a few proposed algorithms. Saad suggested removing one synchronization point at the expense of numerical instabilities (see Saad (1985), Saad (1989)). Meurant (1987) proposed a version of the CG algorithm that eliminates one synchronization point in Algorithm 2.1.1 and increases the data locality factor to 2. However, the cost of his improvement reduces the numerical stability of the algorithm and adds one inner product. Chronopoulos proposed a different version of the CG algorithm named the s-step conjugate gradient algorithm (see Chronopoulos and Gear (1991)).
The s-step conjugate gradient is based on the idea of generating s orthogonal vectors in one iteration step by first building $r^{(j)}, Ar^{(j)}, \ldots, A^{(s-1)}r^{(j)}$ and orthogonalize and update the current approximation to the solution in the resulting subspace. The s-step improves the data locality and the parallelism from the classical CG algorithm since the arrays are loaded only once per iteration. Also, in the s-step the number of synchronization points is reduced to one (see Chronopoulos (1989)). The s-step performs $2 \cdot n$ more floating point operations per iteration than the classical CG, which is not critical for large values of s because a large value of s not only increase the granularity of parallel subtasks but may also reduce the number of iteration steps before reaching convergence. However, the s-step introduces numerical instabilities by explicitly computing $A^{(s-1)}$. Also, depending on the spectrum of the matrix A the set of vectors $r^{(j)}, Ar^{(j)}, \ldots, A^{(s-1)}r^{(j)}$ may convergence to a vector in the direction of the dominant eigenvectors and not converge to the solution of the linear system. The weaknesses of the s-step algorithm are further reviewed in Saad (1988).
D'Azevedo, Eijkhout, and Romine (1993) introduced two new stable variants to the CG algorithm. In these versions, the authors substitute the second inner product by equivalent algebraic expressions. They have shown that these version of the algorithm are stable and claim an improvement on the performance of the classical CG algorithm of 5% to 13%. A different approach was presented by Demmel, Heath, and van der Vorst (1993) in which one overlaps communication with useful computations. Their original approach was proposed for preconditioned CG. For the non-preconditioned CG one has to split the computation (3.4) into two phases to overlap communication with useful computations. And in general, the improvement from overlapping communication with useful computation will be determined by the speed of the communication network, the computing processors, and the size of the system to be solved.

2.2 A stable Block-CG algorithm

In the previous section, the matrix A is assumed to be SPD. The fact that A is positive definite guarantees the termination of the algorithm in absence of round off errors. However, Algorithm 2.1.1 may still work, without guarantee, for non positive-definite matrices.
Some algorithms based on the CG methods have been designed for solving symmetric indefinite linear systems. A few examples of these algorithms are CGI (see Modersitzki (1994)), MINRES and SYMMLQ (see Paige and Saunders (1975)). And in some cases, one can use a preconditioner matrix H which is a good approximation to A^{-1} and SPD.
Here we consider a Block-CG algorithm for SPD matrices and later in Chapter 6 we will study the Block-CG method for accelerating the rate of convergence of a another basic block row projection method.
In the Block-CG algorithm the term *"block"* refers to multiple solution vectors, and the number of these solutions defines the block size. Let s be the block size for the system of linear equations

$$HX = K. \tag{2.2.1}$$

Here, the vectors x and b from (2.1.1) are replaced by the X and K matrices respectively, and both matrices are of order $n \times s$. Now we write the block form for Algorithm 2.1.1.

Algorithm 2.2.1 (Block Conjugate Gradient)

(1) $X^{(0)}$ is arbitrary, $P^{(0)} = R^{(0)} = K - HX^{(0)}$

(2) For $j = 0, 1, \ldots$, until convergence do:

(2.1) $$X^{(j+1)} = X^{(j)} + P^{(j)} \left(P^{(j)^T} H P^{(j)} \right)^{-1} R^{(j)^T} R^{(j)}$$

(2.2) $$R^{(j+1)} = R^{(j)} - H P^{(j)} \left(P^{(j)^T} H P^{(j)} \right)^{-1} R^{(j)^T} R^{(j)}$$

(2.3) $$P^{(j+1)} = R^{(j+1)} + P^{(j)} \left(R^{(j)^T} R^{(j)} \right)^{-1} R^{(j+1)^T} R^{(j+1)}$$

(3) Stop

However, there is a change from the Krylov space considerations used while building Algorithm 2.1.1. Equation (2.1.6) is rewritten here as

$$\mathcal{K}^{(j)} \left(H; R^{(0)} \right) = \text{Span} \left\{ R^{(0)}, HR^{(0)}, \ldots, H^{(j)} R^{(0)} \right\} \tag{2.2.2}$$

With (2.1.6) a space of size j is generated, and with 2.2.2 inside the Block-CG algorithm a space of size $s \times j$ is generated. Furthermore, the columns in the $R^{(j)}$ matrix may become almost

linearly dependent when convergence is about to be reached. A solution to this problem is the orthonormalization of the P and R matrices. As is given by Ruiz (1992), the stabilized Block-CG algorithm is depicted in Algorithm 2.2.2

Algorithm 2.2.2 (Stabilized Block Conjugate Gradient)

(1) $X^{(0)}$ is arbitrary, $R^{(0)} = K - HX^{(0)}$

(2) $\overline{R}^{(0)} = R^{(0)}\gamma_0^{-1}$ such that $\left(\overline{R}^{(0)^T}\overline{R}^{(0)} = I\right)$

(3) $\overline{P}^{(0)} = \overline{R}^{(0)}\beta_0^{-1}$ such that $\left(\overline{P}^{(0)^T}\overline{HP}^{(0)} = I\right)$

(4) For $j = 0, 1, \ldots$, until convergence do:

(4.1) $\lambda_j = \beta_j^{-T}$

(4.2) $X^{(j+1)} = X^{(j)} + \overline{P}^{(j)}\overline{P}^{(j)^T}R^{(j)}$ where $R^{(j)} = K - HX^{(j)}$

(4.3) $\overline{R}^{(j+1)} = \left(\overline{R}^{(j)} - \overline{HP}^{(j)}\lambda_j\right)\gamma_{j+1}^{-1}$ such that $\left(\overline{R}^{(j+1)^T}\overline{R}^{(j+1)} = I\right)$

(4.4) $\alpha_j = \beta_j\gamma_{j+1}^T$

(4.5) $\overline{P}^{(j+1)} = \left(\overline{R}^{(j+1)} + \overline{P}^{(j)}\alpha_j\right)\beta_{j+1}^{-1}$ such that $\left(\overline{P}^{(j+1)^T}\overline{HP}^{(j+1)} = I\right)$

(5) Stop

In step (4.2) of Algorithm 2.2.2, we do not explicitly compute $R^{(j)} = K - HX^{(j)}$ because this computation involves the $HX^{(j)}$ matrix-matrix product and this product is very expensive when computed in parallel (e.g., processors need to exchange data, which implies more communication and synchronization points). From steps (2) and (4.3) in Algorithm 2.2.2, it can be shown by recurrence that the residuals are given by

$$R^{(j)} = \overline{R}^{(j)}\left(\prod_{i=j}^{0}\gamma_i\right).$$

Therefore, the update of the $X^{(j+1)}$ solution in step (4.2) can be rewritten as

$$X^{(j+1)} = X^{(j)} + \overline{P}^{(j)}\overline{P}^{(j)^T}\overline{R}^{(j)}\left(\prod_{i=j}^{0}\gamma_i\right). \tag{2.2.3}$$

In general, it can be proved that for the Block-CG algorithm the H-norm of the error at each iteration is bounded by its reduced condition number $\kappa = \lambda_n/\lambda_s$, where the λ_j's are the eigenvalues of the H matrix (sorted in increasing order,) and s is the block size (see O'Leary (1980)). This observation represents an advantage of the Block-CG over its non-block counterparts where the error at each iteration is bounded by its condition number $\kappa' = \lambda_j/\lambda_1$.

2.3 Performance of Block-CG vs Classical CG

We first start by comparing the complexity of the Classical CG Algorithm 2.1.1, and the Stabilized Block-CG Algorithm 2.2.2. We focus on the number of floating point operations (FLOP count) per iteration as depicted in Table 2.1. In Chapter 3, we present results from sequential runs of computer implementations of both algorithms to compare their convergence rates.
In Table 2.1, the matrix A is SPD of dimension n. The sparsity pattern of A is used through all of the computations, thus the FLOP count in a matrix-vector operation is a function of the sparsity pattern of the matrix A defined by

$$\theta(A) = \text{flop\_count}(A \times b),$$

for any vector $b \in \Re^n$. In the Block-CG algorithm, s is the block size and usually $s \ll n$.
The FLOP count from one iteration of the Block-CG is greater than s-times the FLOP count from one iteration of the Classical CG. Because of the stability issues discussed in Section 2.2, a few FLOPs more are performed in the orthonormalization of the matrices $\overline{R}^{(j)}$ (steps 2, and 4.3 of Algorithm 2.2.2), $\overline{P}^{(j)}$, and $\overline{HP}^{(j)}$ (steps 3, and 4.4 of Algorithm 2.2.2) . Also the FLOP count increases with the use of Cholesky factorizations to compute $\gamma_{(j)}^{-1}$ (steps 2, and 4.3 of Algorithm 2.2.2) and $\beta_{(j)}^{-1}$ (steps 3, and 4.4 of Algorithm 2.2.2). Both $\gamma_{(j)}$ and $\beta_{(j)}$ have dimensions $s \times s$, and we use the routine DPOTRF from the LAPACK library (Anderson, Bai, Bischof et al. (1995)) to perform the Cholesky factorizations. The increase in the FLOP count from the Cholesky factorizations becomes significant for large values of s.
From Table 2.1, the FLOP count per iteration of the Classical-CG Algorithm 2.1.1 is

$$FLOP\_count = \theta(A) + 10n + 2, \tag{2.3.1}$$

and for one iteration of the Block-CG Algorithm 2.2.2

$$FLOP\_count = (s \cdot \theta(A)) + 12(n \cdot s^2) + 4(n \cdot s) + \frac{11s^3}{3} + s^2 + \frac{s}{3}. \tag{2.3.2}$$

As summarized in (2.3.1) and (2.3.2), the FLOP count from one iteration of Block-CG is much greater than the FLOP count from one iteration of Classical CG for large values of s. Therefore, a large value of s deteriorates the performance of a sequential Block-CG implementation. On the other hand, a large value of s increases the granularity of the associated parallel subtasks, and improves the performance of a parallel Block-CG implementation. Also, in the absence of roundoff errors, the number of synchronization points is implicitly reduced with a large value of s because the number of iterations required to reach convergence is also reduced.
The value of s should be a compromise between the FLOP count, the acceleration in the convergence rate of the Block-CG due to a greater value of s, and resources available in the system. Nikishin and Yeremin (1995) have proposed a Variable Block Preconditioned Conjugate Gradient (VBPCG) algorithm that provides a possibility for adjusting the block size during the preconditioned Block-CG iterations. The goal of varying the block size is to find a compromise between the block size, the convergence rate, the FLOP count and the degree of parallelism. While they have shown that VBPCG algorithm reduces the total FLOP count from the Block preconditioned CG, they have also concluded that performance of the VBPCG algorithm depends on the spectral properties of the preconditioned matrix.

In Chapter 5, we study a compromise between the factors that influence the performance of different parallel implementations of the Block-CG Algorithm 2.2.2 in distributed memory environments.

2.4 Stopping criteria

The results from experiments to be presented in next chapter come from sequential runs of computer implementations of the Classical CG and Block-CG algorithms. Furthermore, the stopping criterion used in this implementations is based on the theory of Oettli and Prager (see Oettli and Prager (1964)) for backward error analysis.
Let $x^{(j)} \in \mathcal{R}^n$, $A \in \mathcal{R}^{n\times n}$ and $b \in \mathcal{R}^n$, and define $\mathrm{r}(x^{(j)}) = b - Ax^{(j)}$. For any matrix E, $\mathrm{E} \geq 0$, and a vector f, $\mathrm{f} \geq 0$, define

$$\omega = \max_i \frac{|\ \mathrm{r}(x^{(j)})_i\ |}{(\mathrm{E}\ |\ x^{(j)}\ |\ +\mathrm{f})_i}, \tag{2.4.1}$$

with

$$\omega = \frac{0}{0} = 0, \qquad \text{and} \qquad \omega = \frac{\rho}{0} = \infty.$$

If $\omega \neq \infty$, there exists a matrix δA and a vector δb with

$$|\ \delta A\ | \leq \omega \mathrm{E} \quad \text{and} \quad |\ \delta b\ | \leq \omega \mathrm{f},$$

such that,

$$(A + \delta A)\, x^{(j)} = b + \delta b$$

and ω is the smallest number for which such δA and δb exist. Therefore, we have the solution to a nearby problem when we compute a small ω for a given E and f.
At the end of each iteration we compute an ω that satisfies (2.4.1) and we stop the iterations when either ω is reduced to a given threshold value or ω cannot be reduced anymore.
There are different ways of choosing E and f (see for instance Arioli, Demmel, and Duff (1989), and Arioli, Duff, and Ruiz (1992)). Here we have used

$$\mathrm{E} = \|A\|_\infty e e^T \quad \text{and} \quad \mathrm{f} = \|b\|_\infty e,$$

where e is a column vector of all 1's. This choice of E and f corresponds to the normwise backward error defined by

$$\omega = \frac{\|r(x^{(j)})\|_\infty}{\|A\|_\infty \|x^{(j)}\|_1 + \|b\|_\infty}, \tag{2.4.2}$$

and we stop iterations after $\omega \leq c$, where c is the threshold value that determines the termination. A comparison between the normwise backward error and other stopping criteria is presented by (Arioli, Duff, and Ruiz (1992)), and the authors have empirically shown that in some cases it is possible to drive ω to near the machine precision. Furthermore, the authors consider ω as a useful stopping criterion for iterative methods because the spectral properties of the matrices are implicitly considered in ω. ω uses the norms of the matrices to estimate the

Phase	Classical CG Algorithm 2.1.1		Block-CG Algorithm 2.2.2	
	Step	FLOP count	Step	FLOP count
Startup	(1)	$\theta(A) + n$	(1)	$(s \cdot \theta(A)) + (n \cdot s)$
			(2)	$3(n \cdot s^2) + \frac{s^3}{3} + \frac{s^2}{2} + \frac{s}{6}$
			(3)	$(s \cdot \theta(A)) + 4(n \cdot s^2) + \frac{s^3}{3} + \frac{s^2}{2} + \frac{s}{6}$
	Total:	$n + \theta(A)$	Total:	$2(s \cdot \theta(A)) + 7(n \cdot s^2) + (n \cdot s) + \frac{2s^3}{3} + s^2 + \frac{s}{3}$
Iteration cycle	(3.1)	$2n$	(4.2)	$2(n \cdot s^2) + 2s^3$
	(3.2)	$\theta(A)$	(4.3)	$4(n \cdot s^2) + 2(n \cdot s) + \frac{s^3}{3} + \frac{s^2}{2} + \frac{s}{6}$
	(3.3)	$2n$	(4.4)	s^3
	(3.4)	$2n$	(4.5)	$(s \cdot \theta(A)) + 6(n \cdot s^2) + 2(n \cdot s) + \frac{s^3}{3} + \frac{s^2}{2} + \frac{s}{6}$
	(3.5)	1		
	(3.6)	$2n$		
	(3.7)	$2n$		
	(3.8.1)	$2n$		
	(3.9)	1		
	Total:	$\theta(A) + 10n + 2$	Total:	$(s \cdot \theta(A)) + 12(n \cdot s^2) + 4(n \cdot s) + \frac{11s^3}{3} + s^2 + \frac{s}{3}$

Table 2.1: Complexity of the Classical CG and the Block-CG algorithms. The FLOP count in the Block-CG step (4.2) comes from (2.2.3).

error. Consequently, ω, as many iterative methods, is sensitive to scalings of the matrices.

In many applications, people use a different error estimation that uses less information than the normwise backward error. For instance, computing an error estimate using the residual norms. In this case one sets

$$\mathrm{E} = 0 \quad \text{and} \quad \mathrm{f} = \|r(x^{(0)})\|,$$

thus the backward error at each iteration is estimated with:

$$\omega_1 = \frac{\|r(x^{(j)})\|}{\|r(x^{(0)})\|}. \tag{2.4.3}$$

and the iteration process stops when $\omega_1 \leq c$. Quite often, one looks for only a few correct digits in the approximation to the solution of a linear system of equations. In these cases ω_1 is commonly used.
Furthermore, the threshold value c can be related to the truncation error that comes from the discretization of the original problem. Thus, the maximum number of correct digits in an approximation depends on the truncation error (or discretization error) from discretizing the original continuous problem and writing out the system of linear equations.
We also present in Chapter 3 some results from runs of the Classical CG and Block-CG using a stopping criterion based on ω_1.

Chapter 3

Sequential Block-CG experiments

The purpose of the following experiments is to examine the behavior of the Classical CG and the Block-CG algorithms described in the previous chapter and conclude with some incentives for the parallelization of the Block-CG Algorithm.
As illustrated in (O'Leary (1980)), Block-CG is expected to converge faster than Classical CG for linear systems of equations with clusters of eigenvalues that are separated. In addition, Block-CG can simultaneously solve a linear system of equations with multiple right-hand sides. Some advantages from using Level 3 BLAS routines inside the Block-CG iteration are shown in this chapter. Furthermore, the granularity of Block-CG is greater than the granularity of Classical CG and we anticipate to gain more from implementing the Block-CG algorithm in vector and parallel computer systems.
In the first three sections, we use two test problems that were previously used for testing a Lanczos algorithm with partial reorthogonalizations (Simon (1984)). This version of the Lanczos algorithm is named LANPRO from LANczos with Partial ReOrthogonalization. The matrices used in the tests of the LANPRO algorithm are available in the Harwell-Boeing sparse matrix collection (Duff, Grimes, and Lewis (1992)).
In the Harwell-Boeing collection these seven matrices are labeled NOS1 to NOS7. Here we used the problems LANPRO (NOS2) and LANPRO (NOS3). These problems are particularly interesting because Simon (1984) compares the LANPRO algorithm with Classical CG and presents results from the solution of the problem LANPRO (NOS2) in which Classical CG poorly converges after more than n iterations.
The LANPRO test problems come from finite element approximations to problems in structural engineering. LANPRO (NOS2) results from a discretization using a biharmonic operator on a beam. In this case, the beam has one end fixed and one end free (see Figure 3.1 (a)). The corresponding stiffness matrix was computed using a finite element approximation program named FEAP for Finite Element Analysis Program (see Zienkiewikz and Taylor (1989)).
The biharmonic equation of a plate flexure is given by

$$\frac{\partial^4 w}{\partial x^4} + \frac{\partial^4 w}{\partial x^2 y^2} + \frac{\partial^4 w}{\partial y^4} + q\frac{12\,(1-v^2)}{Et^3} = 0 \tag{3.0.1}$$

where w is the displacement. In this problem, w is the result of applying a unit load at about the middle of the beam and there is no displacement at the fixed end of the beam. Thus, the boundary conditions at the fixed end are

$$\frac{\partial w}{\partial x} = \frac{\partial w}{\partial y} = w = 0. \tag{3.0.2}$$

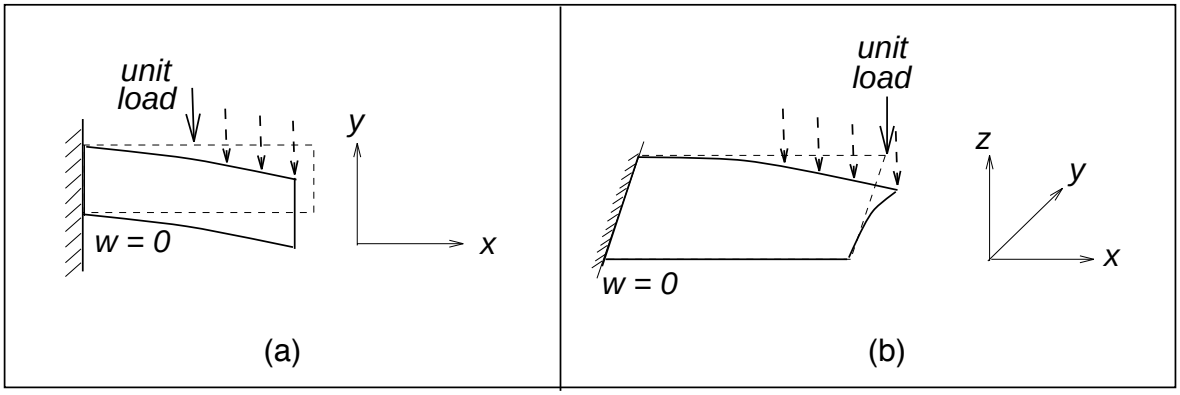

Figure 3.1: (a) A beam with one end fixed and the other end free. A unit load at about the middle of the beam. The stiffness matrix corresponds to the LANPRO (NOS2) problem. (b) A plate with one end fixed and the other free. A unit load is applied to one of the free corners of the plate. The stiffness matrix corresponds to the LANPRO (NOS3) problem.

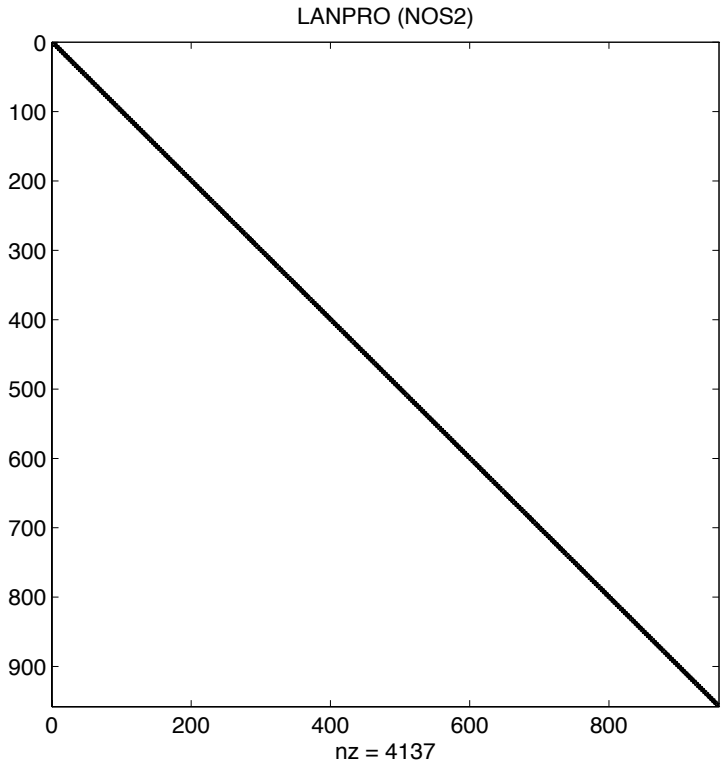

Figure 3.2: Sparsity pattern of LANPRO (NOS2) matrix from the Harwell-Boeing Sparse Matrix Collection.

In the LANPRO (NOS2) problem, 240 elements were used in the finite element approximation with 3 degrees of freedom in each element. Figures 3.2 illustrates the sparsity pattern of the test problem. The stiffness matrix is SPD of order 957 with 2547 nonzero elements. Figure 3.3 illustrates the eigenspectrum of this matrix.

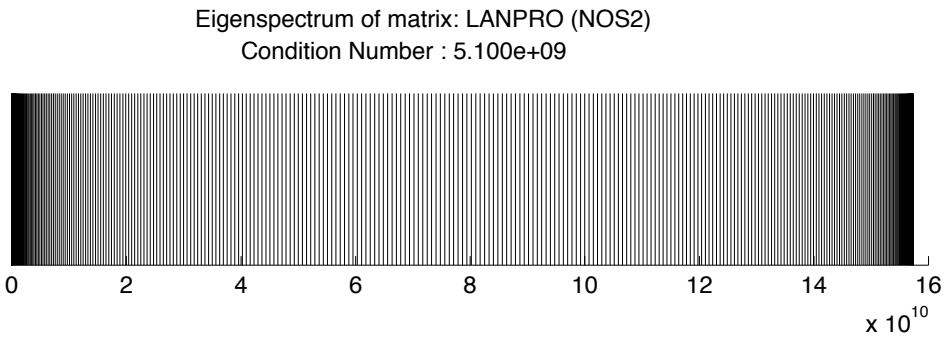

Figure 3.3: Eigenspectrum of LANPRO (NOS2) matrix.

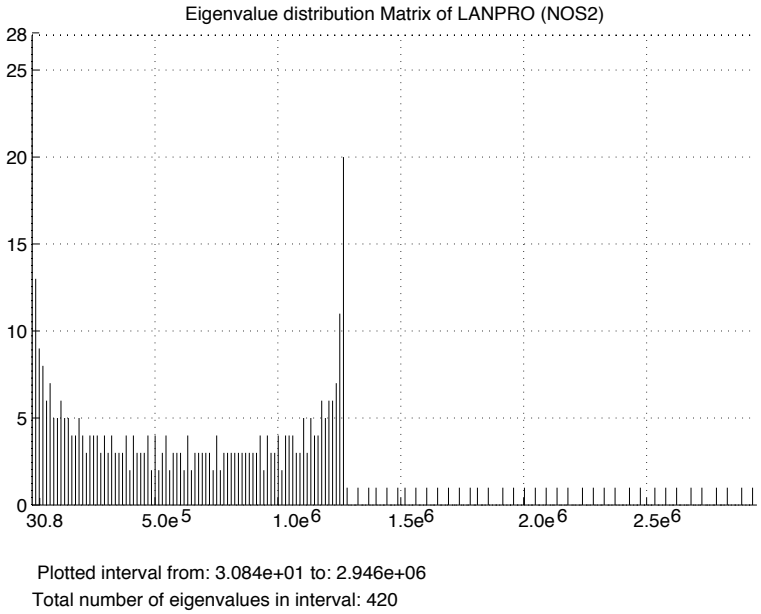

Figure 3.4: Eigenvalues at lower end of the eigenspectrum. Notice that there are 28 eigenvalues around 30.84

LANPRO (NOS3) comes from applying a biharmonic operator on a rectangular plate. In this problem the plate has one side fixed and the others free (see Figure 3.1 (b)). The problem is discretized using the biharmonic operator (3.0.1) with the boundary conditions (3.0.2) at the

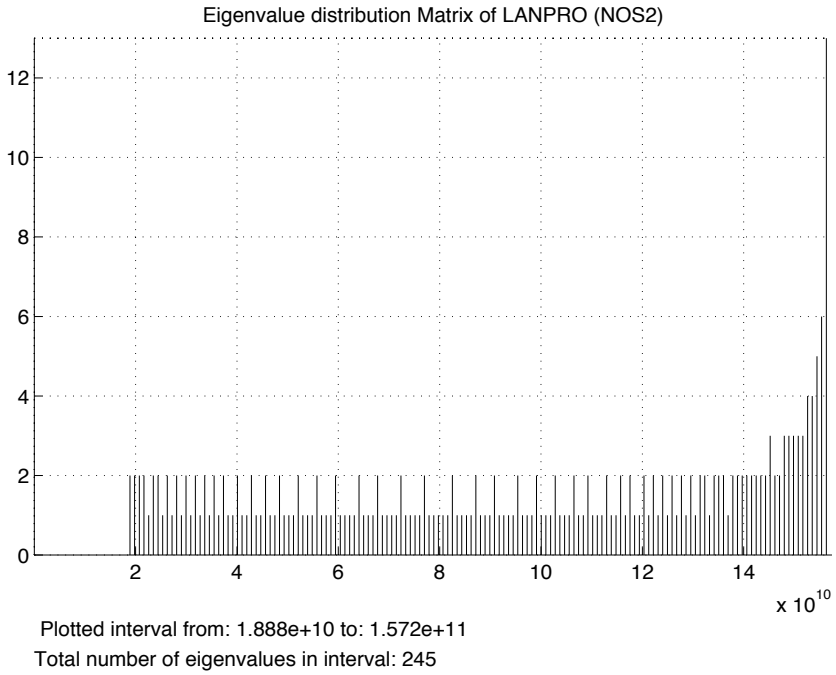

Figure 3.5: Eigenvalues at the center of the eigenspectrum.

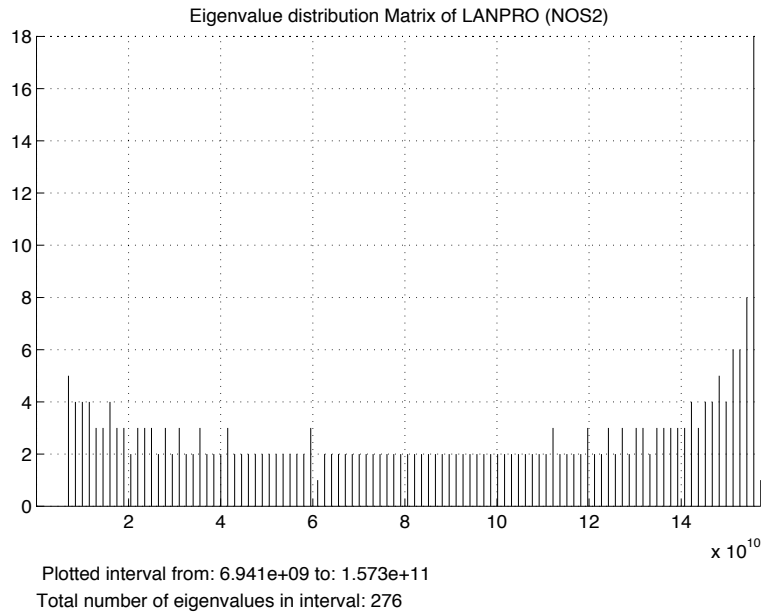

Figure 3.6: Eigenvalues at upper end of the eigenspectrum.

fixed end.

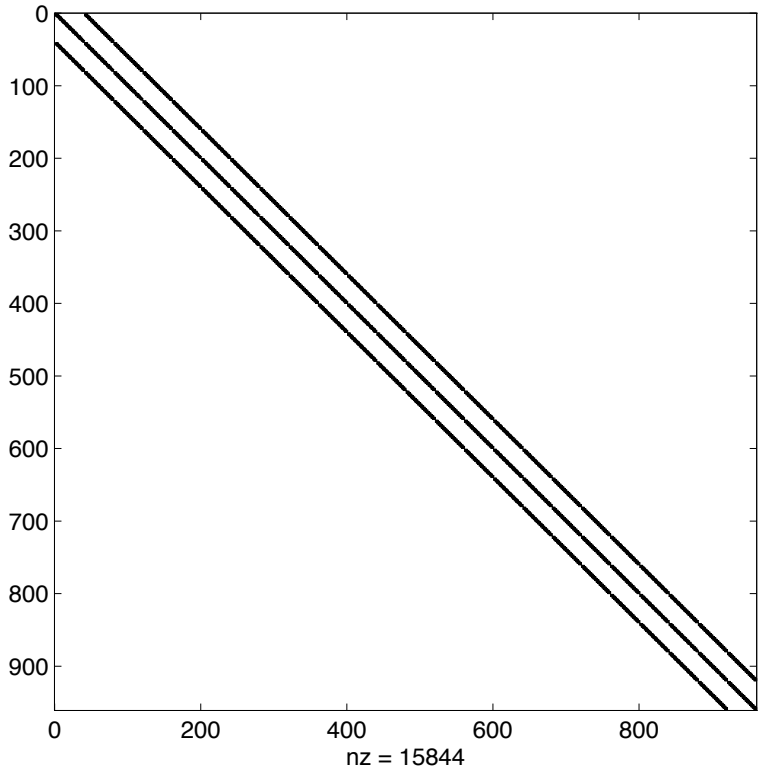

Figure 3.7: Sparsity pattern of LANPRO (NOS3).

The stiffness matrix was also computed using FEAP. The stiffness matrix is SPD of order 960 with 8402 nonzero elements. The sparsity pattern of this matrix is shown in Figure 3.7, and Figure 3.8 illustrates the eigenspectrum of the LANPRO (NOS3) matrix.
As shown in Figure 3.3, the test problem LANPRO (NOS2) has two large clusters of eigenvalues to the extremes of the spectrum, and a large condition number of 5.1×10^9. As can be seen in Figure 3.8, the LANPRO (NOS3) problem has several clusters of eigenvalues and these clusters are not as separated as in LANPRO (NOS2). Also the condition number of LANPRO (NOS3), 3.772×10^4, is relatively small compared to the one of the LANPRO (NOS2).

In Section 3.4, we use a problem arising from oil reservoir modeling. This problem is included in the Harwell-Boeing sparse matrix collection under the name of SHERMAN4. The problem comes from a three dimensional simulation model on a $16 \times 23 \times 3$ grid. The problem has been discretized using a seven-point finite-difference approximation with one equation and one unknown at each grid point.
The results in this chapter are from sequential runs of implementations of the Classical CG Algorithm 2.1.1 and the Block-CG Algorithm 2.2.2. Results from parallel runs of both algorithms will be presented in the next chapter. The runs reported in this chapter were performed on an IBM RS6000 system 550 with theoretical peak performance of 83.2 Mflops. The Mflops term is

the computational rate measured in millions of floating-point operations computed in a second and is different from a FLOP count that represents the number of floating-point operations.

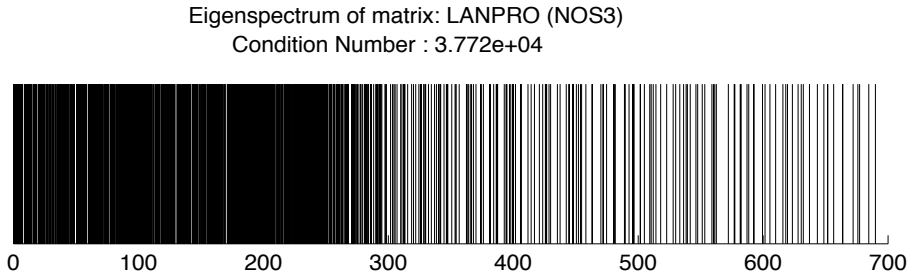

Figure 3.8: Eigenspectrum of LANPRO (NOS3) matrix.

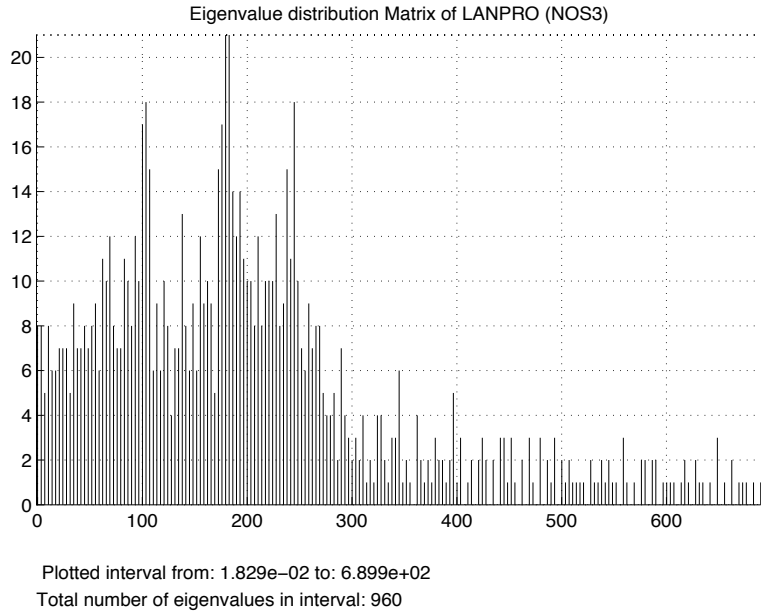

Figure 3.9: Distribution of Eigenvalues for LANPRO (NOS3).

3.1 Solving the LANPRO (NOS2) problem

Block size	Iteration count	Equivalent iterations	Normwise Backward error (ω)
1	16470	16470	0.22×10^{-8}
8	2058	16464	0.99×10^{-16}
16	726	11616	0.76×10^{-16}
32	305	9760	0.93×10^{-16}
64	100	6400	0.34×10^{-16}

Table 3.1: Sequential runs of CG and Block-CG programs. Stop iterations after $\omega \leq 1.0 \times 10^{-16}$. LANPRO (NOS2) problem.

In this first experiment, we use the LANPRO (NOS2) problem, and we randomly choose the block sizes of 8, 16, 32 and 64, to examine some effects of the block size on the performance of Block-CG. In this case, we stop iterations when $\omega \leq 1.0 \times 10^{-16}$ (ω is the normwise backward error from (2.4.2)).
Figure 3.10 shows the convergence curves of Classical CG, labeled Block-CG(1), and four instances of Block-CG. In this figure, the number of iterations represents the iteration count of the Classical CG and the Block-CG algorithms.
In Figure 3.10, only 2060 iterations of Classical CG have been reported although in this experiment Classical CG cannot reduce ω below 1.22×10^{-8} even after 16470 iterations. In the same figure, the iteration count for reducing ω to 1.0×10^{-16} decreases as the block size is increased. On the other hand, Block-CG computes s orthogonal vectors at each iteration, while Classical CG computes only one. Therefore, the iteration count has to be multiplied by the block size to reflect the number of equivalent or normalized iterations.
Figure 3.11 shows the convergence curves using equivalent iterations. The results from this experiment are summarized in Table 3.1. Table 3.2 summarizes the results from the similar runs as in Table 3.1 after computing only n orthogonal vectors.

The solution of a linear system is in the eigenspace of the matrix A and any iterative method that successively multiplies the matrix A times a vector, or another matrix as in Block-CG, will include information from eigenvalues at the upper end of the eigenspectrum into the solution during the first iterations. Information from eigenvalues at the lower end of the eigenspectrum is likely to be included only during the last iterations of the method.
In finite arithmetic, information from the smallest eigenvalues is difficult to include in the solution when the matrices have a high condition number and there are clusters of eigenvalues at the lower end of the spectrum. For instance, two very small eigenvalues will converge to the same value after a few multiplications of the matrix A in finite arithmetic. Similar convergence behavior has been observed in our experiments solving the LANPRO (NOS2), and this explains the slow convergence rate of both Classical CG and Block-CG.

Block size	Iteration count	Equivalent iterations	Normwise Backward error (ω)	Execution Time (secs)	Mflops
1	957	957	0.45×10^{-8}	8.39	2.8
8	120	960	0.68×10^{-7}	8.92	11.4
16	60	960	0.62×10^{-7}	8.98	21.5
32	30	960	0.15×10^{-6}	15.99	23.2
64	15	960	0.27×10^{-6}	27.14	28.1

Table 3.2: Sequential runs of Classical CG and Block-CG programs with a fixed number of equivalent iterations. LANPRO (NOS2) problem.

In Classical CG, the computational weight is in the $Ap^{(j)}$ product, thus $\theta(A)$ has a great influence in the total execution time. As shown in Tables 3.2 and 3.4, the execution time of Block-CG with block sizes 32 and 64 is significantly greater than the execution time of Block-CG with smaller block sizes. This effect means that FLOP count in Block-CG can be greatly influenced by large values of s.
The size of the matrix A, and its sparsity pattern are constant through the Block-CG iterations, thus the ratio of $\theta(A) \times s$ over the total FLOP count decreases as the block size is increased. Clearly, increasing the block size will cause the FLOP count from operations that involve the $s \times s$ matrices to increase. For instance, in the Block-CG Algorithm 2.2.2 the factorization of the $\gamma^{(j)}$ and $\beta^{(j)}$ matrices becomes more expensive as s is increased and this can be easily verified in (2.3.2). Table 3.3 shows the influence of $\theta(A) \times s$ on the total FLOP count for the runs shown in Table 3.2.

$\theta(A) = 8274$					
Block size	1	8	16	32	64
% Total of FLOP count	33.7	7.9	4.2	2.2	1.1

Table 3.3: Percentage of Total FLOP count generated by $\theta(A)$. LANPRO (NOS2) problem.

The results shown in Tables 3.1 and 3.2 are verified using a different stopping criterion based on (2.4.3), stopping iterations when $\omega_1 \leq 10 \times 10^{-5}$. The results are shown in Table 3.4, and

as shown in Simon (1984), the convergence rate of Classical CG solving the LANPRO (NOS2) problem is very slow. In Table 3.4, the behavior of Block-CG is the similar to the one observed with a different stopping criterion (Tables 3.1 and 3.2).

Stopping criterion: $\omega_1 \leq 1.0 \times 10^{-5}$			
Block size	Iteration count	Equivalent iterations	Execution Time (secs)
1	323	323	1.32
8	122	976	4.16
16	124	1984	10.68
32	67	2144	23.74
64	40	2560	41.56

Table 3.4: Sequential runs of Classical CG and Block-CG programs. Using $\omega_1 \leq 1.0 \times 10^{-5}$ as stopping criterion. LANPRO (NOS2) problem.

Empirically, it has been observed that Block-CG has better chances to include information from the smallest eigenvalues into the solution with a block size close to the size of the cluster of eigenvalues at the lower end of the spectrum. One reason is that more information from spectrum is gathered at each iteration. In infinite precision, this implies that fewer matrix multiplications are performed to obtain the relevant information from the entire spectrum and include it in the solution. And in finite arithmetic, the reduction in the number of matrix multiplications will also reduce the perturbation in the solution due to roundoff errors from implicitly computing $A^{(J)}$.
Therefore, in some cases a large block size is recommended for Block-CG.

3.2 Solving again the LANPRO (NOS2) problem

Block size	Iteration count	Equivalent iterations	Normwise Backward error (ω)	Execution Time (secs)	Mflops
1	957	957	0.45×10^{-8}	8.39	2.8
196	3	588	0.75×10^{-9}	26.05	32.1

Table 3.5: Sequential runs of CG and Block-CG programs. LANPRO (NOS2) problem.

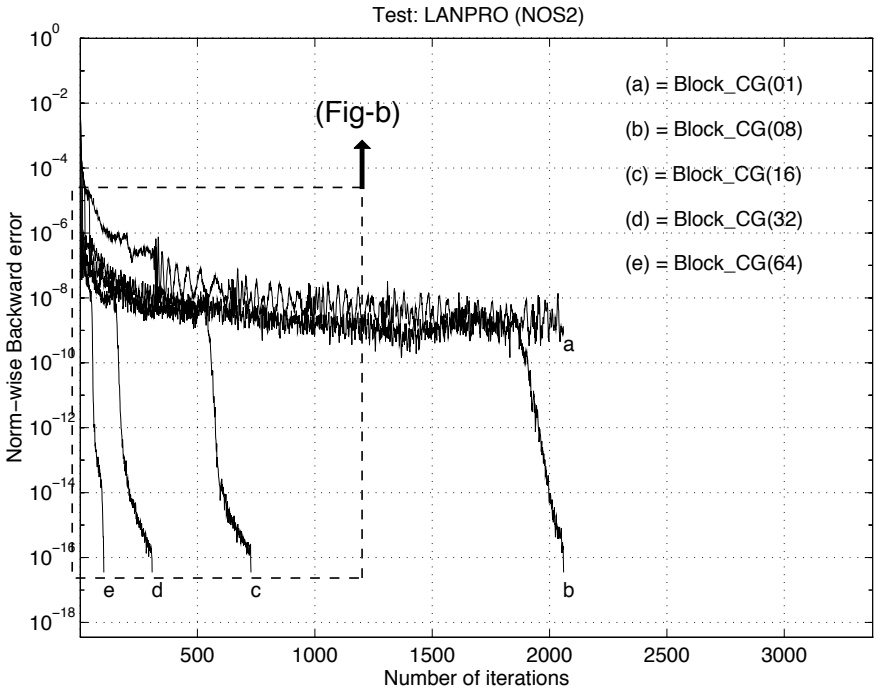

Fig-a

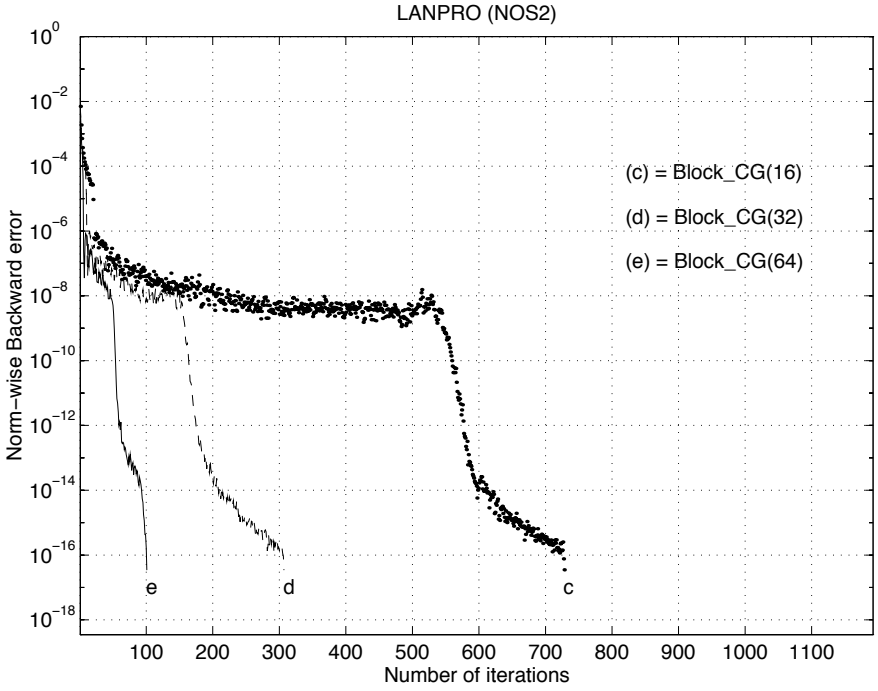

Fig-b

Figure 3.10: (a) Convergence curves of Block-CG with different block sizes. (b) Zoom of some convergences curves. Test problem: LANPRO (NOS2).

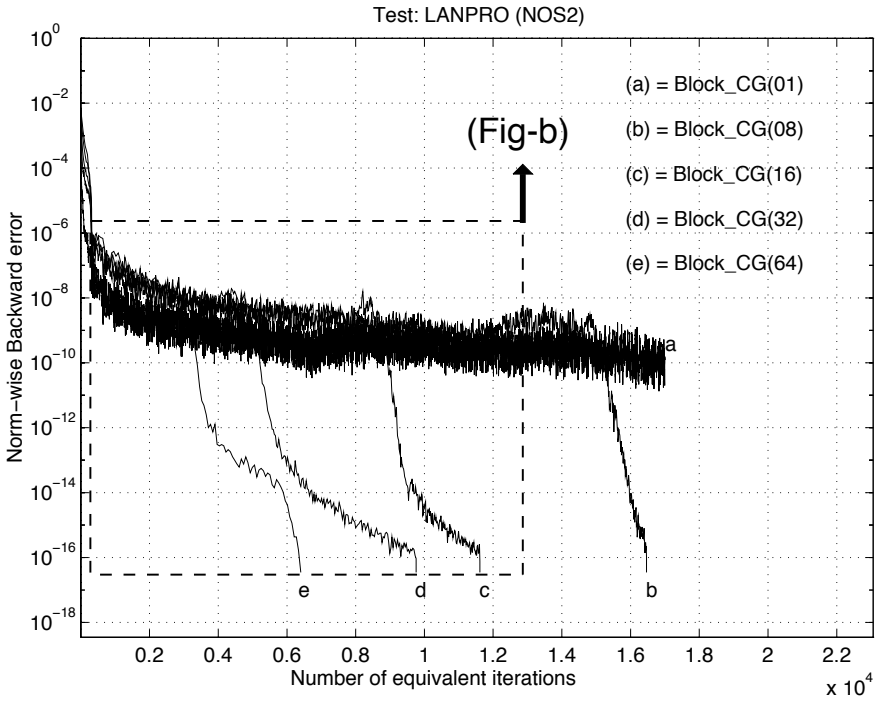

Fig-a

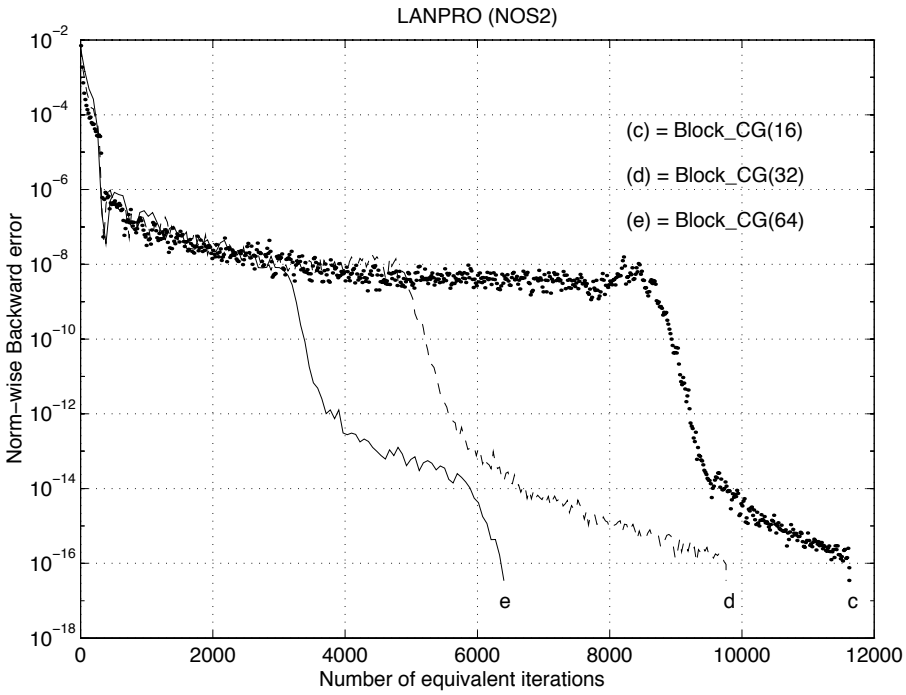

Fig-b

Figure 3.11: Convergence curves of Block-CG with different block sizes. In each curve, the number of iteration is multiplied by its block size to show equivalent iterations. (b) Zoom of some convergences curves. Test problem: LANPRO (NOS2).

Again, we use the LANPRO (NOS2) problem. Figures 3.4, 3.5, and 3.6 show the distribution of eigenvalues at the lower end, the center, and the upper end of the eigenspectrum respectively. In these figures, The eigenvalues are sorted in ascending order.
There are 201 eigenvalues in the cluster of eigenvalues at the lower end of the eigenspectrum of LANPRO (NOS2) (see Figure 3.3). In this cluster, the eigenvalues lie between 1.0×10^{-5} and 9.9×10^{-5}. After a few runs of Block-CG varying the block sizes around 201, we have chosen to report the results with block size of 196 because Block-CG with block sizes between 196 and 205 have exhibited the same behavior during the experiments, and 196 is the smallest value.
Table 3.5 shows the results from sequential runs of Classical CG and Block-CG with block size 196. Again, the execution time of Block-CG has drastically increased with a block size of 196 when compared to the execution time of Classical CG. This increase is due to Block-CG operations that involve 196×196 matrices. When solving the LANPRO (NOS2) using a block size of 196, the $\theta(A)$ parameter is only 0.3% of the total FLOP count.
In Table 3.6, the stopping criterion is based on (2.4.3), and iterations are stopped after reducing the residual norm below 1.0×10^{-5}. In this case, Classical CG is stopped earlier than in the experiment reported in Table 3.5.

Stopping criterion: $\omega_1 \leq 1.0 \times 10^{-5}$			
Block size	Iteration count	Equivalent iterations	Execution Time (secs)
1	323	323	1.32
196	3	588	26.05

Table 3.6: Sequential runs of Classical CG and Block-CG with block size 196. Using $\omega_1 \leq 1.0 \times 10^{-5}$ as stopping criterion. LANPRO (NOS2) problem.

In the computational results in Sections 3.1 and 3.2, it is observed that the Mflop rate increases as the block size is increased. This effect is due to the use of Level 3 BLAS routines in Block-CG instead of the Level 2 BLAS routines that are used in Classical CG. Mainly, the use of Level 3 BLAS routines improves the ratio of FLOPs per memory reference.

3.3 Solving the LANPRO (NOS3) problem

In this experiment, we use the test problem LANPRO (NOS3) and we choose the four block sizes 4, 8, 16, and 32 for Block-CG. The clusters of eigenvalues in this test problem are not very distant one from the other as shown in Figure 3.8. The condition number is smaller than the one from LANPRO (NOS2). Therefore, all the eigenvalues are easily approximated even when using small block sizes.
The convergence curves for this experiment are shown in Figures 3.12 and 3.13. Figure 3.12 plots the convergence curves using ω, the normwise backward error, versus the iteration count, and in Figure 3.13 the convergence curves were plotted using ω versus the equivalent itera-

Block size	Iteration count	Equivalent iterations	Normwise Backward error (ω)	Execution Time (secs)	Mflops
1	306	306	0.91×10^{-16}	1.91	7.8
4	137	548	0.89×10^{-16}	3.28	13.7
8	95	760	0.90×10^{-16}	5.27	18.9
16	62	992	0.53×10^{-16}	8.35	28.4
32	35	1120	0.28×10^{-16}	13.26	35.3

Table 3.7: Sequential runs of CG and Block-CG programs. LANPRO (NOS3) problem.

Block size	Iteration count	Equivalent iterations	Normwise Backward error (ω)	Execution Time (secs)	Mflops
1	306	306	0.91×10^{-16}	1.91	7.8
4	76	304	0.15×10^{-5}	1.89	13.3
8	38	304	0.62×10^{-5}	2.38	17.2
16	19	304	0.95×10^{-5}	2.64	27.5
32	10	320	0.16×10^{-4}	4.17	34.9

Table 3.8: Sequential runs of Classical CG and Block-CG programs with a fixed number of equivalent iterations. LANPRO (NOS3) problem.

tions. Considering the iteration count in Figure 3.12, Block-CG converges faster as the block size is increased. However in Figure 3.13, the computational work necessary to reduce ω below 1.0×10^{-16} increases as we increase the block size, and Classical CG converges faster, in terms of equivalent iterations, than all of the four Block-CG instances.
The size of LANPRO (NOS3) is 960 and Block-CG with block sizes of 16 and 32 compute more than 960 orthogonal vectors, thus Block-CG may have converged to a erroneous solution in both cases.
Table 3.7 summarizes the convergence information from Figures 3.12 and 3.13.
Classical CG takes 306 iterations to converge and in Table 3.8 the number of equivalent iteration has been fixed around 306 to compare the computational work between Classical CG and the different instances of Block-CG. At this point in the computations, Classical CG has performed better in approximating the solution to LANPRO (NOS3) than Block-CG instances.

Stopping criterion: $\omega_1 \leq 1.0 \times 10^{-5}$				
Block size	Iteration count	Equivalent iterations	Execution Time (secs)	Mflops
1	206	206	1.20	8.0
4	97	388	1.99	12.9
8	69	552	3.37	17.5
16	44	704	5.83	27.2
32	30	960	9.55	35.1

Table 3.9: Sequential runs of CG and Block-CG programs. Solving the LANPRO (NOS3) problem.

Table 3.9 summarizes the results from solving the LANPRO (NOS3) problem with the same block sizes. Comparing the results from Table 3.9 and Table 3.7, it is evident the advantage of using the stopping criterion based on (2.4.3), because even with block 32 there are only 960 orthogonal vectors computed before reaching the threshold value. Whereas using (2.4.2) as the stopping criterion, (see Table 3.7,) more equivalent iterations are performed for the block sizes of 16 and 32.

Table 3.10 shows the percentage of the total FLOP count generated by $\theta(A)$. Notice, that the percentages have increased compared to those shown in Table 3.3 for the LANPRO (NOS2) problem. One of the reasons is that the LANPRO (NOS3) stiffness matrix is more dense than the one of the LANPRO (NOS2) problem. As a result of increasing percentages, in Tables 3.7, 3.8 and 3.9, it can be seen that the differences between the execution times are smaller than the ones reported in the tables for the LANPRO (NOS2) problem after varying the block sizes.
Again in this experiment, the Level 3 BLAS effect improves the Mflop rate as the block size is also increased.

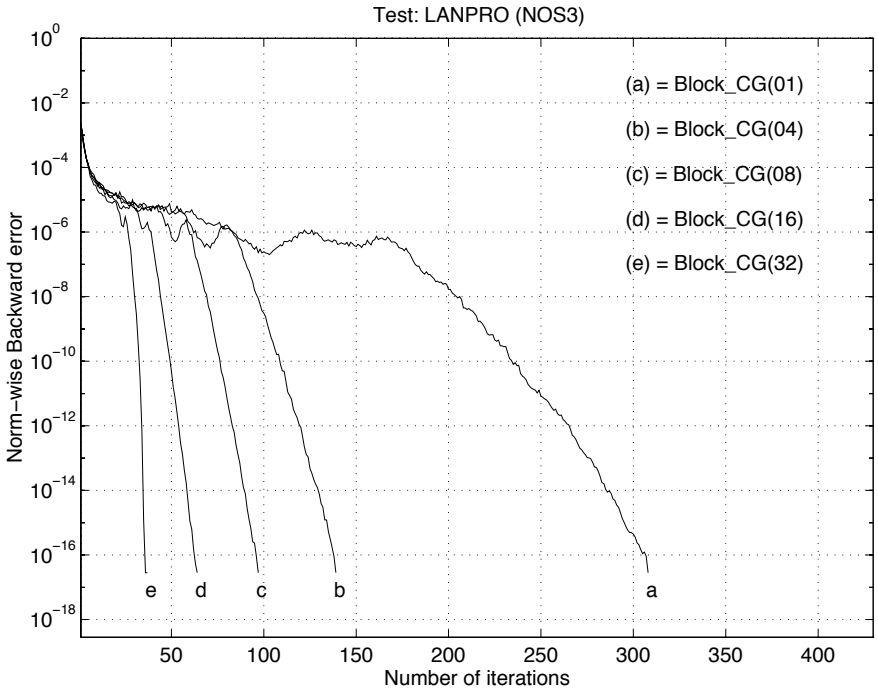

Figure 3.12: Convergence curves of Block-CG with different block sizes. Test problem: LANPRO (NOS3).

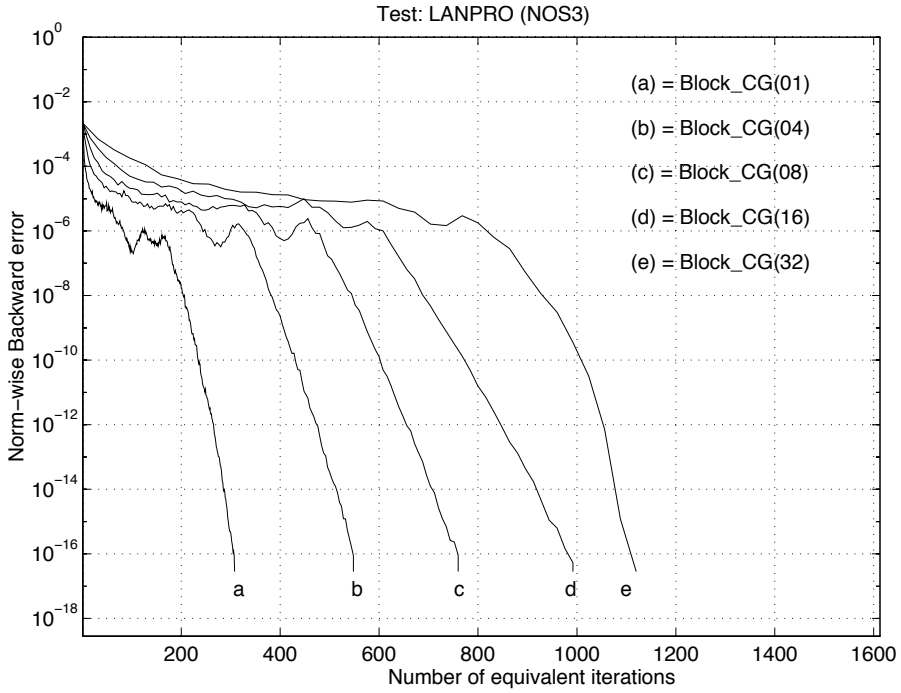

Figure 3.13: Convergence curves of Block-CG with different block sizes. In each curve, the number of iteration is multiplied by its block size to show equivalent iterations. Test problem: LANPRO (NOS3).

$\theta(A) = 31688$					
Block size	1	4	8	16	32
% Total of FLOP count	65.8	38.2	24.3	14.0	7.5

Table 3.10: Percentage of Total FLOP count generated by $\theta(A)$. LANPRO (NOS3) problem.

3.4 Solving the SHERMAN4 problem

As mentioned earlier, the SHERMAN4 matrix comes from an oil reservoir model. The SHERMAN4 matrix has only a symmetric pattern but is not SPD. In this experiments we solve instead A^TA, where A is the SHERMAN4 matrix, because A^TA is SPD. The clusters of eigenvalues in A^TA are more separated than in the original SHERMAN4 matrix. The spectrum of the original SHERMAN4 matrix is shown in Figure 3.15 and the eigenspectrum of A^TA from the SHERMAN4 matrix is shown in Figure 3.16. Given distribution of eigenvalues in A^TA, we anticipate that Block-CG will convergence faster than Classical CG.
The original SHERMAN4 problem will be solved in Chapter 11 using a block row projection method that is accelerated with Block-CG.
The sparsity pattern of A^TA from SHERMAN4 is shown in Figure 3.14. The matrix has 10346 nonzero entries and a condition number of 4.74×10^6.
The block sizes 1, 4, 8, 16, 32, and 64 were chosen for this experiment. In the results presented in Table 3.11, the stopping criterion was $\omega_1 \leq 1.0e-5$, where ω_1 is the residual norm from (2.4.3).

In Table 3.11, it can be seen that for this case Block-CG performs better than Classical CG. The table shows that Block-CG with block sizes of 4 and 8 have computed with almost the same accuracy the solution in fewer iterations than Classical CG. With the block sizes of 16 and 32 a more accurate solution is computed in fewer equivalent iterations.
Although, in this case, the execution of Block-CG with block sizes 16, 32, and 64 took more time than Classical CG, it is expected that Block-CG will perform better in parallel environments because of the Mflop rates shown in Table 3.14.

Now, we repeat the same experiment using $w \leq 1.0 \times 10^{-12}$, normalized backward error, as the stopping criterion. The threshold value has been chosen using information that was obtained during the runs from the first experiments reported in Table 3.11.
Figures 3.17 and 3.18 show convergence curves using the iteration count and equivalent iterations, respectively. Tables 3.12 and 3.13 summarize the results from the convergence curves. Table 3.12 supports the results from Table 3.11, and again Block-CG runs faster than Classical CG for the block sizes of 4 and 8. In addition, Table 3.13 shows that the block sizes of 16 and 32 have computed a solution with the requested accuracy after 864 equivalent iterations while Classical CG has not.

Block size	Iteration count	Equivalent iterations	Normwise Backward error (ω_1)	Execution Time (secs)
1	977	977	0.88×10^{-5}	5.33
4	237	948	0.91×10^{-5}	4.52
8	109	872	0.69×10^{-5}	5.01
16	50	800	0.81×10^{-6}	5.81
32	24	768	0.80×10^{-6}	7.91
64	12	768	0.16×10^{-5}	12.82

Table 3.11: Sequential runs of CG and Block-CG programs. SHERMAN4 problem.

Block size	Iteration count	Equivalent iterations	Normwise Backward error (ω)	Execution Time (secs)
1	944	944	0.77×10^{-12}	5.33
4	254	1016	0.42×10^{-12}	5.01
8	114	912	0.52×10^{-12}	5.30
16	54	864	0.79×10^{-12}	6.24
32	27	864	0.68×10^{-12}	8.92
64	16	1024	0.77×10^{-12}	16.82

Table 3.12: Sequential runs of CG and Block-CG programs. SHERMAN4 problem.

Block size	Iteration count	Equivalent iterations	Normwise Backward error (ω)	Execution Time (secs)	Mflops
1	864	864	0.29×10^{-8}	4.94	7.7
4	216	864	0.11×10^{-6}	4.21	16.5
8	108	864	0.56×10^{-9}	5.11	23.4
16	54	864	0.11×10^{-12}	6.39	33.5
32	27	864	0.19×10^{-12}	9.41	46.5
64	14	896	0.21×10^{-10}	15.3	56.5

Table 3.13: Sequential runs of CG and Block-CG programs with fixed number of iterations. SHERMAN4 problem.

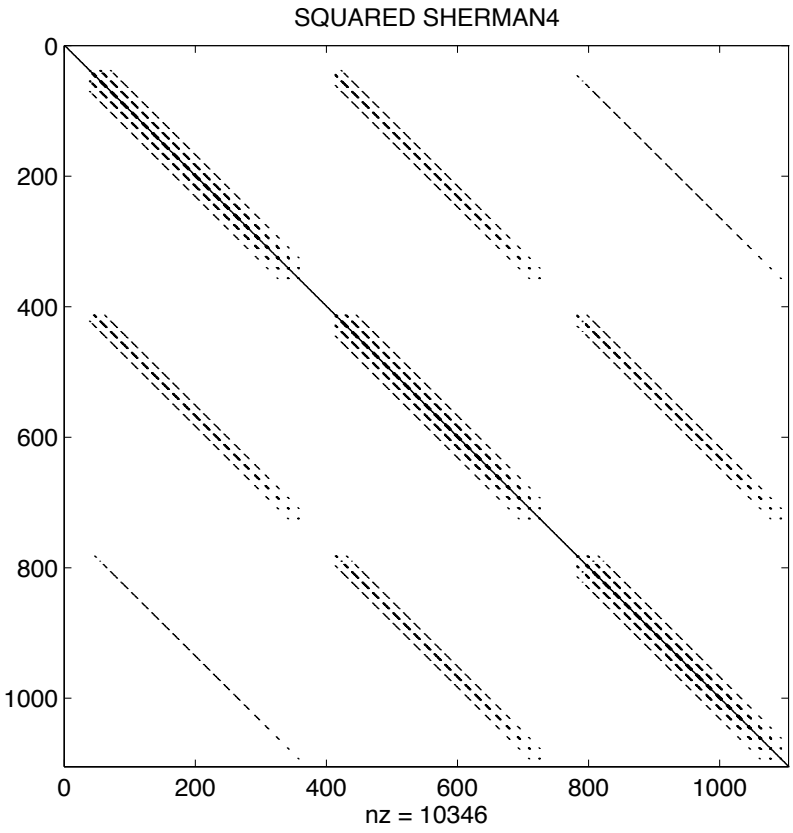

Figure 3.14: Sparsity pattern of $A^T A$ SHERMAN4 matrix.

$\theta(A) = 20692$						
Block size	1	4	8	16	32	64
% Total of FLOP count	55.4	26.4	15.7	8.7	4.5	2.3

Table 3.14: Percentage of Total FLOP count generated by $\theta(A)$. SHERMAN4 problem.

As shown in Table 3.14, the percentage of the total FLOP count generated by $\theta(A)$ decreases as the block size is increased. Furthermore, these percentages are significantly decreased for the block sizes of 16, 32, and 64, and it implies that in these cases the total FLOP count, and execution time are dominated by the block size.
As in the previous experiments, the Level 3 BLAS effect increases the Mflop rate as the block size is increased.

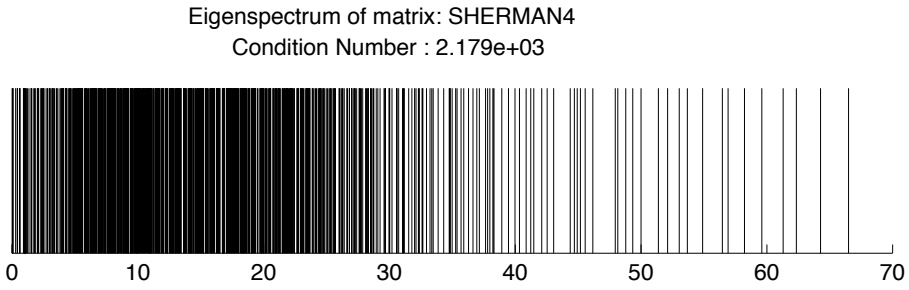

Figure 3.15: Eigenspectrum of SHERMAN4 matrix.

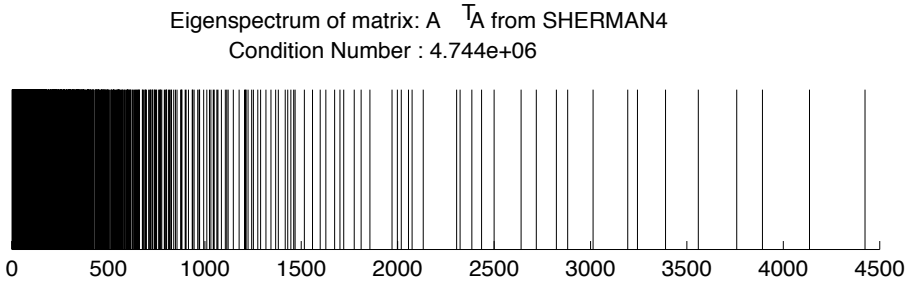

Figure 3.16: Eigenspectrum of $A^T A$ from SHERMAN4 matrix.

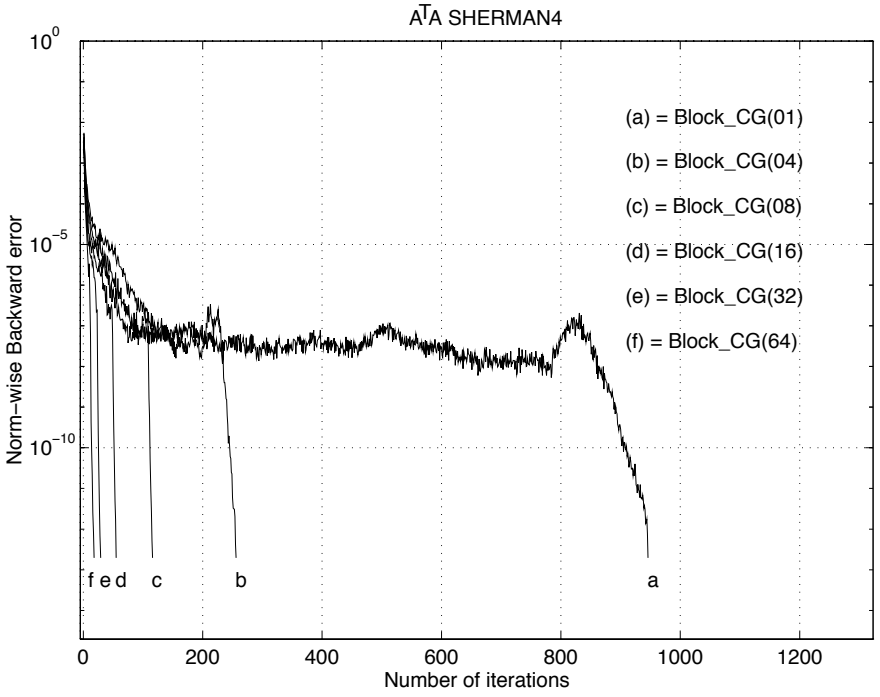

Figure 3.17: Convergence curves of Block-CG with different block sizes.

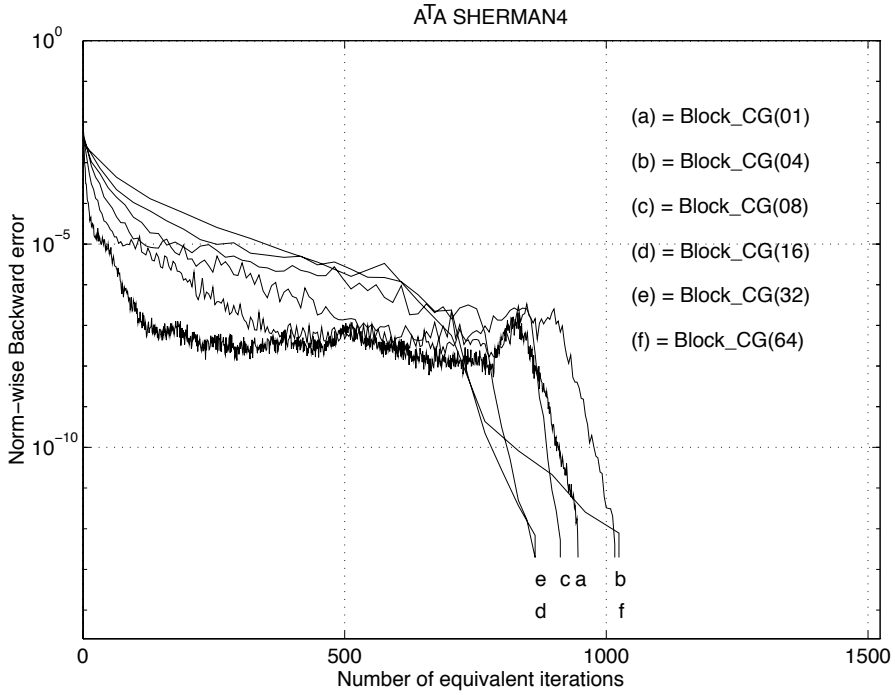

Figure 3.18: Convergence curves of Block-CG with different block sizes. In each curve, the number of iteration is multiplied by its block size to show equivalent iterations.

3.5 Remarks

	LANPRO (NOS2)			LANPRO (NOS3)			SHERMAN4 (A^TA)		
	Exec. Time			Exec. Time			Exec. Time		
block size	Classc CG	Block CG	Perf ratio	Classc CG	Block CG	Perf ratio	Classc CG	Block CG	Perf ratio
4	–	–	–	4.84	1.99	2.4	21.32	4.47	4.8
8	10.56	4.16	2.5	9.68	3.37	2.9	42.64	5.01	8.5
16	21.12	10.68	2.0	19.36	5.83	3.3	85.25	5.81	14.7
32	42.24	23.74	1.8	38.78	9.55	4.1	170.56	7.91	21.6
64	84.48	41.56	2.0	–	–	–	341.12	12.82	26.6
196	258.72	26.05	9.9	–	–	–	–	–	–

Table 3.15: Summary of results from previous experiments. Comparison between Block-CG and solving the linear system s times with Classical CG. The performance ratio is the execution time of Classical CG divided by Block-CG. Execution time is in seconds. For each problem, only a selected number of block sizes were reported in the previous sections of this chapter. Therefore, there are some empty entries in the table.

In some cases, it is not trivial to choose a suitable block size for the Block-CG algorithm without additional information from the problem to be solved. For instance, a linear system of equations with a high condition number may require a prior analysis of the eigenspectrum, and computing the eigenspectrum is more expensive than solving the linear system of equations.
From Block-CG results presented in this chapter, the Mflop rate increases as the block size is increased. The Mflop rates reported in Tables 3.2, 3.5, 3.7, 3.8, 3.9, and 3.13 show that as the granularity is increased the Mflop rate is increased. Furthermore, the increasing Mflop rate in Block-CG can produce computational gains on vector and parallel computers even for linear systems in which Classical CG converges a few iterations faster than Block-CG.
In a parallel distributed environment, the number of synchronization points of Classical CG are implicitly reduced in Block-CG. In Block-CG the information that needs to be communicated in one iteration equals the information that need to be communicated in s iterations of Classical CG. Thus, the number of synchronization points while computing the orthogonal vectors is reduced as the block size of Block-CG is increased. In this case, Block-CG is preferred over Classical CG even if Block-CG communicates longer messages than Classical CG. Indeed, the possible bottlenecks in the communication are associated with the volume of messages being exchanged and the expensive start-up or latency time.
When solving a linear system of equations with multiple right-hand sides, Block-CG will perform

better than Classical CG since the s solutions are computed simultaneously.
Table 3.15 compares the performance of Block-CG and Classical CG for solving a linear system of equations with s right-hand sides. The information in Table 3.15 has been extracted from results presented in previous sections. The execution times of Classical CG presented in previous tables have been multiplied by the block size to estimate the execution time of Classical CG for solving the linear system of equations s times.
In all of the results presented in this chapter, using Block-CG has been faster than s applications of Classical CG. On the other hand, this means that in none of the cases has Block-CG cost more than s times Block-CG although at each Block-CG iteration the FLOP count is increased by a factor of s when compared with a Classical CG iteration.
Increasing the block size in Block-CG will also increase the granularity of problem to be solved and this effect makes Block-CG more suitable for parallelization. In Chapter 4 we study the parallelization of the Block-CG Algorithm 2.2.2 for distributed computing environments. And in Chapter **??**, we present results from parallel experiments that support our expectation that Block-CG is suitable for parallel environments.

Chapter 4

Parallel Block-CG implementation

In this chapter, we present three different parallel versions of the Block-CG Algorithm 2.2.2 for distributed memory architectures (see also Drummond, Duff, and Ruiz (1993)). First, we present partitioning and scheduling strategies used in the three implementations. Then we discuss and compare the three parallel distributed implementations.

4.1 Partitioning and scheduling strategies

At this point, we look for a problem partitioning strategy to manage the creation of subproblems that can be solved in parallel, and a strategy to distribute these tasks among the processing elements. Here we recall that the problem in hand is to find a solution X to the linear system of equations:

$$HX = K \tag{4.1.1}$$

where H is a $n \times n$ symmetric positive definite matrix (SPD), X and K are $n \times s$ matrices, and s is the number of solution vectors in the system or block size for Block-CG.
The stabilized Block-CG, Algorithm 2.2.2, introduced in Chapter 2 will be used in the solution of (4.1.1).

4.1.1 Partitioning strategy

First, a column partition of the matrix H into l submatrices is performed ($l = 4$ in Figure 4.1). A column partition has been preferred over a row partition for the following reasons:

- There is no need to send the full X and K matrices to every computing element (CE) during the task distribution. Rather, each CE receives one or more parts of these matrices (e.g., in Figure 4.1, $X_1, X_2, X_3, X_4, K_1, K_2, K_3$, and K_4).
- The number of messages that need to be sent is independent of a column or row partition. However, When computing the $HX^{(0)}$ or the $\overline{HP}^{(J)}$ products in parallel, shorter messages are sent with a column partitioning strategy than with a row one. In the latter, it may be required to send the X and P matrices in full to perform the parallel products.
- In the FORTRAN language, it is easier and faster to access sequential elements in a matrix column-wise than row-wise.

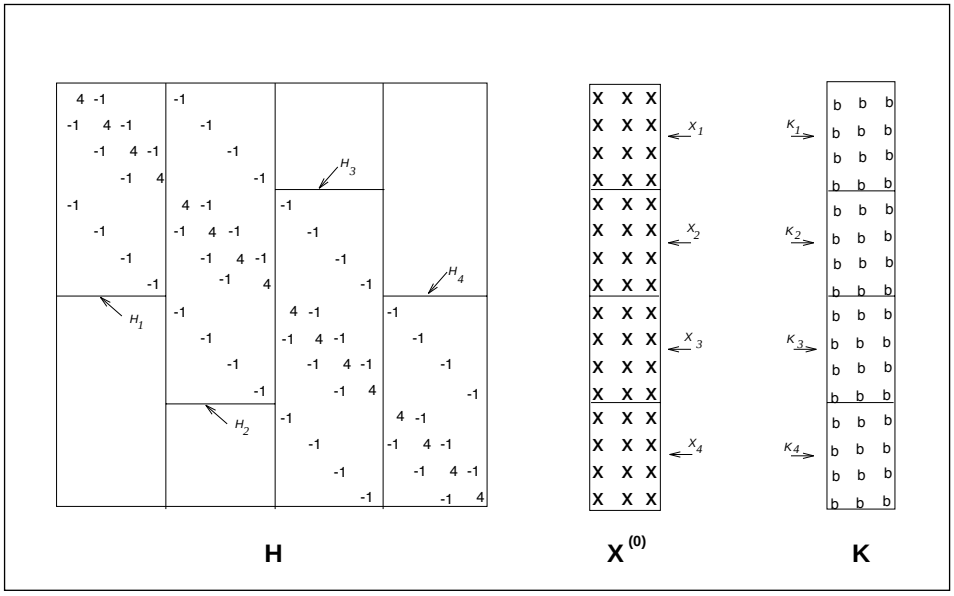

Figure 4.1: Column-partition strategy of the linear system of equations H X = K (initial definition of tasks).

In the partitioning strategy, the parameter k is defined as the number of columns to be included in every submatrix. If k does not divide n exactly the last submatrix will have less than k columns. These submatrices are labeled $H_1, H_2, .., H_l$.
A second step is to identify row boundaries for every submatrix H_i. The row boundaries for a submatrix H_i are defined by the first and last rows in H_i with nonzero entries. As a result, every submatrix H_i can be considered as a rectangular matrix of dimension $n_i \times k_i$, where n_i is the number of rows between the row boundaries for H_i, and k_i is equal to k except that probably k_l may be smaller. The partition of the X and K matrices is performed after completing the partition of the H matrix.
Partitioning the X and K matrices result in submatrices $X_1, X_2, \ldots, X_l$ and $K_1, K_2, \ldots, K_l$ respectively. The X_i submatrices need to be product compatible with the H_i submatrices. The matrix K is partitioned into submatrices of k-rows each (again, except the last submatrix that may have less than k rows.) Figure 4.1 shows a 4-column partition for a system of linear equations with a block size for Block-CG equal to three. For now on, we define a task as a triplet of matrices $< H_i, X_i, K_i >$. The resulting tasks from the partition will be performed in parallel under different programming models to be introduced in Section 4.2.
In parallel processing, a reduce operation gathers partial results from parallel computations and merges these partial results to obtain a global one. In Algorithm 2.2.2, the operations

$$R^{(0)} = K - HX^{(0)} \qquad \text{in step (1)}, \tag{4.1.1}$$

and

$$\overline{R}^{(j+1)} = \overline{R}^{(j)} - HP^{(j)} \qquad \text{in step (4.3)}, \tag{4.1.2}$$

involve the products $H_i X_i^{(0)}$ and $H_i P_i^{(j)}$ respectively. When these products are computed in parallel, one of the CEs performs a reduce operation to obtain the global result for the matrix-

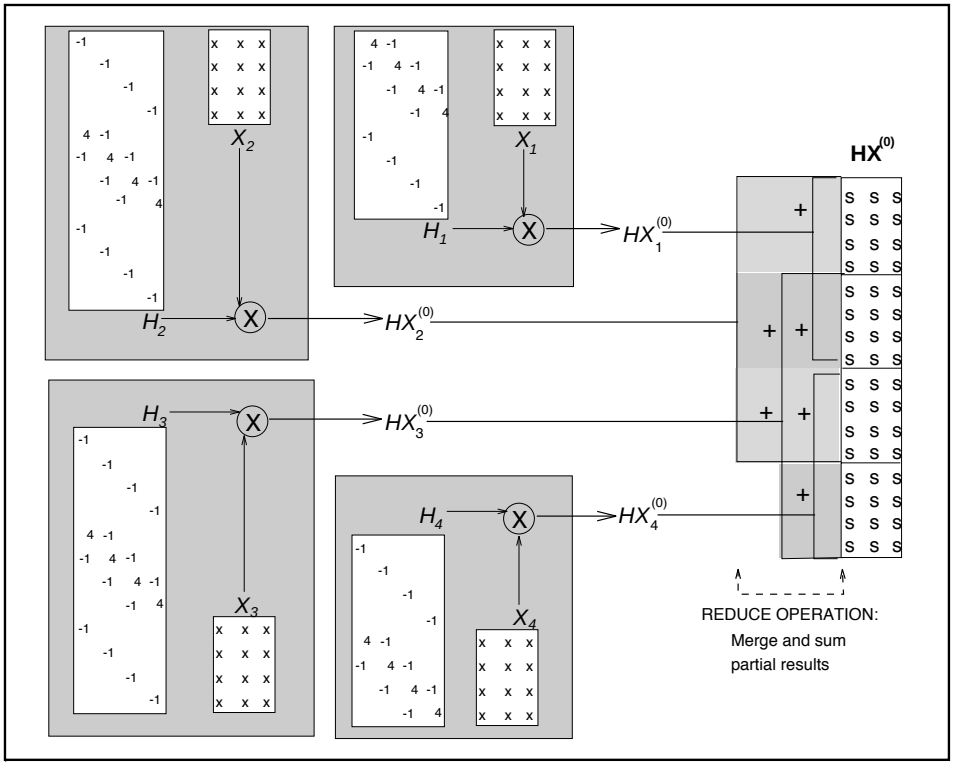

Figure 4.2: Example of a reduce operation. The $HX_i^{(0)}$ matrices are the partial results from the H_iX_i products. The global $HX^{(0)}$ is obtained after merging the partial results and adding the rows of the partial products that overlap.

matrix product. The partial results, $HX_i^{(0)}$ and $HP_i^{(j)}$, have dimensions $n_i \times s$ and if the H_i submatrices correspond to the blocks on the diagonal of the matrix H, then $\sum_{i=1}^{l} n_i = n$ and the global result from the product is obtained by simply merging the partial results.

On the other hand, if there are row overlaps between the H_i submatrices, then $\sum_{i=1}^{l} n_i \geq n$, and the global result from the product is obtained by merging the partial results and adding the rows from the partial results that overlap. Figure 4.2 shows an example of row overlaps between the submatrices $HX_1^{(0)}, HX_2^{(0)}, HX_3^{(0)}$, and $HX_4^{(0)}$ for the 4-column partitioning strategy illustrated in Figure 4.1. Whether these partial products are computed in a single processor or in a cluster of processors, the information overlaps require data manipulation to obtain a correct answer to the matrix-matrix product. For instance, in Figure 4.2, the second piece of the $HX^{(0)}$ matrix is obtained after the addition of the rows that overlap the three partial matrices $HX_1^{(0)}, HX_2^{(0)}$, and $HX_3^{(0)}$.

In a parallel distributed environment, this reduce operation can be centralized in one CE or distributed among all the CEs. In the former, each CE sends back its partial result from the product to the CE performing the reduce operation. In the distributed reduce, every CE

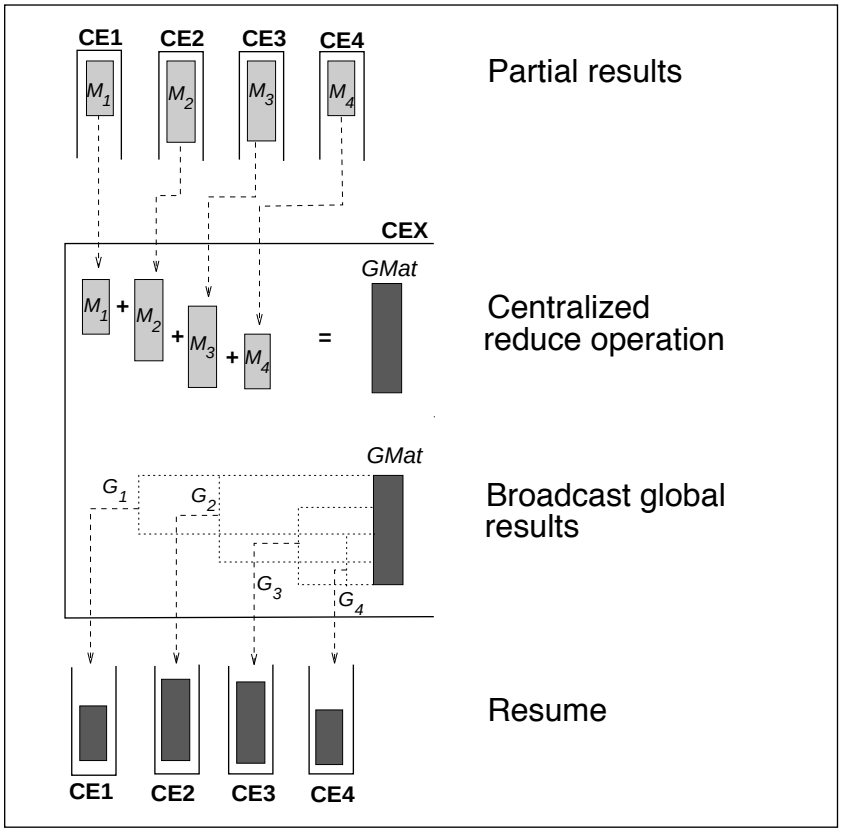

Figure 4.3: Centralized reduce operation. CEX is the computing element responsible for gathering partial results, merging them and broadcasting back parts of the global results to other CEs.

has information of the neighboring CEs with which it needs to exchange data, and the reduce operation is also perform in parallel. Figure 4.3 illustrates a centralized reduce operation, and Figure 4.4 illustrates a distributed reduce operation.
After the completion of the reduce operation in the $HX^{(0)}$ product, each CE gets back an updated $HX_i^{(0)}$ which is a copy of n_i rows of the global $HX^{(0)}$ matrix. If the centralized reduce is used then the CE performing the reduce operation sends back these updated matrices to the other CEs. However, the update of these matrices is done implicitly in the distributed reduce.

In the parallel computations of the i^{th} task triplet, the operations in (4.1.1) and (4.1.2) become

$$R_i^{(0)} = K_i - HX_i^{(0)}$$

and

$$\overline{R}_i^{(j+1)} = \overline{R}_i^{(j)} - HP_i^{(j)}.$$

Since the dimensions of the K_i matrix are $k_i \times s$, a strategy must be defined to select k_i rows from the n_i rows available in $HX^{(0)}$. Otherwise, if all of the n_i rows are used in the computations,

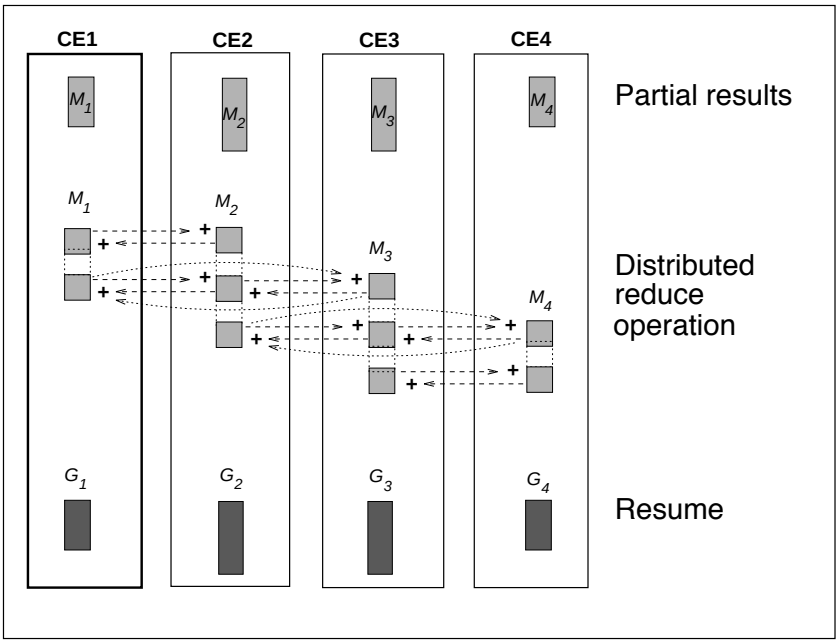

Figure 4.4: Distributed reduce operation. Data exchanges are performed in parallel between the CEs. At the end each CE has a part of the global result.

there will be duplicated data in one or more CEs. The duplicated data will perturb the Block-CG computations when not handled properly. Moreover, the selected k_i rows for the i^{th} task triplet must not overlap with the k_j rows selected for the j^{th} task triplet ($i \neq j$). This strategy is depicted in Figure 4.5.
In Figure 4.5, $GMat$ can be any global matrix that result from a parallel distributed operation. The Mat_i matrices are in the local memory of a CE. The shaded areas in the Mat_i matrices represent the k_i rows that are used in the computations associated with i^{th} task triplet.
The strategy is based on the following assumptions:

- The matrix H is SPD, thus it has a full diagonal
- k_i is the number of columns in H_i, and $\sum_{i=1}^{l} k_i = n$
- The matrix $GMat$ has n rows

The expression in (4.1.3) finds the first of the k_i rows in Mat_i, where Fc_i is the column number in H of the first column in H_i, and Fr_i is the row number in H of the first row in H_i. Then

$$FirstRow(Mat_i) = |Fc_i - Fr_i| + 1 \tag{4.1.3}$$

($|.|$ represents absolute value).

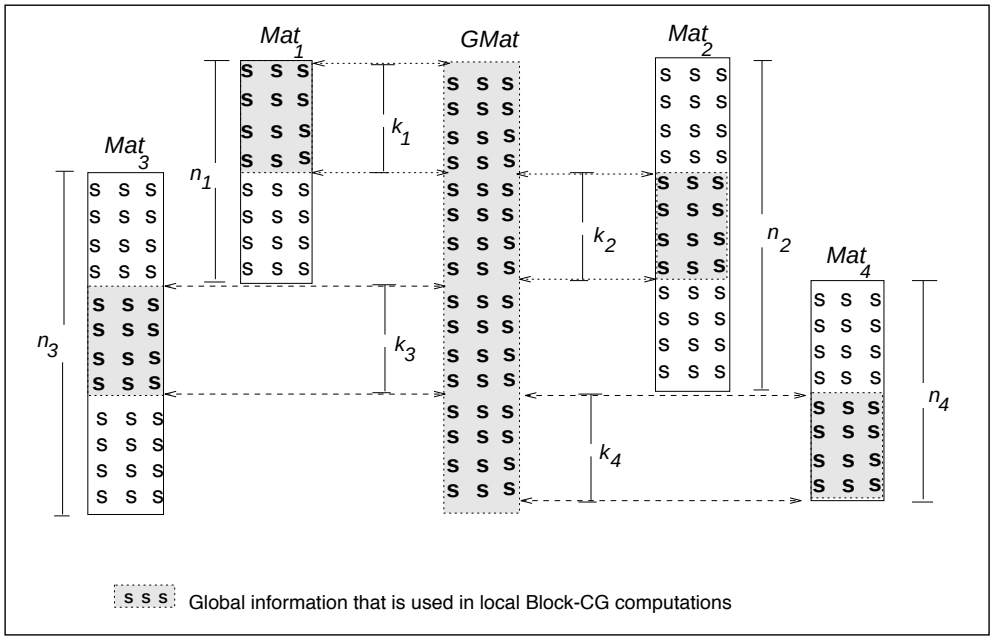

Figure 4.5: Distribution of a global result matrix $GMat$. The Mat_i matrices represent either the $R_i^{(0)}$, $\overline{R}_i^{(j+1)}$, $\overline{P}_i^{(j+1)}$, or $\overline{HP}_i^{(j+1)}$ matrices used in the parallel computations of the Block-CG algorithm.

4.1.2 Scheduling strategy

Among of the factors that determine the quality of a parallel implementation are:

- The design of the parallel algorithm
- The size of problems to be solved
- The computing environment
- The number of CEs used
- The efficient use of each CE

A parallel scheduler can be seen as a tool for tuning the last factor given the other four.
In this section, we introduce a static scheduler that is used in Section 4.2 inside the parallel implementations of the Block-CG Algorithm 2.2.2. This static scheduler tries to evenly distribute the total workload among the CEs. A balanced distribution of the workload is important to use all of the CEs efficiently in a parallel run.
This static scheduler considers two parameters for distributing the workload among the CEs. The first parameter is the size of each subproblem that results from the problem partition (e.i., number of rows times number of columns) and the second is the number of overlapping rows between different subproblems. Therefore, this scheduler is designed for homogeneous computing environments. In Chapter 7, we present a more general scheduler for heterogeneous computing

environments.
First of all, the static scheduler used in the parallel Block-CG implementations calls a function that computes a weight factor per task triplet $\mathcal{T}_i$. For each task $\mathcal{T}_i$ there is a $\theta_1(\mathcal{T}_i)$ that is the number of nonzeros in H_i. Also, there is a $\theta_2(\mathcal{T}_i)$ that is the total number of row overlaps between the submatrix H_i and all of the other submatrices.
The function that assigns a weight factor to each task is defined by:

$$\mathcal{W}(\mathcal{T}_i) = \theta_1(\mathcal{T}_i) + \theta_2(\mathcal{T}_i). \tag{4.1.1}$$

The $\theta_1(\mathcal{T}_i)$ determines the FLOP count associated with $\mathcal{T}_i$, and can thus be used as an estimate of the computational work associated with the task. One the other hand, $\theta_2(\mathcal{T}_i)$ is used to estimate the communication associated with $\mathcal{T}_i$.
Once the weight factors are computed, the scheduler sorts the tasks in descending order by their weight factors in a $work - list$. The scheduler then dispatches these tasks to the available CEs in the order they appear in the $work - list$.
To maintain workload balance among the CEs, the scheduler keeps track of the workload already assigned to each CE, and assigns the next task in the $work - list$ to one of the CEs with the lowest workload factor. This scheduling strategy can be seen as a solution to a binpacking problem (see for instance Dror (1990), Mizuike, Ito, Kennedy, and Nguyen (1991), and Mayr (1988)).

4.2 Implementations of Block-CG

We present three different parallel implementations of the Block-CG Algorithm 2.2.2. First, we develop an *All-to-All* parallel version in which we parallelize the entire Block-CG algorithm. The other two implementations are based on a *master-slave* computing model, with different approaches to determine which sections of the algorithm are performed in parallel and which ones are performed sequentially.
We have used the P4 library of parallel routines (Butler and Lusk (1992)) to manage the parallel environments on different machines (also we have developed versions using the PVM library of routines (Beguelin, Dongarra, Geist et al. (1992), and Geist, Baguelin, Dongarra et al. (1994))). The computing elements (CEs) can all be physically part of the same computer, or a network of heterogeneous distributed processors. For the remainder of this chapter, we focus our discussion on the processes and the tasks they perform in each implementation, rather than on the CEs because processes are the active entities in the execution of a program and in some computer platforms the user does not have direct control over the CEs.

4.2.1 All-to-All implementation

In this implementation, we perform a full parallel implementation of the Block-CG algorithm at the expense of redundant computations that are carried out in parallel by different processes referred to here as the *worker* processes. At first there is an *initiator* process which creates the other *worker* processes and generates the task triplets $< H_i, X_i, K_i >$. Later, the *initiator* process becomes one of the *workers* and works on some of the generated tasks.
One or more tasks can be assigned to a *worker* process. After a worker process receives one or more tasks it builds a set of matrices (one set per task) that corresponds to the set of matrices used in Algorithm 2.2.2. Figure 4.6 illustrates an example of a task triplet and the set of locally generated matrices associated with the task. During an iteration a *worker* process communicates

to its neighbors the results from its computations of H_iP_i. Each *worker* needs to communicate only to neighbors with tasks that have information that overlap its tasks. Then, each *worker* broadcasts the $R_i^T R_i$ and the $P_i^T H P_i$ matrices to all the other *workers*.

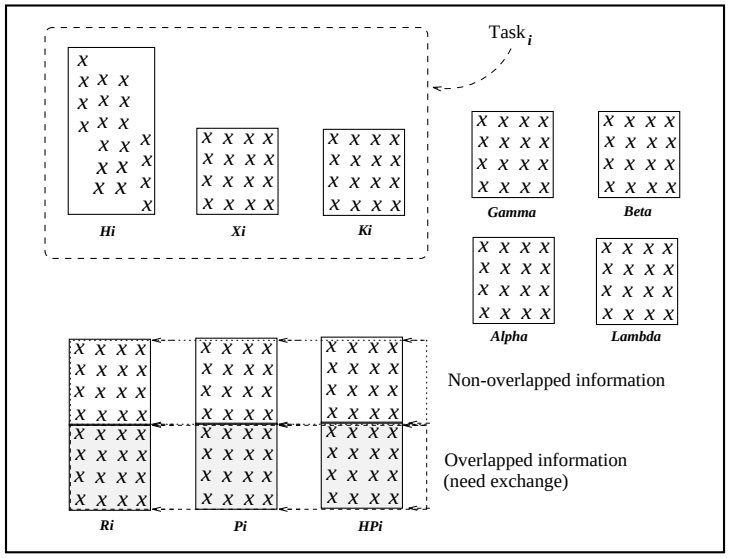

Figure 4.6: Block-CG matrices kept locally by each process. There is a set of matrices per task assigned to a process.

Every *worker* performs an iteration of the Block-CG, and tests whether the stopping criterion has been met. If the stopping criterion has not been met, the *worker* processes start a new iteration. At the end, when the stopping criterion is satisfied, each *worker* process reports its part of the solution to the *initiator* process which assembles the solution of the whole system. Figure 4.7 captures the interactions between processes in the $All - to - All$ implementation. Clearly, the communication between processes has an impact on the performance of the three parallel Block-CG implementations that are presented in this chapter. Thus, we study the number of messages sent at each iteration for all of the three parallel Block-CG implementations. Let m_k be the number of neighboring $workers$ with which $worker_k$ will exchange partial results, and p be the total number of $workers$.
For every product $H_iX_i^{(0)}$ and H_iP_i :

$$\mathcal{M} = \sum_{k=1}^{p} m_k \qquad \text{messages,}$$

and the length of these messages varies according to the number of rows $worker_k$ needs to exchange with each of its neighboring $workers$.

For every product $R^T R$ (all to all):

$$p \cdot (p-1) \qquad \text{messages of length } s \cdot s$$

For every product $P^T HP$ (all to all):

$$p \cdot (p-1) \qquad \text{messages of length } s \cdot s$$

Leading to : $2p(p-1)+\mathcal{M}$ messages of lengths varying from $s \cdot s$ to $\max(n_i) \cdot s$, where n_i is the number of rows in H_i.

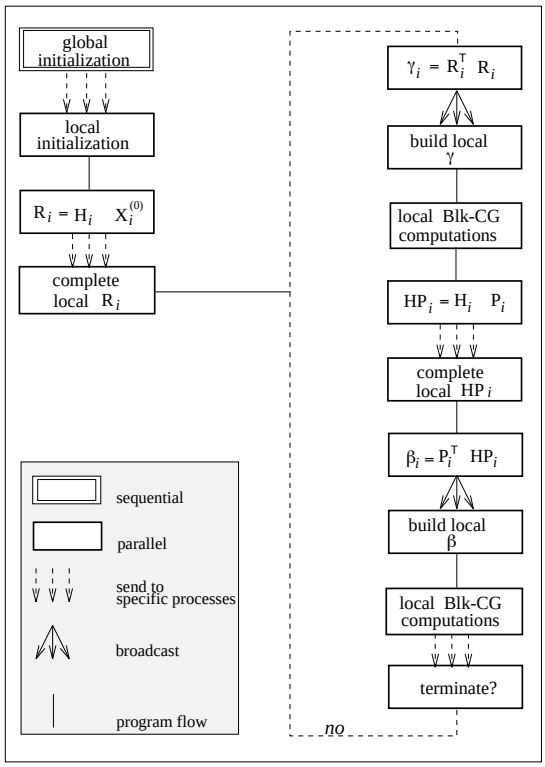

Figure 4.7: Scheme of the $All-to-All$ parallel Block-CG implementation.

4.2.2 Master-Slave: distributed Block-CG implementation

In this implementation, we use a $master-slave$ programming model in which the *master* process monitors the program execution, while the Block-CG algorithm is run at the *slave* processes' level. Initially a *master* process is created which in turn creates the *slave* processes and generates the tasks ($< H_i, X_i, K_i >$). The *master* calls the static scheduler to distribute the tasks. After scheduling the tasks and dispatching them to the *slave* processes, the *master* process performs three reduce operations at each iteration.
In the first reduce, the *master* collects from *slave* processes their results from computing the products $R_i^T R_i$. The *master* then combines these partial results to build the full $R^T R$. The resulting matrix is referenced as γ in the Block-CG Algorithm 2.2.2. This matrix is later factorized by the *master* and broadcasted back to the *slave* processes. During this reduce operation, only the *master* is active while the *slave* processes wait for the answer (this problem was previously identified as synchronization point in Chapter 2 for the CG algorithm).
In the second reduce, the *master* collects the partial products $P_i^T H P_i$ from the *slave* processes. This time the matrix corresponds to the β matrix in the Block-CG Algorithm 2.2.2. The master also factorizes and broadcasts back the results from β to the *slave* processes. In the third

reduce operation, the master runs the test for termination. To the test for convergence, the master gathers information that is local to the *slave* processes (local residuals, and $\|X_i^{(j)}\|_1$,) and when the stopping criterion is met, the *master* process signals *termination* to the *slave* processes. Otherwise, the *master* process signals *continuation* to the *slave* processes and the *slaves* start a new parallel iteration.
The *slave* processes initially receive from the *master* process at least one task to execute. Then, the *slave* processes build locally their partial matrices and these matrices are kept in memory until the *master* process signals *termination*. In Figure 4.6, the partial matrices β_i, and γ_i are computed locally by each *slave* process. The first and second reduce operations transform these partial matrices into global β, and γ respectively.

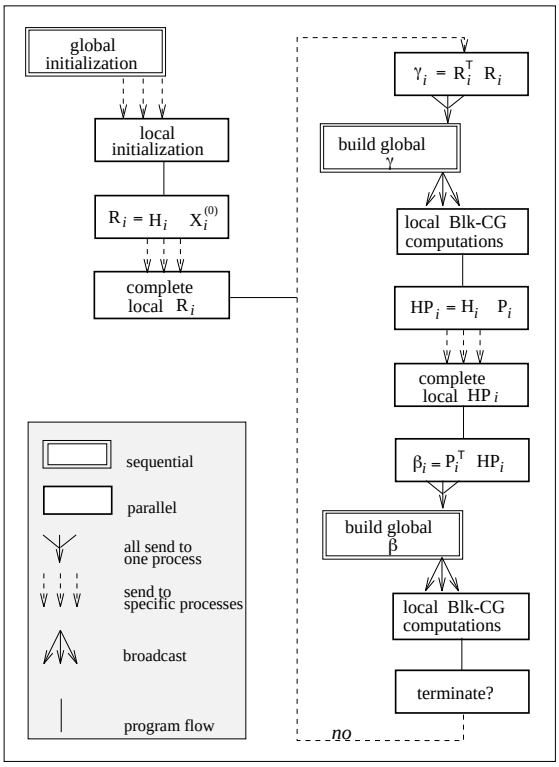

Figure 4.8: Scheme of the $Master - Slave$: distributed parallel Block-CG implementation.

As mentioned before, there may exist row overlaps between two or more tasks while computing the H_iX_i and H_iP_i products. A process working on a task with row overlaps has to communicate with its neighbors to complete its part of the global product. In addition, one or more tasks with rows that overlap may be scheduled to the same *slave* process and in this case the overlap is handled locally.
To study the average number of messages sent at every iteration step we use the same set of assumptions made for the $All - to - All$ implementation (Section 4.2.1).

For every product H_iP_i :

$$\mathcal{M} = \sum_{k=1}^{p} m_k \qquad \text{messages.}$$

The length of these messages varies according to the number of rows $slave_k$ will exchange with each of its neighbor *slaves*.

For every product $R_i^T \cdot R_i$ (one message in every direction between the *master* and $slave_k$):

$$2 \cdot (p-1) \qquad \text{messages of length } s \cdot s$$

For every product $P_i^T \cdot HP_i$ (one message in every direction between the *master* and $slave_k$):

$$2 \cdot (p-1) \qquad \text{messages of length } s \cdot s$$

Which results in : $4(p-1) + \mathcal{M}$ messages of lengths varying from $s \cdot s$ to $\max(n_i) \cdot s$, where n_i is the number of rows in H_i.
Figure 4.8 captures the interaction between the *master* process and the *slave* processes under this parallel implementation.

4.2.3 Master-Slave: centralized Block-CG implementation

Lastly, we present an implementation based on a $master-slave$ programming model that differs from the previous one in the roles of the *master* and the *slave* processes. In the following parallel implementation only the $\overline{HP}^{(j)}$ product is parallelized.
The *master* process generates the tasks by dividing the matrix H from 4.1.1 into submatrices H_i. Here the initial tasks are no longer the task triplets $< H_i, X_i, K_i >$, but $< H_i, X_i >$ for the first iteration, and $< H_i, P_i >$ for the main iteration (see Figure 4.9). The *master* process spawns a group of *slave* processes, and then distributes the H_i submatrices among the *slave* processes. The *master* process executes all the steps from the Block-CG algorithm except for the products HP which are performed in parallel by the *slave* processes. Therefore, the *master* process partitions and distributes the matrix P in a way that will be product-compatible with the already distributed H_i's. In the reduce operation, the *slave* processes send their partial $\overline{HP}_i^{(j)}$ to the *master* process and the *master* assembles the global $\overline{HP}^{(j)}$ matrix.
Figure 4.9, shows the interaction between the *master* and the *slave* processes in this parallel implementation. At the end of an iteration, the *master* process performs the test for termination and signals *termination* or *continuation* to the *slave* processes.
We conduct the same study as before to calculate the number of messages sent at each iteration.
For every product H_iP_i :

$$2 \cdot (p-1) \qquad \text{messages of length } n_i \cdot s$$

$(p-1)$ in every direction between the *master* process and the *slave* processes. Resulting in $2p-2$ messages being sent at each iteration step.
Table 4.1 summarizes the number of messages that are sent at each iteration of the Block-CG Algorithm 2.2.2 under each of the parallel implementations presented here. The third implementation, $master-slave$: centralized Block-CG, involves fewer messages than the other two implementations. However, the messages sent in the $master-slave$: centralized Block-CG

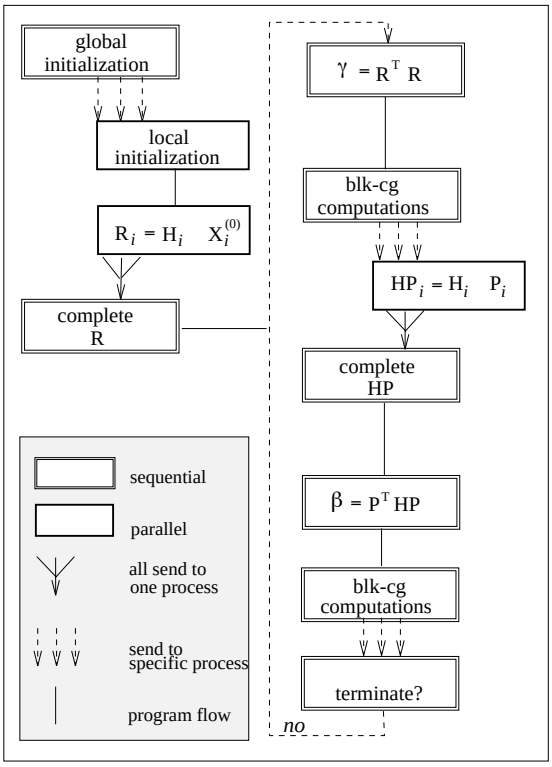

Figure 4.9: Scheme of the $Master - Slave$: centralized parallel Block-CG implementation.

implementation are longer than the messages sent in any of the other two implementations. In this implementation the *master* process sends at each iteration the $P_i^{(j)}$ submatrices to each *slave* process, while in the other two implementations this information is kept locally and processes only exchange parts of the $\overline{HP}_i^{(j)}$ and small $s \times s$ matrices.

Implementation	Number of Messages
All-to-all	$2p(p-1) + \sum_{k=1}^{p} m_k$
Master-Slave: distributed Block-CG	$4(p-1) + \sum_{k=1}^{p} m_k$
Master-Slave: centralized Block-CG	$2p - 2$

Table 4.1: Messages being sent per parallel Block-CG implementation.

Chapter 5

Parallel Block-CG experiments

The purpose of this chapter is to compare the performance of the three parallel Block-CG implementations described in Chapter 4, and to show advantages from the parallelization of the Block-CG algorithm.
In this chapter, we assume through all the experiments that a parallel run of Block-CG with block size of one is numerically equivalent to a run of Classical CG as can be verified from Algorithms 2.1.1 and 2.2.2. Thus, the results from runs of Classical CG are presented under the columns labeled Block-CG with block size of one in all of the tables in this chapter.
In the three parallel Block-CG implementations, a run with only one CE involves extra overhead that is not present in a sequential run of the Block-CG implementation. In the $All - to - All$ implementation, the overhead is very small and is due to data manipulation provided for handling the parallelism. In the other two implementations, the overhead comes from creation of one *slave* process, and this overhead is more significant than the one in the $All - to - All$ implementation.

The matrix that results from squaring the SHERMAN4 matrix (see Section 3.4,) is used in the first parallel experiment. Next, the LANPRO (NOS2) and LANPRO (NOS3) problems are solved in parallel. For these two problems, the block sizes have been chosen based on the results and remarks from Chapter 3.

In Section 5.6 a different problem is used. It comes from a finite difference model of Laplace's equation (see for example Strang (1986))

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0. \tag{5.0.1}$$

When the right-hand side of 5.0.1 is different from 0, the equation is Poisson's equation.
Applying a central difference approximation to the second derivates

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x+h) - 2u(x+h) + u(x-h)}{h^2} \tag{5.0.2}$$

and

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u(y+h) - 2u(y+h) + u(y-h)}{h^2}. \tag{5.0.3}$$

The right-hand side approaches the left-hand side as h goes to zero.

The differential equations $\frac{\partial^2 u}{\partial x^2} = f$ and $\frac{\partial^2 u}{\partial y^2} = f$ are replaced by a finite differences approximation of the values of u at the meshpoints $x = h, 2h, ..., Nh$.
Figure 5.1 (a) shows a 5-point molecule resulting from the combination of 5.0.2 and 5.0.3. The boundary conditions at the edge of the grid are $u = 0$. The points in the grid are labeled from 1 to n^2 ($n = N^2$), and there is one linear equation per grid point. The resulting matrix A is depicted in Figure 5.1 (b). The matrix is SPD with $+4$'s along the main diagonal and -1's in the other diagonals. Because of the boundary conditions, some -1's are removed from the other diagonals.

A problem based on the Poisson's equation is solved in Section 5.6. The grid size is 64, thus the matrix A is of order 4096, with 20224 nonzero elements.

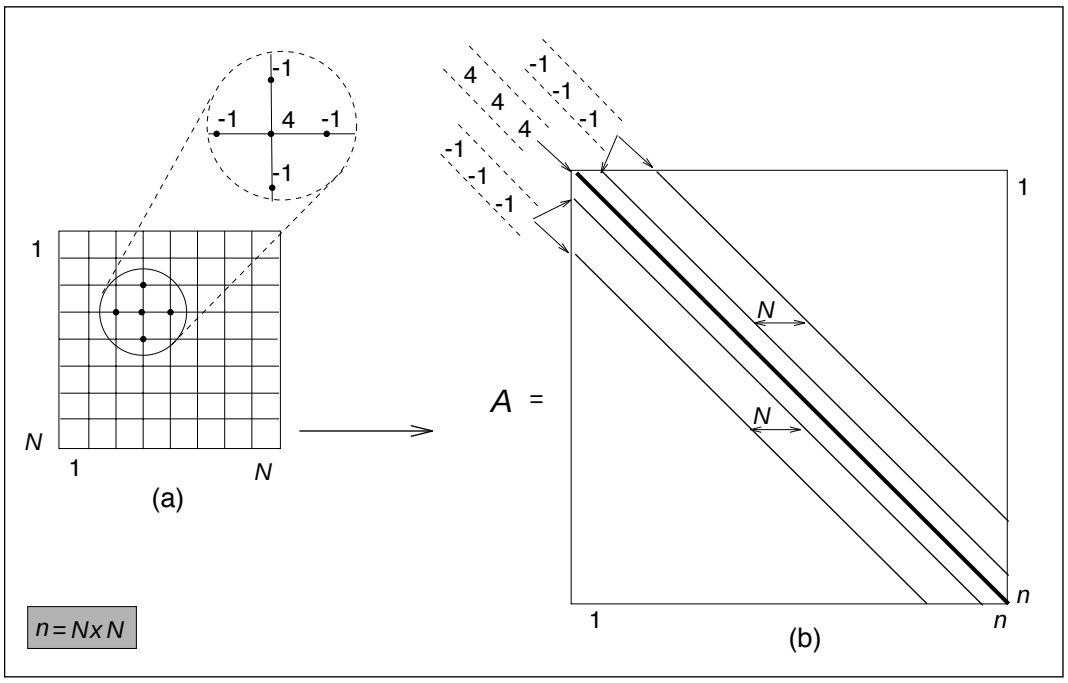

Figure 5.1: 5-point discretization of Laplace's equation using finite differences. (a) 5-point approximation. (b) Structure of matrix A that results from discretization.

In this chapter, the performance of the parallel implementations is measured in terms of two parameters, the speedup and the efficiency. The speedup is defined by:

$$\text{speedup} = \frac{\text{Execution time from best sequential run}}{\text{Execution time from parallel run using } p \text{ CEs}}$$

and the efficiency by:

$$\mathcal{E} = \frac{\text{speedup}}{p}.$$

A speedup is a factor of the reduction in the execution time when moving from a sequential implementation to a parallel implementation and ideally a speedup equals the number of CEs that are used during a parallel run. In some less favorable cases, the parallel implementation requires more time than the sequential implementation, thus $0 \leq speedup \leq 1$. The efficiency gives an estimation of the contribution of the p processors to the speedup and ideally it is equal to one.
The following parallel experiments were run on three different parallel computer systems. The first platform is a Thin node SP2 with 32 RS6000-370 processors. Each node has a theoretical peak performance of 130 Mflops. The nodes are connected through a high performance switch with latency time of 40 microseconds and a bandwidth of 30 to 25 Mbytes per second.
The second platform is a BBN TC2000 computer system installed at Argonne National Laboratory. The TC2000 system is a virtual shared memory computer, and peak performance of 20 MFlops per node. The nodes are interconnected through a multistage network processor called the Butterfly switch.
The last computing platform is a network of SUN Sparc 10 workstations. Each workstation has a peak performance of 40 MFlops, and they are connected through ETHERNET. The peak performance of the ETHERNET network is 2.5 Mbytes per second.

5.1 Parallel solution of the SHERMAN4 problem

Some advantages of using Block-CG over Classical CG were presented in Section 3.4 with the results from sequentials runs of Block-CG in the solution of the A^TA from SHERMAN4 problem. In this section, the results from parallel runs of Block-CG on the A^TA SHERMAN4 are presented. The block sizes of 4, 8, 16 and 32 are used in these experiments. Tables 5.1 to 5.6 summarize the results from the three different parallel Block-CG implementations.
The results from parallel runs of the Block-CG $All-to-All$ implementation are shown in Tables 5.1 and 5.2. The runs were performed on a SP2 computer. The bottlenecks from parallelizing Block-CG are shown in these tables by the low speedups obtained. Increasing the block size gives better speedups. However, the efficiency factors decrease as the number of CEs is increased.
The execution time of a sequential run of Block-CG solving the A^TA SHERMAN4 problem and using block size of 8 is almost the same as the execution time of a sequential run of Classical CG on the same problem. As shown in Section 3.4, the block size of 8 converges in fewer equivalent iterations than the Classical-CG, and after parallelizing the Block-CG algorithm the execution time for solution of the A^TA SHERMAN4 problem is reduced by almost 63% with a block size of 8 and computing on 8 CEs.

In Tables 5.3 and 5.4 the results from parallel runs of the Block-CG $Master-Slave$: distributed implementation are presented. The runs were performed on a SP2 computer system.
The parallel Classical CG implementation using the $Master - Slave$: distributed scheme also reports low speedups, but they are better than the speedups obtained with the $All - to - All$ implementation and this is due to the number of messages being exchanged in each implementation at both synchronization points.

In the $All-to-All$ implementation, the execution of the synchronization points takes more time than in the $Master-Slave$: distributed implementation because of the number of messages being exchanged. And in both implementations only a scalar is exchanged in a message which does not compensate for the expense of the communication over the computations.
Increasing the block size improves the performance of the $Master-Slave$:distributed implementation. In Table 5.3, the execution time from the solution of the A^TA SHERMAN4 problem using 8 CEs and a block size of 8 is reduced by almost 75% from the sequential version. In Table 5.4, the performance is improved even more when increasing the block size to 16 because a reduction on the execution time of almost 84% from the sequential implementation is obtained. In this case, the execution time is reduced by almost 80% from the execution of the sequential Classical-CG.
Even though in some cases Block-CG takes a few more iterations to converge than Classical CG, with the $Master-Slave$: distributed implementation is possible to present an example in which the performance of Block-CG is improved with the parallel implementation.
According to the results reported in Table 3.11, Block-CG with a block size of 32 is computationally more expensive than Classical CG. However the performance of Block-CG with a block size of 32 is also improved with the results presented in Table 5.4.

Execution times of Block-CG sequential version: block size 1 = 6.1 secs, block size 4 = 5.7 secs, block size 8 = 6.0 secs									
Number of CEs	block size 1			block size 4			block size 8		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	6.1	1.0	1.00	5.9	0.9	0.96	6.1	0.9	0.98
2	4.8	1.3	0.65	4.6	1.2	0.62	3.7	1.6	0.81
4	3.7	1.7	0.43	1.8	3.2	0.79	2.2	2.7	0.68
8	6.7	0.9	0.11	3.5	1.6	0.20	2.3	2.6	0.33

Table 5.1: Results from the $All-to-All$ parallel implementation of Block-CG on the solution of the A^TA SHERMAN 4 problem. The times shown in the table are in seconds.

In Tables 5.5 and 5.6, the results from the $Master-Slave$: centralized parallel Block-CG implementation are presented. This parallel implementation of Classical CG and Block-CG performs very poorly, and the reason is the granularity of the work being performed in parallel that is very small compared to the granularity of the problem performed sequentially.
Increasing the granularity of the H_iP_i products may improve the behavior of this implementation. Thus, increasing the computational weight of the parallel products compensates for the expensive cost of distributing the P_i matrices at each iteration.

Execution times of Block-CG sequential version: block size 16 = 7.4 secs block size 32 = 10.3 secs						
Number of CEs	block size 16			block size 32		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	7.6	0.9	0.97	10.6	0.9	0.97
2	4.1	1.8	0.90	6.0	1.7	0.86
4	2.2	3.4	0.84	3.9	2.6	0.66
8	2.7	2.7	0.34	3.4	3.1	0.39

Table 5.2: Results from the $All - to - All$ parallel implementation of Block-CG on the solution of the A^TA SHERMAN4 problem. The times shown in the table are in seconds.

Execution times of Block-CG sequential version: block size 1 = 6.1 secs block size 4 = 5.7 secs block size 8 = 6.0 secs									
Number of CEs	block size 1			block size 4			block size 8		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	7.3	0.8	0.80	6.3	0.9	0.91	6.4	0.9	0.94
2	4.7	1.3	0.65	3.9	1.5	0.73	3.9	1.5	0.77
4	3.5	1.7	0.43	2.1	2.7	0.68	2.0	3.0	0.75
8	3.8	1.6	0.20	1.8	3.2	0.40	1.3	4.6	0.58

Table 5.3: Results from the $Master - Slave$: distributed parallel implementation of Block-CG on the solution of the A^TA SHERMAN 4 problem. The times shown in the table are in seconds.

Execution times of Block-CG sequential version: block size 16 = 7.4 secs block size 32 = 10.3 secs						
Number of CEs	block size 16			block size 32		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	7.8	0.9	0.95	10.6	0.9	0.97
2	4.1	1.8	0.90	6.1	1.7	0.84
4	2.3	3.2	0.80	4.0	2.6	0.64
8	1.2	6.2	0.77	2.6	4.0	0.50

Table 5.4: Results from the $Master - Slave$: distributed parallel implementation of Block-CG on the solution of the A^TA SHERMAN4 problem. The times shown in the table are in seconds.

Execution times of Block-CG sequential version: block size 1 = 6.1 secs block size 4 = 5.7 secs block size 8 = 6.0 secs									
Number of CEs	block size 1			block size 4			block size 8		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	7.4	0.8	0.82	7.5	0.8	0.76	6.8	0.9	0.88
2	7.9	0.8	0.39	8.6	0.7	0.35	7.9	0.8	0.38
4	9.2	0.7	0.17	10.8	0.5	0.13	8.3	0.7	0.18
8	10.2	0.6	0.08	12.6	0.5	0.06	9.8	0.6	0.08

Table 5.5: Results from the $Master - Slave$: centralized parallel implementation of Block-CG on the solution of the A^TA SHERMAN 4 problem. The times shown in the table are in seconds.

Execution times of Block-CG sequential version: block size 16 = 7.4 secs block size 32 = 10.3 secs						
Number of CEs	block size 16			block size 32		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	8.6	0.9	0.86	10.6	0.9	0.97
2	10.2	0.7	0.36	11.4	0.9	0.45
4	12.3	0.6	0.15	12.2	0.8	0.21
8	13.9	0.5	0.07	16.8	0.6	0.08

Table 5.6: Results from the $Master-Slave$: centralized parallel implementation of Block-CG on the solution of the A^TA SHERMAN4 problem. The times shown in the table are in seconds.

Also, in a shared memory parallel environment this implementation may perform better, because the communications are avoided and the access to shared memory can be easily synchronized in a less expensive way than exchanging messages. In the next sections we perform a more detailed analysis of this implementation when solving the LANPRO NOS2 and NOS3 problems.

5.2 Parallel solution of the LANPRO (NOS2) problem

The purpose of this experiment is to compare the performance of the sequential and parallel implementations of the Classical-CG Algorithm 2.1.1 and the Block-CG Algorithm 2.2.2 using a large block size for Block-CG.
In this experiment, the LANPRO (NOS2) problem is solved in parallel in the SP2 computer. The block sizes of 1 and 196 are chosen for the experiment based on the sequential Block-CG experiments presented in Chapter 3.
To be able to compare the sequential and parallel implementations on the SP2 computer, a selection of the sequential experiments from Chapter 3 are repeated in this section. Table 5.7 summarizes the results from sequential runs on the SP2.
The results from the solution of the LANPRO (NOS2) problem using the $All-to-All$ parallel Block-CG implementation are presented in Table 5.8. Although, the computations of Block-CG with block size 196 continue to be more expensive than with Classical CG, a parallel run of Block-CG on 8 CEs has reduced the execution time of the sequential Block-CG run by almost 76%.
The parallel $All-to-All$ implementation performs very poorly for Classical CG. An analysis of the run from the $All-to-All$ implementation has shown that the communication time between CEs is a dominant factor in the total execution time. This is due to the small granularity of the

LANPRO problem	Block size	Normwise Backward error ω	Execution time
NOS2	1	0.45×10^{-8}	3.4
NOS2	196	0.75×10^{-9}	54.9

Table 5.7: Sequential runs of Classical CG and Block-CG implementations for solving the LANPRO (NOS2) problem on a SP2 computer system. The times shown in this table are in seconds

parallel Classical CG operations. Basically, in Classical CG, all of the BLAS 3 operations from Block-CG are replaced by BLAS 1 and BLAS 2 operations in the Classical CG.
In these experiments, the highest speedup reported for the $All - to - All$ parallel Block-CG implementation is attained with 8 CEs. The efficiency suggests that 50% of the total computational power of the 8 nodes has been used.

Execution times of Block-CG sequential version: block size 1 = 3.4 secs block size 196 = 54.9 secs						
Number of CEs	block size 1			block size 196		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	3.7	0.9	0.92	59.4	0.9	0.92
4	3.1	1.1	0.27	28.3	1.9	0.48
8	8.9	0.4	0.05	13.9	4.0	0.50
16	15.1	0.2	0.01	14.2	3.9	0.24

Table 5.8: Results from the $All - to - All$ parallel implementation of Block-CG on the solution of the LANPRO (NOS2) problem. The times shown in the table are in seconds.

Table 5.9 shows the results from the solution of the LANPRO (NOS2) problem using the $Master - Slave$: distributed parallel Block-CG implementation. The execution time from the sequential Block-CG is reduced by almost 82% with the parallel Block-CG using 16 CEs. The highest speedup is attained with 16 processors, but the efficiency is reduced as we increased the number of processors.
Lastly, Table 5.10 summarizes results from runs of the $Master - Slave$: centralized parallel

Execution times of Block-CG sequential version: block size 1 = 3.4 secs block size 196 = 54.9 secs						
Number of CEs	block size 1			block size 196		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	3.9	0.9	0.87	57.8	0.9	0.95
4	2.8	1.2	0.30	17.5	3.2	0.80
8	3.1	1.1	0.14	11.6	4.7	0.59
16	4.4	0.8	0.05	10.1	5.4	0.34

Table 5.9: Results from the $Master - Slave$: distributed parallel implementation of Block-CG on the solution of the LANPRO (NOS2) problem. The times shown in the table are in seconds.

Block-CG implementation when solving the LANPRO (NOS2) problem. The Amdahl's law (Amdahl (1967)) helps to explain to the poor performance of this implementation. A parallel implementation may have sequential and parallel sections. The sequential sections are sometimes called synchronization points. Moreover, if σ is the ratio of computations performed in sequence, then $1-\sigma$ is the ratio of computations perform in parallel. The Amdahl's law, defines an upper bound, $\mathcal{S}$, for the speedup

$$\begin{aligned} \mathcal{S} &= \lim_{p\to\infty} \frac{1}{\sigma + \dfrac{1-\sigma}{p}} \\ &= \frac{1}{\sigma}. \end{aligned}$$

In the $Master - Slave$: centralized Block-CG implementation only the part of the total FLOP count that is related to $\theta(A)$ is performed in parallel. Table 3.3 shows the percentages of the total Block-CG FLOP count that comes from $\theta(A)$ for the problem LANPRO (NOS2) problem. A 33.7% of the total FLOP count comes from $\theta(A)$ using a block size of 1, and this factor is reduced to 0.3% with a block size of 196.
In these cases, σ is equal to 0.663 when the block size is 1 and equal to 0.997 when the block size is 196. Thus, $\mathcal{S}$ is equal to 1.5 and 1.0 for the block sizes of 1 and 196, respectively. The lack of speedups in the results shown in Table 5.10 are explained by the small values of $\mathcal{S}$. This is not the case for the other two implementations that have higher ratios between the parallel and sequential sections.

Execution times of Block-CG sequential version: block size 1 = 3.4 secs block size 196 = 54.9 secs						
Number of CEs	block size 1			block size 196		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	5.6	0.6	0.61	71.2	0.8	0.77
4	3.9	0.9	0.22	68.6	0.8	0.20
8	4.0	0.9	0.12	68.2	0.8	0.10
16	4.5	0.8	0.05	110.1	0.5	0.03

Table 5.10: Results from the $Master-Slave$: centralized parallel implementation of Block-CG on the solution of the LANPRO (NOS2) problem. The times shown in the table are in seconds.

5.3 Parallel solution of the LANPRO (NOS3) problem

The LANPRO (NOS3) problem introduced in Chapter 3 is solved in these experiments. Table 5.11 summarizes the results from sequential runs of Block-CG on the SP2 computer. Based on the results presented in Section 3.3, the block sizes of 1, 4, and 8 were chosen.

LANPRO problem	Block size	Normwise Backward error ω	Execution time
NOS3	1	0.91×10^{-16}	2.5
NOS3	4	0.89×10^{-16}	4.8
NOS3	8	0.90×10^{-16}	7.5

Table 5.11: Sequential runs of Classical CG and Block-CG implementations for solving the LANPRO (NOS3) problem on a SP2 computer system. The times shown in this table are in seconds

Table 5.12 presents results from runs of the $All-to-All$ parallel implementation of Block-CG on the solution of the LANPRO (NOS3) problem. For all of the block sizes used in this case, the granularity is smaller than the granularity of the block size of 196 on the LANPRO (NOS2) problem, thus the speedups have decreased compared to the speedups shown in Table 5.8.

In this case, the maximum speedups are attained using 4 CEs with the block sizes of 4 and 8. The parallel performance of Block-CG with block size of 1 is very poor in this case because of the required synchronizations discussed in Section 2.1, and the low granularity of Classical CG.

Execution times of Block-CG sequential version: block size 1 = 2.5 secs block size 4 = 4.8 secs block size 8 = 7.5 secs									
Number	block size 1			block size 4			block size 8		
of CEs	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	2.5	1.0	1.00	4.8	1.0	1.00	7.6	0.9	0.99
4	1.7	1.5	0.38	1.8	2.7	0.67	2.4	3.1	0.78
8	4.1	0.6	0.08	3.7	1.3	0.16	4.0	1.9	0.23
16	11.6	0.2	0.01	13.2	0.4	0.02	10.6	0.7	0.04

Table 5.12: Results from the $All - to - All$ parallel implementation of Block-CG on the solution of the LANPRO (NOS3) problem. The times shown in the table are in seconds.

Table 5.13 shows the results of runs of the $Master - Slave$: distributed Block-CG implementation. As in the case of the $All - to - All$ Block-CG implementation the speedups have also decreased when compared to the speedups reported in Table 5.9. The maximum speedup with a block size of 4 is attained with 8 CEs, and with a block size of 8 is attained with 16 CEs. The execution times for Block-CG are shorter than the execution times of the sequential and parallel Classical CG implementations. The efficiency of 8 CEs in the case of block size of 8 is higher than the efficiency attained with the same 8 CEs in the case of block size of 4 and this is due to and increase in the granularity after increasing the block size from 4 to 8.

The results of the $Master - Slave$: centralized implementation are presented in Table 5.14. Again, the implementation reported poor speedups. From information in Table 3.10, and the Amdahl's law, it can be deduced that the upper bounds of the speedups in this experiments are 3.2 for block size 1, 1.6 for block size 4, and 1.4 for block size of 8. However, the maximum speedup reported from these experiments for block size of one is 0.9, and 1.4 for the block sizes of 4 and 8. The poor performance of the parallel Classical CG is due to a straight parallel implementation of Classical CG without the considerations discussed in Section 2.1.

Lastly, Table 5.15 shows comparative results from different runs of sequential Classical CG and parallel version of Block-CG. The purpose of this table is to illustrate the advantages of the parallel Block-CG over a sequential Classical CG even for those cases presented in Chapter 3 where Block-CG was a few equivalent iterations more expensive than Classical CG.

Execution times of Block-CG sequential version: block size 1 = 2.5 secs block size 4 = 4.8 secs block size 8 = 7.5 secs									
Number	block size 1			block size 4			block size 8		
of CEs	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	3.0	0.8	0.83	5.1	0.9	0.94	7.7	0.9	0.97
4	1.5	1.7	0.42	2.8	1.7	0.43	2.4	3.1	0.78
8	1.5	1.7	0.21	1.3	3.7	0.46	1.5	5.0	0.63
16	1.9	1.3	0.08	1.3	3.7	0.23	1.3	5.8	0.36

Table 5.13: Results from the *Master – Slave*: distributed parallel implementation of Block-CG on the solution of the LANPRO (NOS3) problem. The times shown in the table are in seconds.

Execution times of Block-CG sequential version: block size 1 = 2.5 secs block size 4 = 4.8 secs block size 8 = 7.5 secs									
Number	block size 1			block size 4			block size 8		
of CEs	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	4.7	0.5	0.53	7.3	0.7	0.66	9.9	0.7	0.75
4	2.7	0.9	0.23	4.6	1.0	0.26	5.7	1.3	0.33
8	2.8	0.9	0.11	3.5	1.4	0.17	5.2	1.4	0.18
16	3.0	0.8	0.05	3.5	1.4	0.09	5.2	1.4	0.09

Table 5.14: Results from the *Master – Slave*: centralized parallel implementation of Block-CG on the solution of the LANPRO (NOS3) problem. The times shown in the table are in seconds.

LANPRO problem	Sequential Classical CG exec. time (secs)	Parallel Block-CG						
		Block size	All-to-All		Master-Slave: Distributed		Master-Slave: Centralized	
			N CEs	Time (secs)	N CEs	Time (secs)	N CEs	Time (secs)
NOS2	3.4	1	4	3.1 ↗	4	2.8 ↗	4	3.9
NOS2	3.4	196	8	13.9	16	10.1	8	68.2
NOS3	2.5	1	4	1.7 ↗	4	1.5 ↗	4	2.7
NOS3	2.5	4	4	1.8 ↗	8	1.3 ↗	8	3.5
NOS3	2.5	8	4	2.4 ↗	16	1.3 ↗	8	5.2

Table 5.15: Time comparison between runs of sequential Classical CG and parallel Block-CG implementations. The ↗ is used for the cases where the a parallel Block-CG implementation has performed better than sequential implementation of Classical CG.

For each of the LANPRO problems there is one or more entries in Table 5.15. Each entry in the table compares runs from sequential Classical CG versus the best time obtained in the solution of the same problem with each parallel Block-CG implementation. Along with an entry for a parallel implementations, there is a column that identifies the number of CEs used in the solution of the problem, and another entry that identifies the block size.
Entries in Table 5.15 that are marked with a ↗ are the favorable cases in which a parallel Block-CG implementation has performed better than the sequential Classical CG implementation. Therefore, the execution time in those cases has been reduced by the effects of parallelism despite the extra FLOP count generated by Block-CG. In the $Master - Slave$: centralized Block-CG implementation none of the best times improves the performance of the sequential Classical CG. In the other two implementations almost of all of the cases are favorable, except in the solution of the LANPRO (NOS2) problem with a block size of 196.
In the sequential Block-CG implementation, the solution of LANPRO (NOS2) with the large block size required far more FLOPs than Classical CG. In the $Master - Slave$:distributed Block-CG implementation, the large block size enlarges the sequential sections where the *master* performs the centralized reduce operations and factorizes the $\gamma_{(j)}$ and $\beta_{(j)}$ matrices. In addition, the length of the messages exchanged between the *master* and *slave* processes is drastically increased (see Table 4.1.)
In the $All - to - All$ Block-CG implementation, the FLOP count from redundant operations is drastically increased with the large block size and as in the $Master - Slave$: distributed implementation the length of the messages is also increased. Furthermore, in the $All - to - All$ implementation, the long messages are more expensive because they are broadcast to all the *workers*.

5.4 Fixing the number of equivalent iterations

In the parallel Classical CG and Block-CG implementations it has been observed that there are numerical and computational issues that influence the performance of a parallel Block-CG implementation. These issues can be summarized by

- The structure of the iteration matrix (e.g., size, sparsity pattern, etc.,)
- The block size used in Block-CG
- The number of equivalent iterations
- The stopping criterion
- The computing platform
- The number of CEs used in a computer run
- The computational scheme of the parallel implementation

The first four issues are related to the numerics of the problem being solved. The influence of the computing platform and the number of CEs is studied in Sections 5.5 and 5.6.
So far when comparing the parallel Block-CG implementations, the results have been influenced mainly by some of the numerical issues. For instance, the Block-CG iterations are stopped after reducing the normwise backward error, or the residual error below a threshold value. Therefore, the iteration count has varied as the block size was also varied, and the execution time of a parallel Block-CG run depended not only on the parallel implementation but also on the iteration count.
In this section, the purpose of the experiments is to study the influence of each parallel implementation on the performance of Block-CG. The number of equivalent iterations is fixed to isolate some of the issues related with the computational scheme from the ones related to the stopping criterion.
To preserve the numerical meaning in the following experiments, Table 5.16 shows the quality of the solution after k iterations of Classical CG and Block-CG when solving the LANPRO (NOS3) problems. The block sizes of 4 and 8 are used in the parallel Block-CG.

LANPRO problem	Block size	Orthogonal vectors computed	Normwise Backward error ω
NOS3	1	304	0.14×10^{-15}
NOS3	4	304	0.15×10^{-5}
NOS3	8	304	0.62×10^{-5}

Table 5.16: Work done after k equivalent iterations of Classical CG and Block-CG.

In Tables 5.17, 5.18, and 5.19, the Block-CG iterations are fixed at 304. When the execution time of the parallel Block-CG is less than the execution time of the parallel Classical CG, a − is placed next to the execution time to represent the reduction in the execution time. In these cases, the effect of increasing the block size speeds the execution of a parallel Block-CG implementation. On the other hand, a + is placed next to the entries in the table when the execution time from the parallel Block-CG is more than the execution time of the parallel Classical CG. Entries without a − or a + sign are assumed that take almost the same time for both Block-CG and Classical CG.
Table 5.17 shows the results from this experiment with the $All - to - All$ Block-CG implementation. As observed earlier, the parallel Classical CG using an $All - to - All$ scheme simply increases the execution time as the number of CEs is increased because the cost of the operations performed in parallel is smaller than the cost of the communications. In this case the communication time has a great influence on the overall execution time, and in distributed computing environments this communication becomes bottleneck of parallelizing the Classical CG Algorithm.
In the parallel Block-CG with block sizes of 4 and 8, it is observed that the execution time is reduced as the number of CEs is increased. Increasing the block size also speeds the execution time, and in Table 5.17 this effect is seen when moving from block size of 4 to 8 on 4 CEs.

Number of CEs	Parallel Classical CG	Parallel Block-CG Block size	
		4	8
1	2.5	2.7 +	3.1 +
4	1.7	3.2 +	1.0 −
8	4.1	3.1 −	2.9 −
16	11.6	9.9 −	9.6 −

Table 5.17: Execution times from runs of the $All - to - All$ parallel Block-CG implementation. Times are in seconds.

Table 5.18 shows the results from repeating the same experiment with the $Master - Slave$ distributed Block-CG implementation. This implementation presents a better compromise between the communication that is overlapped with useful computations, and this effect is also present in the parallel Classical CG where the execution time is reduced as the number of CEs is increased.
In the same table, it is observed that the execution times of the parallel Block-CG do not changed much when changing the block size from 4 to 8. With a block size of 8 the iteration count is 38 and with a block size of 4 is 76. Thus, with the block size of 8 the number of communications is reduced by half, and this compensates for the increase in the FLOP count as the block size is increased.
In almost all of the cases reported in Table 5.18, the execution time of the parallel Block-CG is

Number of CEs	Parallel Classical CG	Parallel Block-CG Block size	
		4	8
1	3	2.9 −	3.2 +
4	1.5	1.0 −	1.0 −
8	1.5	0.7 −	0.7 −
16	1.9	0.7 −	0.6 −

Table 5.18: Execution times from runs of the $Master - Slave$: distributed parallel Block-CG implementation. Time are in seconds.

less than the execution time of Classical CG.
Table 5.19 shows the results from running this experiment with the $Master - Slave$: centralized implementation. Contrary to the other two implementations, the granularity of the problem does not compensate for the cost of communication. Thus, increasing the block size also increases the size of the sequential section and the length of the messages being sent, while decreasing the percentage of the total FLOP count that is performed in parallel.

Number of CEs	Parallel Classical CG	Parallel Block-CG Block size	
		4	8
1	4.7	4.8 +	4.7
4	2.7	2.7	3.2 +
8	2.8	2.6 −	4.1 +
16	3.0	2.7 −	4.8 +

Table 5.19: Execution times from runs of the $Master - Slave$: centralized parallel Block-CG implementation. Times are in seconds.

5.5 Comparing different computing platforms

The computing platform has an influence in the performance of a parallel implementation. The three parallel Block-CG implementations were designed for general parallel distributed computing systems, and do not include specifically tunned algorithms to enhance the performance on any specific computer system. Nevertheless, the relation between the speed of the processors and the speed of the communication network can enhance the performance of a parallel implementation, and in the case of Block-CG can even favor one implementation over another.
The purpose of this section is to perform parallel runs of the Block-CG implementations on three different computing platforms. In each computing platform, a different library subroutine was used to compute the execution times, and for some platforms the execution times are more accurate than others. Thus, we do not attempt to compare the computational performance of one computer system against another.
In Table 5.20, the results of solving the LANPRO (NOS3) problem on the SP2 computer are presented. In this case, the block size has been fixed at 4. We observe that the $Master - Slave$ distributed Block-CG implementation reports the best speedups compared to the other two implementations. However, the maximum speedup in this case is obtained with 8 CEs with an efficiency factor of only 0.46.
In the same table, it is observed that the $All - to - All$ implementation provides the maximum efficiency factor with one or more CEs. When using one CE, this implementation generates less overhead than the other two, and the execution time may be in some cases slightly longer than the execution time of the sequential Block-CG implementation.
The maximum efficiency attained with more than one CE is attained by the $All - to - All$ Block-CG implementation using 4 CEs. In the $All - to - All$ implementation, distributed reduce operations are used. In a computing platform with a fast communication network, the use of distributed reduce operations is more favorable than the use of centralized ones. However, increasing the number of CEs in a parallel implementation that uses distributed reduce operations will have a greater impact on the speedups and efficiency than increasing the number of CEs in a parallel implementation that uses centralized reduce operations. This effect is also observed in Table 5.20, when increasing the number of CEs from 4 to 16. With 8 and 16 CEs, the $All - to - All$ implementation has the lowest efficiency of the three implementations.

The results in Table 5.21 come from runs on the BBN TC2000 computer. The problem LANPRO (NOS3) is solved with a block size of 4. The processors in the BBN TC2000 are slower than the processors in the SP2. The BBN TC2000 has also a fast network switch and the combination of relatively slow processors and a relatively fast network reduces the overall expense of communication over useful computations.
This last effect is reflected in the speedups shown in Table 5.21 that have increased, for the $All - to - All$ and $Master - Slave$: distributed Block-CG implementations, from the speedups reported with the SP2 computer. As remarked earlier, in the $Master - Slave$: centralized implementation the speedup is less affected by the the speed of the network and the speed of the processors, and the S upper bounds for this implementation when solving the LANPRO (NOS3) problem was also shown in Section 5.3. The S upper bounds computed in Section 5.3 are consistent with the results in Tables 5.20, 5.21, and 5.22.

Lastly, the results from runs on the network of SUN Sparc 10 stations are shown in Table 5.22. Here the network is very slow compared with the networks in the other two computing platforms. As a result, the efficiency rates reported in the three implementations are smaller than those reported in the two previous tables.

Execution time of Block-CG sequential version : 4.8 secs									
Number CE	All-to-All			Master-Slave: Distributed			Master-Slave: Centralized		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	4.8	1.0	1.00	5.1	0.9	0.94	7.3	0.7	0.66
4	1.8	2.7	0.67	2.8	1.7	0.43	4.6	1.0	0.26
8	3.7	1.3	0.16	1.3	3.7	0.46	3.5	1.4	0.17
16	13.2	0.4	0.02	1.3	3.7	0.23	3.5	1.4	0.09

Table 5.20: Performance of parallel Block-CG implementations while solving LANPRO (NOS3) problem. Times in table are in seconds. The parallel runs were performed in a SP2 computer.

Block-CG Sequential Time : 64.9 millisecs									
Number CE	All-to-All			Master-Slave: Distributed			Master-Slave: Centralized		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	65.0	1.0	1.00	65.5	1.0	0.99	68.4	0.9	0.94
2	34.1	1.9	0.98	34.3	1.9	0.94	61.9	1.1	0.52
4	19.5	3.3	0.83	19.5	3.3	0.83	54.0	1.2	0.30
8	14.1	4.6	0.58	12.7	5.1	0.64	52.2	1.2	0.15
12	20.9	3.1	0.26	11.3	5.7	0.48	53.6	1.2	0.10
16	48.1	1.4	0.08	11.9	5.4	0.34	58.4	1.1	0.07

Table 5.21: Performance of parallel Block-CG implementations while solving LANPRO (NOS3) problem. Times in table are in seconds. The parallel runs were performed in a BBN TC2000 computer.

In this case, the use of centralized reduce operations in the $Master - Slave$: distributed implementation becomes very expensive as the number of CEs is increased. This is effect comes from the serial delivery of messages through a slow network. As shown in Table 5.22, the problem is solved faster with the $All - to - All$ implementation than with the other two implementations.

Execution time of Block-CG sequential version: 16.5 secs									
Number CE	All-to-All			Master-Slave: Distributed			Master-Slave: Centralized		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	20.0	0.8	0.82	23.7	0.7	0.69	34.4	0.5	0.48
2	12.1	1.4	0.68	15.5	1.1	0.53	23.2	0.7	0.35
3	10.0	1.6	0.55	13.3	1.2	0.41	19.7	0.8	0.28
4	9.4	1.7	0.44	29.4	0.6	0.14	29.9	0.6	0.14

Table 5.22: Performance of parallel Block-CG implementations while solving LANPRO (NOS3) problem. Times in table are in seconds. The parallel runs were performed in a network of Sparc 10 workstations.

5.6 Parallel solution of POISSON problem

The purpose of this section is to compare runs of the three parallel Block-CG implementations as in Section 5.5 after increasing the granularity of the problem to be solved. In these experiments the problem to be solved comes from the discretization of Poisson's equation. A block size of 4 is used in all the experiments in this section.
Table 5.23 presents the results from parallel runs on the SP2 computer. In the $All - to - All$ and $Master - Slave$:distributed implementations the increase in the granularity has improved their performance. This is not the case for the $Master - Slave$ centralized implementation in which the increase in the granularity has also increased the computational weight of the sequential sections.
In the $All - to - All$ implementation, the efficiency with 4 CEs is higher than the efficiency obtained in the LANPRO (NOS3) problem, and in this case the $Master - Slave$: distribute also exhibits the same efficiency with 4 CEs. In the $All - to - All$ implementation, the efficiency decreases at faster rate than in the $Master - Slave$: distributed implementation. For these two implementations, the speedups and efficiency rates are again increased when repeating this experiment on the BBN TC2000 computer.

Table 5.23 shows the results from runs of this experiment on a BBN TC2000 computer. The $Master - Slave$: distributed implementation shows a slow decrease in the efficiency rates that

Execution time of Block-CG sequential version : 13.8 secs									
Number CE	All-to-All			Master-Slave: Distributed			Master-Slave: Centralized		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	14.2	0.9	0.97	14.1	0.9	0.98	19.2	0.7	0.72
2	7.2	1.9	0.95	7.2	1.9	0.95	14.6	0.9	0.45
4	4.9	2.8	0.70	4.0	3.5	0.88	10.7	1.3	0.33
8	5.4	2.6	0.33	2.5	5.5	0.69	11.3	1.2	0.15
12	7.0	2.0	0.17	1.8	7.7	0.64	11.4	1.2	0.10
16	11.5	1.2	0.08	1.6	8.6	0.54	12.0	1.2	0.08

Table 5.23: Performance of parallel Block-CG implementations solving a Poisson's equation problem. Times in table are in seconds. The parallel runs were performed on a SP2 computer.

results in a very high speedup of 11 on 16 CEs.

The results from the same runs on the network of SUN Sparc 10 workstations are presented in Table 5.25. The times obtained when solving the LANPRO (NOS3) problem on the SUN Sparc workstation are reconfirmed with the results in Table 5.25. The $All-to-All$ Block-CG implementation has performed better than the other two implementations on the network of workstations, and in the $All-to-All$ and $Master-Slave$: distributed implementations the speedups and efficiency have increased as the granularity has also been increased.

5.7 Remarks

In this chapter we have shown the advantages of parallelizing the Block-CG Algorithm. In some cases it has been shown that the parallel Block-CG will perform better than the sequential Classical CG even when a few more FLOPs are computed with the Block-CG implementation. In distributed computing environments, the poor performance of the $Master-Slave$: centralized Block-CG implementation proves that only parallelizing the HP products does not compensate for the use of more than one CE given the small ratios between the parallel and sequential computations. However, this implementation may perform better in the event that the matrix H comes from a preconditioner that needs to be recomputed at each iteration.
Additionally, the $Master-Slave$: centralized Block-CG implementation is more trivial to implement than the other two parallel Block-CG implementations presented in this chapter, and has fewer critical sections (i.e., sections of a parallel code where only one CE is allow to update the date at a time) than its counterparts in shared memory environments because these critical

Block-CG Sequential Time : 279.1 millisecs									
Number CE	All-to-All			Master-Slave: Distributed			Master-Slave: Centralized		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	278.8	1.0	1.00	279.4	1.0	0.99	315.7	0.9	0.88
2	143.1	1.9	0.98	143.4	1.9	0.97	301.9	0.9	0.46
4	71.4	3.9	0.98	71.2	3.9	0.98	278.1	1.0	0.25
8	40.7	6.8	0.85	38.8	7.2	0.90	273.3	1.0	0.25
12	40.7	6.9	0.58	29.7	9.4	0.78	279.4	1.0	0.08
16	57.8	4.8	0.30	25.5	11.0	0.69	283.6	1.0	0.06

Table 5.24: Performance of parallel Block-CG implementations solving a Poisson's equation problem. Times in table are in seconds. The parallel runs were performed on a BBN TC2000 computer.

Execution time of Block-CG sequential version : 62.5 secs									
Number CE	All-to-All			Master-Slave: Distributed			Master-Slave: Centralized		
	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$	Time	speedup	$\mathcal{E}$
1	64.2	0.9	0.97	66.0	0.9	0.95	133.3	0.5	0.47
2	35.1	1.8	0.90	39.3	1.6	0.80	96.3	0.7	0.33
3	26.0	2.4	0.80	31.0	2.0	0.67	87.0	0.7	0.24
4	23.2	2.7	0.67	38.0	1.6	0.41	93.1	0.7	0.18

Table 5.25: Performance of parallel Block-CG implementations solving a Poisson's equation problem. Times in table are in seconds. The parallel runs were performed on a network of SUN Sparc 10 workstations.

sections appear every time a reduce operation is performed.
The granularity of the problem, the number of CEs used in a run and the characteristics of the network have a greater impact on the performance of the $All - to - All$ implementation than in the other two implementations. Therefore, the $All - to - All$ implementation will perform better than its counterparts in some cases when a fair compromise is found between the number of CEs and the granularity of the subproblems being solved.
A fair compromise can only be found after a few runs of the $All - to - All$ parallel Block-CG implementation using a different number of CEs at each run. Clearly, tuning the parallel efficiency is only worthwhile when the linear system of equations is solved more than once with different values for the coefficients of the equations each solution (these types of systems usually appear in many application fields like structural engineering, climate modeling, operations research, *etc.*)
From experiments reported in this chapter, it known that when a fair comprise is found the $All - to - All$ implementation performs better than the other two implementations in terms of the parallel efficiency (see for instance Table 5.25). On the other hand this fair compromise also depends on the computing platform being used and when running in time-sharing mode the fair compromise will also depend on the computational load of the computer system. Alternatively, the $Master - Slave$: distributed implementation has exhibit a more consistent performance even when moving from one computing platform to the other, and when solving linear systems of different sizes.
The $Master - Slave$: distributed implementation has exhibited linear speedups because of its balance between communication and computations. Moreover, the speedups are increased as the granularity of the problem is increased. Chapter 6, we use the $Master - Slave$: distributed Block-CG implementation inside a block Cimmino iteration to accelerate its convergence rate. The resulting implementation is extended to general unsymmetric sparse systems. Furthermore, inside a block Cimmino iteration the synchronization points are overlapped with useful computations.
In general, the $Master - slave$: distributed implementation suits distributed computing environments better.

Chapter 6

Solving general systems using Block-CG

The Block-CG algorithm only guarantees convergence in the solution of symmetric positive definite systems. With the use of a preconditioner the algorithm can be extended to the solution of general unsymmetric systems. As this can be done by using Block-CG as an acceleration procedure inside another basic iterative method. In this chapter, we study two iterative row projection methods for solving general systems of equations, and use the Block-CG algorithm to accelerate the rate of convergence of a row projection method.
An overview of the block Cimmino method is presented in Section 6.1, and the block Kaczmarz method is briefly introduced in Section 6.2. An implementation of the block Cimmino method accelerated with the stabilized Block-CG Algorithm is presented in Section 6.3.
General issues from a computer implementation of the block Cimmino method accelerated with the Block-CG algorithm are discussed in Section 6.4, and the corresponding parallel implementation is presented in Section 6.5.

6.1 The block Cimmino method

The block Cimmino method is a generalization of the Cimmino method (see Cimmino (1938)). Basically, the linear system of equations

$$Ax = b, \tag{6.1.1}$$

where A is a $m \times n$ matrix, is partitioned into l subsystems, with $l \leq m$, such that

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_l \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_l \end{pmatrix}. \tag{6.1.2}$$

Notice, that the term "block" in block Cimmino refers to the matrix blocks that result from (6.1.2) and not to the number of right-hand side as in Block-CG.
Figure 6.1 illustrates a geometric interpretation of the block Cimmino method in a planar case. The block method computes a set of l row projections, and a combination of these projections is used to build the next approximation to the solution.
Algorithm 6.1.1 describes the block Cimmino iteration. the matrix A_i^+ is the Moore-Penrose pseudoinverse of the submatrix A_i defined by

$$A_i^+ = A_i^T \left(A_i A_i^T \right)^{-1},$$

and $P_{\mathcal{R}(A_i^T)}$ is a projector onto the range of A_i^T given by

$$P_{\mathcal{R}(A_i^T)} = A_i^+ A_i,$$

Algorithm 6.1.1 (Block Cimmino)

(1) $x^{(0)}$ is arbitrary

(2) For $j = 0, 1, \ldots$, until convergence do:

(2.1) For $i = 1, 2, \ldots l$ do:

(2.1.1) $\delta_i^{(j)} = A_i^+ b_i - P_{\mathcal{R}(A_i^T)} x^{(j)}$

(2.2) $x^{(j+1)} = x^{(j)} + \nu \sum_{i=1}^{l} \delta_i^{(j)}$

(3) Stop

One of the advantages of the block Cimmino method is the natural parallelism in Step (2.1), and for this reason its parallel implementation for distributed computing environments is studied in Section 6.5.
In Algorithm 6.1.1, the Moore-Penrose pseudoinverse A_i^+ is used. However, the block Cimmino method will converge for any other pseudoinverse of A_i (Campbell and Meyer (1979)). Thus, we use the generalized pseudo-inverse

$$A_{\overline{G^{-1}}} = G^{-1} A_i^T \left(A_i G^{-1} A_i^T \right)^{-1},$$

where G is an ellipsoidal norm matrix.
When solving for the δ_i's in Algorithm 6.1.1, the augmented systems approach (Bartels, Golub, and Saunders (1970) and Hachtel (1974)) is used because it is computationally more reliable

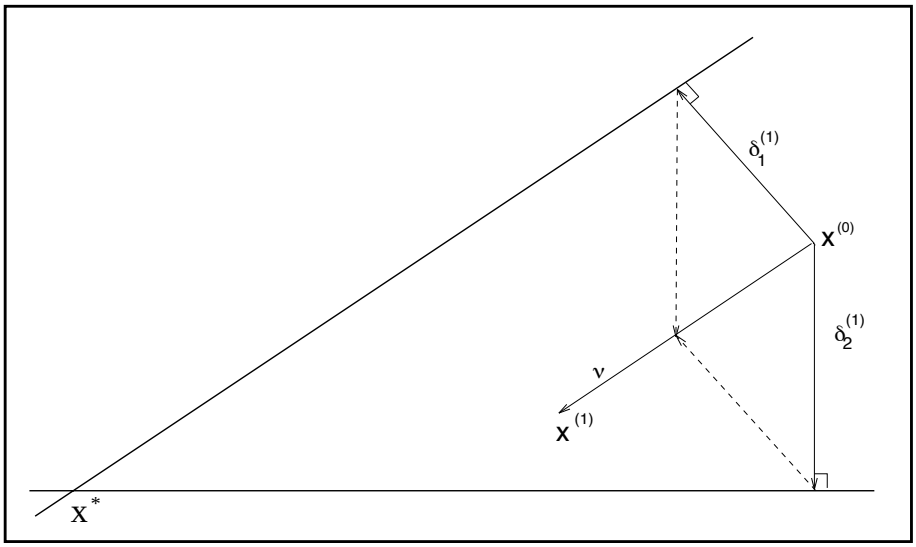

Figure 6.1: Block Cimmino geometric interpretation. In this case, a two block row partition is depicted.

and less expensive than the normal equations. Here, the augmented system equations for A_i's are written as

$$\begin{bmatrix} G & A_i^T \\ A_i & 0 \end{bmatrix} \begin{bmatrix} u_i \\ v_i \end{bmatrix} = \begin{bmatrix} 0 \\ b_i - A_i x \end{bmatrix}$$

with solution

$$\begin{aligned} v_i &= -\left(A_i G^{-1} A_i^T\right)^{-1} r_i \\ u_i &= A_{iG^{-1}}^{-}(b_i - A_i x) \\ &= \delta_i. \end{aligned} \tag{6.1.3}$$

In Step (2.2) of Algorithm 6.1.1, the choice of the relaxation parameter ν has an impact on the convergence of the algorithm.
It can be verified that the block Jacobi iteration matrix applied to the system

$$\begin{cases} AA^T y &= b \\ x &= A^T y, \end{cases} \tag{6.1.4}$$

(see Hageman and Young (1981)) is similar to the iteration matrix of the block Cimmino method (see for instance Ruiz (1992)). For this reason in a study of row projection methods, Elfving refers to the block Cimmino method as the "Block-row Jacobi" method (Elfving (1980)). In

Elfving's paper, he shows that the convergence of the block row method depends on the relaxation parameter ν and the spectral radius of the matrix defined by

$$M = \sum_{i=1}^{l} P_{R(A_i^T)} = \sum_{i=1}^{l} A_i^T \left(A_i A_i^T\right)^{-1} A_i. \tag{6.1.5}$$

If $b \in \mathcal{R}(A)$ and $x^{(0)} \in \mathcal{R}(A^T)$, then the block-row converges towards the minimum norm if and only if

$$0 < \nu < \min\left(2, \left(\frac{2}{\rho(M)}\right)\right).$$

The value of ν giving the optimal asymptotic rate of convergence is

$$\nu = \frac{2}{\mu_{max} + \mu_{min}},$$

where μ_{max} and μ_{min} are the largest and smallest nonzero eigenvalues of the matrix M.

6.2 The block Kaczmarz method

The block Kaczmarz is another row projection method and is a generalization of the Kaczmarz method (Kaczmarz (1939)). As in the block Cimmino method, the block structure from the original matrix is obtained by partitioning (6.1.1) as (6.1.2). And a block Kaczmarz algorithm can be defined as in Algorithm 6.2.1. Figure 6.2 illustrates a geometric interpretation of the Block Kaczmarz algorithm.

Algorithm 6.2.1 (Block Kaczmarz)

(1) $x^{(0)}$ is arbitrary

(2) For $j = 0, 1, \ldots$, until convergence do:

(2.1) $z^{(1)} = x^{(j)}$

(2.2) For $i = 1, 2, \ldots l$ do:

(2.2.1) $$z^{(i+1)} = \left(I - \nu P_{\mathcal{R}(A_i^T)}\right) z^{(i)} + \nu A_i^+ b^i$$

$$= z^{(i)} + \nu A_i^+ \left(b_i - A_i z^{(i)}\right)$$

(2.3) $x^{(j+1)} = z^{(l+1)}$

(3) Stop

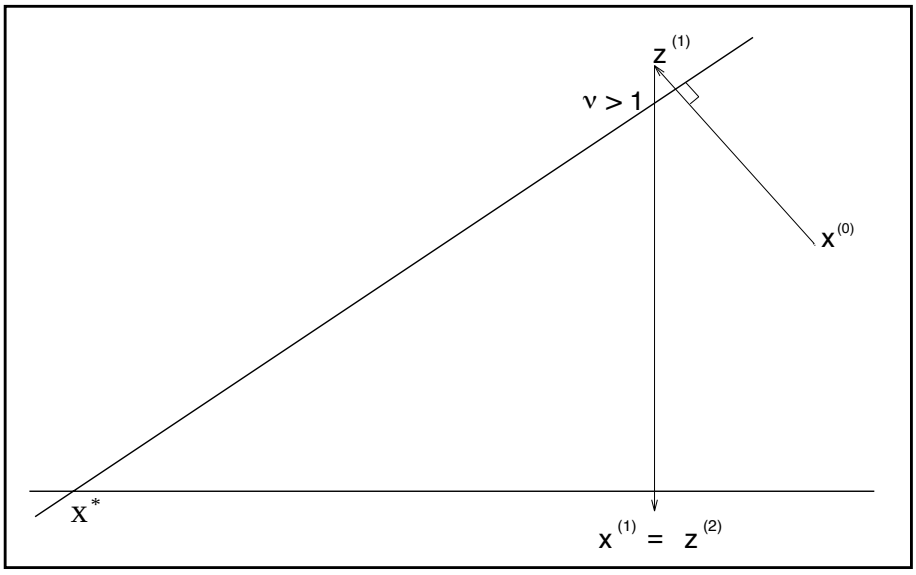

Figure 6.2: Block Kaczmarz geometric interpretation. In this case, a two block row partition is depicted.

Contrary to the block Cimmino algorithm, block Kaczmarz is sequential in nature. The Kaczmarz projections are computed one at a time in Step (2.2). This is also illustrated in Figure 6.2 in the planar case. The current iteration is projected onto the different subspaces using the updated value from one subspace to compute the next one. If ν is greater than one, then an overprojection is performed otherwise an underprojection is performed. After computing all the projections over each subspace the new iterate is obtained (Step (2.3)).
The Kaczmarz method has also been studied by Elfving (1980), and he shows that the method is equivalent to the block SOR applied to the system (6.1.4). Thus Elfving refers to the Kaczmarz method as the block-row SOR.
The relaxation parameter ν in the block Kaczmarz method does not depend on the spectral radius of the matrix as the block Cimmino method. Furthermore, the block Kaczmarz converges towards the minimum norm solution when $b \in \mathcal{R}(\mathcal{A})$ and $x^{(0)} \in \mathcal{R}(\mathcal{A}^{\mathcal{T}})$. The differences between the convergence of the block Cimmino and the block Kaczmarz methods are similar to those between the Jacobi and SOR methods for solving symmetric positive definitive linear systems (see Hageman and Young (1981)).

The iteration matrix of the block Kaczmarz method,

$$\prod_{i=1}^{l} (I - \nu P_{\mathcal{R}(A^{iT})}),$$

is not symmetric, and in general its eigenvalues are not real but complex. As for the symmetric SOR, or SSOR method, a symmetric version of the block Kaczmarz method can be derived, thus the Kaczmarz method is also referred as the block SSOR method. The symmetric block

Kaczmarz method has a symmetric matrix with real eigenvalues, and with these properties it can be accelerated with the Block-CG Algorithm. Kamath and Sameh (1988) presents a study on the robustness of the conjugate gradient acceleration for block row projection methods as the block SSOR.

One of the advantages of the block Kaczmarz method over the block Cimmino method is that the relaxation parameter is independent of the spectral radius of the matrix. On the other hand, one can improve the solution time of a system of equations by exploiting the natural parallelism of the block Cimmino method, whereas the symmetric block Kaczmarz method can only be parallelized for some matrices with a special structure in which a block partitioning will allow the computations of two or more subspaces independently and in parallel (see Arioli, Duff, Noailles, and Ruiz (1992)). Comparisons between the block Cimmino method and block SSOR method can be found in the works of Bramley (1989), Bramley and Sameh (1990), and Arioli, Duff, Noailles, and Ruiz (1989).

The Block-CG acceleration for basic stationary iterative methods does not require that the basic iterative method is convergent. In the block Cimmino method, the rate of convergence can still be slow even with the optimal ν, and this motivates the study of a procedure to accelerate its convergence rate in the next section. A similar procedure can be derived for the symmetric block Kaczmarz method (see Ruiz (1992)).

6.3 Block Cimmino accelerated by Block-CG

The block Cimmino method is a linear stationary iterative method, with a symmetrizable iteration matrix (for a definition of symmetrizable iterative method see Hageman and Young (1981), page 21). The use of ellipsoidal norms ensures the conditions that make the block Cimmino iteration matrix SPD.
Any basic iterative method for the solution of

$$Ax = b$$

may be expressed as

$$x^{(j+1)} = Qx^{(j)} + k, \tag{6.3.1}$$

where Q is the iteration matrix. The block Cimmino iteration matrix is $(I - \nu M)$. From (6.1.5) and Step (2.1.1) in Algorithm 6.1.1, the expression in (6.3.1) becomes

$$\begin{aligned} x^{(j+1)} &= (I - \nu M)x^{(j)} + k \qquad (6.3.2) \\ &= (I - \nu M)x^{(j)} + \nu \sum_{i=1}^{l} A_i^+ b_i. \end{aligned}$$

Let D be the block diagonal matrix with diagonal blocks $A_i A_i^T$, thus the matrix M can be rewritten as

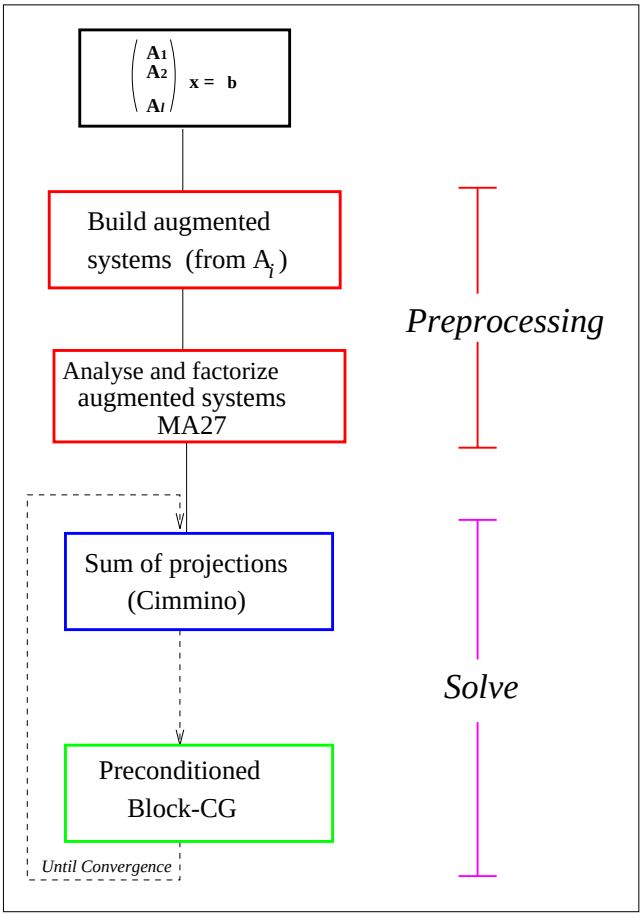

Figure 6.3: The block Cimmino method accelerated with the stabilized Block-CG algorithm.

$$M = A^T D^{-1} A \tag{6.3.3}$$

From (6.3.3), the matrix M is SPD if A is square and has full rank. The $(I - \nu M)$ is symmetric semidefinite for $\nu > 0$ and SPD if A is square and has full rank. Using the generalized pseudo inverse in (6.3.3)

$$x^{(j+1)} = \left(I - \nu M^{(G)}\right) x^{(j)} + \nu \sum_{i=1}^{l} A^{-}_{iG^{-1}} b_i$$

with $M^{(G)}$ defined as

$$M^{(G)} = \sum_{i=1}^{l} A_{\bar{iG}^{-1}} A_i.$$

A SPD block Cimmino iteration matrix can be used as a sort of preconditioning matrix for the Block-CG method. We recall that Block-CG will simultaneously search the next approximation to the system's solution in s-Krylov subspaces and in the absence of roundoff errors will converge to the system's solution in a finite number of steps. Figure 6.3 depicts the Block-CG acceleration described in Algorithm 6.3.1. And in Algorithm 6.3.1 it can be seen that the convergence of the algorithm is independent of the relaxation parameter ν.

Algorithm 6.3.1 (Block-CG acceleration for block Cimmino)

(1) $X^{(0)}$ is arbitrary, $P(0) = R(0)$

(2) For $j = 0, 1, \ldots$, until convergence do:

(2.1) $R^{(j)}$ is the pseudo residual vector,

$$R^{(j)} = \nu \left(\sum_{i=1}^{l} A_{\bar{iG}^{-1}} B_i - M^{(G)} X^{(j)} \right)$$

(2.2) $X^{(j+1)} = X^{(j)} + \lambda_j$

(2.3) $P^{(j+1)} = P^{(j+1)} + \alpha_j P_j$

(2.4) $\lambda_j = R^{(j)^T} G R^{(j)} \left(P^{(j)^T} G M^{(G)} P^{(j)} \nu \right)^{-1}$

(2.5) $\alpha_j = R^{(j+1)^T} G R^{(j+1)} \left(R^{(j)^T} G R^{(j)} \right)^{-1}$

(3) Stop

6.4 Computer implementation issues

At first, the matrix A is partitioned row-wise into l blocks according to (6.1.2). And inside a block, the columns with only zero elements are not explicitly stored.
In Figure 6.4(a), the linear system is partitioned into three blocks of rows. In Figure 6.4(b), the blocks are illustrated after discarding the columns with only zero elements. A section of columns is defined as the group of contiguous columns with nonzero elements, and all the columns in one section appeared exactly in the same blocks. In Figure 6.4(c), there are seven sections of columns. S_1 is the section of contiguous columns with nonzero elements that appeared A_1 only. S_2 is the section of contiguous columns with nonzeros that appeared in A_1 and A_2'. S_3 is the section of contiguous columns with nonzeros that are in A_1, A_2', and A_3, and so forth.
The number of sections of columns depends on the sparsity pattern of the columns of the matrix A, and this number can be very large. If the iterative scheme is designed to manipulate the

blocks by sections of columns, as is the case of this implementation, then a large number of sections of columns may prevent us from solving the system of equation in a reasonable amount of time. Therefore, an amalgamation parameter is used to scan groups of columns instead of a single column at time.
The purpose of removing the columns with only zero elements from the blocks, and clustering the columns in sections of columns is to avoid performing unnecessary operations on operands with zero value. In addition, storage space is saved because the zero elements are not explicitly stored.
After analysing the structure of the blocks, the data structures for handling the augmented systems are generated. Figure 6.5 illustrates the augmented systems from the block row partition in Figures 6.4(a-c).
After building the l augmented systems, (6.1.3) is solved using the sparse symmetric linear solver MA27 from the Harwell Subroutine Library (Duff and Reid (1983) and AEA Technology (1993)). The MA27 solver is a frontal method which computes the LDL^T decomposition. The MA27 solver has three main phases; Analyse, Factorize, and Solve.
During the MA27 Analyse phase, the minimum degree criterion is used to choose the pivots from the diagonal. The order of eliminations is represented in a elimination tree that is parsed using a depth-first search. In the MA27 Factorize phase, the matrix is decomposed using the assembling and elimination ordering from the MA27 Analyse phase. The MA27 Solve phase uses the factors generated in the previous phase to solve the system of equations.

With the generation of the augmented systems the size of the subproblems in (6.1.2) is increased, and using the sparsity pattern of the augmented systems will minimize the storage space required to store the augmented subsystems.
In general, the ellipsoidal norms matrices, G_i's, are diagonal matrices. The nonzero elements in a G_i are stored in an one dimension array of length less than or equal $c_i^{(G)}m$, where $c_i^{(G)}$ is maximum number of nonzeros in a row of G_i and $c_i^{(G)} \ll m$. For A_i and A_i^T, only A_i needs to be stored since A_i^T can be implicitly obtained from A_i. The A_i matrices are stored in a general sparse matrix format.
The zero blocks in the lower right corner of each augmented system are not explicitly stored in memory. Thus, the only increment in the storage space requirement comes from the nonzero elements in the ellipsoidal norm matrices.

The solve phase of the Block-CG acceleration is depicted in Figure 6.3 and this is an implementation of Algorithm 6.3.1. At each iteration, the augmented systems are solved to obtain the δ_i's. The sum of the δ_i's is used as the residual vector in the Block-CG iteration and this procedure is repeated until convergence is reached.

6.5 Parallel implementation

This section deals with the parallel distributed implementation of the block Cimmino method with a Block-CG acceleratioin studied in Section 6.3. The parallel implementation of Block-CG acceleration is designed for general distributed computing environments including heterogeneous computing environments.
The $Master - Slave$: distributed parallel Block-CG implementation presented in Section 4.2.2 is used inside the parallel Block-CG acceleration. A few additions are made to the parallel

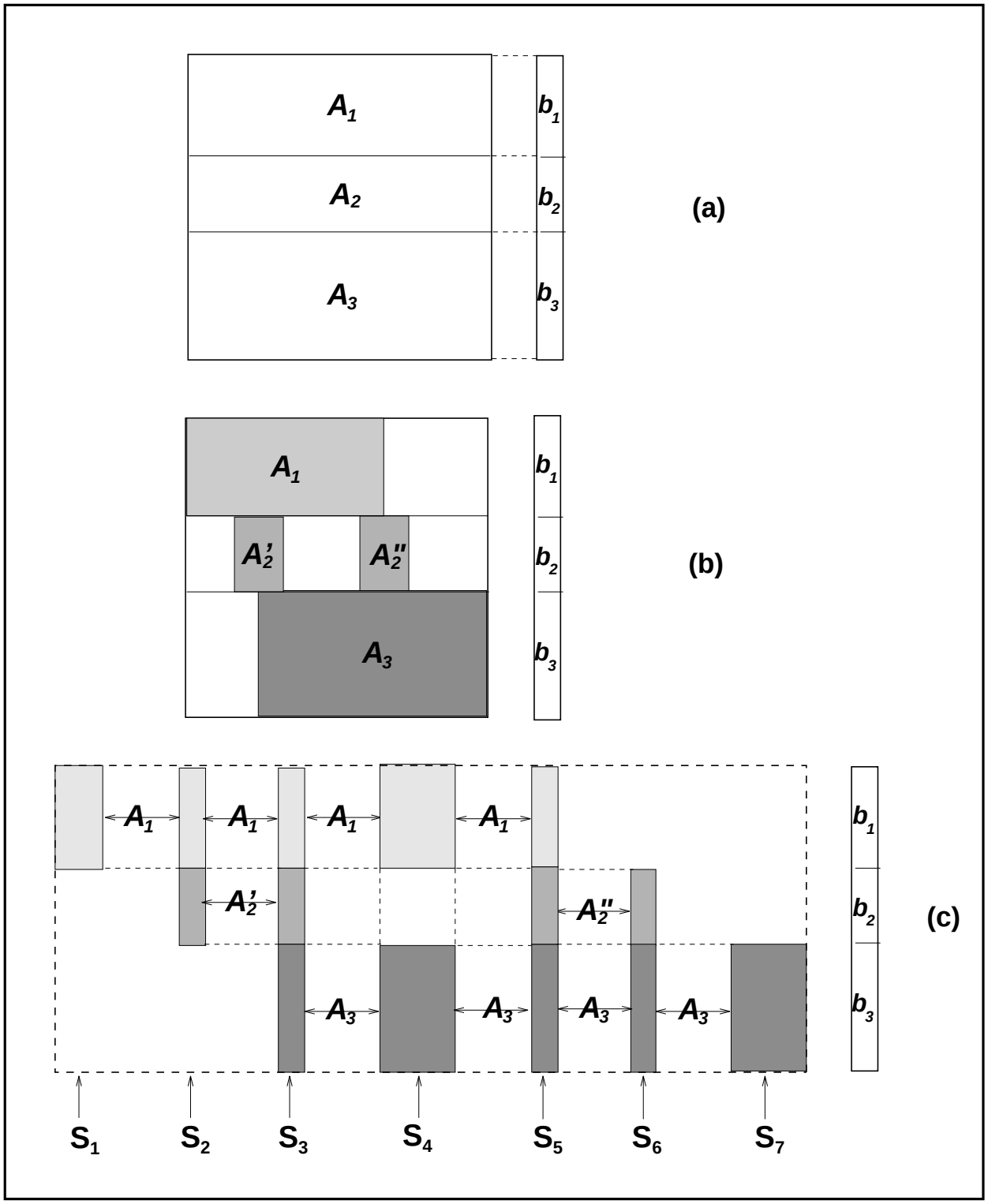

Figure 6.4: Row partitioning of the original linear system of equations and identification of column sections. In (a) resulting subsystems from 6.1.2. In (b) the same subsystems after identifying the blocks of nonzeroes in each subsystem. (c) Identification of column sections.

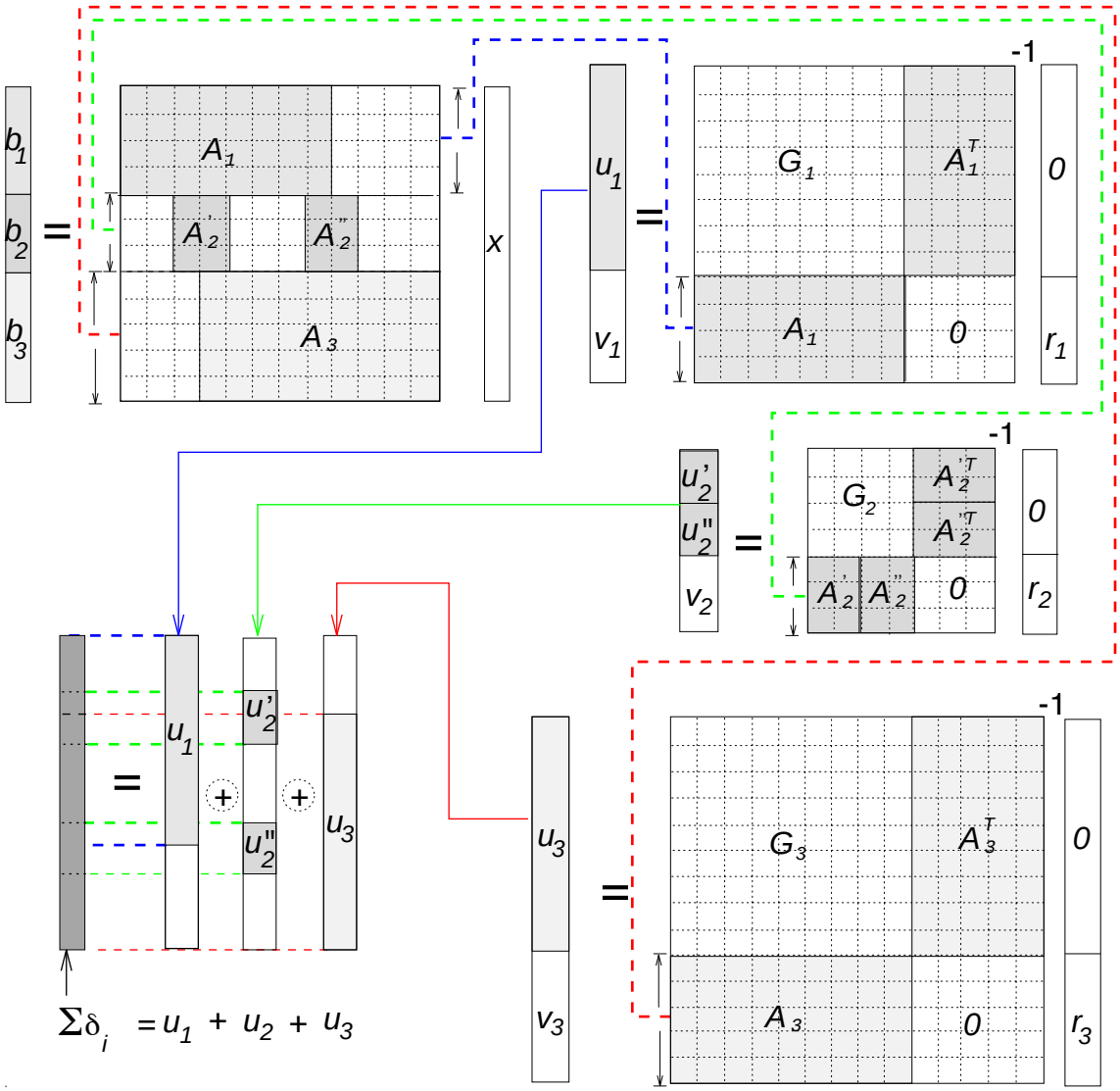

Figure 6.5: Building the augmented systems from resulting subsystems in 6.1.2. Illustration of the augmented systems corresponding to the example of system partition presented in Figure 6.4

Block-CG implementation to accommodate the used of the Cimmino iteration matrix.
Figure 6.6 illustrates the computational flow of the parallel implementation of the block Cimmino method accelerated with the Block-CG algorithm.

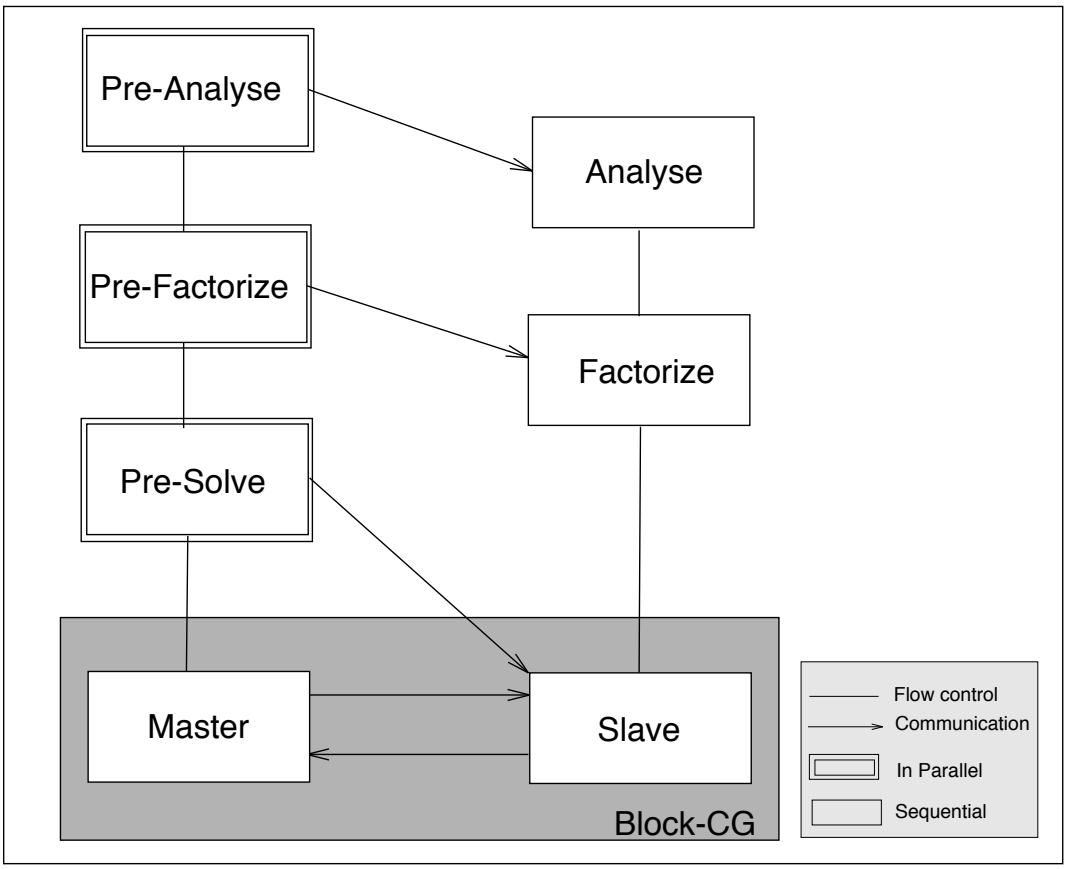

Figure 6.6: Scheme of the parallel Block-CG acceleration for the block Cimmino method.

In the Pre-Analyse phase, the *master* process is responsible for partitioning the system of equations, and identifying the sections of columns. These partitions are specified by the user as an input parameter.
With the sections of columns, the *master* is able to identify data overlaps between the different blocks. This information is later transmitted to the *slave* processes, and with this information the *slave* processes build their local information about their computational neighborhood (e.g., identification of neighbor *slave* processes, and data it needs to communicate to each neighbor *slave* process).
Before the *master* process sends informtion to the *slave* processes, the *master* process calls a scheduler to evenly distribute the workload among the *slave* processes, and this distribution is based on the partitioning of the system and data overlappings between the sections of columns. After scheduling the solution of the different subsystems, the *master* process dispatches to each *slave* information from one or more subsystems. During this first dispatch, the *master* process

sends to the *slave* processes the sparsity pattern of each augmented system, and the description of the corresponding sections of columns.
In the MA27 Analyse phase only the structure of the matrix is used. Later, the numerical values are used in the MA27 Factorize phase. For this reason during the first dispatch the *master* process sends only the structure of the augmented systems to the *slave* processes, and each *slave* process calls the routine that implements the MA27 Analyse phase on one or more augmented systems. While the *slave* processes call the routine that performs the MA27 Analyse, the master prepares the data messages for the parallel numerical factorization of the subsystems.
Once the messages are prepared, the *master* process sends to each *slave* process the numerical data that corresponds to each augmented system. In this way the second dispatch of information is expected to overlap with the parallel computations of the MA27 Analyse phase that is performed in parallel by the *slave* processes.
With the numerical information, the *slave* processes call the routine that performs the MA27 Factorize phase, and at the same time the *master* process prepares the messages with the right-hand side values and sends them to the *slave* processes.
Lastly, the *master* and *slave* processes take their roles in the Block-CG computations as described in the $Master - Slave$: distributed parallel Block-CG implementation (from Section 4.2.2). However, some procedures need to be modified to accommodate the use of the block Cimmino iteration matrix in Block-CG.
Basically, there are two main differences between the data manipulation in the parallel Block-CG and the one in the parallel Block-CG acceleration. As discussed in Section 4.1.1 the system of equations is partitioned column-wise in Block-CG and it is partitioned row-wise in the block Cimmino method, and to overcome this difference the blocks of rows are manipulated in sections of columns (see Figure 6.4).
Secondly in the $Master - Slave$: distributed Block-CG implementation, there is a distributed reduce operation in which the *slave* processes exchange the results from partial products and each *slave* builds locally a part of the global product. Furthermore, the use of (4.1.3) helps to identify the contributions of each *slave* process to the global solution. In the Block-CG acceleration, (4.1.3) cannot be used because the matrix A may not have a full diagonal.
Two solutions to this problem are considered. The first one distributes *section ownership* roles between the *slave* processes. Thus, a *slave* process monitors a reduce operation on one or more sections of columns. In other words, the distributed reduced operation becomes a centralized reduced operation, and the *slave* process that *owns* a section of columns is responsible for collecting local results from neighbors and broadcasting back the global results.
The number of sections of columns varies with the sparsity pattern of the matrix A and the amalgamation parameter discussed in Section 6.4. A large number of sections of columns will congest the networks and degrade the overall performance of the parallel implementation.
The assignment of the *section ownership* roles is not trivial because the parameters that determine the computational weight involved in a centralized reduce operation are only known after a few iterations of the Solve phase. Calibrating the workload after a few iterations of the Solve phase is computationally expensive in the case of the Block-CG acceleration because of all the local data structures that have already been generated by a *slave* process.
Alternatively, a distributed reduce operation can be used at the expense of redundant computations that are performed locally by two or more *slave* processes. In this case, each *slave* process manipulates the data it receives from its neighbor processes and eliminates the redundant data while building a part of the global solution.

In comparison with the parallel Block-CG implementations (see Chapter 4), the granularity of the parallel Block-CG has been increased with the use of the block Cimmino iteration matrix. The increase on the granularity, and the effects of the overlaps between sections require a better scheduling strategy than the scheduling strategy presented in Chapter 4 to overcome the communication bottlenecks and benefit from the parallel scheme that overlaps communications with useful computations.

Chapter 7

A scheduler for heterogeneous environments

One of the advantages of working in heterogeneous computing environments is the ability to provide certain level of computing performance proportional to the number of resources available in the system and their different computing capabilities. A scheduler is regarded as a strategy or policy to efficiently manage the use of these resources. In parallel distributed environments with homogeneous resources, the level of performance is commensurate with the number of resources present in the system.
A scheduler for parallel iterative methods in heterogeneous computing environments is presented in this chapter. An overview of some scheduling techniques is introduced in Section 7.1. In Section 7.2 we present a static scheduler for heterogeneous environments and justify its application in parallel Block-CG like iterative methods. The different modules of the scheduler can be reused for other parallel iterative, direct or semi-direct methods.
In heterogeneous computing environments, the scheduler not only considers information from the tasks to be executed in parallel but it must also consider information about the capabilities of the CEs. In Section 7.3, a syntax for specifying heterogeneous environments is defined.
Finally, in Section 7.4 an example of an application of the scheduler is presented. In the example, the scheduler distributes a set of tasks arising from the parallel Block-CG acceleration of the block Cimmino method presented in Chapter 6.

7.1 Taxonomy of scheduling techniques

There is a great variety of procedures for solving the problem of distributing a set of resources to a group of consumers in an optimal manner, and this problem is known to be NP-complete (see Garey and Johnson (1979)). In general, all these procedures receive different names depending on the discipline in which they are used. In the context used here, the scheduler can be characterized by a mapping function that assigns work to a set of CEs, thus in some literature the problem has been referred as the mapping problem (see for instance Heddaya and Park (1994), Talbi and Muntean (1993)).
Several authors have classified the different scheduling strategies in a taxonomy. A taxonomy of different scheduling strategies is presented in Figure 7.1, which is an extension of the taxonomies presented by Casavant and Kuhl (1988), Talbi and Muntean (1993). In this chapter, a static greedy scheduling strategy is used and a short overview of all of the strategies in Figure 7.1 will

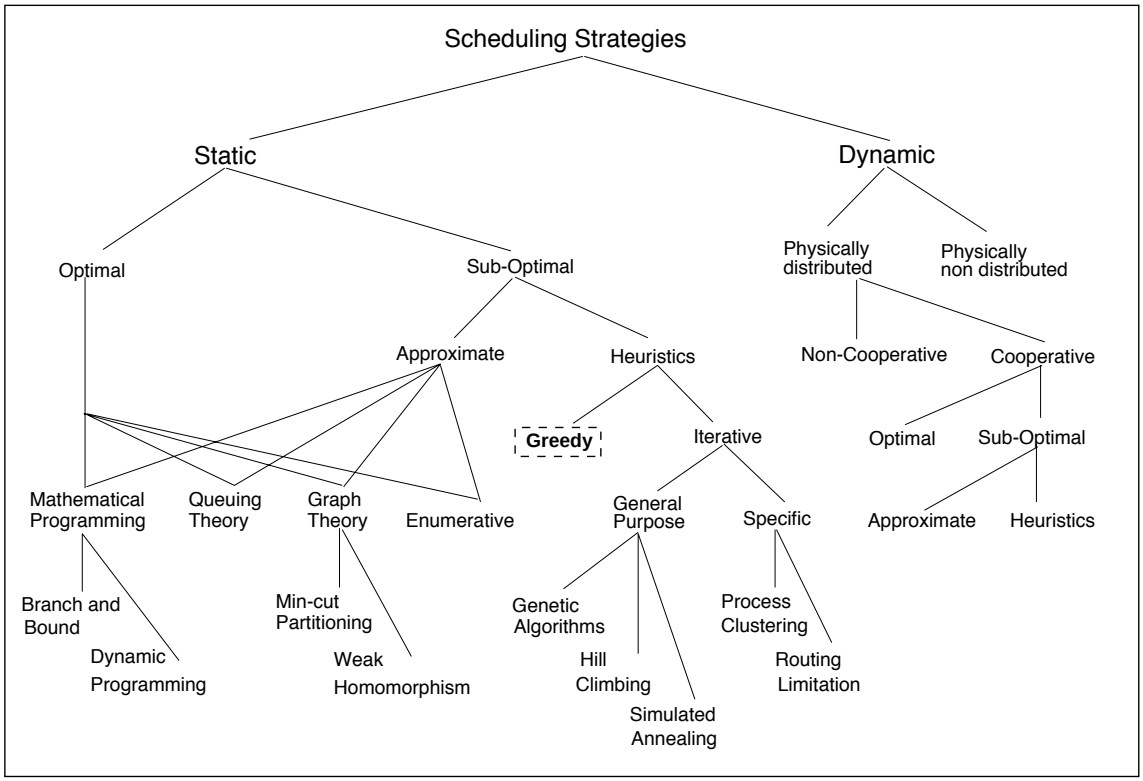

Figure 7.1: Taxonomy of some scheduling strategies

help to understand our choice for a static greedy scheduling strategy.

The first classification in the taxonomy illustrated in Figure 7.1 is related to the time the scheduling strategy is applied. In a pure static strategy, the scheduling decision is made before the parallel processing starts. In a dynamic strategy, decisions for the distribution of tasks are constantly made throughout the parallel processing. Therefore, a dynamic strategy is preferred when dynamic creation of tasks or allocation of processors is allowed.

In a dynamic scheduling strategy very little information is known about the needs of the tasks and characteristics of the computing resources. Dynamic strategies can be performed distributed when many processors can run the scheduling strategy (see Gao, Liu, and Railey (1984)), or non-distributed in which the dynamic scheduling strategy is centralized in a single processor (see Stone (1978)).

The classification of cooperative or non-cooperative relates to whether or not the different processors consult one another while distributely performing a dynamic scheduling strategy. In the cooperative case a global optimal scheduling solution can be sought. In some cases, the high cost of computing an optimal scheduling solution can be reduced by accepting a relatively good scheduling solution, and in this case a sub-optimal scheduling strategy is used. The same scheduling strategies under the static sub-optimal classification can be used for the dynamic sub-optimal one.

Here, a static scheduling strategy is considered because in iterative methods generally it is possible to collect a priori information from the parallel tasks before the parallel processing phase begins. Also, data locality is an important issue for several iterative methods and moving data

around will only increase the communication cost that represents already a bottleneck in the performance of some parallel iterative methods.
In a static optimal scheduling strategy the assignment of tasks is based on an optimal criterion function which may involve the minimization of the total execution time, or maximization of the usage of available resources in the system (see for instance, Bokhari (1981b), Gabrielian and Tyler (1984), Shen and Tsai (1985)). The size of the search space in this problem grows exponentially with the number of resources and tasks involved in a parallel run. Furthermore, the problem has been proven to be NP-complete when the number of processors goes to infinity (see Chretienne (1989)).
Our efforts are now reduced to sub-optimal scheduling strategies because of the exponential cost of optimal scheduling strategies. A sub-optimal strategy can be based on a search for a good approximation to the solution inside a given search space. Several search algorithms have been developed under this principle and some of these algorithms are used in static optimal scheduling strategies. These strategies are not studied in more detail here and the reader is referred to the following studies of static approximate strategies

- Mathematical-Programming (Bokhari (1981b), Gabrielian and Tyler (1984), Ma, Lee, and Tsuchiya (1982))
- Queuing-Theory (Kleinrock (1976), Kleinrock and Nilsson (1981))
- Graph-Theory (Bokhari (1979), Shen and Tsai (1985))
- Enumerate (Shen and Tsai (1985)).

Static heuristic scheduling strategies make the most realistic assumptions about a priori knowledge concerning the tasks, and individual performance of each processor (see for instance Efe (1982)), and for this reason they are preferred in this work over static approximate strategies.
In a static greedy scheduling strategy, part of a scheduling solution is given and this solution is expanded until a complete schedule of tasks is obtained. Only one task assignment is made at each expansion. On the other hand, iterative based scheduling strategies are initialized by a complete schedule of tasks and this schedule is improved through iterations. At the end of each iteration the schedule is evaluated using a function, and the output of this function is tested against a stopping criterion (e.g., minimum execution time, maximum utilization of allocated processors).
Depending on the function that evaluates the current schedule, an iterative strategy can be based on some well known general purpose strategies or specific scheduling solutions that have been proposed for certain problems and in Figure 7.1 the instances of Specific scheduling strategies are Process Clustering (Lo (1988)), and Routing limitation (Bokhari (1981a)) which are not further discussed here. Some examples of General Purpose strategies are Genetic Algorithms, Hill Climbing, and Simulated Annealing.
Genetic algorithms are stochastic search techniques introduced by Holland (1975), and they are regarded as optimization algorithms based on a space search in which each point is represented by a string of symbols. Each string combination is assigned a value based on a fitness function. The algorithm starts with a basic set of initial points in the space called the basic population, a new set of points is generated at each iteration. The process is called reproduction because the new generated points are combinations of the current population. Then, some of these new generated points are discarded using the fitness function for the elimination criteria. From the

points that survive, the algorithms generate a new sequence and this iteration is repeated for a given number of generations.
Hill-Climbing algorithms (Johnson, Papadimitriou, and Yannakakis (1985)) find a global minimum only in convex search spaces. It starts with a complete schedule, and tries to improve it by local transformations. At the end of each iteration the new schedule is evaluated and if the cost of the move from the old schedule to the knew one is positive the move is accepted. This process is repeated until there are no more possible changes to the scheduling solution that will further reduce the cost of the function. Thus, a local minimum is found rather than a global minimum.
Simulated annealing algorithms scan a search space using Markov chains to apply a sequence of random local transformations to a system that has been submitted to a high temperature. These transformations affect the temperature in the system until a state of equilibrium is reached. Simulated annealing algorithms (Kirkpatrick, Gelatt, and Vecchi (1983)) are sequential in nature and difficult to parallelize (Greening (1990)), and scheduling strategies based on these algorithms are more costly than the two previous iterative strategies (Talbi and Muntean (1993)).

7.2 A static scheduler for block iterative methods

In this section, we propose a scheduler for synchronous block iterative methods. The proposed scheduler has a number of parameters that are tuned to suit the particular scheduling needs from a block iterative method. In the taxonomy illustrated in Figure 7.1, the proposed scheduler is classified as a static greedy scheduling strategy.
A static greedy strategy is preferred for the following reasons:

- In many synchronous block iterative methods, a task is associated to a partition of the system of equations and these partitions are preserved through the iterations. Thus, no dynamic tasks are generated and there is no advantage of using a dynamic scheduling strategy over a static one,

- Finding an optimal distribution of the workload is ideal but is not a necessary condition for parallel block iterative methods. Furthermore, optimal strategies are more expensive to implement and compute than the sub-optimal ones,

- As discussed earlier, heuristic strategies are preferred over approximate ones because heuristic strategies make the most realistic assumptions about a priori knowledge concerning the tasks, and the CEs,

- A greedy strategy is preferred over the iterative ones because iteratives scheduling strategies are generally more expensive to compute than greedy, and it is not always possible to find a fitness function, or a good criterion to stop iterations.

In general, the scheduling problem for block iterative methods on heterogeneous computing environments is first approached by a symbolic description of the partition of problem to be

solved (system of equations) and the heterogeneous computing environment.
Without loss of generality, the linear system of equations

$$Ax = b \tag{7.2.1}$$

can be divided into l subsystems of equations, and the resulting subsystems are not necessarily mutually exclusive (e.g., some equations can be repeated in more than one subsystem).
In general, (7.2.1) can be replaced by a different expression when solving a different type of problem with a block iterative method.
The division of the system (7.2.1) into the l subsystem can be represented by an undirected graph of subsystems given by

$$G_s = (V_s, E_s), \tag{7.2.2}$$

where V_s is a set of the vertices $\{V_{s_i},\ i = 1, 2, \ldots l\}$, and there is a vertex per subsystem of equations. E_s is the set of edges that represent a row or column overlap between subsystems. If any two subsystems have rows or columns that overlap, then there is an edge between them and depending on the number of rows or columns an integer value is associated with each edge to represent the potential communication between the two subsystems. The value associated with an edge is greater than zero, and edges with a zero value are reserved for subproblems that are mutually exclusive, thus the edges are not explicitly included in the graph.
In the other hand, the heterogeneous computing environment is described as a metacomputer with p CEs. CEs are identified by some computing characteristics that differentiate one from the other (e.g., computer name, computing speed, number of processors, architecture, *etc.*).
Similarly to the graph of subsystems, a graph of CEs is defined by

$$G_{ce} = (V_{ce}, E_{ce}), \tag{7.2.3}$$

where V_{ce} is the set of vertices $\{V_{ce_i},\ i = 1, 2, \ldots p\}$, and each vertex represents a CE in the metacomputer. Moreover, there are three different types of vertices. V_{ce_R} is identified as the root vertex for monitoring the parallel processing in the metacomputer. Usually, the parallel environment is generated from the CE represented by root vertex.
A vertex in G_{ce} can also be a single processor or a cluster of processors. For the latter case, the cluster is represented by a undirect subgraph of CEs, $G_{ce_i} = (V_{ce_i}, E_{ce_i})$, that expresses the interconnection between the CEs inside a cluster. A cluster can be shared or distributed memory.
In 7.2.3, E_{ce} is the set of edges in the metacomputer's graph, and each edge represent the interconnection network between the different CEs. Figure 7.2 depicts an example of a CE graph. In Figure 7.2, G_{ce_2} is a distributed cluster and V_{ce_5} to V_{ce_8} are the CEs inside this cluster. G_{ce_4} is a shared memory cluster, and in this case is represented by a graph with no edges since the CEs are connected by a pool of memory rather than a physical communication network.

The scheduling strategy can be defined as a mapping function

$$f : V_s \longrightarrow V_{ce} \tag{7.2.4}$$

such that

$$f(v_{s_i}) = v_{ce_j}, \quad i = 1, 2, \ldots l, \quad \text{and} \quad j = 1, 2, \ldots p.$$

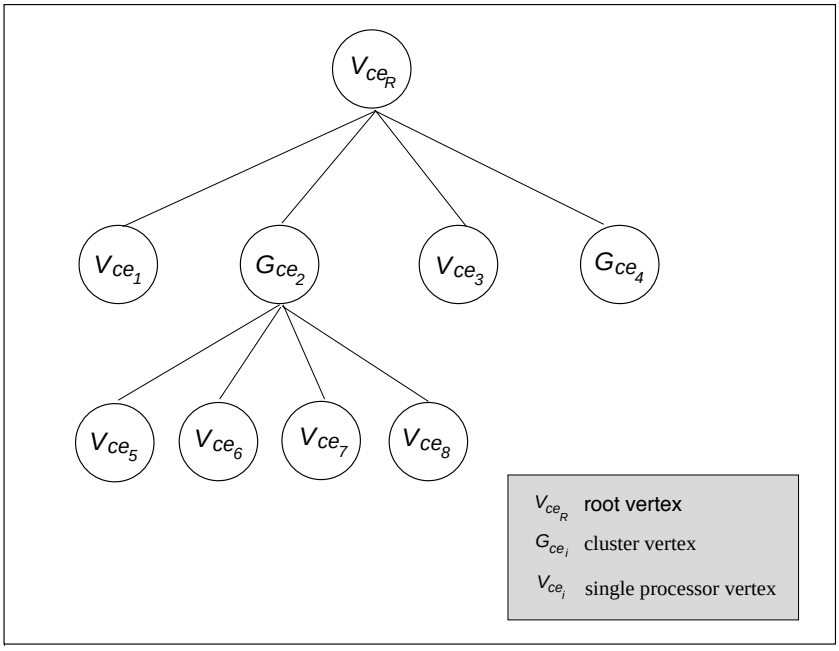

Figure 7.2: An example of a graph of CEs.

Some authors have studied (7.2.4) as continuous function (for instance Prasanna and Musicus (1991)) with an infinite number of CEs, and any fraction, between 0 and 1, of a CE can be used to solve a subsystem. However, in practice (7.2.4) is taken to be a discrete function and a subsystem can only by assigned to one CE.
To balance the workload distribution each v_{s_i} is assigned a computational weight using a discrete function defined as

$$W : V_s \longrightarrow \mathcal{N}$$

$\mathcal{N}$ is the set of natural numbers. Then

$$W(v_{s_j}) = \sum_{i=1}^{r} \lambda_{j_i} \theta_{j_i}, \tag{7.2.5}$$

where $\lambda_{j_i} \in \mathcal{R}$ are scalars and θ_{j_i} are parameters relevant to the subproblem (e.g., number of nonzeros, size of the subproblem, etc.). The λ_{j_i} scalars control the significance of the θ_{j_i} parameters. Similarly, a function to measure different computational aspects from the CEs is defined. The function outputs a number that is used as a priority for assigning work to the CEs, such that a CE with a highest priority will be assigned tasks before a CE with a lower priority.

$$P : V_{ce} \longrightarrow \mathcal{N} \tag{7.2.6}$$

$$P(v_{ce_j}) = \sum_{i=1}^{r} \gamma_{j_i} p_{j_i},$$

the $\gamma_{j_i} \in \mathcal{R}$ are scalars that control the significance of the p_{j_i} parameters or specifications from each CE.
Additionally, a workload function is defined as

$$\Omega : V_{ce} \longrightarrow \mathcal{N} \tag{7.2.7}$$

such that

$$\Omega(v_{ce_j}) = \sum_{i=1}^{l_j} W(v_{s_{j_i}})$$

$\left\{v_{s_{j_i}}\right\}_{i=1}^{l_j}$ is the set of subsystems assigned to v_{ce_j}.

A static greedy scheduling strategy is described in Algorithm 7.2.1 for distributing tasks to a group of heterogeneous CEs.

Algorithm 7.2.1 (A Static Scheduling Algorithm)

(1) Sort in descending order the v_{s_j} according to $W(v_{s_j})$,
l = number of v_{s_j}'s

(2) p = number of CEs in the metamachine,
if $(l < p)$ then $p = l$

(3) Sort in descending order the v_{ce_j} according to $P(ce_j)$

(4) for each $v_{ce_j} \in V_{ce}$ do:

(4.1) Assign v_{s_j} to v_{ce_j}, and update $\Omega(v_{ce_j})$

(5) for each $v_{s_i} \in V_s$ not yet assigned do:

(5.1) Find best $v_{ce_j} \in V_{ce}$ to work on v_{s_i}
considering E_s and $\Omega(v_{ce_j})$

(6) Stop

The purpose of considering E_s in Step (5.1) of Algorithm 7.2.1 is to minimize the communication between the CEs and the purpose of considering $\Omega(v_{ce_j})$ in the same step is to keep the workload distribution balanced. However, the order in which these two parameters are considered leads to two different scheduling strategies.
If the set of edges is considered first, then the priority is to reduce the communication between the CEs. In this case, an attempt is made to assign to the same CE two or more subsystems that potentially will communicate a lot, the $\Omega(v_{ce_j})$ parameter is also used here to avoid overloading a CE.
If the $\Omega(v_{ce_j})$ are considered first, then the priority is to balance workload among the CEs. For each task in the not-yet-assigned list of tasks, the processor with the lowest $\Omega(v_{ce_j})$ is sought, and the set of edges is used to resolve conflicts between two or more CEs with the same computational weights.

7.3 Heterogeneous environment specification

The heterogeneous environment is described in an input file or *hesfile* (for **h**eterogeneous **e**nvironment **s**pecification file). A hesfile contains a line per CE in the metamachine. Each line specification contains a list of keywords and values from the following syntax,

hostname	[**ct**	or	**CT** = DISTRIB, SHARED, SINGLE]
	[**nm**	or	**NM**= *hostname1,* [*hostname2, ..*]]
	[**np**	or	**NP** = *integer number*]
	[**sp**	or	**SP** = *integer number*]

the keywords and corresponding values between squared brackets (e.g., [ ],) are optional. The hostname is mandatory and if none of the other keywords are specified the default values are used.
CT specifies the cluster type, and the default value for this keyword is SINGLE for a single processor. The SINGLE value is provided for declaration of workstations.
If the cluster type is set to DISTRIB (e.g., ct = DISTRIB), then the *hostname* is a distributed memory cluster. A distributed memory cluster is used for specifying a distributed memory machine, a virtual shared memory machine, or a subset of a network of workstations that are connected through a special communication network.
In the last case of the distributed cluster, the CEs are addressed independently, thus the keyword **NM** is used for declaring a list of machines names in the distributed cluster.
The number of CEs in a cluster is specified with the keyword **NP**, and the default is one.
The keyword **SP** is used for specifying the speed of a CE or an output value from 7.2.6.

7.4 A scheduler for a parallel block Cimmino solver

Basically, the different phases of the parallel Block-CG acceleration are depicted in Figure 6.3. The basic iteration matrix in this case is the block Cimmino iteration matrix, thus the subproblems are obtained after l row partitions of the system of equations (7.2.1), and the computational weight factors are computed as follows,

$$
\begin{aligned}
W(v_{s_i}) &= \alpha_1\theta_{i_1} + \alpha_2\theta_{i_2} + \alpha_3\theta_{i_3} \\
\theta_{i_1} &= \text{size of the subsystem of equations} \\
\theta_{i_2} &= \text{number of nonzeros} \\
\theta_{i_3} &= \text{number of edges}
\end{aligned}
$$

The scalars α_1, α_2, and α_3 determine the relevance of one parameters against the others.
The values associated to the edges in G_s are the number of columns that overlap between pairs

of subsystems.
The parallel implementation of the Block-CG acceleration is based on the *Master – Slave* distributed Block-CG implementation, thus the CEs communicates through messages. The PVM 3.3 system library has been used for handling the message passing. Thus, the syntax of the hesfile accepts all the PVM 3.3 keywords defined for a hostfile specification (see Geist, Baguelin, Dongarra et al. (1994)).
PVM 3.3 provides standard information from each CE in a data structure called **pvmhostinfo**, and this information can be modified through the PVM keywords. From the information in the hesfile and the information supplied by the **pvmhostinfo**, the nodes of the G_{ce} graph are generated. Figure 7.3 depicts a node from the G_{ce} graph.

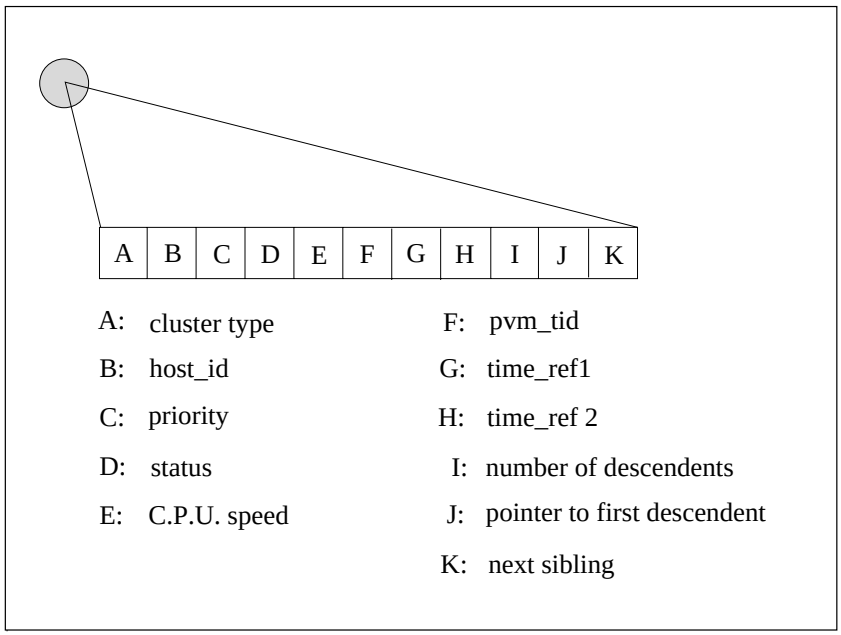

Figure 7.3: Structure of a node in the G_{ce} graph.

The fields host_id and pvm_tid are included to interface the solver with the PVM system. The cluster type is provided by the user in the hesfile. The fields number of descendents, the pointer to first descendent, and next sibling are provided for traversing the G_{ce} graph.
The fields status, time_ref1 and time_ref2 are used for monitoring the execution of a CE. The priority field is set after computing (7.2.6). The speed field is taken from the PVM system by default, however the user can specify it directly with keywords from the hesfile or PVM hostfile. Algorithm 7.2.1 is used to schedule the solution of the subsystems on one of the CEs. The priority is given to the communication because of the synchronization points of the Block-CG method.

The partitions of the system of equations (7.2.1) can be uniform or nonuniform. If the system of equations is uniformly partitioned, then each subsystem will have the same number of rows, and the computational weights will not vary much from one subsystem to the other. In this case

an approximate average workload per processor is defined by

$$\overline{w} = \frac{\sum_{i=1}^{l} W(v_{s_i})}{\sum_{i=1}^{p} Nprocs(v_{ce_i})} \tag{7.4.8}$$

$Nprocs$ is a function that returns the number of processors specified in the declaration of each CE.
On the other hand, if the system is nonuniformly partitioned, then the sizes and computational weights from one subsystem to another may vary a lot. In this case, an upper-bound for the computational workload is defined by

$$[(\overline{w} \times l) + (\overline{w} \times \delta)] \times Nprocs(v_{ce_i}) \tag{7.4.9}$$

with $\delta \in \mathcal{R}$, and $0 \leq \delta \leq 1$.
Other authors have suggested tuning the workload distribution by minimizing the workload imbalanced after each assignment of a subsystem (see for instance Talbi and Muntean (1993)). However, in the parallel block Cimmino accelerated with the Block-CG algorithm, the priority for assigning subsystems to the CEs is on the values associated with the edges in G_s, thus we suggest using (7.4.8 or 7.4.9) to keep the workload balanced.

Chapter 8

Scheduler experiments

The purpose of this chapter is to validate the use of the static greedy scheduler proposed in Section 7.4 (also in Arioli, Drummond, Duff, and Ruiz (1995)). The following experiments were run on a network of SUN SPARC 10 and IBM RS6000 workstations as shown in Table 8.1.

ID	Computing Element	Number of Processors
A	Cluster IBM RS6000 With FDDI Network	3
B	IBM RS6000 350	1
C	IBM RS6000 320H	1
D - H	SUN SPARC 10	1

Table 8.1: List of available computing elements.

An unsymmetric non-diagonally dominant matrix that comes from a finite volume discretization of the Navier Stokes equation coupled with chemistry is used in the numerical experiments. The discretization is performed using a curvilinear mesh of 69 by 60 points on five variables, and this leads to a matrix of order $69 \times 60 \times 5 = 20700$. The original system is partitioned into 20 blocks. For the first experiment, the system of equations is partitioned as described in Table 8.2 and in the second experiment the partition of the system of equations is shown in Table 8.3. The resulting G_s for the first and second partition are depicted in Figures 8.1 and 8.2, respectively.

Each subsystem will become a task to be performed by one of the computing elements in the metacomputer. Three different scheduling strategies are compared in the following sections to analyse the influence of the scheduling strategy on the overall execution time of the parallel block Cimmino solver accelerated with the Block-CG algorithm.
The first strategy is a sequential distribution of subsystems to CEs, in which subsystems are distributed one by one to the next CE in a circular queue of CEs. In this case neither communication nor workload balancing issues are considered.
The second strategy uses Algorithm 7.2.1 and in Step 5 the focus is to balance the workload among the CEs. In this strategy, the CE with the lowest workload value is the candidate for

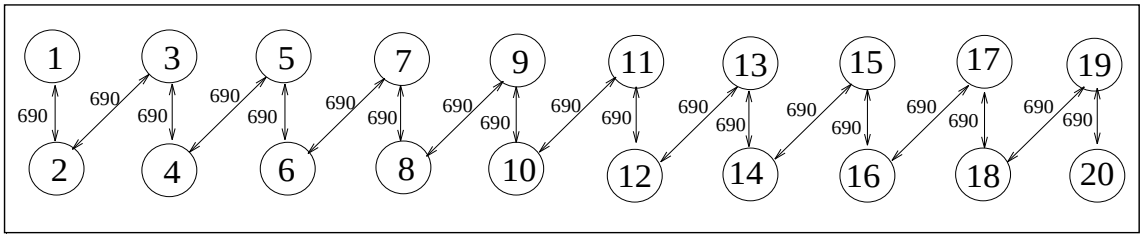

Figure 8.1: Graph of subsystems according to first partition.

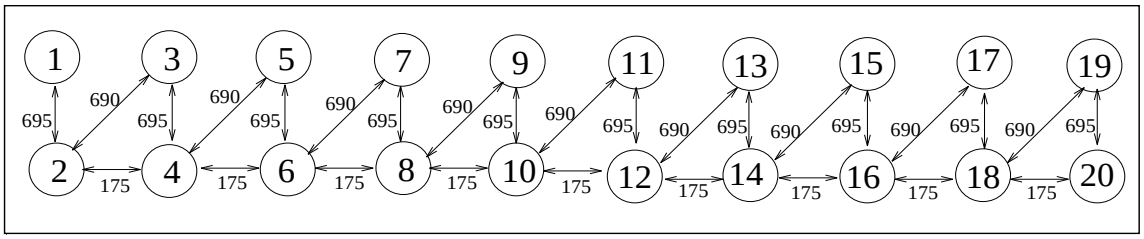

Figure 8.2: Graph of subsystems according to second partition.

solving the subsystem in turn. If there is more than one candidate the subsystem is scheduled to the CE that has been previously assigned a set of subsystems with the highest potential of communication with the subsystem in turn.
The purpose of the third strategy is to reduce the communication cost per iteration by scheduling two or more subsystems with a high potential of communication to the same CE. Using 7.4.9 to find upper bounds for the workload per CE. The third strategy also uses Algorithm 7.2.1 and in Step (5) the emphasis is to minimize the communication.

Subsystem Number	Number of Rows	Size of Augmented System	Number of Nonzeros
1, 20	1035	2415 × 2415	25380
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19	1035	2760 × 2760	27450

Table 8.2: Partition I of system equations into subsystems.

8.1 Scheduling subsystems with a nearly uniform workload

As show in Table 8.2, the subsystems have almost the same weight factor, and the purpose of this experiment is to test the performance of the parallel Cimmino implementation under different distribution strategies.

Subsystem Number	Number of Rows	Size of Augmented System	Number of Nonzeros
1	518	1383×1383	12015
2, 4, 6, 8, 10 12, 14, 16, 18	1552	3797×3797	40820
3, 5, 7, 9, 11 13, 15, 17, 19	518	1728×1728	14085
20	1552	3452×3452	38750

Table 8.3: Partition II of system equations into subsystems.

ID	scheduling strategy	Computing Elements A	B	C	D	E	F	G	H
STG1	In Sequence (Random)	1 2 3 11 12 13	4 14	5 15	6 16	7 17	8 18	9 19	10 20
STG2	First Consider Weight Factor	2 3 4 12 13 14	5 15	6 16	7 17	8 18	9 19	10 1	11 20
STG3	First Consider Communication	2 3 1 4 5 6	7 8	9 10	11 12	13 14	15 16	17 18	19 20

Table 8.4: Distributions of subsystems under three scheduling strategies. Each subsystem is identified in the table by the number of its corresponding subsystem.

ID Strategy	Computing Elements A	B	C	D	E	F	G	H	Execution Time (Secs)
STG1	2	2	2	2	2	2	2	2	22.7
STG2	3	2	2	2	2	2	3	2	23.2
STG3	1	2	2	2	2	2	2	1	18.6

Table 8.5: Number of neighbors per computing element under three strategies

The results from the first experiments are shown in Tables 8.4 and 8.5. Each CE has to communicate with two neighbors. The first strategy has performed better than the second strategy because the weight of the subsystems is already balanced by the first partition, and the attempt to balance the workload using Step (5) only generates more communication.
In the third strategy the number of neighbors is reduced by one, and in this case the third strategy reduces the execution time by saving communication.

8.2 Scheduling subsystems with non-uniform workload

ID	scheduling strategy	Computing Elements A	B	C	D	E	F	G	H
STG1	In Sequence (Random)	1 2 3 11 12 13	4 14	5 15	6 16	7 17	8 18	9 19	10 20
STG2	First Consider Weight Factor	4 6 8 7 9 5	10 11	12 13	14 15	16 17	18 19	2 1	20 3
STG3	First Consider Communication	4 3 5 2 6 1	8 7	10 9	12 11	14 13	16 15	18 17	20 19

Table 8.6: Distributions of subsystems under three scheduling strategies. Each subsystem is identified in the table by the number of its corresponding subsystem.

ID Strategy	Computing Elements A	B	C	D	E	F	G	H	Execution Time (Secs)
STG1	2	3	2	4	2	4	2	3	34.4
STG2	3	2	2	2	2	2	2	2	17.0
STG3	1	2	2	2	2	2	2	2	10.2

Table 8.7: Number of neighbors per computing element under three strategies

In this experiment, the subsystems have different sizes, and some require more communication than others. Thus, the performance of the three scheduling strategies is studied to conclude with a strategy that best suits the parallel Cimmino implementation.
The results from the second experiments are summarized in Tables 8.6 and 8.7. Table 8.6 depicts the distribution of subsystems and Table 8.7 presents the number of neighbors per CE and the total execution time. Clearly, the third strategy reduces the execution time because the number of communications are also reduced and the communication is a bottleneck in parallel implementations of conjugate gradient based methods.
The second strategy has performed better than the first one because it balanced the workload among the CEs and as a result CEs had to wait less at the synchronization points. In the first experiment, this effect did not appear because the subsystems were of almost the same size,

however, in the second problem, these subsystem vary in size and number of neighbors.

8.3 Remarks

The parallel performance of the block Cimmino solver depends a great deal on the distribution of tasks among the CEs. The first scheduling strategy should be reserved for cases in which the linear system is evenly partitioned into blocks of rows, the number of neighbors per CE is the same, and the CEs have the same computing capabilities. In practice, these trivial partitionings are not performed and we dedicate Chapters 10 and 11 to study the effects of the partitioning on the behavior of the block Cimmino solver and its parallel implementation.
The second scheduling strategy should be preferred when the sizes of the blocks vary more than the variations in the number of neighbors between blocks. The first and second scheduling strategies are more trivial and less costly to implement than the third strategy.
The third scheduling strategy performs better than its counterparts, and in the previous two experiments, the third strategy has improved the performance of the parallel Cimmino implementation. Thus, this strategy is used in the experiments presented in Chapter 9 and it is compared against the second scheduling strategy in Chapter 11.

Chapter 9

Block Cimmino experiments

In this chapter, we present runs of the parallel Cimmino with Block-CG acceleration. In these experiments, trivial block row partitionings are performed on the linear system (6.1.1) to obtain (6.1.2). Later in Chapter 10, we study partitioning strategies that may improve the overall performance of the iterative solver in terms of its convergence rate and accuracy approximating the real solution of the system of equations.
The GRE_1107 problem is used in the first experiment. This problem comes from the Harwell-Boeing matrix collection (see Duff, Grimes, and Lewis (1992)). The unsymmetric matrix arises in the simulation of computer systems. The sparsity pattern of GRE_1107 is shown in Figure 9.1. The matrix is very ill-conditioned, and Arioli, Demmel, and Duff (1989) have shown that its classical condition number in the infinity norm ($\|A\|_\infty \|A^{-1}\|_\infty$) is equal to $1.8e^{10}$.
In the second experiment, we use the FRCOL problem that comes from a two dimensional model developed by Perrel for studying the effects of a body entering the atmosphere at a high mach number. The problem is discretized using a finite volume discretization of the Navier Stokes equations coupled with chemistry. There are two velocities and one energy at each grid point. From the chemistry, there are two species which leads to two density variables per mesh point. Thus, there is a total of five variables per mesh point.
The discretization is performed in a curvilinear mesh of 69×60 points, leading to block tridiagonal matrices of order $69 \times 60 \times 5 = 20700$.
In Section 9.1, we solve the GRENOBLE_1107 problem using the block Cimmino accelerated with the Block-CG, and study the impact of different block sizes on the performance of the parallel solver. In these experiments, the block size for the Block-CG acceleration and the number of CEs in the system are varied.
Afterwards, in Section 9.2 we solve the FRCOL problem, and fix the block size to isolate the effects of the block size on the rate of convergence that may speedup the parallel execution of the iterative solver from the speedups arising from the parallelism. Thus, in the experiments of Section 9.2, only the number of CEs in the system is varied.

9.1 Solving the GRENOBLE_1107 problem

The GRE_1107 matrix is partitioned into seven subsystems. The first six subsystems have 159 rows each (three blocks of 53 rows, $53 \times 3 = 159$) , and the last subsystem has 153 rows (three blocks of 51 rows, $51 \times 3 = 153$). The experiments are run on the 32 Thin node SP2 computer at CNUSC. The maximum number of CEs to be used in these experiments is seven because there

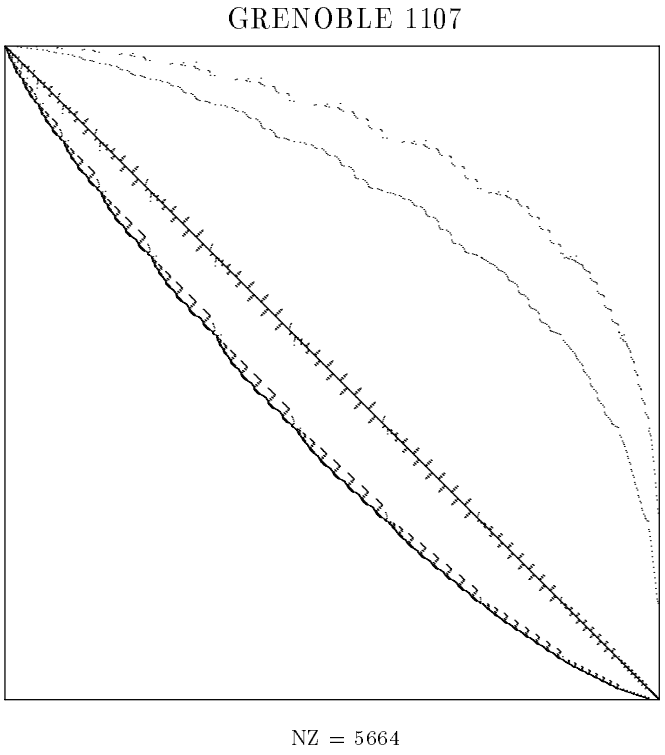

Figure 9.1: Sparsity pattern of GRE_1107 matrix.

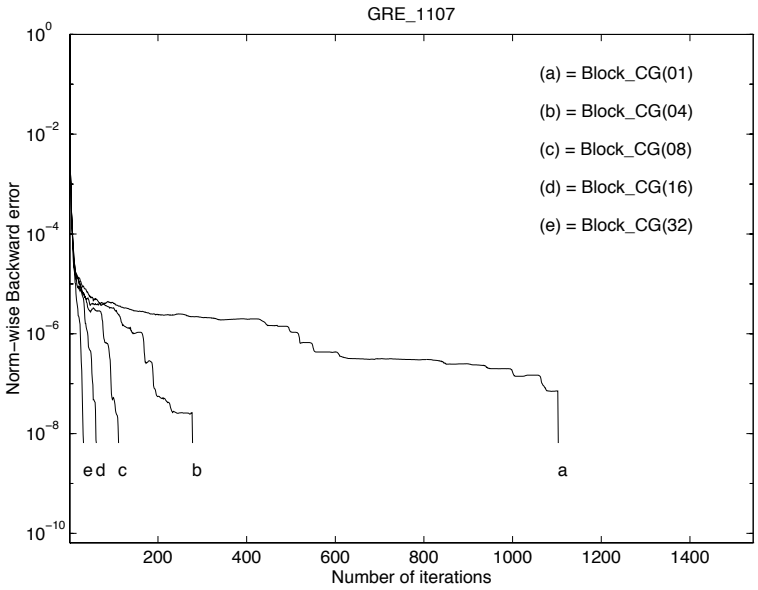

Figure 9.2: Convergence curves with iteration count. Test problem: GRE_1107.

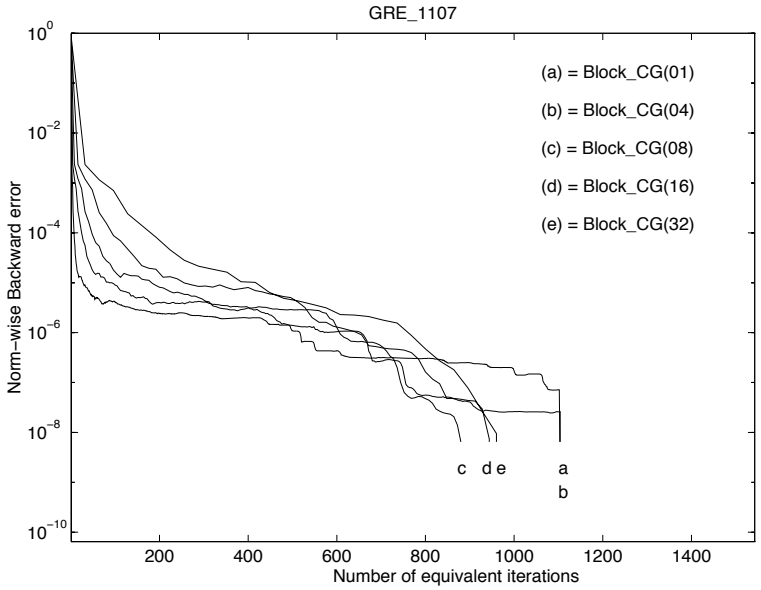

Figure 9.3: Convergence curves with equivalent iterations (the iteration count is multiplied by the block-size). Test problem: GRE_1107.

are seven subsystems.
Figures 9.2 and 9.3 show the convergence curves corresponding to these experiments. the block sizes of 8, 16, and 32 converge in fewer iterations than the Classical CG acceleration.

In Tables 9.1 to 9.3, the execution time of the Solve phase has been separated from the other two phases because, in some cases, the output of the Analyse and Factorize phases can be reused to solve the same linear system of equations with a different set of right hand sides. Thus, only the Solve phase is rerun in these cases. For these experiments, we first focus our attention on the total run time, and then the run time of the Solve phase by itself.
A close look at the times shown in Tables 9.1 to 9.3 reveals that most of the computational weight is in the Solve phase. And thus the Block-CG implementation has a great impact on the performance of this parallel block Cimmino implementation.
As shown in Figure 9.3, the Block-CG acceleration with a block size of four is as fast, in terms of equivalent iterations, as the CG acceleration (reported in figure with a block size of one). However in Table 9.1, it is shown that using the Block-CG with a block size of four is faster than using Classical CG.
In all cases, the speedups increase as we increase the number of CEs. And the efficiencies obtained in these experiments are higher than the ones obtained with the parallel Block-CG (see Chapter 5). This is due to an increase in the granularity of the tasks performed in parallel. And in this implementation of the Block-CG acceleration, the increase in the granularity comes from the computations of the orthogonal projections from the block Cimmino method.

Sequential times: Block size 1, total run : 63.0 secs, Solve phase: 62.4 secs Block size 4, total run : 43.6 secs, Solve phase: 42.2 secs												
N of CEs	Block size 1						Block size 4					
	Total run			Solve phase			Total run			Solve phase		
	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$
2	43.7	1.4	0.70	42.6	1.5	0.75	36.1	1.2	0.60	34.6	1.2	0.60
3	30.3	2.1	0.70	27.8	2.2	0.73	24.3	1.8	0.60	21.5	2.0	0.67
4	24.2	2.6	0.65	19.5	3.2	0.80	18.5	2.4	0.60	15.3	2.8	0.70
7	22.9	2.8	0.40	15.6	4.0	0.57	15.6	2.8	0.40	10.0	4.2	0.60

Table 9.1: Results from the parallel Block Cimmino with Block-CG acceleration. On the solution of the GRE_1107 problem. Times in this table are in seconds.

Sequential times: Block size 8, total run : 36.9 secs, Solve phase: 36.1 secs Block size 16, total run : 42.7 secs, Solve phase: 42.0 secs												
N of CEs	Block size 8						Block size 16					
	Total run			Solve phase			Total run			Solve phase		
	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$
2	35.6	1.1	0.55	33.3	1.1	0.55	38.7	1.1	0.55	36.3	1.2	0.60
3	24.3	1.5	0.50	21.4	1.7	0.57	26.3	1.6	0.53	24.3	1.7	0.57
4	15.3	2.4	0.60	12.9	2.8	0.70	18.7	2.3	0.57	15.2	2.8	0.70
7	11.4	3.5	0.50	8.6	4.2	0.60	12.1	3.5	0.50	9.7	4.3	0.61

Table 9.2: Results from the parallel Block Cimmino with Block-CG acceleration. On the solution of the GRE_1107 problem. Times in this table are in seconds.

Sequential times: Block size 32, total run : 47.8 secs Solve phase: 46.5 secs						
N of CEs	Block size 32					
	Total run			Solve phase		
	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$
2	44.3	1.1	0.55	42.7	1.1	0.55
3	34.3	1.4	0.46	32.3	1.3	0.48
5	24.9	1.9	0.47	20.6	2.3	0.57
7	16.5	2.9	0.41	13.5	3.4	0.49

Table 9.3: Results from the parallel Block Cimmino with Block-CG acceleration. On the solution of the GRE_1107 problem. Times in this table are in seconds.

9.2 Solving the FRCOL problem

Here, the FRCOL problem is solved using the parallel implementation of block Cimmino method accelerated with the Block-CG stabilized algorithm. In this experiment, the size of the block for the Block-CG acceleration is fixed to 8 to isolate the performance of the parallel implementation from that related to the block size. The linear system is partitioned into 10 blocks of rows of 2070 rows each. Therefore, a maximum of 10 CEs are used in the parallel execution.
The FRCOL problem is larger than the GRENOBLE_1107 problem. Thus, the grain size of the parallel task has been increased. As shown in Table 9.4, this increase has favored the performance of the parallel Cimmino implementation, and the efficiencies obtained are above 0.72 when considering the total run time and above 0.75 when only considering the execution time of the Solve phase. Thus, most of the computational resources are being used during the parallel executions.

The efficiency in the Solve phase increases as the number of CEs is increased, while the efficiency of the total run starts to decrease after five CEs. This is due to the communication in the first two phases and, as shown in Table 9.5, the percentage of the total run time consumed in the Solve phase decreases as the number of CEs is increased, whereas the percentage of the total run time consumed by the *master* to *slave* communication in the Analyse and Factorize phases increases as the number of CEs is increased. Therefore, the efficiency of the total run time decreases despite the reduction in the total run time due to an increase on the number of CEs.

Sequential times: Block size 8, total run : 274.3 secs Solve phase: 257.8 secs						
N of CEs	Block size 8					
	Total run			Solve phase		
	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$
2	186.2	1.5	0.75	171.1	1.5	0.75
3	104.6	2.6	0.87	94.6	2.7	0.91
5	62.3	4.4	0.88	56.0	4.6	0.92
8	44.9	6.1	0.76	33.9	7.6	0.95
10	38.2	7.2	0.72	27.2	9.5	0.95

Table 9.4: Results from the parallel Cimmino with Block-CG acceleration. On the solution of the FR_COL problem. Times in this table are in seconds.

Consumption of run time in	Number of CEs				
	2	3	5	8	10
Communication from first two phases	5.4	5.8	6.7	14.3	23.7
Solve Phase	91.9	90.4	89.9	75.5	71.2

Table 9.5: Percentages of the total run time consumed in the Communication of the Analyse and Factorized phases and the execution of the Solve phase.

9.3 Remarks

The block Cimmino with Block-CG acceleration appears to be a reliable solver which suits well the solution of some practical problems (for instance Benzi, Sgallari, and Spaletta (1995), and O'Leary (1993)). The solver can be easily tuned to reduce the number of iterations and improve the performance of a parallel run.
In some cases, the efficiency of the parallel runs of block Cimmino increases as the number of CEs is increased. And the maximum number of the CEs to be used in a run is limited by the number of blocks of rows from partitions of the linear system.
The partition of a linear system should be driven by the nature of the problem being solve. In the next chapter, we study natural partitioning and preprocessing strategies to improve the performance of the block Cimmino method.

Chapter 10

Partitioning strategies

Preprocessing a linear system of equations improves the performance of most linear solvers and in some cases a more accurate approximation to the real solution is found. For instance, numerical instabilities are encountered in the solution of some linear systems and it is necessary to scale the systems before they are solved. These instabilities occur even when the most robust computer implementations of direct or iterative solvers are used.
Also, performing some permutations of the elements of the original system can substantially reduce the required computing time for the solution of large sparse linear systems by improving the rate of convergence of some iterative methods (e.g., SOR, and Kaczmarz methods), and reducing fill-in in direct methods.
The aim of this chapter is to study preprocessing strategies to derive natural block partitionings of the form (6.1.2) from general linear systems of equations. We study two different preprocessing strategies to be applied to the original matrix A. These preprocessing strategies are based on permutations that transform the matrix AA^T into a matrix with a block tridiagonal structure. Transforming the matrix AA^T into a block tridiagonal matrix provides a natural partitioning of the linear system for row projection methods because these methods use the normal equations to compute their projections. Therefore, the resulting natural block partitioning should improve the rate of convergence of block row projection methods as block Cimmino (Section 6.1), and block Kaczmarz (Section 6.2).

10.1 Ill conditioning in and across blocks

An ill-conditioned matrix A has some linear combinations of rows that are almost equal to zero, and, as mentioned by Bramley and Sameh (1990), these linear combinations may occur inside blocks or across blocks after row partitionings of the form (6.1.2). If the method used for solving the subproblems is sensitive to ill-conditioning within the blocks, then the method may converge to the wrong solution.
Assuming that the projections in the block Cimmino are computed on the subspaces exactly, then the rate of convergence of the block Cimmino method depends only on the conditioning across the blocks. Therefore, a combination of a robust method for computing the projections and a partitioning strategy that minimizes the ill-conditioning across the blocks is sought.

To study the ill-conditioning across the blocks, we consider the linear system of equations resulting from the partitioning (6.1.2), and the **QR** decomposition of the submatrices A_i^T. Then,

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

Writing again the sum of projections of the block Cimmino method in matrix form (from Section 6.1), we have

$$M = \sum_{i=1}^{l} A_i^T \left(A_i A_i^T \right)^{-1} A_i, \tag{10.1.1}$$

and, replacing the A_i's by the **QR** factors,

$$
\begin{aligned}
M &= \sum_{i=1}^{l} Q_i R_i \left(R_i^T Q_i^T Q_i R_i \right)^{-1} R_i^T Q_i^T \\
&= \sum_{i=1}^{l} Q_i R_i \left(R_i^T R_i \right)^{-1} R_i^T Q_i^T \\
&= \sum_{i=1}^{l} Q_i Q_i^T \\
&= (Q_1 \ldots Q_l)(Q_1 \ldots Q_l)^T
\end{aligned} \tag{10.1.2}
$$

Using the theory of the singular value decomposition (see Golub and Kahan (1965), Golub and Van Loan (1989)), it can be seen that the nonzero eigenvalues of $(Q_1 \ldots Q_l)(Q_1 \ldots Q_l)^T$ are also the nonzero eigenvalues of $(Q_1 \ldots Q_l)^T(Q_1 \ldots Q_l)$. Therefore, the spectrum of the matrix M from (10.1.1) is the same as the spectrum of the matrix

$$
\begin{pmatrix}
I_{m_1 \times m_1} & Q_1^T Q_2 & \cdots & \cdots & Q_1^T Q_l \\
Q_2^T Q_1 & I_{m_2 \times m_2} & Q_2^T Q_3 & \cdots & Q_2^T Q_l \\
\vdots & & \ddots & & \vdots \\
Q_l^T Q_1 & \cdots & \cdots & & I_{m_l \times m_l}
\end{pmatrix} \tag{10.1.3}
$$

where the $Q_i^T Q_j$ are the 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 Björck and Golub (1973)). These principal angles are recursively defined by

$$\cos(\Psi_{(i,j)_k}) = \max_{u_{(i,j)} \in \mathcal{R}(A_i^T)} \left[\max_{v_{(i,j)} \in \mathcal{R}(A_j^T)} \left(\frac{u_{(i,j)}^T v_{(i,j)}}{\|u_{(i,j)}\| \, \|v_{(i,j)}\|} \right) \right] \tag{10.1.4}$$

$$= \frac{u_{(i,j)_k}^T v_{(i,j)_k}}{\|u_{(i,j)_k}\| \quad \|v_{(i,j)_k}\|}$$

subject to

$$u_{(i,j)}^T u_{(i,j)_p} = 0 \quad \text{for } p = 1, \ldots, l-1, \text{ and}$$

$$v_{(i,j)}^T v_{(i,j)_p} = 0 \quad \text{for } p = 1, \ldots, l-1,$$

l varying from 1 to $m_{ij} = \min\left[\dim(\mathcal{R}(A_i^T)), \dim(\mathcal{R}(A_j^T))\right]$. The set of vectors $\left\{u_{(i,j)_1}, \ldots, u_{(i,j)_{m_{ij}}}\right\}$ and $\left\{v_{(i,j)_1}, \ldots, v_{(i,j)_{m_{ij}}}\right\}$ are called the principal vectors between the subspaces $\mathcal{R}(A_i^T)$ and $\mathcal{R}(A_j^T)$.

Notice, the principal angles satisfy

$$0 \leq \Psi_{(i,j)_1} \leq \ldots \leq \Psi_{(i,j)_{m_{ij}}} \leq \frac{\pi}{2},$$

and if all of the m_{ij} principal angles are equal to $\frac{\pi}{2}$, it means that $\mathcal{R}(A_i^T)$ is orthogonal to $\mathcal{R}(A_j^T)$, and the wider the principal angles are the closer the block Cimmino iteration matrix is to the identity matrix. Furthermore, the closer the iteration matrix is to the identity, the faster the convergence of the Block-CG acceleration should be.
The principal angles by themselves do not provide information regarding the spectrum and ill-conditioning of the resulting iteration matrix, therefore it is necessary to study partitioning strategies for which there exists a strong relation between these principal angles and the spectrum of the iteration matrix.

Without loss of generality, assume that the matrix A is partitioned in two blocks

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}, \tag{10.1.5}$$

where A_1 and A_2 have m_1 and m_2 rows respectively, and further assume that $m_1 \geq m_2$.
With (10.1.5), the block Cimmino iteration matrix is defined as $C = I - \nu(P_1 + P_2)$, where $P_1 = P_{\mathcal{R}(A_1^T)}$ and $P_2 = P_{\mathcal{R}(A_2^T)}$. From (10.1.1), $M = P_1 + P_2$, and from (10.1.2) it can be reduced to $(Q_1 Q_2)(Q_1 Q_2)^T$. Therefore from (10.1.3), the spectrum of the block Cimmino iteration matrix for the (10.1.5) partition is given by

$$\begin{pmatrix} (1-\nu) I_{m_1 \times m_1} & -\nu Q_1^T Q_2 \\ -\nu Q_2^T Q_1 & (1-\nu) I_{m_1 \times m_1} \end{pmatrix}$$

As mentioned in Section 6.3, the block Cimmino with the Block-CG acceleration is independent of the relaxation parameter ν. Thus, if $\nu = 1$ then

$$\begin{pmatrix} 0_{m_1 \times m_1} & -Q_1^T Q_2 \\ -Q_2^T Q_1 & 0_{m_2 \times m_2} \end{pmatrix} \tag{10.1.6}$$

The following observations follow from (10.1.6)

- Since $m_1 \leq m_2$, the rank of the rectangular matrices $Q_1^T Q_2$, and $Q_2^T Q_1$ is not greater than m_2. Thus, there are at most $2m_2$ nonzero eigenvalues.
- The use of the Block-CG acceleration guarantees the finite termination of the block Cimmino method in exact arithmetic. In this particular case, it should not take more than $2m_2$ steps to reach convergence. Therefore, block Cimmino with the Block-CG acceleration will work better for small values of m_2.
- The singular values of the matrix $Q_1^T Q_2$ and the eigenvalues of the matrix (10.1.6) are strongly related. Indeed, a singular value decomposition of the matrix $Q_1^T Q_2$ (following Golub and Van Loan (1989)) is given by

  $$Q_1^T Q_2 = U \Sigma V^T,$$

  with U and V orthogonal matrices with dimensions $m_1 \times m_1$ and $m_2 \times m_2$ respectively. Σ is a rectangular matrix with dimensions $m_1 \times m_2$, and $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_{m_2})$. The matrix (10.1.6) is equal to

  $$\begin{pmatrix} U & 0 \\ 0 & V \end{pmatrix} \begin{pmatrix} 0 & -\Sigma \\ -\Sigma^T & 0 \end{pmatrix} \begin{pmatrix} U^T & 0 \\ 0 & V^T \end{pmatrix}$$

  And has the same eigenvalues as the matrix:

  $$\begin{pmatrix} 0 & -\Sigma \\ -\Sigma^T & 0 \end{pmatrix}.$$

  The eigenvalues of matrix (10.1.6) are $\{\pm\sigma_i, i = 1, \ldots, m_2\}$ and correspond to the cosines of the principal angles between the two subspaces $\mathcal{R}(A_1^T)$ and $\mathcal{R}(A_2^T)$.

10.2 Two-block partitioning

In the following sections, we will study two preprocessing strategies to permute the rows of the matrix A based on permutations that transform the AA^T matrix into a block tridiagonal matrix. Matrices with block tridiagonal structures are commonly found in discretization of partial differential equations and in practice these systems are solved using different iterative schemes. Coefficient matrices from other general systems of equations can also be permuted to block tridiagonal matrices using matrix reordering techniques (see for example Duff, Erisman, and Reid (1989)).

Two-block partitioning refers to a partitioning strategy in which the columns inside a block of rows intersect the columns of at most two other blocks. A two-block partitioning can be applied to matrices with a block diagonal structure and in some cases this will speed the convergence of the linear solver.
For instance, let the matrix A with blocks of size $b \times b$ be partitioned into five blocks of rows as illustrated here

$$\left(\begin{array}{cccccccccccc}
* & * & & & & & & & & & & \\
* & * & * & & & & & & & & & \\
 & * & * & * & & & & & & & & \\
\hline
 & & * & * & * & & & & & & & \\
 & & & * & * & * & & & & & & \\
\hline
 & & & & * & * & * & & & & & \\
 & & & & & * & * & * & & & & \\
 & & & & & & * & * & * & & & \\
\hline
 & & & & & & & * & * & * & & \\
 & & & & & & & & * & * & * & \\
\hline
 & & & & & & & & & * & * & * \\
 & & & & & & & & & & * & * \\
 & & & & & & & & & & &
\end{array}\right) = \left(\begin{array}{c} A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \end{array}\right), \qquad (10.2.1)$$

where each A_i has $k_i \times b$, and each $k_i \geq 2$, for $i = 1, 2, \ldots, 5$. Further, for $i = 1, 2, 3$, we observe that the pair of subspaces $\mathcal{R}(A_i^T)$ and $\mathcal{R}(A_{i+2}^T)$ are orthogonal, and the sum of these orthogonal projections onto these two subspaces correspond to the orthogonal projection onto the direct sum of these orthogonal subspaces, viz:

$$P_{\mathcal{R}(A_i^T)} + P_{\mathcal{R}(A_i^T)} = P_{\mathcal{R}(A_i^T) \oplus \mathcal{R}(A_{i+2}^T)}.$$

Therefore, the partitioning defined in (10.2.1) is numerically equivalent to partitioning in two blocks, B_1 and B_2 as in (10.2.2),

$$\left(\begin{array}{cccccccccccc}
* & * & & & & & & & & & & \\
* & * & * & & & & & & & & & \\
 & * & * & * & & & & & & & & \\
 & & & & * & * & * & & & & & \\
 & & & & & * & * & * & & & & \\
 & & & & & & * & * & * & & & \\
 & & & & & & & & & * & * & * \\
 & & & & & & & & & & * & * \\
\hline
 & & * & * & * & & & & & & & \\
 & & & * & * & * & & & & & & \\
 & & & & & & * & * & * & & & \\
 & & & & & & & * & * & * & &
\end{array}\right) = \left(\begin{array}{c} A_1 \\ A_3 \\ A_5 \\ A_2 \\ A_4 \end{array}\right) = \left(\begin{array}{c} B_1 \\ \\ \\ \\ B_2 \end{array}\right), \qquad (10.2.2)$$

where

$$B_1 = \left\{ \bigcup_i A_i/i \text{ is odd} \right\}, \qquad B_2 = \left\{ \bigcup_i A_i/i \text{ is even} \right\}$$

From Section 10.1, the number of nonzero eigenvalues in the iteration matrix is at most $2 * \min(m_1, m_2)$ when using a two-block partition. In (10.2.2), m_1 and m_2 are the number of rows in B_1 and B_2 respectively. Therefore, varying the number of rows in B_1, and B_2 will also vary the number of nonzero eigenvalues of the iteration matrix.
Without loss of generality, we go back to one of the assumptions made in Section 10.1, $m_1 \geq m_2$, the size of the interface block B_2 can at least be of size $2 \times b$, which is the minimum required for making the A_i's in B_1 orthogonal.
In the parallel implementation of the block Cimmino method accelerated with Block-CG, the compromise is to find a matrix partitioning that reduces the size of the interface block B_2 which implicitly improves the rate of convergence of Block-CG acceleration, and at the same time enables a fair distribution of the workload.

10.3 Preprocessing Strategies

Given the general linear system of equations

$$Ax = b, \tag{10.3.1}$$

with the matrix A of dimensions $m \times n$. The first preprocessing strategy finds a permutation of the normal equations

$$B = PAA^TP^T \tag{10.3.2}$$

such that the matrix B has a block tridiagonal structure. An implementation of the Cuthill-McKee Algorithm (see for instance Cuthill and McKee (1969), George (1971), Duff, Erisman, and Reid (1989)) for ordering symmetric matrices is used. Afterwards, the system of equations

$$\hat{A}x = \hat{b} \tag{10.3.3}$$

is solved, with $\hat{A} = PA$, and $\hat{b} = Pb$. From the block tridiagonal structure of the matrix B, the block row partition (6.1.2) is defined by partitioning the blocks of rows in $\hat{A}$ with respect to the diagonal block in the block tridiagonal structure of B. Doing this, the row partitioning preserves the advantages of the two-block partitioning described in Section 10.2. Figure 10.1 illustrates the row partitioning of $\hat{A}$ from the diagonal block structure of B.

In the second preprocessing strategy, the matrix AA^T is first normalized

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

Afterwards, remove all the nonzero elements in $\overline{B}$ which are value under a tolerance value τ in absolute, viz

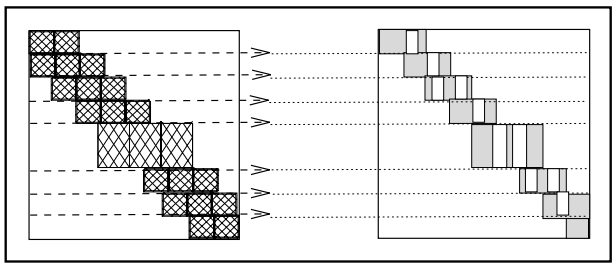

Figure 10.1: Row partitioning of $\hat{A}$ from block tridiagonal structure of B.

$$\overline{\overline{B}} = \mathbf{remove}(\overline{B}, \tau),$$

permute $\overline{\overline{B}}$ using the Cuthill-McKee Algorithm

$$\hat{B} = P\overline{\overline{B}}P^T \tag{10.3.4}$$

and solve (10.3.3). In this case the row partitioning for $\hat{A}$ is defined by the block tridiagonal structure of $\hat{B}$.

The first preprocessing strategy always delivers a matrix A with a two-block partitioning. The partitioning obtained with the second strategy is close to a two-block partitioning, and this will become clearer with the partitioning experiments in the next chapter.
Additionally, since some nonzero elements under a tolerance value are removed from $\overline{B}$ in the second preprocessing strategy, the matrix $\hat{B}$ has a smaller bandwidth than the matrix $\overline{B}$. Therefore, $\hat{B}$ has more blocks on the diagonal than $\overline{B}$ and consequently the matrix $\hat{A}$ can be partitioned into more blocks of rows.
Clearly, the second preprocessing strategy is a more flexible partitioning strategy.

We perform a last step after using either preprocessing strategy. In this last step the columns that belong to the same subset of blocks are grouped to expedite the identification of sections of columns in the block Cimmino solver (see Section 6.4). The need for this last step will become more evident in the experiments in the next chapter (particularly compare Figure 11.3 with Figure 11.4 and Figure 11.9-a with Figure 11.9-b).
In the last preprocessing step, the sparsity pattern of $\hat{A}$ is stored in a sparse matrix C, such that the matrix C has a 1 in all places that the matrix $\hat{A}$ has a nonzero element.
Then the block row partitioning from the matrix $\hat{A}$ is applied to the matrix C, such that, $\overline{C}_i$ is the i-th block of the matrix C. If l is the number of blocks of rows in the matrices $\hat{A}$ and C, then we use a matrix D, of dimension $l \times n$, to store the sparsity pattern of each block in C.
As depicted in Figure 10.2, if there is at least one 1 in the j-th column of the block matrix $\overline{C}_i$, then $D(i,j) = 1$.

After scanning the sparsity pattern of the blocks in $\hat{A}$, an integer label is computed for each column of the matrix $\hat{A}$ in the following manner

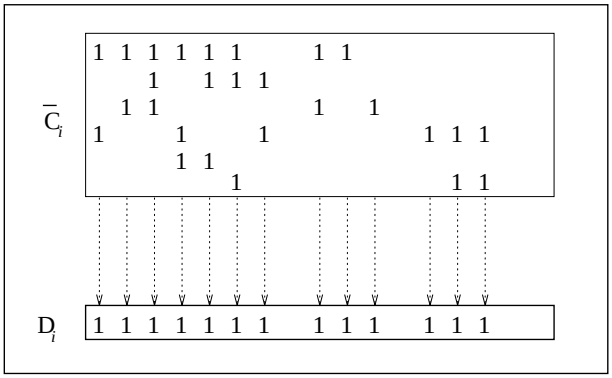

Figure 10.2: Scanning the sparsity pattern of a block of rows.

$$\text{label}(j) = \sum_{p=1}^{l} 2^{(p-1)} D(p,j) \qquad \text{for } j = 1, \ldots n \tag{10.3.5}$$

After computing the labels, they are sorted in ascending order and the results from the sort define a column permutation to be applied to the matrix $\hat{A}$.
Notice that these column permutations will group all the columns that belong to the same subset of blocks, and the resulting matrix will have sections of columns inside the blocks of rows and these sections of columns are the ones used in the parallel Cimmino implementation described in Section 6.5.

Chapter 11

Partitioning Experiments

Now, we perform some more runs of the parallel block Cimmino implementation using the Block-CG acceleration, and focus on the effects of the preprocessing strategy on the performance of the method. In the experiments reported in Chapters 5, 8, and 9, we have used trivial partitionings based on the size of the linear systems and the number of available CEs in the computational platform. In this chapter, all of the previous efforts invested in the parallelism are combined with the preprocessing strategies for solving the linear system of equations and the advantages of using the output from a preprocessing phase are emphasized.
The experiments in this chapter were run on a SP2 Thin node.
In Section 11.1, the parallel block Cimmino implementation is used for solving the SHERMAN4 linear system from the Harwell-Boeing matrix collection (Duff, Grimes, and Lewis (1992)). The symmetric matrix SHERMAN4 comes from the discretization of partial differential equations extracted from an oil reservoir modeling program. The matrix arises from a three dimensional simulation model on a $16 \times 23 \times 3$ grid using a seven-point finite difference approximation with one equation and one unknown per grid block (Simon (1985)).
Section 11.2 contains the results from experiments of runs of the parallel block Cimmino implementation solving the GRENOBLE 1107 problem introduced in Chapter 9.

11.1 Solving the SHERMAN4 problem

The matrix SHERMAN4 is a symmetric matrix of order 1104. The spectrum of the matrix was shown earlier in Chapter 3, Figure 3.15. The experiments reported in this section were selected from a large set of experiments varying the block size for the Block-CG acceleration. To focus on the effects from the preprocessing strategies, the block size in the Block-CG has been fixed at 4. In these experiments, we have used ω as a stopping criterion (see Section 2.4) for the block Cimmino iterations, and the error is reduced below 1.0×10^{-12}.
The original pattern of the matrix SHERMAN4 is shown in Figure 11.1 and the results from parallel runs of the block Cimmino implementation solving the original system (i.e., without any data preprocessing) are reported in Table 11.1 under the column labeled **Original matrix** A.
The parallel block Cimmino program can be invoked from the command line with several parameters to specify the computing environment, the problem to be solved, and other values that are meaningful to the parallel solver (for instance the block size for the Block-CG acceleration, the threshold value for the stopping criterion, the block partitioning strategy, the size of the integer and double working arrays, *etc.*).

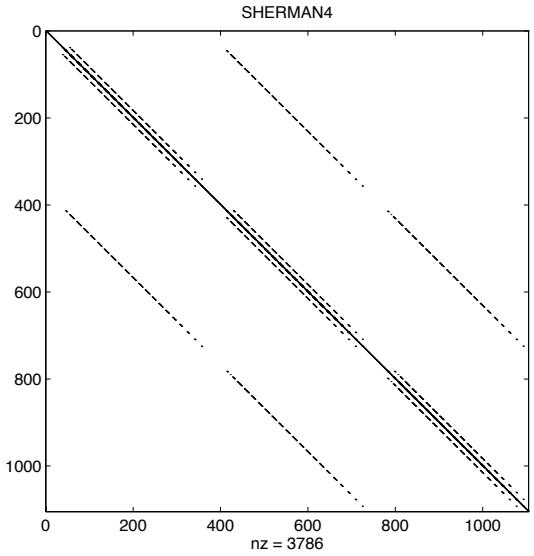

Figure 11.1: Sparsity pattern of the matrix SHERMAN4.

The following is an example of a call to the block Cimmino solver from the UNIX command line (note that the only purpose for including this type of example in this chapter is to summarize the different parameters of the parallel solver that were used in a given set of runs. Further, the examples should not be confused with a user guide to the parallel block Cimmino solver).

```
command[1]> blkcimmino -hesfile sp2env -datafile sherman4.rua \
          -partitions 10 110 110 110 110 110 110 110 110 110 114 \
          -blocksize 4 -threshold 1.0D-12 -scheduling 3
```

As mentioned in Section 7.3, the computational environment is specified in a HESfile, and in the above example the HESfile is the `sp2env`. The linear system is stored in the Harwell-Boeing sparse matrix format in the `sherman4.rua` file. In the first experiment, a trivial row block partition of the linear system is performed.
The first argument following the `-partitions` keyword is the number of blocks of rows in the linear system of the form (6.1.2). In these experiments, the blocks have almost the same number of rows (110, only the last block has 114 rows). the `-blocksize` keyword is used for the block size of the Block-CG acceleration.
The value of the `-threshold` keyword is used for the stopping criteria. As mentioned in Chapter 7, we have implemented three scheduling strategies for the block Cimmino solver. In the results reported in Table 11.1, only the second and third scheduling strategies are used because the first scheduling strategy is a random scheduling distribution. And from experiments presented in Chapter 8, there are no significant advantages for using the first scheduling strategy over the other two.
Recall that the goal in the second scheduling strategy is to balance the workload, while in the third scheduling strategy is to minimize first the communication cost and preserve, as much as possible, the balanced workload.

For the second set of experiments, a trivial column permutation of the original matrix A is performed. The goal of the column permutation is to group the columns that belong to the same blocks. The column labeling (10.3.5) is used to find these column permutation.
Figure 11.2 shows the matrix SHERMAN4 after permuting its columns. The results from these experiments are presented in Table 11.1 under the column labeled **After column permutation**.

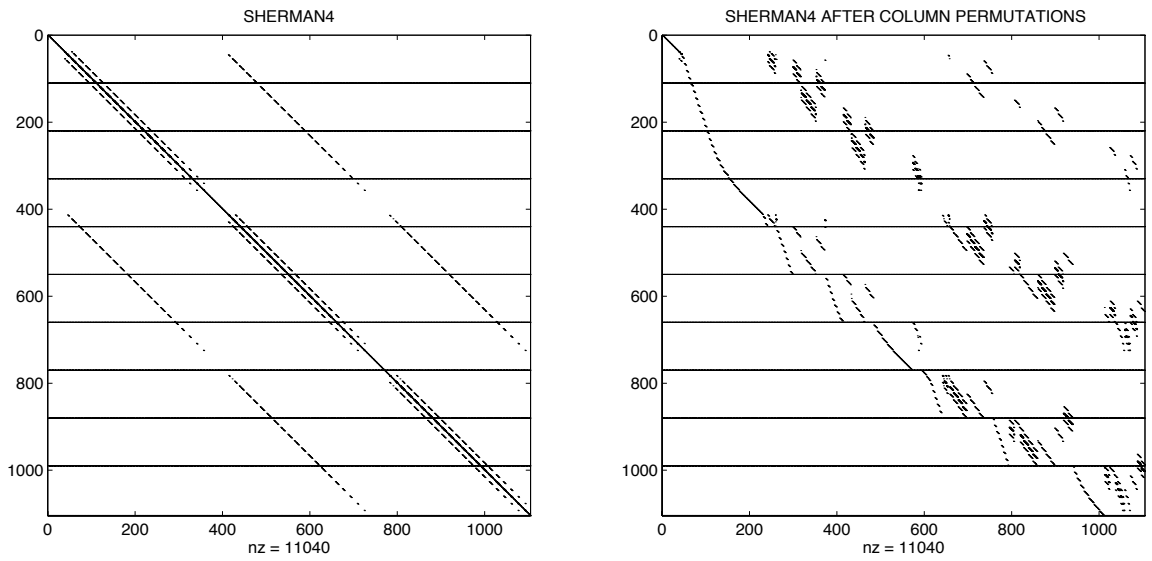

Figure 11.2: Sparsity pattern of the SHERMAN4 matrix before and after column permutation using the column labeling (10.3.5). The matrix SHERMAN4 has been partitioned into blocks of rows.

Recall that block Cimmino is numerically insensitive to column permutations. Thus, the reduction in the number of overlaps between the blocks of rows may only improve the parallel execution of the block Cimmino method because the communication is implicitly reduced.
In the third round of experiments, the first preprocessing strategy from Section 10.3 is used. Figure 11.3 shows the matrix SHERMAN4 after the first preprocessing strategy. From the block tridiagonal structure of the normal equations from SHERMAN4, the matrix is partitioned into blocks of rows. Later, the columns of the resulting matrix are permuted using the column labeling (10.3.5). The sparsity pattern of the matrix SHERMAN4 after the first preprocessing strategy and column permutations is shown in Figures 11.4-a, and Figures 11.4-b.
Figure 11.4-a is the result of applying the column permutations to the matrix SHERMAN4 after the Reverse Cuthill McKee algorithm is used inside the first preprocessing strategy, and Figure 11.4-b is the result of using the Cuthill McKee algorithm instead in this preprocessing phase. The partitionings shown in both figures are two-block partitionings. Here, we prefer Figure 11.4-b because there are five blocks that are independent from the rest while in Figure 11.4-a we have obtained only four. In both cases, a great improvement in the parallel execution is expected because there are independent blocks of rows.
The following is an example of the command line and parameters for running the parallel block Cimmino experiments on the matrix SHERMAN4 after using the first preprocessing strategy with the column permutations

```
command[1]> blkcimmino -hesfile sp2env -datafile sherman4_stg1.rua\
```

```
-partitions 10 110 110 110 110 110 102 113 111 103 125\
-threshold 1.0D-12 -scheduling 3
```

Again, the blocks of rows have nearly the same size, however the number of column overlaps between the blocks of rows varies from 0 to 2, and this implies nonuniform communication between the CEs during a parallel block Cimmino run. Results from the parallel block Cimmino runs are presented in Table 11.1 under the column labeled **Preprocessing strategy I**.

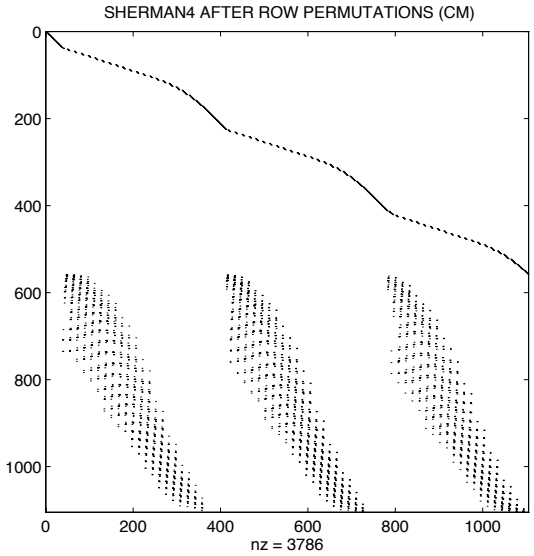

Figure 11.3: Matrix SHERMAN4 after applying the preprocessing strategy I.

In the last experiments of this section, the second preprocessing strategy is applied to the matrix SHERMAN4. Figure 11.5 shows the sparsity pattern of the normal equations from SHERMAN4 after symmetric permutation. Figure 11.5-a shows the results of using the Reverse Cuthill-Mckee algorithm and Figure 11.5-b shows the results of using the Cuthill-McKee algorithm. The normalized nonzero entries of AA^T are plotted in Figure 11.6.
The tolerance value is chosen to be 0.2 and the nonzero entries less than the tolerance value are removed from the matrix AA^T. The rows and columns of the new matrix AA^T are permuted using the Reverse Cuthill-McKee algorithm or the Cuthill-McKee algorithm. Figure 11.7 shows the result of permuting the matrix using the Reverse Cuthill-McKee algorithm.

The sparsity pattern of the SHERMAN4 matrix after completion of the second preprocessing strategy is shown in Figure 11.8. This matrix is obtained after permuting the rows of the original SHERMAN4 matrix using either the Reverse Cuthill-McKee algorithm (Figure 11.8-a) or the Cuthill-McKee algorithm (Figure 11.8-b) inside the second preprocessing strategy with the column permutations based on the column labeling (10.3.5).

The following is an example of the command line and parameters for running the parallel block Cimmino experiments on the matrix SHERMAN4 after using the second preprocessing strategy

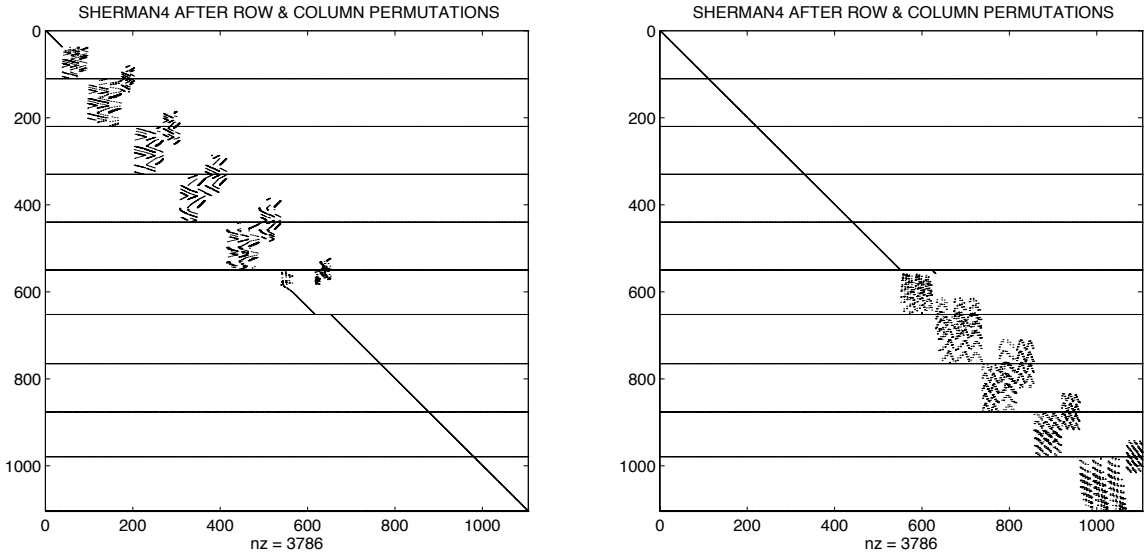

Figure 11.4: Matrix SHERMAN4 after applying the preprocessing strategy I and permuting the columns using the column labeling (10.3.5). In (a) the Reverse Cuthill-McKee algorithm is used inside the preprocessing phase, and in (b) the Cuthill-McKee algorithm is used.

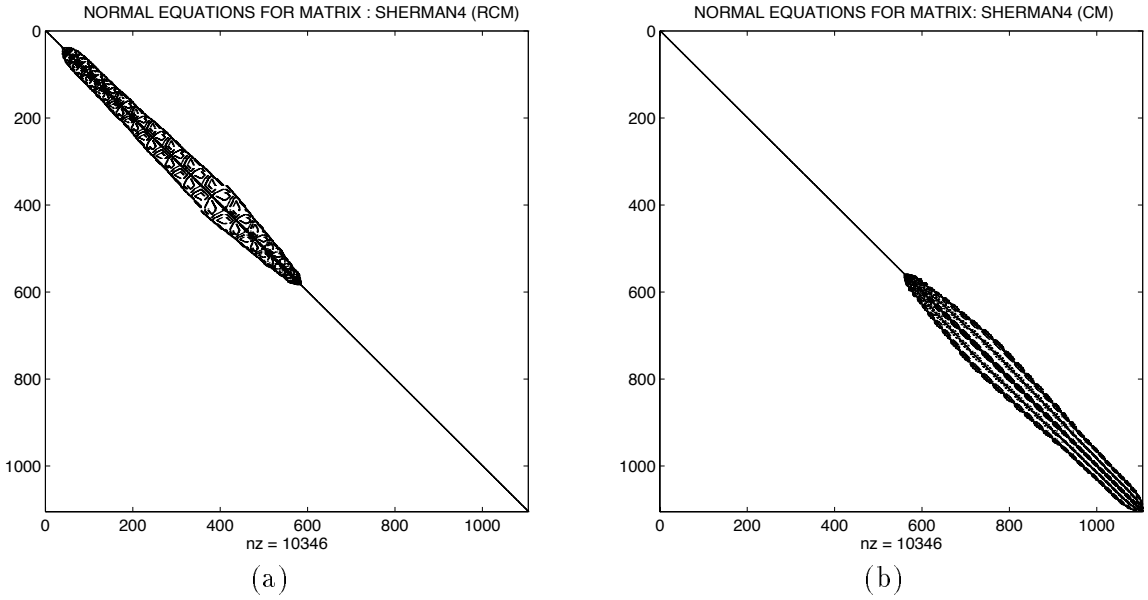

Figure 11.5: Block tridiagonal structures of the normal Equations from SHERMAN4 using the Reverse Cuthill-Mckee algorithm in (a) and the Cuthill McKee algorithm in (b).

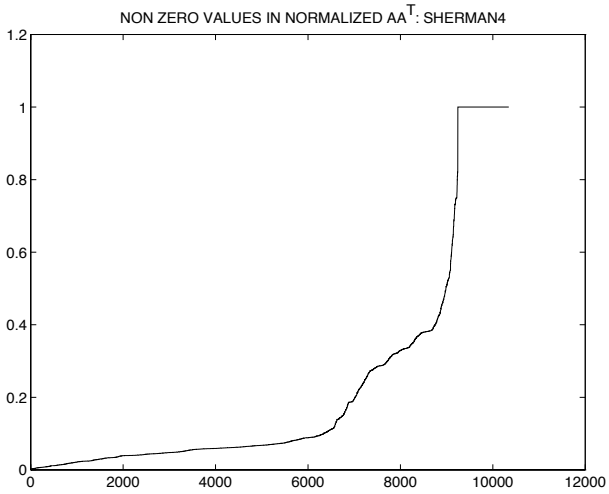

Figure 11.6: Normalized nonzeros in the normal equations matrix SHERMAN4.

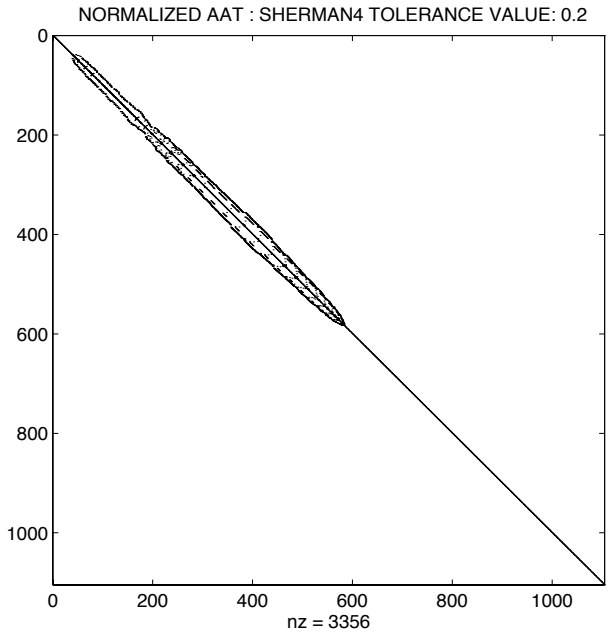

Figure 11.7: Sparsity pattern of the normal equations from SHERMAN4 after removing the nonzero entries less than 0.2 and using the Reverse Cuthill-McKee algorithm to permute the matrix to a block tridiagonal form.

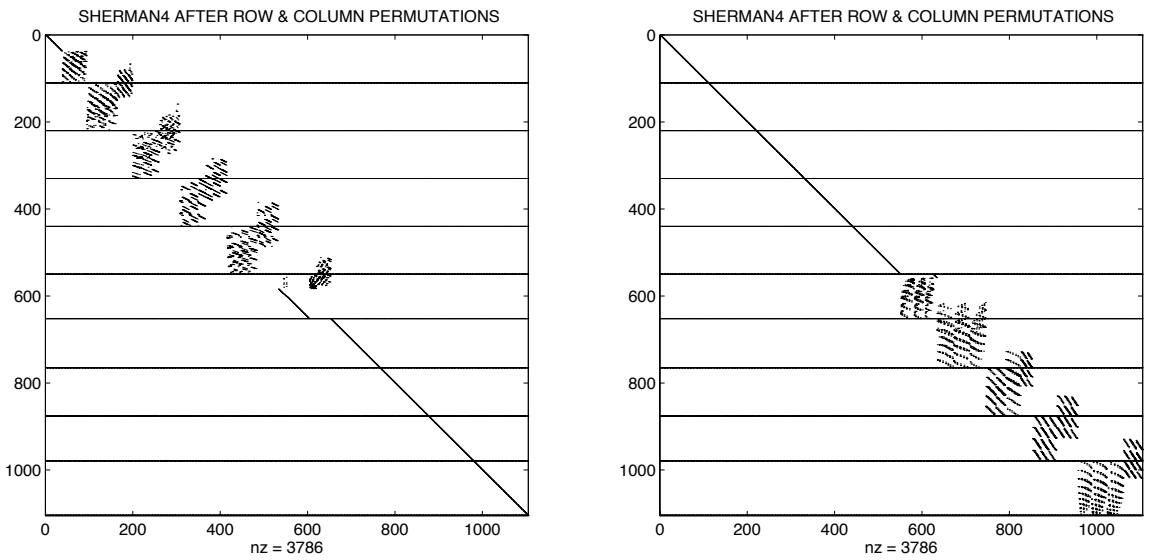

Figure 11.8: Matrix SHERMAN4 after applying the preprocessing strategy II and permuting the columns using the column labeling (10.3.5). In (a) the Reverse Cuthill-McKee algorithm is used inside the preprocessing phase, and in (b) the Cuthill-McKee algorithm is used.

and performing the column permutations

```
command[1]> blkcimmino -hesfile sp2env -datafile sherman4_stg2.rua \
          -partitions 10 110 110 110 110 110 122 114 115 118 85\
          -threshold 1.0D-12 -scheduling 3
```

As shown in Figure 11.8, this partitioning does not lead to a two-block partitioning but numerically it is not that different because we have kept the largest entries from AA^T.
After the second preprocessing strategy, there is more freedom in the way the permuted matrix is partitioned into blocks of rows since the Cuthill McKee ordering (or the Reverse Cuthill McKee ordering) is applied to a sparser matrix than the original AA^T. In this section, we have chosen only 10 blocks to fairly compare with the results from the other three preprocessing strategies. However, in the next section we present another example in which more blocks are obtained using the second preprocessing strategy.

In the SHERMAN4 problem, the performance of the Cimmino solver is substantially improved by the preprocessing strategies I and II. First of all, the number of iterations has been reduced from 70 to 52 with the first preprocessing strategy, and from 70 to 55 with the second. This proves that the second preprocessing strategy does not differ too far numerically from a two-block partitioning. Furthermore, when either preprocessing strategy has been used, the independent blocks of rows reduce the communication required by the original linear system.
Using a trivial partitioning of the SHERMAN4 matrix without preprocessing will lead to nearly uniform tasks and this is the reason for the comparable results using the scheduling strategy 2 with that using the scheduling strategy 3 with nearly uniform tasks. However, this is not the case when any of the preprocessing strategies are used.
When the preprocessing strategies have been used, the resulting blocks do not only differ in size

Comp env.	Original matrix A			After column permutation			Preprocessing strategy I			Preprocessing strategy II		
Seq. run	25.3 secs			24.5 secs			7.7 secs			8.3 secs		
	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$
stg2 2 CEs	13.7	1.9	0.95	12.4	2	1.00	6.0	1.3	0.64	6.2	1.3	0.64
stg3 2 CEs	13.1	1.9	0.95	13.1	1.9	0.95	4.0	1.9	0.95	4.3	1.9	0.95
stg2 4 CEs	9.2	2.8	0.70	8.5	2.9	0.72	3.9	2.0	0.50	4.1	2.1	0.53
stg3 4 CEs	8.5	3.0	0.75	8.3	3.0	0.74	2.3	3.4	0.84	2.5	3.3	0.83
stg2 8 CEs	5.0	5.1	0.63	5.1	4.8	0.60	3.0	2.6	0.32	3.4	2.5	0.31
stg3 8 CEs	4.2	6.0	0.75	4.5	5.4	0.68	1.2	6.4	0.80	1.4	5.9	0.74
stg2 10 CEs	4.2	6.0	0.60	3.2	7.7	0.77	2.3	3.4	0.34	2.4	3.5	0.35
stg3 10 CEs	3.4	7.4	0.74	2.8	8.8	0.88	0.9	8.6	0.86	1.0	8.3	0.83

Table 11.1: Results from the parallel Block Cimmino implementation with Block-CG acceleration using four different preprocessing strategies in the solution of the SHERMAN4 problem. Times in this table are in seconds. stg2 and stg3 correspond to the scheduling strategies STG2 and STG3, respectively, from Chapter 8.

but also in the number of column intersections between blocks of rows. Therefore, the scheduling strategy 2 distributes the tasks in a less favorable manner than the scheduling strategy 3.
The minimum execution time reported in Table 11.1 is 0.9 secs, for a speedup of 8.6 on 10 processors using the partitioning strategy I and the scheduling strategy 3. The execution time in this case represents an improvement of 96.5% of the execution time from a sequential block Cimmino run solving the original matrix SHERMAN4.

11.2 Solving the GRENOBLE_1107 problem

Now we perform similar experiments as in the previous section with the GRENOBLE_1107 problem. We try to study the effects of varying the tolerance value in the second strategy, and also the number of block partitions.
In Table 11.2, the results from runs of the parallel block Cimmino implementation are presented. The results in the column labeled **Original matrix** A come from experiments using the block row partitioning shown in Figure 11.9-a, and the results in the second column labeled **After column permutation** are from experiments using the block partitioning shown in Figure 11.9-b.
The following is an example of the call to the block Cimmino solver that was used in the set of experiments that generated the first two columns of Table 11.2.

```
command[1]> blkcimmino -hesfile sp2env -datafile gre_1107.rua \
           -partitions 7 159 159 159 159 159 159 153\
           -blocksize 8 -threshold 1.0D-08 -scheduling 3
```

Applying the first preprocessing strategy to the GRENOBLE_1107 problem leads to a matrix with the sparsity pattern shown in Figure 11.10-a, later the columns of this matrix are permuted using the column labeling (10.3.5). And as expected, the output of the first preprocessing strategy leads to a two-block partitioning, illustrated in Figure 11.10-b.

In this case, an example of a call to the block Cimmino solver is

```
command[1]> blkcimmino -hesfile sp2env -datafile gre_1107_stg1.rua \
           -partitions 7 102 98 130 160 190 214 213 -blocksize 8 \
           -threshold 1.0D-12 -scheduling 3
```

Results from these experiments are reported in Table 11.2 under the column labeled **Preprocessing strategy I**.

In the first and second preprocessing strategies, the normal equations from GRENOBLE_1107 are computed, and the sparsity pattern of the matrix AA^T is shown in Figures 11.11-a and 11.11-b. The normalized nonzero entries are plotted in Figure 11.12.

Two tolerance values are selected for testing the second preprocessing strategy. First, 0.2 is used and about half of the nonzero entries are removed from AA^T. The resulting matrix is reordered

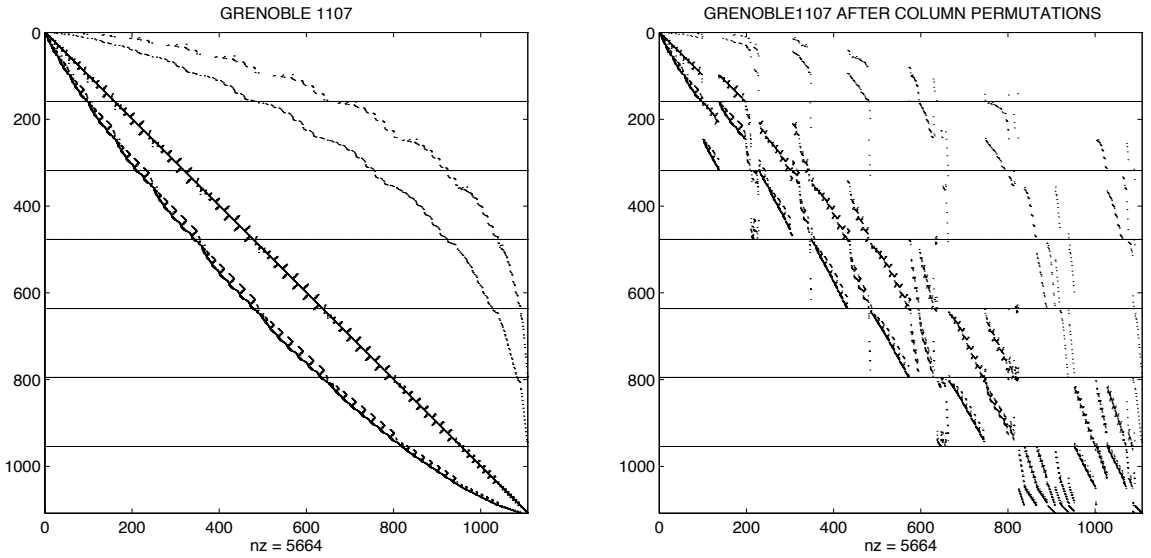

Figure 11.9: (a) GRENOBLE_1107 original sparsity pattern, and (b) after grouping the columns according to the block partitions.

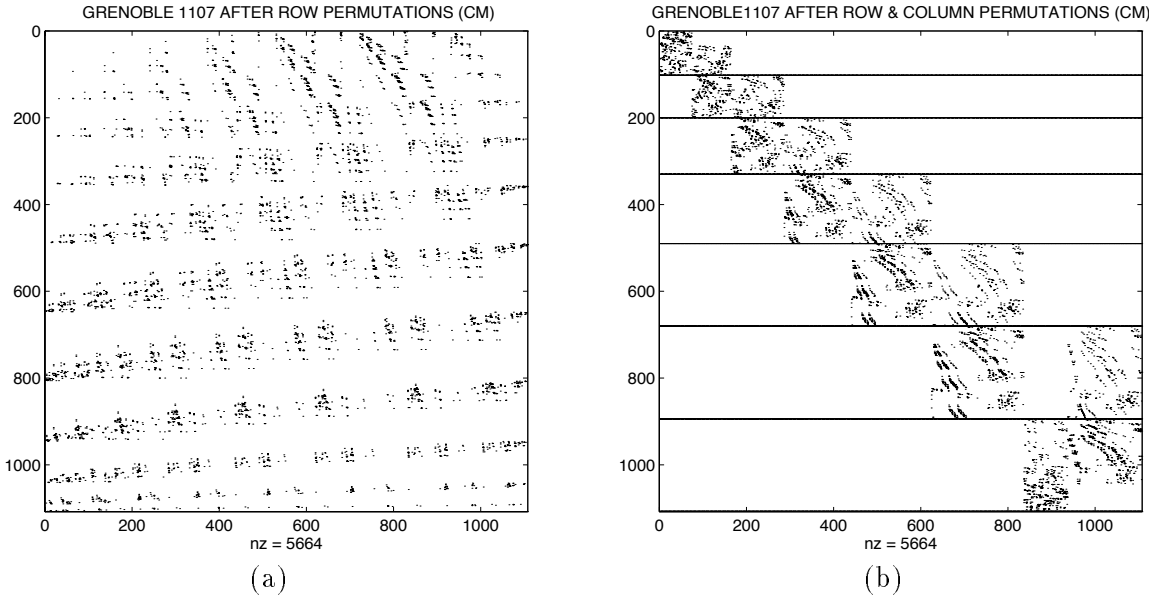

Figure 11.10: (a) GRENOBLE_1107 sparsity pattern after the first preprocessing strategy using Cuthill-McKee for reordering the normal equations, and (b) the same preprocessed matrix after column permutations.

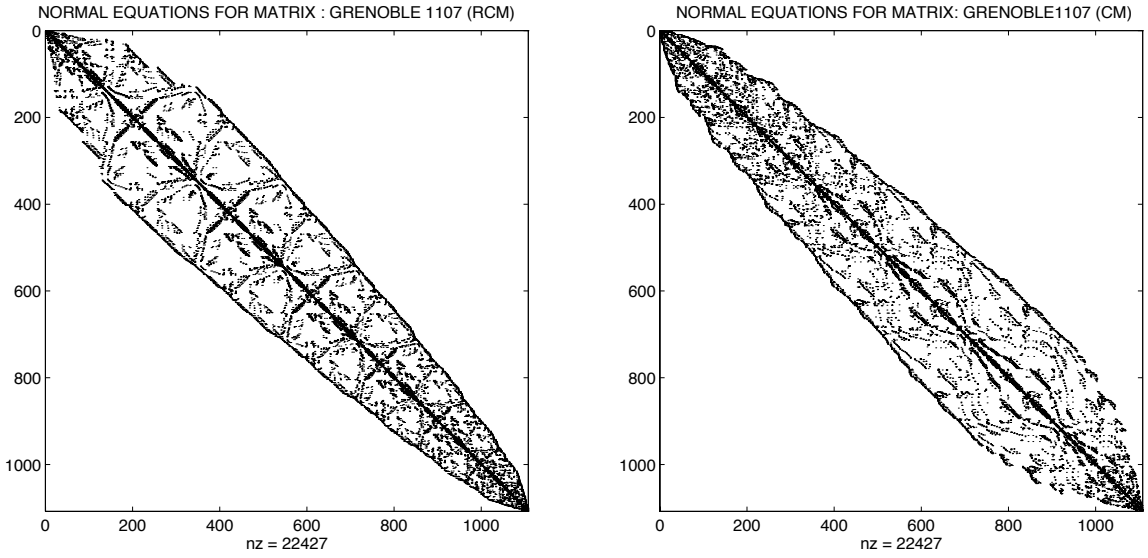

Figure 11.11: Sparsity pattern of the normal equations from GRENOBLE_1107. In (a) the Reverse Cuthill-McKee ordering is used and in (b) the Cuthill-McKee ordering is used.

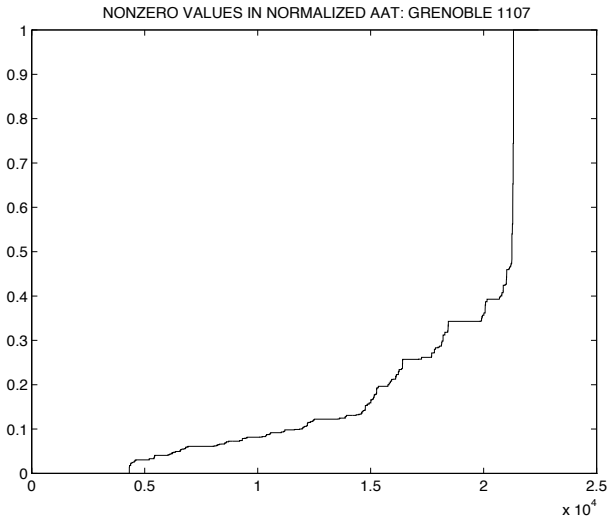

Figure 11.12: Normalized nonzeros in the normal equations from GRENOBLE_1107 matrix.

with the Cuthill McKee algorithm.
Figure 11.13-a, shows the sparsity pattern of AA^T after removing the nonzero entries less than 0.2, and permuting the matrix to a block tridiagonal form. Figure 11.13-b shows the results of using the second preprocessing strategy. The truncated matrix AA^T in Figure 11.13-a has a smaller bandwidth than the normal equations from the original linear system, shown in Figure 11.11, and the blocks of rows in Figure 11.13-b have fewer column overlappings than those in Figure 11.9-a and Figure 11.9-b.

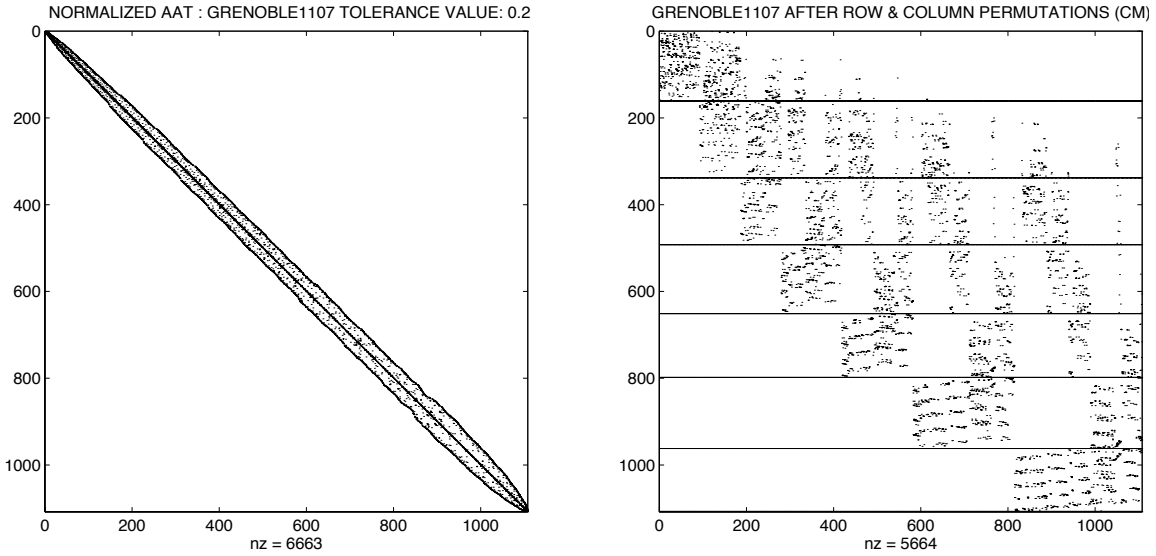

Figure 11.13: (a) GRENOBLE_1107 sparsity pattern after the second preprocessing strategy using Cuthill-McKee for reordering the normal equations and a tolerance value of 0.2. And (b) the same preprocessed matrix after column permutations.

Choosing a higher tolerance value, for instance 0.8, will lead to an almost diagonal matrix (Figure 11.14-a), and the output matrix from the second preprocessing strategy (Figure 11.14-b) is almost the same matrix as that shown in Figure 11.9-b. Therefore, removing a large number of nonzero entries from the AA^T matrix will reduce the effects of the second preprocessing strategy. In Table 11.3, the experiments with the tolerance value of 0.2 are presented under the column labeled **Preprocessing Strategy II - 7 blocks**. The following is an example of the call to the block Cimmino solver for these experiments

```
command[1]> blkcimmino -hesfile sp2env -datafile gre_1107_stg2.rua \
          -partitions 7 161 177 154 159 147 164 145 -blocksize 8 \
          -threshold 1.0D-08 -scheduling 3
```

One of the advantages of using the second preprocessing strategy over the first one is the freedom for partitioning the blocks of rows. Thus, we repeat the last experiments using the second preprocessing strategy with a tolerance value of 1.0, with the system partitioned in 12 blocks of rows. Figure 11.15 illustrates the sparsity pattern of the resulting matrix. An example of the calls to the block Cimmino solver used in these experiments is

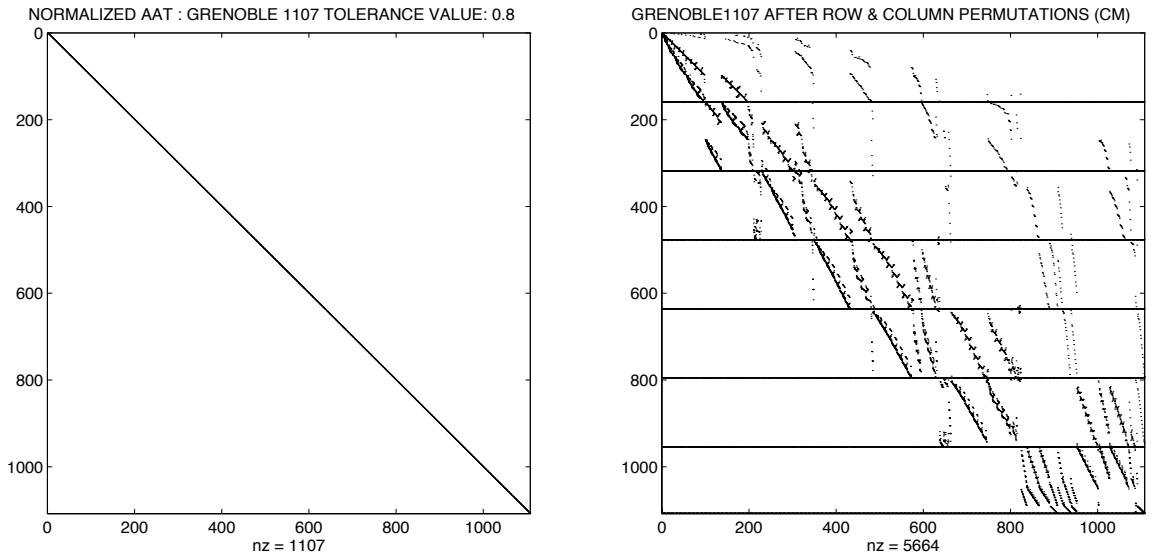

Figure 11.14: (a) GRENOBLE_1107 sparsity pattern after second preprocessing strategy using Cuthill-McKee for reordering the normal equations and a tolerance value of 0.2. And (b) the same preprocessed matrix after column permutations.

```
command[1]> blkcimmino -hesfile sp2env -datafile gre_1107_stg2b.rua \
          -partitions 12 37 77 93 95 109 124 125 119 123 111 66 28 \
          -blocksize 8 -threshold 1.0D-12 -scheduling 3
```

There is a reduction in the number of column overlaps between the blocks of rows in Figure 11.15, and those in Figure 11.13. Furthermore, the number of rows in each block is non-uniform as in the trivial partitionings shown in Figures 11.9-a and 11.9-b.

The GRENOBLE_1107 problem is interesting because the preprocessing strategy does not change the behavior of the block Cimmino solver. That is to say, the distribution of eigenvalues inside the blocks of rows does not change after applying any of the preprocessing strategies. Thus, the algorithm takes exactly the same number of steps to converge in all of the experiments (i.e., 140 iterations).
The results in Tables 11.2 and 11.3 show that using the second preprocessing strategy with a few more blocks will improve the parallel performance of the solver because the number of column overlaps between the blocks is reduced, and with more blocks more tasks can be generated and more CEs can be used.
The maximum speedups in these cases are obtained with the preprocessing strategy I on 7 CEs (6.1), and preprocessing strategy II on 10 CEs (8.0) using the third scheduling strategy. The runs with the first preprocessing strategy reported almost the same parallel behavior, in terms of efficiencies, as the second preprocessing strategy with 12 blocks. Thus, the use of the second preprocessing strategy is sensitive to the number of block partitions.

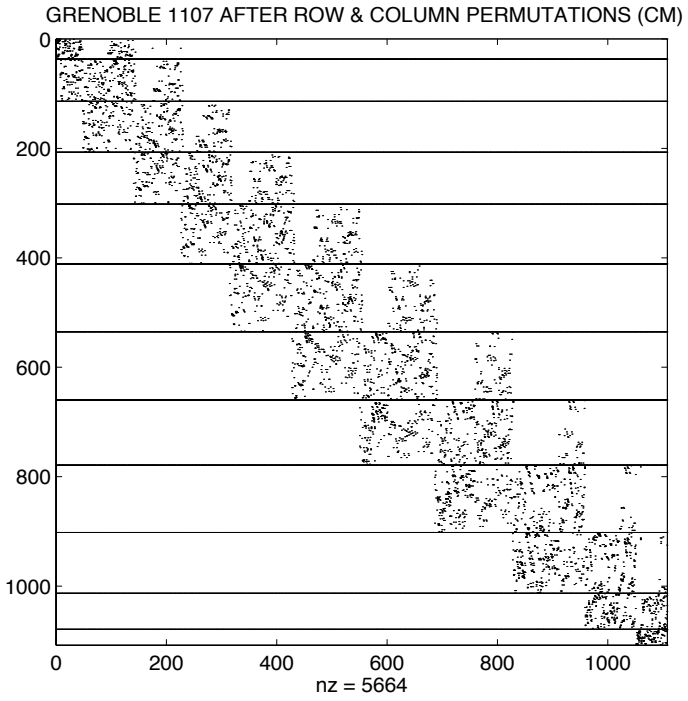

Figure 11.15: Sparsity pattern of the GRENOBLE_1107 after preprocessing strategy II with a tolerance value of 0.1. In this case 12 blocks of rows are defined.

Comp env.		Original matrix A			After column permutation			Preprocessing strategy I		
Seq. run		36.1 secs			37.3 secs			37.9 secs		
		Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$
2 CEs	stg2	34.2	1.1	0.55	32.0	1.2	0.60	26.7	1.4	0.70
	stg3	33.3	1.1	0.55	31.5	1.2	0.60	19.1	2.0	1.00
4 CEs	stg2	13.9	2.6	0.65	11.7	3.2	0.80	18.5	2.1	0.53
	stg3	12.9	2.8	0.70	10.1	3.7	0.92	11.4	3.3	0.83
7 CEs	stg2	10.0	3.6	0.52	9.5	3.9	0.56	11.5	3.3	0.47
	stg3	8.6	4.2	0.60	8.0	4.7	0.67	6.2	6.1	0.87

Table 11.2: Results from the parallel Block Cimmino with Block-CG acceleration using three different preprocessing strategies on the solution of the GRENOBLE_1107 problem. Times in this table are in seconds. stg2 and stg3 correspond to the scheduling strategies STG2 and STG3, respectively, from Chapter 8.

Comp env.	Preprocessing strategy II (7 blocks)			Preprocessing strategy II (12 blocks)		
Seq. run	38.1 secs			30.2 secs		
	Time	Spdup	$\mathcal{E}$	Time	Spdup	$\mathcal{E}$
stg2 2 CEs	30.0	1.3	0.64	24.1	1.3	0.64
stg3 2 CEs	19.2	2.0	1.00	15.7	1.9	0.96
stg2 4 CEs	20.1	1.9	0.47	13.7	2.2	0.55
stg3 4 CEs	11.3	3.4	0.84	8.4	3.6	0.90
stg2 7 CEs	12.0	3.4	0.49	8.9	3.4	0.49
stg3 7 CEs	6.8	5.6	0.80	5.0	6.0	0.86
stg2 10 CEs	-	-	-	7.8	3.9	0.39
stg3 10 CEs	-	-	-	3.8	8.0	0.80

Table 11.3: Results from the parallel Block Cimmino with Block-CG acceleration varying the number of blocks of rows in the preprocessing strategy II on the solution of the GRENOBLE_1107 problem. Times in this table are in seconds. stg2 and stg3 correspond to the scheduling strategies STG2 and STG3, respectively, from Chapter 8.

11.3 Remarks

From experiments in Ruiz (1992), the use of the augmented system approach is not very efficient when the blocks have a small number of rows compared to their number of columns. Thus, the first preprocessing strategy is preferred because it delivers a two-block partitioning with less computational effort than the second preprocessing strategy.
Alternatively, if, for some linear systems, partitionings with small blocks must be preferred, then the normal equations approach of Bramley (1989), and Bramley and Sameh (1990) should be used inside the block Cimmino solver because it performs better in terms of efficiency. And in these cases, the solver should be combined with the second preprocessing strategy to have more flexibility in the number and sizes of the blocks of rows.
The above experiments suggest that preprocessing strategies can greatly improve the performance of the parallel Cimmino solver. Furthermore, the advantages of using either preprocessing strategy will depend on the problem being solved, and the computing environment.
If a large number of CEs is available, and they are connected through a fast interconnection network, then the second preprocessing strategy can be used to generate as many blocks of rows as the number of CEs in the system. Nevertheless, a compromise must be found between the size of tasks that are generated from the block row partitions, the efficiency of the CEs, and the method used for computing the normal equations inside the block Cimmino iterations.
On the other hand, if the computational system has only a few CEs, or there are possible bottlenecks in the communication between the CEs (for instance, the bus in an ETHERNET network), then the first preprocessing strategy should be used because it produces a two-block partitioning of the original system which implicitly reduces the communication cost, and delivers bigger blocks of rows.

Chapter 12

Conclusions

Currently, there is much active work in developing Krylov based iterative methods for the solution of linear systems and eigenvalue problems. At the same time, there is a great demand for accelerating the performance of these methods on the solution of large systems of equations using efficiently the computational resources.
Various authors have studied different techniques to improve the performance of parallel implementations of conjugate gradient-type methods. Parallel implementations of Classical CG have attempted to reduce the number of synchronization points embedded in the Classical CG algorithm and in this way minimize its execution time. However, these efforts have led to either break the robustness of Classical CG or increase its complexity, and in less favorable implementations both drawbacks are found.
In general, increasing the complexity of any algorithm pays off when the more complex algorithm either extends the usefulness of the original or improves its computational rate. In both cases, the robustness and correctness of the original algorithm must be preserved. Furthermore, for computational purposes, if there is an increase in the storage requirements, then it should be linear in the size of the system of equations.

We have studied a stabilized version of the Block-CG algorithm 2.2.2 which is a generalization of the CG method. In exact arithmetic, Block-CG reduces the number of iterations of Classical CG by a factor of its block size s. In practice, this block size is chosen to be much smaller than the size of the linear system. In the stabilized Block-CG Algorithm 2.2.2, there is an increase in the storage requirements from the Classical CG Algorithm 2.1.1. This increase is only a factor of the block size times the size of the linear system.
The computational rate of the stabilized Block-CG implementation is increased as the block size is increased and we show this effect in Chapter 3. Some results from experiments are shown in Figure 12.1. In the figure, results from runs of Classical CG are reported with a block size of one. The increase in the complexity of the algorithm is caused by the use of rectangular matrices of size $n \times s$ instead of vectors of size n, and extra orthonormalizations to preserve the stability of the algorithm.
Additionally, in the Block-CG algorithm the matrix-vector operations from Classical CG are replaced by matrix-matrix operations. Thus, Level 3 BLAS routines are used in Block-CG implementations instead of Level 1 BLAS routines that are used by Classical CG. The use of the Level 3 BLAS routines causes an increase in Mflop rates as shown in Figure 12.1, due to better data locality.

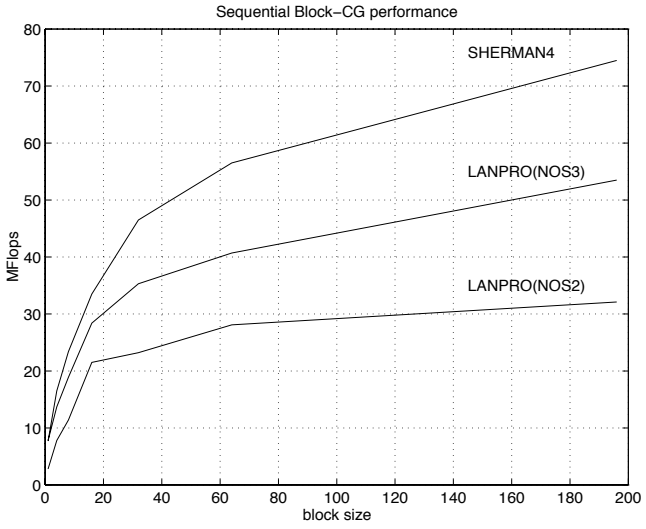

Figure 12.1: Plot of Mflops against block size from runs of a stabilized Block-CG implementation on a 550 IBM RS6000 workstation.

Several authors have studied the parallelization of the Classical CG, Block-CG, and preconditioned versions of these algorithms. We have shown, in Chapter 5, that the Block-CG is more suitable for parallel computing environments than the Classical CG because of its granularity and the implicit reduction in the number of synchronizations. From the results of the three parallel implementations of the Block-CG, we have concluded that the trivial parallelization of the HP products is not as efficient as parallelizing the entire algorithm.
Basically, the computational weight of the HP products decreases as the block size is increased. This effect, summarized in Figure 12.2, motivates our development of the other two parallel implementations of the Block-CG in which most of the Block-CG operations are performed in parallel.

The Block-CG works well as an acceleration technique inside basic iterative methods for the solution of general linear systems. We have studied a reliable implementation of the block Cimmino iteration with Block-CG acceleration. In this case, we have increased the complexity of the stabilized Block-CG algorithm to be able to solve a larger set of linear systems and have improved its overall solution time using a parallel implementation.
The performance of the parallel implementation of the block Cimmino method with Block-CG acceleration is more sensitive to the strategy used for partitioning the system into blocks of rows and the computational environment than the Block-CG algorithm is to column partitioning. We have presented two examples of preprocessing strategies to find natural partitionings of the linear system of equations.
The first preprocessing strategy provides a two-block partitioning of the original system of equations. A two-block partitioning of the linear system improves the performance of the parallel implementation by reducing the communication between blocks of rows. Additionally, a two-block partitioning, as presented in Section 10.2, can accelerate the rate of convergence of block Cimmino.

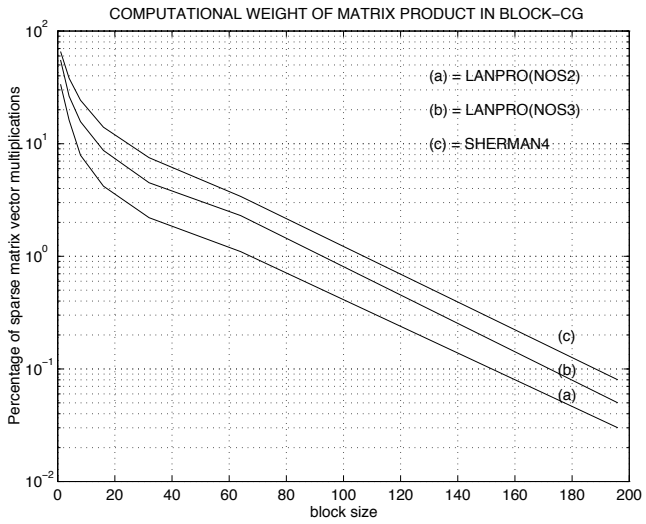

Figure 12.2: Ratio of floating point operations from the HP products over the total number of floating point operations in a Classical CG or Block-CG iteration.

However, a two-block partitioning may deliver blocks of rows with extremely non-uniform workload that may lead to unbalanced workload distributions, and thus affects the efficiency of the parallel implementation.
The second preprocessing strategy can be used to derive a more flexible partitioning than the two-block one. Generally, this strategy delivers more blocks than the first preprocessing strategy. Thus, more CEs can be used during a parallel run of the block Cimmino implementation. The second preprocessing strategy may yield very small blocks of rows, and as stated in Chapter 11. The augmented system approach on small blocks with fewer rows than columns is not very efficient. Therefore, it is recommended to regroup contiguous blocks of rows into bigger blocks. Inside the block Cimmino implementation we have used the $Master - Slave$: distributed implementation of the Block-CG. In this case, the granularity of the tasks generated by the Block-CG iteration has increased and the performance of the parallel block Cimmino algorithm is sensitive to the distribution of tasks among available CEs.
We have studied different scheduling strategies to distribute tasks among heterogeneous CEs. Basically, in many iterative methods, the number and sizes of the tasks remain almost constant from one iteration to the other. Furthermore, in several implementations, these tasks create data structures that are local to one CE and expensive to move around. For these reasons we have preferred a static scheduler over a dynamic one.
Although we have designed a scheduler for heterogeneous environments, the homogeneous cases can be regarded as instances of these. In heterogeneous environments, for each CE, its theoretical speed and the number of available processors are used as parameters to the scheduler. After partitioning the system of equations into block of rows, heuristics about the computational effort required on a block of rows can be derived. In our implementation, we use for each block of rows the number of column overlaps with other blocks, the size of the augmented system and the number of nonzero elements as parameters to the scheduler.

Both sets of heuristics, from each CE and each block of row, have lead us to implement a static scheduler of the greedy subclass. And after isolating some numerical issues that accelerate the rate of convergence of the block Cimmino solver, we have shown an improvement in the performance of the parallel block Cimmino solver due to a proper selection of the parameters passed to scheduler, and the scheduling strategy used.
For instance, Figure 12.3 illustrates an efficiency surface obtained with various runs of the block Cimmino with Block-CG acceleration in which the block size and the number of processors are varied while the number of equivalent iterations is kept constant. In the experiments of Figure 12.3, the first preprocessing strategy and third scheduling strategy were used. Using a different scheduling or preprocessing strategy lead to totally different efficiency results. In Figure 12.3, the experiments in which a block size of one was used in the Block-CG acceleration reported the lowest efficiencies. In these cases the efficiency decreases as the number of CEs is increased. For larger block sizes, the efficiency surface is smoother even when the number of CEs is increased.

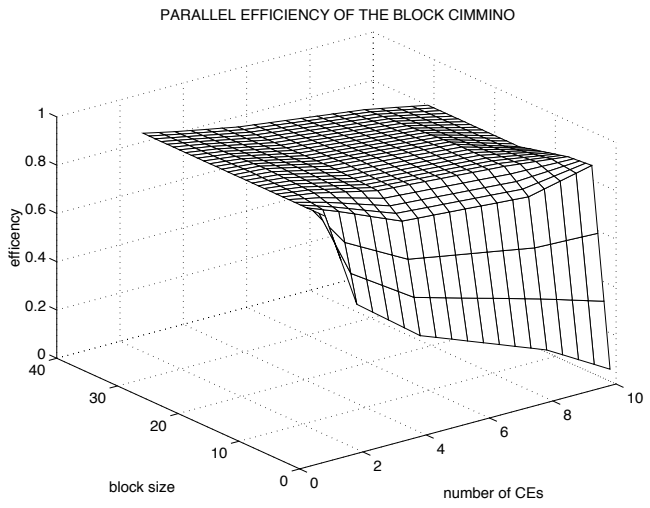

Figure 12.3: Efficiency surface from parallel runs of the block Cimmino on the solution of the SHERMAN4 problem.

The parallel strategies developed in the implementation of the block Cimmino with Block-CG acceleration can be easily accommodated to implementations of other iterative schemes. The static greedy based scheduler can be tuned to accept other input parameters from the computational environment and the linear system being solved. The strategies used in the manipulation of distributed operations like the reduce are fully reusable in other parallel programs.

The modular structure of the parallel block Cimmino solver allows reusability of some of the modules inside other iterative schemes. For instance, the module with Block-CG acceleration can be reused inside the block Kaczmarz iteration with a new module for computing the Kaczmarz projections. Furthermore, an iterative scheme has been proposed recently by Arioli and Ruiz (1995) which the authors call BlockCGSI. The BlockCGSI algorithm combines the stabilized Block-CG with the theory of subspace iteration to overcome some of the breakdowns

of the Block-CG due to ill-conditioned systems and roundoff errors. BlockCGSI increases the complexity of the original Block-CG algorithm in order to extend the applicability to general linear system of equations.

Moreover, any of the parallel Block-CG implementations presented in Chapter 4 can be reused in an implementation of the new iterative scheme. Based on experiments reported in Chapter 5, the $Master - Slave$: distributed or the $All - to - All$ implementations should be just as efficient for the new iterative scheme.

The static scheduler from Chapter 7 can also be reused for distributing the tasks among the CEs. Similarly, the software modules used in the block Cimmino and Block-CG implementations to perform the reduce operations, to gather information from the computational neighborhoods and to handle scattered data structures in parallel distributed environments can also be reused in a parallel implementation of BlockCGSI.

Henceforth, we fully expect that BlockCGSI, as well as other block iterative schemes, can show superior performance when implemented using the parallel concepts introduced in this work.

Bibliography

AEA Technology, (1993), *HARWELL SUBROUTINE LIBRARY*, Didcot, England. A catalogue of subroutine (Release 11).

G. Amdahl, (1967), *Validity of the single processor approach to achieving large scale computer capabilities*, in AFIPS Conference Proceedings., vol. 30.

E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, (1995), *LAPACK Users' Guide*, SIAM, Philadelphia, USA, second ed.

M. Arioli and D. Ruiz, (1995), *Block Conjugate Gradient with Subspace Iteration for Solving Linear Systems*, Tech. Rep. RT/APO/95/3, ENSEEIHT - IRIT, Toulouse, France. To appear in the proceedings of the second IMACS International Sympsium on Iterative Methods in Linear Algebra, Bulgaria, June 1995.

M. Arioli, J. W. Demmel, and I. S. Duff, (1989), *Solving Sparse Linear Systems with Sparse Backward Error*, SIAM J. Matrix Anal and Applics., **10**, 165–190.

M. Arioli, A. Drummond, I. S. Duff, and D. Ruiz, (1995), *A Parallel Scheduler for Block Iterative Solvers in Heterogeneous Computing Environments*, in Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing, Philadelphia, USA, SIAM, 460–465.

M. Arioli, I. S. Duff, and D. Ruiz, (1992), *Stopping Criteria for iterative solvers*, SIAM J. Matrix Anal and Applics., **13**, 138–144. Also, Technical Report TR/PA/91/58 CERFACS, France.

M. Arioli, I. S. Duff, J. Noailles, and D. Ruiz, (1989), *Block Cimmino and Block SSOR Algorithms for solving linear systems in a parallel environment*, Tech. Rep. TR/89/11, CERFACS, Toulouse, France.

M. Arioli, I. S. Duff, J. Noailles, and D. Ruiz, (1992), *A Block Projection Method For Sparse Matrices*, SIAM J. Sci. Stat. Comput., 47–70.

R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, (1993), *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM.

R. H. Bartels, G. H. Golub, and M. A. Saunders, (1970), *Numerical techniques in mathematical programming*, in Nonlinear Programming, K. R. J.B. Rosen, O. L. Mangasarian, ed., Academic Press, New York.

A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, (1992), *A User's Guide to PVM Parallel Virtual Machine*, Tech. Rep. ORNL/TM-11826, Oak Ridge National Laboratory, Tennessee 37831.

M. Benzi, F. Sgallari, and G. Spaletta, (1995), *A parallel block projection method of the cimmino type for finite Markov chains*, in Computations with Markov chains, proceedings of the 2nd international workshop on the numerical solution of Markov chains, W. J. Stewart, ed., Boston, Kluwer Academic publisher, 65–80.

A. Björck and G. H. Golub, (1973), *Numerical methods for computing angles between linear subspaces*, Math. Comp., **27**, 579–594.

S. H. Bokhari, (1979), *Dual processor scheduling with dynamic reassignment*, IEEE Trans. Software Eng., **SE-5**, 326–334.

S. H. Bokhari, (1981a), *On the mapping problem*, IEEE Trans. on Comp., **C-30**, 207–214.

S. H. Bokhari, (1981b), *A shortest tree algorithm for optimal assignments across space and time in a distributed processor system*, IEEE Trans. Software Eng., **SE-7**, 335–341.

R. Bramley and A. Sameh, (1990), *Row projection methods for large nonsymmetric linear systems*, Tech. Rep. 957, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL. Also, SISSC, Vol. **13**, January 1992.

R. Bramley, (1989), *Row projection methods for linear systems*, PhD Thesis 881, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL.

R. Butler and E. Lusk, (1992), *User's Guide to the p4 Parallel Programming System*. Mathematics and Computer Science Division, Argonne National Laboratory.

S. L. Campbell and J. C. D. Meyer, (1979), *Generalized inverses of linear transformations*, Pitman, London.

T. L. Casavant and J. G. Kuhl, (1988), *A Taxonomy of scheduling in General-Purpose Distributed Computing Systems*, IEEE transactions on software engineering, **14**, 141–154.

P. Chretienne, (1989), *Task scheduling over distributed memory machines*, in Parallel and distributed Algorithms, M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, eds., Amsterdam, Elsevier Science, 165–176.

A. T. Chronopoulos and C. W. Gear, (1991), *S-Step iterative methods for symmetric linear systems*, in Supercomputing '91, Los Alamos, CA, IEEE Computer Society Press, 578–587.

A. T. Chronopoulos, (1989), *Towards efficient parallel implementation of the CG method applied to a class of block tridiagonal linear systems*, Computer and Applied Mathematics, **25**, 153–168.

G. Cimmino, (1938), *Calcolo approssimato per le soluzioni dei sistemi di equaziono lineari*, in Ricerca Sci. II, vol. 9, I, 326–333.

E. Cuthill and J. McKee, (1969), *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings 24th National Conference of the Association for Computing Machinery, New Jersey, Brandon Press, 157–172.

E. D'Azevedo, V. Eijkhout, and C. Romine, (1993), *Reducing Communication Costs in the Conjugate Gradient Algorithm on Distributed Memory Multiprocessors*, Tech. Rep. CS-93-187, University of Tennessee, Knoxville, Tennessee. Also, LAPACK WORKING NOTE 56.

J. W. Demmel, M. T. Heath, and H. A. van der Vorst, (1993), *Parallel Numerical Linear Algebra*, Acta Numerica 93.

Dror, (1990), *Cost Allocation: The Traveling Salesman, Binpacking, and the Knapsack*, Applied Mathematics and Computation, **35**.

A. Drummond, I. S. Duff, and D. Ruiz, (1993), *A Parallel Distributed Implementation of The Block Conjugate Gradient Algorithm*, Tech. Rep. TR/PA/93/02, CERFACS, Toulouse, France.

I. S. Duff and J. K. Reid, (1983), *The multifrontal solution of indefinite sparse linear systems*, ACM Trans. Math. Softw., **9**, 302–325.

I. S. Duff, A. M. Erisman, and J. Reid, (1989), *Direct Methods For Sparse Matrices*, Oxoford University Press, New York, USA.

I. S. Duff, R. G. Grimes, and J. G. Lewis, (1992), *Users' Guide for the Harwell-Boeing Sparse Matrix Collection*, Rutherford Appleton Laboratory, Chilton DIDCOT Oxon OX11 0QX. RAL-92-086.

K. Efe, (1982), *Heuristic models of task assignment scheduling in distributed systems*, Computer, **15**, 50–56.

T. Elfving, (1980), *Block-iterative methods for consistent and inconsistent linear equations*, Numer. Math., **35**, 1–12.

A. Gabrielian and D. B. Tyler, (1984), *Optimal object allocation in distributed computer systems*, in Proceeding from the 4th International conference on distributed computing system, 84–95.

C. Gao, J. W. S. Liu, and M. Railey, (1984), *Load balancing algorithms in homogeneous distributed systems*, in Proceedings from International Conference on Parallel Processing, 302–306.

M. R. Garey and D. S. Johnson, (1979), *Computers and intractability: A guide to the theory of NP-completeness*, Freeman, San Francisco.

A. Geist, A. Baguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, (1994), *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Network Parallel Computing*, MIT Press, Cambridge, Massachusetts.

A. George, (1971), *Computer implementation of the finite-element method*, PhD thesis, Department of Computer Science, Stanford University, Stanford, California. Report STAN CS-71-208.

G. H. Golub and W. Kahan, (1965), *Calculating the singular values and pseudo-inverse of a matrix*, SIAM J. Numer. Anal., **2**, 205–225.

G. H. Golub and C. F. Van Loan, (1989), *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, second ed.

D. R. Greening, (1990), *Parallel simulated annealing techniques*, Physica D: Nonlinear phenomena, **42**, 293–306.

G. D. Hachtel, (1974), *Extended applications of the sparse tableau approach - finite elements and least squares.*, in Basic question of design theory, W. R. Spillers, ed., North Holland, Amsterdam.

L. A. Hageman and D. M. Young, (1981), *Applied Iterative Methods*, Academic Press, London.

A. Heddaya and K. Park, (1994), *Mapping parallel iterative algorithms onto workstation networks*, in Proceedings from 3rd. International Symposium on High Performance Computing.

M. R. Hestenes and E. Stiefel, (1954), *Methods of conjugate gradient for solving linear systems*, J. Res. Natl. Bur. Stand., **49**, 409–436.

J. H. Holland, (1975), *Adaptation in natural and artificial systems*, Ann Arbor: University of Michigan Press.

D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis, (1985), *How easy is local search?*, in Proceedings from Annual Symposium of Foundation of Computer Science, 39–42.

S. Kaczmarz, (1939), *Angenäherte Auflösung von Systemen linearer Gleichungen*, Bull. intern. Acad. polonaise Sci. lettres (Cracouie); Class sci. math. natur.: Seira A. Sci. Math., 355–357.

C. Kamath and A. Sameh, (1988), *A projection method for solving nonsymmetric linear systems on multiprocessors*, Parallel Computing, **9**, 291–312.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, (1983), *Optimization by simulated annealing*, Science, **220**, 671–680.

L. Kleinrock and A. Nilsson, (1981), *On optimal scheduling algorithms for time-shared systems*, ACM, **28**, 477–486.

L. Kleinrock, (1976), *Queuing Systems*, vol. 2: Computer Applications, Wiley, New York.

C. Lanczos, (1952), *Solution of systems of linear equations by minimized iterations*, J. Res. Natl. Bur. Stand., **49**, 33–53.

V. M. Lo, (1988), *Algorithms for static task assignment and symmetric contraction in distributed systems*, in Proceedings of the 11th international conference on parallel processing, The Pennsylvania State University Press, 239–244.

P. R. Ma, E. Y. Lee, and M. Tsuchiya, (1982), *A taks allocation model for distributed computing systems*, IEEE Trans. on Comp., **C-31**, 41–47.

E. W. Mayr, (1988), *Parallel Approximation Algorithms*, Technical Report STAN//CS-TR-88-1225, Stanford University, Department of Computer Science.

G. Meurant, (1987), *Multitasking the Conjugate Gradient method on the CRAY X-MP/48*, Parallel Computing, **5**, 267–280.

T. Mizuike, Y. Ito, D. Kennedy, and L. Nguyen, (1991), *Burst Scheduling Algorithms for SS/TDMA Systems*, IEEE trans. on commun., **COM-39, 4**, 533–539.

J. Modersitzki, (1994), *Conjugate Gradient Type Methods for Solving Symmetric, Indefinite Linear Systems*, Tech. Rep. 868, University of Utrecht, Netherlands.

A. A. Nikishin and A. Y. Yeremin, (1995), *Variable Block CG algorithms for solving large sparse symmetric positive definite linear systems on parallel computers, I: General iterative scheme*, SIAM J. Matrix Anal. Appl., **16**, 1135–1153.

W. Oettli and W. Prager, (1964), *Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides*, Numer. Math., **6**, 405–409.

D. P. O'Leary, (1980), *The Block Conjugate Gradient algorithm and related methods*, Linear Algebra Appl., **29**, 293–322.

D. P. O'Leary, (1993), *Iterative methods for finding the stationary vector for Markov chains*, IMA Volume on Linear Algebra, Markov Chains, and Queueing Models, **48**, 125–136.

C. C. Paige and M. A. Saunders, (1975), *Solution of sparse indefinite systems of linear equations*, SIAM J. Numer. Anal., **12**, 617–629.

G. N. S. Prasanna and B. R. Musicus, (1991), *Generalised Multiprocessor Scheduling Using Optimal Control*, ACM, 216–228.

D. Ruiz, (1992), *Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment*, PhD thesis, CERFACS, Toulouse, France.

Y. Saad, (1985), *Practical use of polynomial preconditionings for the Conjugate Gradient method*, SIAM J. Scientific and Statistical Computing, **6**, 865–881.

Y. Saad, (1988), *Krylov subspace methods on supercomputers*, tech. rep., RIACS, Moffett Field, CA.

Y. Saad, (1989), *Krylov subspace methods on supercomputers*, SIAM J. Scientific and Statistical Computing, **10**, 1200–1232.

C. Shen and W. Tsai, (1985), *A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion*, IEEE Trans. Comput., **C-34**, 197–203.

H. D. Simon, (1984), *The Lanczos Algorithm with Partial Reorthogonalization*, Mathematics of Computation, **42**, 115–142.

H. D. Simon, (1985), *Incomplete* **LU** *preconditioned conjugate-gradient-like methods in reservoir simulation*, in Proceedings of the Eight SPE Symposium on Reservoir Simulation, 387–396.

J. Stoer and R. Bulirsch, (1980), *Introduction to Numerical Analysis*, Springer-Verlag, Berlin.

H. S. Stone, (1978), *Critical load factors in two-processor distributed systems*, IEEE Trans. software Eng., **SE-4**, 254–258.

G. Strang, (1986), *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Cambridge, MA.

E. G. Talbi and T. Muntean, (1993), *General heuristics for the mapping problem*, in World Transputer Congress, Aachen, Germany, IO Press.

H. A. van der Vorst, (1992), *Lecture notes on iterative methods.* Mathematical Institute, University of Utrecht, The Netherlands.

O. C. Zienkiewikz and R. L. Taylor, (1989), *The Finite Element Method*, vol. 1 (Basic Formulation and Linear Problems), 2 (Solid and Fluid Mechanics, Dynamics and Non-Linearity), McGraw Hill International Editions, fourth ed.